Web-based Programming for Low-cost Gaming Handhelds

Michał Moskal
mimoskal@microsoft.com
Microsoft Research
USA

Peli de Halleux
jhalleux@microsoft.com
Microsoft Research
USA

Thomas Ball
tball@microsoft.com
Microsoft Research
USA

Abhijith Chatra
abchatra@microsoft.com
Microsoft
USA

James Devine
t-jamdev@microsoft.com
Microsoft Research
UK

Steve Hodges
shodges@microsoft.com
Microsoft Research
UK

Shannon Kao
shakao@microsoft.com
Microsoft
USA

Richard Knoll
riknoll@microsoft.com
Microsoft
USA

Galen Nickel
v-gani@microsoft.com
Microsoft
USA

Jacqueline Russell
jacqchen@microsoft.com
Microsoft
USA

Joey Wunderlich
jowunder@microsoft.com
Microsoft
USA

Daryl Zuniga
daryl.zuniga@gmail.com
Microsoft
USA

ABSTRACT

Low-cost microcontroller boards like the BBC micro:bit are used to engage and inspire students worldwide to learn more about computing. Easy-to-use web-based programming environments and low-cost hardware allow novices to build *physical computing* systems with the micro:bit – systems that sense and respond to the real world. However, devices such as the micro:bit may not capture the attention of every student, as the interests of some may lie in graphic design, animation, or other areas that are not the main focus of physical computing.

Video game creation offers the opportunity for students to engage with computing concepts from a different angle, while keeping open the possibilities for physical computing. To date however, there is no game creation platform that has both the same low-barrier to entry and low-cost hardware as the BBC micro:bit.

We present MakeCode Arcade, a web app for creating video games for gaming handhelds, built on the same technologies as the BBC micro:bit, extended to support multiple microcontrollers and including a 2D game engine and easy-to-use sprite and music editors. We also designed a simple hardware specification that resulted in five different companies creating seven Arcade-compatible handhelds. Qualitative and quantitative evaluation demonstrates that Arcade enables a modern and fully web-based programming experience for low-cost microcontroller-based gaming handhelds.

KEYWORDS

video games, programming, gaming handhelds, web browser

1 INTRODUCTION

Since the introduction of Arduino in 2005 [18], one option open to educators who want to make computer science lessons more engaging is the use of *physical computing* devices based on microcontroller technology. Such devices allow students to build interactive systems that sense and respond to the real world [11]. In classrooms around the world, physical computing has been shown to increase student engagement, especially among girls, and helps with confidence in both educators and students [17].

The BBC micro:bit, shown in Figure 1(a) and (b), is a physical computing device that has seen solid success in the classroom over the last five years [2]. The micro:bit has a retail price of about $15 (US), is half the size of a credit card, and incorporates a 5x5 LED display, two buttons, a programmable microcontroller unit (MCU), an accelerometer, a magnetometer, Micro-USB interface and battery port, as well as a Bluetooth Low Energy (BLE) radio. The micro:bit is now used to teach computer science in over 60 countries, and has to date shipped over 5 million units. A web-based programming environment for the micro:bit, Microsoft MakeCode [9], is a key part of its success. MakeCode enables programs to be created using drag-and-drop blocks (see Figure 1(c)), run in the web browser using the micro:bit simulator (left-side of Figure 1(c)), compiled

to machine code, and finally transferred to the device over a USB cable.

Although the micro:bit offers many exciting and engaging physical computing experiences, its limited 5x5 display makes it less than ideal for video games, which have the potential to further engage students [24]. Our vision is to bring together low-cost yet capable MCU-based hardware with the web-based simplicity of MakeCode to create an end-to-end experience for video game programming, with the potential to add physical computing via external sensors and actuators. Video game programming for MCU-based devices has much to offer: the domain is naturally interesting to many students, encompasses a wide range of computing and programming concepts [20], and the combination of video gaming and physical computing has the potential to unlock new hybrid scenarios [16, 19].

Towards this end, we created a simple and low-cost hardware design for MCU-based gaming handhelds, backed by a new web-based application called MakeCode Arcade, shown in Figure 2, which we developed on top of the open source PXT framework that MakeCode is built upon. With Arcade, users can design game assets (sprites, tile maps, melodies), program a game using a drag-and-drop graphical programming environment, and test/debug/play their games – all via the web browser –and then deploy them to compatible gaming handhelds.

To keep costs down, our hardware design only requires a low-resolution screen, basic sound generation, and cheap tactile controls. This minimal set of features can be supported on low-cost MCUs, but their use introduces processor and memory constraints. These constraints mean that Arcade only supports a retro gaming experience [3]. Although the look and feel of Arcade calls out to retro gaming experiences of the past, the Arcade programming experience is modern and comparable to Scratch [14], both of which use Google's Blockly framework [10]. A wide variety of video games can be implemented with blocks, while the underlying full-featured object-oriented APIs are exposed in TypeScript (JavaScript with types, see www.typescriptlang.org) for more advanced scenarios.

Our main contributions are:

- The design of a gaming handheld specification, with a simple and inexpensive realization using commodity MCUs and hardware components. Our goal here is to spur third-party hardware manufacturers to create Arcade-compatible handhelds with their own unique features – we do not presume to know what combinations of gaming and physical computing will appeal to users.
- The design of layered game APIs that provide a low floor for novices to easily get started using graphical blocks.
- The implementation of the core game engine in a subset of TypeScript called Static TypeScript [4], supported by the PXT framework; the same game engine runs in both the web browser and on the gaming handhelds.
- Major modifications to the PXT framework to support a variety of MCUs, and improve the performance of PXT's compiler and runtime.

We evaluate the deployment of Arcade via:

- A qualitative assessment of seven gaming handhelds that have been produced by third parties in 2019-2020 based on our core design, ranging in retail price from $25 to $50 (US).

- A synthetic benchmark to evaluate game engine performance and characterize the limits of the gaming handhelds.
- Qualitative and quantitative assessment of games written by the Arcade community over three game jam [21] contests run in 2020.

Most of the programs created by the Arcade community in the game jams run successfully on the lowest-powered device produced by the third parties, showing that with careful design it is possible to bring together a modern web-based programming experience with low-cost gaming handhelds.

The rest of the paper is organized as follows: Section 2 discusses background and related work, Section 3 describes the design goals and constraints of Arcade, Section 4 discusses the major challenges in the implementation of Arcade, Section 5 presents our evaluation, and Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

In his book on "retrogames" (old computer games), Aycock[3] says:

> "*One theme that runs throughout this book is the idea of constraints. Retrogame programmers were limited in ways that would be nearly incomprehensible to a modern programmer.*"

Aycock posits that constraints on video games come in three forms: the computing system on which the game runs, constraints on the developer, and constraints on the player. A key point of Arcade is to embrace the constraints of retro video games in terms of their low-resolution graphics and the limited memory and processing power available to their programmers, exemplified by the low-cost MCU-based hardware we choose to use. At the same time, we approach programming from a more modern perspective, leveraging the power of the modern web browser on capable laptop and desktop systems.

Related work breaks roughly into two categories: (1) environments specifically for retro video game creation; (2) game creation environments for novices. Arcade occupies a unique point in the design space, providing a web-based game creation environment for video games, with first-class support for a variety of gaming handhelds.

2.1 Retro game creation environments and communities

The Gamebuino META [1] is a gaming handheld device based on the Arduino Zero that uses the ARM Cortex M0 CPU, with 256 KB flash and 32 KB RAM. The META can be programmed using the Arduino IDE (C/C++) or with CircuitPython. In comparison, Arcade allows web-based programming using Blockly and/or JavaScript. Our analysis of the design space (Section 3) suggests that the faster Cortex M4 CPU with at least 96 KB of RAM is a better starting point for responsive games that use double buffering (to be fair, it has been three years since the debut of the META and we benefit from the fact that Moore's law still applies to MCU-based technology).

GB Studio [13] is a game creation environment that runs on Windows, MacOS, and Linux, and allows the user's game to be exported as a Game Boy compatible ROM file that can be played in several Game Boy emulators (OpenEMU and KiGB). GB Studio's
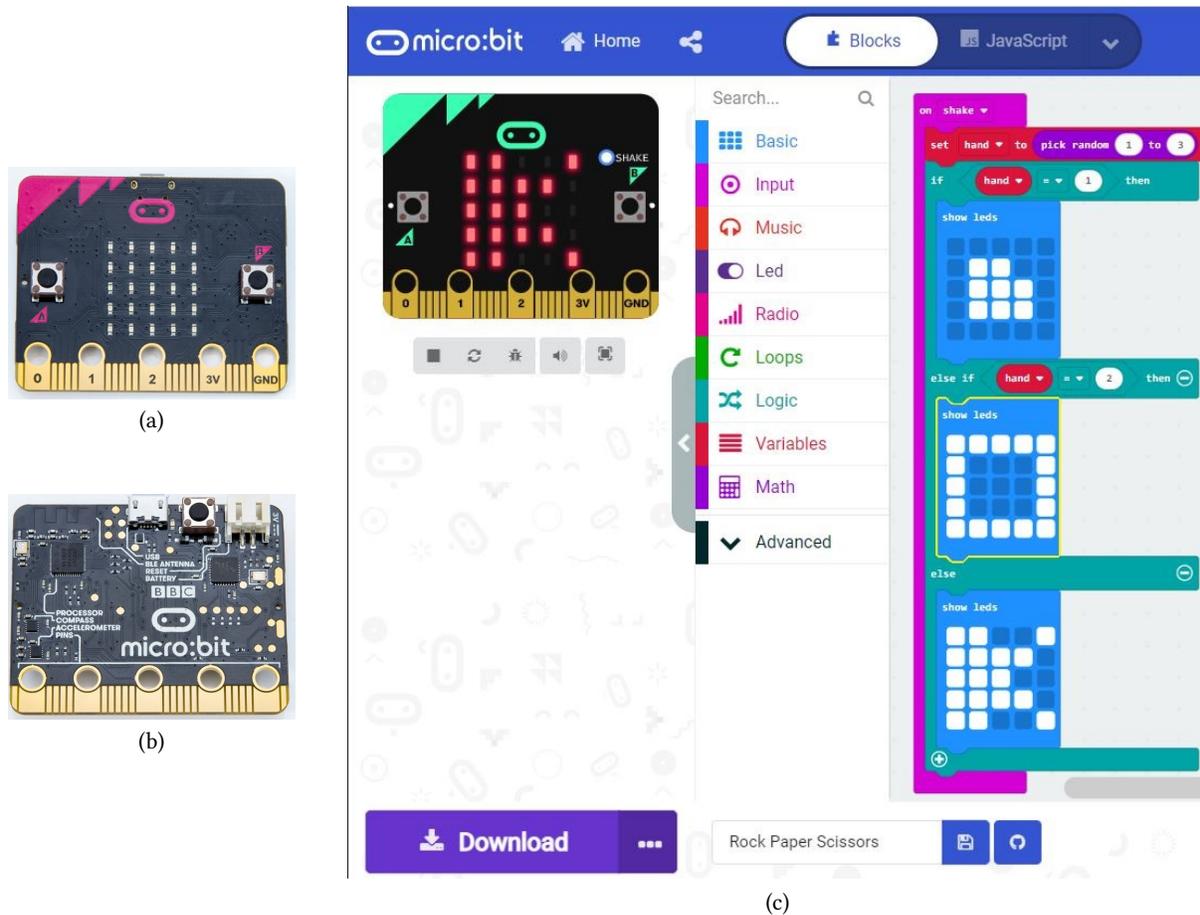
Figure 1: The BBC micro:bit: (a) front; (b) back; (c) MakeCode micro:bit web app, with popular "Rock Paper Scissors" program.

programming model is menu-based and heavily depends on built-in behaviors of a large assortment of sprite kinds. In contrast, Arcade provides a general-purpose programming experience.

There also are small but vibrant communities that focus on writing new games for classic gaming systems (e.g. https://www.retroguru.com). These communities often use the tools of that era (6502, 68000 assembler) and target specific machines (via emulators), or write in ANSI C for multi-platform gaming. ROM hacking, the process of modifying the ROMs of classic games, is also popular but is the domain of the highly technical.

PICO-8 is a very popular application for designing, programming (via the Lua scripting language), and playing retro video games on modern operating systems [23]. As stated on the PICO-8 web site, "A fantasy console is like a regular console, but without the inconvenience of actual hardware." PICO-8 games, as well as the creation environment, are limited to a 128x128 pixel, 16-color display; the PICO-8 virtual machine exposes memory maps and similar 8-bit inspired features to the programmer. PICO-8 gaming requires a full operating system to run the emulator, while Arcade compiles games to run directly on low-resource MCUs. TIC-80 [15] shares PICO-8's approach, also defining a "fantasy console" with intentional limitations copied from the retrogame era.

2.2 Novice game creation environments

There are many game programming environments for children/novices, primarily with the purpose of introducing computing concepts and programming through the medium of video games [24]. Many of the environments use block-based programming and eschew the use of text-based programming languages. As the games run on modern computers (either as a native application or in the web browser), they often take advantage of graphics (for example, smooth 3D graphics made possible by a GPU) and media features that fall outside of the constraints of the gaming handhelds we target. Two of the popular educational environments are:

- **Alice** [6, 12], which uses an agent-based object-oriented paradigm for novice programming of interactive animations. Wernet et. al [22] showed that middle-school students were able to program games with Storytelling Alice (a version of Alice with more support for high-level animations) in about 20 hours over a two-week session – most of these games used 3D.
- **Scratch** [14], which uses Blockly for drag-and-drop programming. Like Alice, the Scratch programming model is actor-based with each sprite having an associated set of
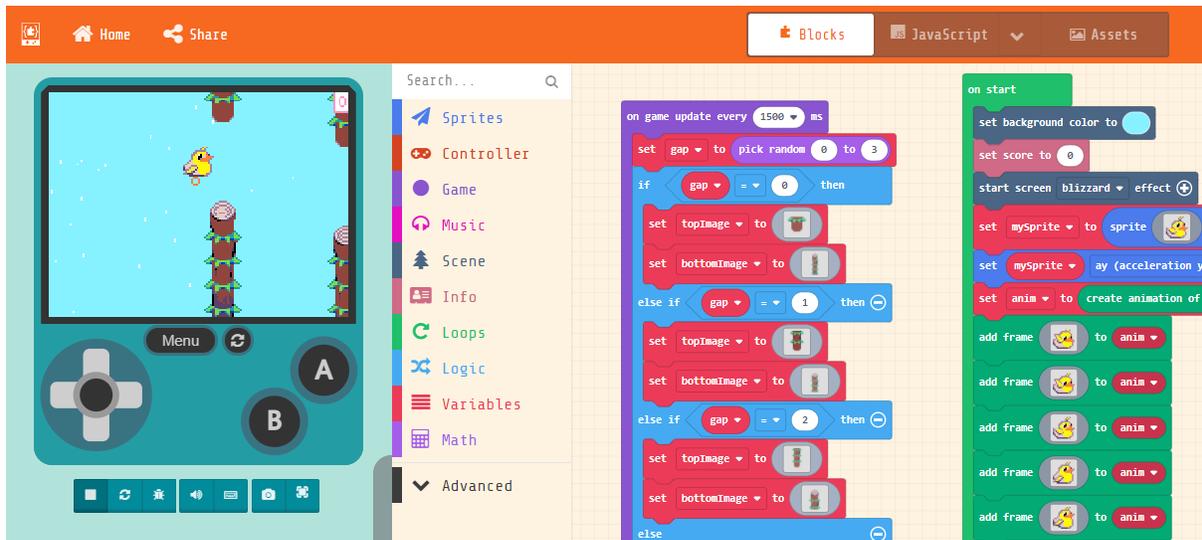
Figure 2: MakeCode Arcade web app (top) and devices (bottom) running the program shown in the web app. Left picture: GHI BrainPad, Kittenbot Meowbit, TinkerGen GameGo. Right picture: experimental micro:bit shield, Adafruit PyBadge.

scripts for the variety of events that can take place (collisions, button-press, etc.) and using message broadcast to coordinate with other sprites.

In contrast to both Alice and Scratch, the Arcade programming model is not actor-based. It uses a standard game update loop in which the programmer can iterate over all sprites to synchronize their activity, as well as traditional event handlers for reacting to user input and colliding sprites. Furthermore, Arcade supports compilation to machine code and deployment to MCU-based hardware.

Although Scratch has the capability to work with hardware such as the BBC micro:bit, the devices are treated as I/O peripherals: the user's program executes in the web browser and I/O commands are sent from/to the peripheral device.

The popularity of Python is ever increasing, especially with its use in data science and machine learning, and Pygame is a well-known python library for creating games [7]. While many Pygame games have a decidedly retro feel to them, they require at least a Raspberry Pi computer to run.

3 ARCADE DESIGN SPACE

The vision of Arcade is to provide a web app that allows users to create a wide variety of video games that can then be compiled and deployed to low-cost MCU-based gaming handhelds. The two major constraints we operated under are as follows: we decided to build Arcade using the PXT framework that underlies the MakeCode web app for the BBC micro:bit, as it already supports coding in the web browser with compilation to ARM-based MCUs; we wanted to create an ecosystem of Arcade hardware, leveraging third party companies to design, manufacture and distribute Arcade-compatible devices.

Arcade was guided by the following three goals: (1) to define a hardware specification that allows third parties to innovate and create novel gaming handhelds that will integrate seamlessly with the Arcade software stack; (2) to create a 2D game engine using Static TypeScript (the core language supported by PXT that is compiled to machine code) that will perform acceptably for a wide variety of games, on both browser and handheld; (3) to provide simple user-facing APIs for programming against the game engine, as well as the ability extend the core game engine with genre-specific libraries.

3.1 Hardware Specification

Cost is a key part of accessibility; to make Arcade as accessible as possible we aimed for a retail price of under $20 (US) for the simplest gaming handhelds. To reach this price and maintain margins typical for consumer electronics, the hardware bill of materials must be around $5 or less.

The dominant costs in gaming handhelds are the LCD screen and the MCU. In the $2 range, LCDs are typically 1.4 inch at 128x128 pixels, or 1.8 inch at 160x128 and 18-bit color. There also are screens in the $4+ range between 2.0 and 3.2 inch at 320x240. To enable market diversity, we decided on a 3:4 aspect ratio and 160x120 resolution as a compromise between the smaller and larger screens. This allows us to use 94% of the area of a 1.8 inch screen and scale up efficiently to 320x240 for more expensive displays.

MCU capability also is important when considering the cost of gaming handhelds. MCUs under $2 typically have no more than 128 KB of RAM and 512 KB of flash (non-volatile memory, which is slow to write and used to store program and data), and run at no more than 120 MHz. The RAM size constraints restrict the resolution and bit-depth of displays. For example, a screen buffer of 160x120 at 18-bit color depth would require a memory buffer size of 42 KB. For smooth gameplay, a second screen buffer is needed. With various sprite operations, the handling of in-game dialogs, and user program variables, the demands of the platform would quickly exceed the capabilities of many low cost MCUs. Instead of increasing hardware cost, we decided to restrict the color depth of the display to 4-bit indexed color.[1] The color palette, referred to by this 4-bit index, can be dynamically changed at run time.

For user input, Arcade minimally requires 4 directional buttons (a d-pad or analog joystick can also be used), two action buttons (A and B), and 1 utility button (menu/pause). Most Arcade devices also have a hardware reset button and/or an on/off switch. Other

hardware components such as a speaker and a USB connector for programming the device, together with assembly and supporting passive components typically comes to less than $1. Optional components, like a battery or enclosure can add a few more dollars. This combination of features and functionality is similar to that of the classic Nintendo Game Boy.

3.2 Retro Game Assets and Core Engine

We focus our attention on supporting 2D graphics, which covers the majority of retro games.[2] The core low-level data structures required are bitmaps for encoding color images, tile maps, and sprite animations. Tile maps are a space-saving device for representing a game level, which is a grid of color images. As images are often repeated in the grid, each entry of a tile map simply refers to the index of a particular image (along with some other meta-data, e.g. whether or not sprites can pass through it). A simple but powerful UI for creating/modifying images, tile maps, and animations is absolutely essential, as users will spend a lot of time crafting game assets. The game engine needs to be designed to work on MCUs, as described above, where memory is severely limited (96 KB of RAM being the minimum).

3.3 High-level APIs and Extensibility

While low-level data structures are needed by the game engine to effectively run on MCUs, we present a high-level view of game abstractions to programmers using a simple object model exposed via TypeScript APIs. The PXT framework supports annotating TypeScript APIs with metadata to expose a further simplified view of these APIs via the Blockly framework.

Despite its restrictions, there are a wide variety of video game genres that are possible within the retro domain: adventure, racing, platform, puzzle, tower defense, sports, shooter, etc. To simplify programming, each of these genres may require additional APIs, so it's important to provide a core set of APIs that support a wide variety of game genres, as well as specific libraries that simplify programming games in a particular genre.

4 ARCADE IMPLEMENTATION, BOTTOM-UP

This section describes the main advances that were required to bring Arcade to life, starting with the hardware design and firmware at the bottom of the stack, and working up to the PXT framework, APIs and user interface. Some design decisions required major changes to the PXT framework, as we will note.

4.1 Hardware and Firmware

PXT was designed to support a single device type per web app [9]. A major challenge we faced was to give manufacturers a choice of MCU and value-added peripheral components to create their Arcade-compatible gaming handheld(s), while ensuring that the machine code generated by PXT would always work.

---

[1] It might have been possible to use 8-bit color, but even with 4 bits we are already often close to RAM limits.

[2] Various pseudo-3D effects, e.g. "mode 7" road racing and ray casting *à la* original Wolfenstein 3D, can be implemented too. But these don't benefit from our sprite and tilemap abstractions.
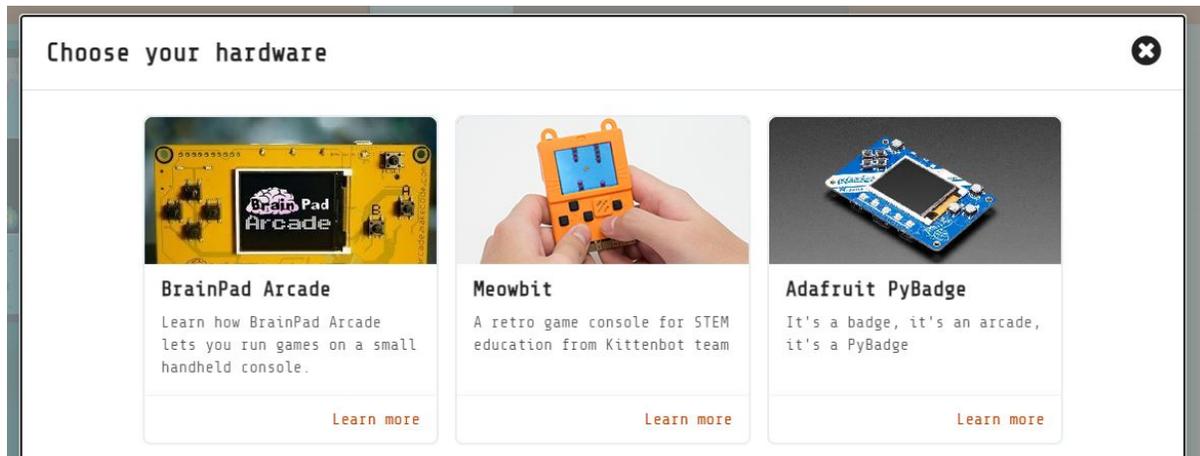
Figure 3: Dialog for choosing device.

While there is little choice for inexpensive, widely available small screens[3], there are many MCUs that fit our needs. Because PXT already generates ARM Thumb code from Static TypeScript, we decided to only support ARM-based MCUs. Arcade currently supports several MCU variants, all of which happen to be based on an ARM Cortex-M4F core:[4]

- D5 - a Microchip ATSAMD51G19A (192 KB RAM, 512 KB flash, 120 MHz);
- F4 - one of the STMicroelectronics STM32F4xx MCUs, ranging from STM32F401xE (96 KB RAM, 512 KB flash, 84MHz) to STM32F412xG (256 KB RAM, 1024 KB flash, 96 MHz);
- N3 - Nordic Semiconductor NRF52833 (128 KB RAM, 512 KB flash, 64 MHz) and N4 - NRF52840 (256 KB RAM, 1024 KB flash, 64 MHz);

All these MCUs have USB device circuitry and expose a virtual USB drive when connected to a computer. The PXT framework compiles a user's program to a binary, which can be copied to that drive to re-program the device. This programming is handled by a bootloader, which is installed by the manufacturer at the top of the MCU's flash memory. The bootloader takes care not to overwrite itself.

While the machine code generated from the user's program and the game engine is the same for all these MCUs, the supporting runtime is different. Even though the MCUs share the same ARM processing core, they have different peripherals for external interfaces (e.g. SPI for talking to the screen, USB for programming, or DAC for sound generation), and different internal features (clocks, timers, access to flash, etc.). Thus, depending on the choice of MCU, a different binary is generated by PXT.[5] As shown in Figure 3, the user chooses the MCU variant by selecting a particular handheld from a grid with pictures. There are also generic fallback options

for each MCU variant; for example, selecting BrainPad, Meowbit, or F4 has the same effect.

On top of MCU variance, third-party manufacturers can build their Arcade handhelds in different ways: they can choose the smaller 1.8 inch screen or a bigger one; they can include various optional peripherals such as an accelerometer, vibration motor, and/or RGB LEDs; they can also connect these periperhals to different physical pins of the MCU. We thus require manufacturers to include configuration data in the bootloader area of the flash. The configuration is a simple integer key to integer value mapping, where the keys are pre-defined in the PXT framework. Typical keys are *accelerometer type*, or *accelerometer SDA pin*. This data is used at runtime to access the appropriate hardware.

Typically, in the embedded world, if a single binary is expected to run on multiple types of hardware, some MCU pins are physically tied to 0 or 1 and used as feature flags. We believe that storing this configuration data in the bootloader space is unique to Arcade, and it allows a much larger space of configurations to be expressed.[6] Hardware designers don't need to ask us for permission for building the devices they want, they just need to specify the configuration data correctly – thus fostering an open ecosystem of hardware.

In addition to the MCUs listed above, Arcade games can run on a variety of more powerful hardware:

- in web browser, as supported by PXT, where the game runs inside the simulator we created (see left-hand side of Figure 2);
- Raspberry Pi Zero, with a powerful and power-hungry ARM11 chip (512MB RAM, 1GHz; note the 1000x more memory and 10x faster clock compared to the MCUs) - via native compilation to Linux ELF format;
- various C++ native apps using libSDL (iOS, Android, Windows, Mac, Linux) via compilation to a custom byte-code that runs in a virtual machine (VM).

---

[3] There are many manufacturers but the screens themselves are commoditized: they look the same and have the same interface.

[4] The compiled TypeScript code runs on various other Cortex cores, and also on ARM11.

[5] It would be theoretically possible to detect the MCU at runtime, and access peripherals accordingly. However, this would significantly increase the size of the runtime, leaving little space for a user's program and game assets.

[6] A yet more flexible option would be to express peripheral configuration data in manufacturer-provided firmware code, but our sense is that this may add too much complexity for hardware manufacturers.

4.2 Language, Compiler, and Runtime

As mentioned before, PXT uses a subset of TypeScript dubbed Static TypeScript (STS) [4]. The main limitation of STS compared to regular TypeScript is the lack of eval and prototype inheritance. As in regular TypeScript, values are ultimately typed at runtime, there is only one numeric type (64-bit IEEE double; 31-bit tagged integers are used as a transparent performance optimization). Classes, interfaces, namespaces, and first-class functions are also supported.

The STS compiler is written in regular TypeScript and can run in a web browser, generating ARM Thumb machine code (as well as continuation-passing-style JavaScript and VM byte-code). The generated binaries are linked with a pre-compiled runtime implemented in C and C++. The runtime contains a garbage collector, methods of built-in data types (numbers, strings, binary buffers, arrays, dictionaries, functions, but also bitmap images), and support code for exceptions, runtime type reflection, etc. In case of MCUs, the runtime builds on CODAL [9], a C++ library that exposes common abstractions for various hardware, as well as a non-preemptive thread scheduler. The Raspberry Pi and VM ports rely instead on regular POSIX APIs.

The Arcade game engine served as a stress test for the STS system, which previously had been used mainly for small user programs and low-level code that didn't make much use of the object system of TypeScript, and so had a very modest memory requirements. We replaced PXT's reference counting framework with a precise mark-and-sweep garbage collector and implemented several previously unsupported language features to make it easier to write the game engine: dynamic type checks and field access, getters/setters, exceptions, and Unicode strings (encoded transparently as UTF-8 to save memory). The games stressed system performance and led to performance optimizations of the runtime and generated code. Also, with the growth of the game engine and user programs, the in-browser compiler had to be made faster, and eventually incremental.

4.3 Game Engine

The Arcade game engine is just over 11,000 lines of Static TypeScript comprising 75 classes, 145 exported functions and 19 enumerations. This count includes 769 lines of meta-data comments that define the mapping from TypeScript APIs to Blockly. Altogether, there are 107 blocks. The game engine is organized in six categories, into which the above APIs and blocks map:

- A *Sprite* has an associated image, (x,y) position, velocity, acceleration, and friction; any number of sprites may be allocated, which means that a game that runs fine in the web browser may run out of memory on the gaming handheld.
- The *Controller* category has functions and event handlers for the d-pad and the A and B buttons; one of the most useful functions binds the d-pad to a sprite to allow the player to move the sprite (by mapping directions to sprite velocities).
- The *Game* category has functions and event handlers for the basic life cycle of the game: splash screen, game update handler, dialogs, and game over screen. More details on the game loop are given below.
- the *Music* category enables the creation of sound effects and music with up to four voices;

- The *Scene* category is critical for games that involve multiple levels – each scene of a game corresponds to a game level with its own set of event handlers, tile map, and camera. Scenes are arranged in a stack, with pop and push operations.
- The *Info* category contains functions for managing the score, player lives and multi-player games.

The game loop sequences all the actions and updates to the current game scene, based on physics and events sent to the sprites. The loop frame interval is timed internally to provide frequent enough updates for smooth scene transitions [5]. In each frame, the following steps take place:

- The controller state is updated, identifying whether buttons are pressed, released, or held down.
- Velocity is set on sprites controlled by the user, depending on which buttons are pressed.
- Physics are applied to the sprites, computing the next positions of all sprites given their velocity, acceleration, and friction values.[7] Collisions are detected and appropriate events are run.
- Game update events are run.
- The current state of the game is rendered to the screen.

The game engine also supports text, sprite animations, particle effects and a system menu.
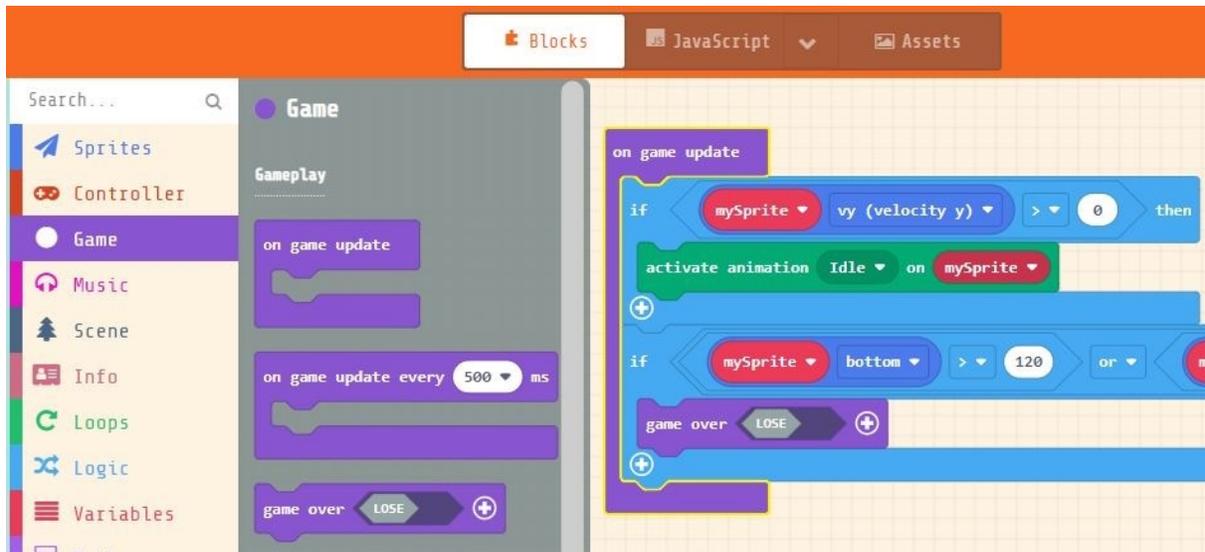
The game engine is implemented with an eye towards performance when compiled to run on hardware. The most prominent example is the internal usage of fixed point math for sprite position, velocity, etc. While all the MCUs that Arcade currently runs on feature a hardware floating point unit (FPU), they all only support single-precision IEEE floats. JavaScript (and hence TypeScript) semantics mandates usage of double-precision floats for all numbers,[8] so the FPU cannot be used, while software floating point is around two orders of magnitude slower. Thus, when numbers fit a 31-bit signed integer (one bit is used as a tag), STS uses integer operations saving both time and memory (64-bit doubles are boxed). All physical properties of sprites can be fractional (eg., a sprite can have a speed of 20 pixels per second, which can be 0.5 pixel per frame, so even the position needs to hold fractional values), and thus they are saved as fixed point values with 8 bits after the binary dot.

4.4 Blocks and TypeScript Editors

Figure 4(a) shows the Arcade web app with the Blocks editor active. On the left-hand side is the Blockly toolbox, which lists the main categories of blocks available to the programmer. As can be seen, the first six categories are the game engine APIs, while the remaining categories are programming constructs. The *Game* category has been selected, showing a list of the available blocks. The first two blocks are top-level event handlers for updating game state. The blocks can be dragged onto the programming canvas, which shows the "on game update" event handler and the user code inside it for determining if a sprite (variable "mySprite") is moving or has gone out of bounds.

---

[7] The physics engine can be turned off or replaced by a different one.
[8] Departing from JavaScript semantics to use single-precision would play havoc with bitwise operations, which all use 32-bit integers. While 64-bit floats can precisely represent 32-bit integers (in fact, up to 53 bits), the 32-bit float cannot.

(a)

(b)

Figure 4: (a) Blockly toolbox for the *Game* namespace and the game update handler; (b) corresponding code in TypeScript.

In Figure 4(b), we have toggled to the TypeScript editor (it's labelled JavaScript for marketing reasons), which shows the Type-Script code corresponding to the blocks in Figure 4(a). Also note that the toolbox is still available: in this case, the user can drag a code snippet (the TypeScript code corresponding to a Block) into the text editor.

The user is free to switch between the various editors. If the user uses a programming construct in the TypeScript editor (for example, a class) that is not editable in the Blocks editor, a greyed-out block representing that code appears in the Blocks editor. All code created in the Blocks editor can be modified in the TypeScript editor.

4.5 Asset Editors

As Figure 5 shows, in addition to the Blocks and TypeScript editors that are standard in the PXT framework, there also is an Assets option that we added to the framework, which gives access to the images, tile maps and animations, a gallery of images, and asset editors. The image editor can be accessed directly from the Blocks and TypeScript editors: for example, line 34 of the TypeScript code in Figure 4(b) references an image literal. To the left of the line number 34 is a paint icon that brings up the image editor directly from the source code editor.
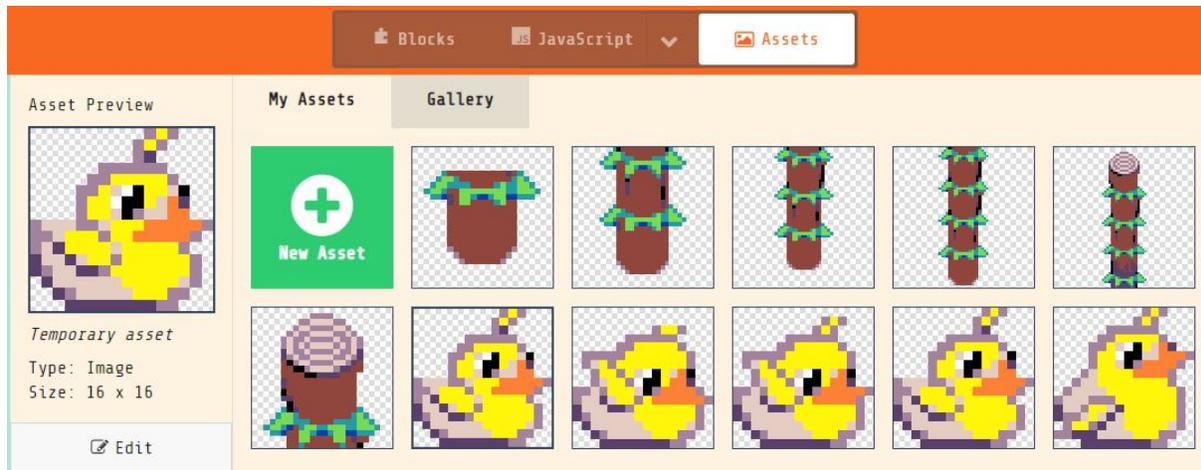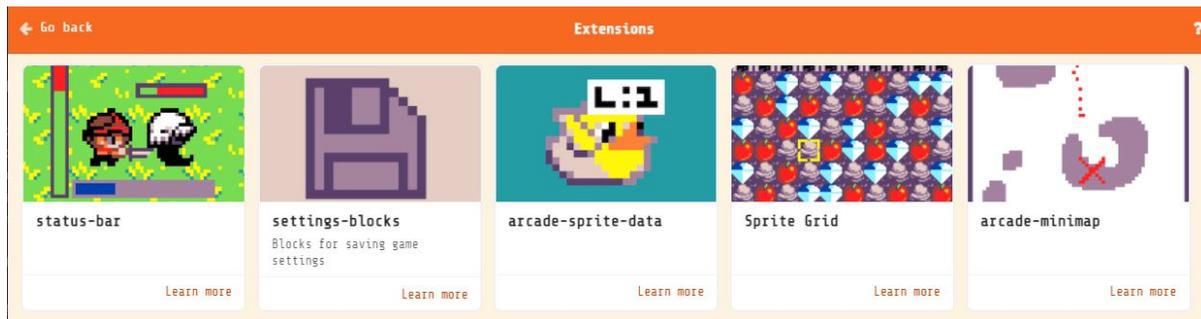
Figure 5: Assets view.

Figure 6: Some Arcade extensions.

4.6 Simulator

The top-left of Figure 2 shows the web-based simulator of the Arcade gaming handheld, which runs in a separate iframe in the web app. The user code and game engine runs in the iframe. The JavaScript of the simulator corresponds directly to our C++ runtime, providing functions so the game engine can write to the 160x120 screen (represented in the DOM) and capturing mouse and keyboard events that trigger d-pad and A/B button events in the game engine.

4.7 Extensions

The PXT framework supports the addition of approved extensions, which are authored in Static TypeScript and hosted in GitHub repos. As shown in the extension dialog of Figure 6, the user can add an extension to the web app, which results in a new category in the toolbox with new blocks. We originally developed sprite animations as a PXT extension, which was useful for experimenting with the APIs and blocks for animations without having to make new releases of the web app; once we were happy with the extension, it was incorporated into the web app.

5 EVALUATION

We started the development of Arcade in January of 2018, first creating our own gaming handheld prototype, writing the game engine, and getting the PXT-based web app to generate code for the prototype. We were then ready to demonstrate the complete system to third-party manufacturers and interest them in producing devices of their own design. The first handheld became available in January of 2019 and six more handhelds have been produced since then; all are still available as of the writing of this paper.

In this section, we evaluate the deployment of Arcade in 2019-2020 in four ways: (1) we describe the seven handhelds that third-party manufacturers produced; (2) we characterize the performance of the game engine in the web browser and on handhelds using a synthetic benchmark; (4) we analyze the performance of 38 games created by the Arcade community over three "game jam" contests run in 2020; (4) we measure attributes of 120 games created by the Arcade community and their use of extensions.

5.1 Gaming handhelds: Third-party manufacturers

Table 1 lists seven different handheld devices made specifically for Arcade. It also includes the micro:bit V2, which can be turned

Table 1: Gaming handhelds created for Arcade. All devices except for Raspberry Pi have 512 KB flash and 1.8 inch screen. In the case of the micro:bit, screen size and battery depends on shield, and the price of shield is estimated at $15+ (US). For Pi Zero (P0), the price depends on how the end-user assembles the system (other devices in the list came ready to play). The minimal bill of materials for a P0 is around three times higher than an F4. See Figure 2 for pictures.

Device	From	MCU	RAM KB	Freq MHz	Price $ (US)	Battery mAh	Expansion port	Case	Remarks
GHI BrainPad Arcade	US	F4	96	84	35	lipo port	micro:bit	no	optional WiFi
Kittenbot Meowbit	CN	F4	96	84	40	400	micro:bit	silicone	SD card port, gyro
TinkerGen GameGo	CN	F4	96	84	37	400	none	silicone + laser-cut	gyro
Adafruit PyBadge LC	US	D5	192	120	25	lipo port	Feather	no	1 RGB LED, no accel
Adafruit PyBadge	US	D5	192	120	35	lipo port	Feather	no	5 RGB LEDs, optional WiFi
Adafruit PyGamer (kit)	US	D5	192	120	50	350	Feather	laser-cut	analog joystick, good speaker
Kitronik Arcade	UK	D5	192	120	41	3xAA	custom	laser-cut	vibration motor, no accel
micro:bit V2 + shield	UK	N3	128	64	30+	in shield	blocked	varies	compass, radio
Raspberry Pi Zero	UK	P0	524288	1000	varies	NA	blocked	varies	ext. screen+controls, no accel

into a Arcade device using a "shield" accessory that adds a screen and buttons, and the RPi Zero for comparison (although it's not a handheld device).

Most handhelds come with an accelerometer and various other on-board components. They expose pins for physical computing scenarios using either a micro:bit-compatible edge connector or one based on Adafruit Feather boards. Some come as bare circuit boards, others have enclosures. They all have either a battery (around 5 hours of game time), or a least a battery port.

The seven handhelds are made by five companies from three countries. They represent a diversity of form-factor, cost, and optional components. We've also seen a number of hobbyists target Arcade, helped by our instructions on how to build a device from existing MCU development boards.

5.2 Core Performance

To get a sense of the number of moving and interacting sprites that each MCU family can process at a reasonable number of frames-per-second (FPS), we wrote a synthetic benchmark that creates N sprites, moving with random speed that bounce off the edges of the screen. On a high-end 2020 MacBook Pro, the benchmark ran in the web browser at 280 FPS for 100 sprites, 60 FPS for 1000 sprites, and 20 FPS for 2000 sprites.

The top graph of Figure 7 plots the FPS of the benchmark as the number of sprites (N) increases. In our experience no flickering was perceptible at frames rate above around 24 FPS, allowing all MCUs to support at least 30 sprites. For the F4 and N3 MCU families,[9] performance drops to around 20 FPS at 40 sprites, while the faster D5 family maintains over 20 FPS up until 70 sprites. At 100 sprites, all devices are basically unusable, while browsers even on low-end Chromebooks perform acceptably.

5.3 Arcade game jams

During 2020, we ran three online "game jam" contests in which Arcade users were challenged to submit games in a particular theme: *Garden* (June 10 - July 1)[10], *Traffic* (Sept. 14 - Oct. 5)[11], *Ocean* (Nov.

30 - Dec. 11)[12]. The main rules were to make a game related to the jam theme and to keep the games safe for children. The rules did not mention or require the use of a Arcade gaming handheld. As part of each contest, our team judged all the entries and selected the top three games and honorable mentions. Over the three game jams, we selected 38 games, which we analyze below.

5.3.1 Game jam game examples. Here we provide a description of three game jam games made with Arcade.

Potato. (Figure 8, left)[13] starts with a large potato growing in the center of a garden scene. Each press of button A encourages the potato to grow, and points are earned as it grows. Hitting the menu button allows you to spend your points on vegetables and plants to add to the garden. Plants and vegetables encourage bees to come to the garden, increasing points earned on each potato growth. Progress is saved between browser sessions and is preserved in flash memory on hardware.

Snail Hike. (Figure 8, middle)[14] is a highly polished game with carefully thought out assets and level design. The main premise of the game is to guide ten snails along a 2D platform towards an exit. Along the way, the player has to solve puzzles to enable snails to reach the exit. The player sprite has complex jump and glide mechanics with polished transition animations.

UFOs Control Traffic. (Figure 8, right)[15] presents a top-down view of a highway with cars travelling from top to bottom. Hovering above the highway is an alien spacecraft (UFO). Vehicles and highway are color coded, and the job of the UFO is to move vehicles to the corresponding highway before they reach the terminating point of the highway. Players have three lives, decremented upon an incorrect matching, and points are awarded upon each correct matching.

5.3.2 Game jam game performance. The Arcade game engine collects basic stats during a game's execution: the current number of allocated sprites and the current FPS. All the games ran without

---

[9] Interestingly, N3 and F4 achieve similar performance despite significant differences in clock speed. We suspect this due to a superior flash caching strategy in N3.
[10] https://arcade.makecode.com/gamejam/garden
[11] https://arcade.makecode.com/gamejam/traffic

[12] https://arcade.makecode.com/gamejam/ocean
[13] https://arcade.makecode.com/42885-92487-13042-52240
[14] https://arcade.makecode.com/27830-69912-67539-85378
[15] https://arcade.makecode.com/63418-25020-58432-17936

Figure 8: Screenshots from three Arcade Game Jam games. Potato (left), Snail Hike (middle), and UFOs Control Traffic (right)

- Three of the games failed to compile because they were too large and would not fit in the space available in the device's non-volatile flash memory.
- Two of the games ran out of memory and one of the games crashed.

In summary, most of the best games submitted by the community for an online game jam ran fine on the lowest-powered Arcade gaming handheld.

5.4 Static game measures and use of extensions

To extend the sample size of games, we also collected creations that have gathered the most "hearts" in the Arcade forum, where users share their games. This dataset has 120 games (out of over 630 games shared in the forum). Of these:

- 11% are written in TypeScript, while the rest are written in blocks.
- 54% use GitHub extensions (between 1 and 12) with a median of three extensions used.
- The median size of TypeScript code (either written directly, or generated from blocks) is 171 lines, the median number of blocks is 199; see Fig. 7, middle for distribution.
- 97% of games encode sprite assets either in source code or with the asset manager (which is a relatively recent addition to Arcade); the median size of these assets is 27,000 pixels.

The most popular GitHub extensions have the following functions: modify game palette and fade between palettes; attach life bars etc. to sprites; create sprites that contain just text; attach custom data to sprites; advanced functionality for tilemaps; store settings in non-volatile flash; start custom timers; attach actions to custom button combos on the controller; create a custom game menu.

6 CONCLUSION & FUTURE WORK

We have presented MakeCode Arcade, a web-based platform for creating video games. The programming environment contains customized blocks for game programming, a simulator for testing gameplay, and editors for creating sprites and sounds. Compiled games can be transferred to compatible devices via USB. Evaluation of community projects shows that low-cost hardware and a modern game programming experience are not mutually exclusive.

We plan to further combine Arcade and physical computing via Jacdac [8] [16], a new protocol for physical computing. Jacdac
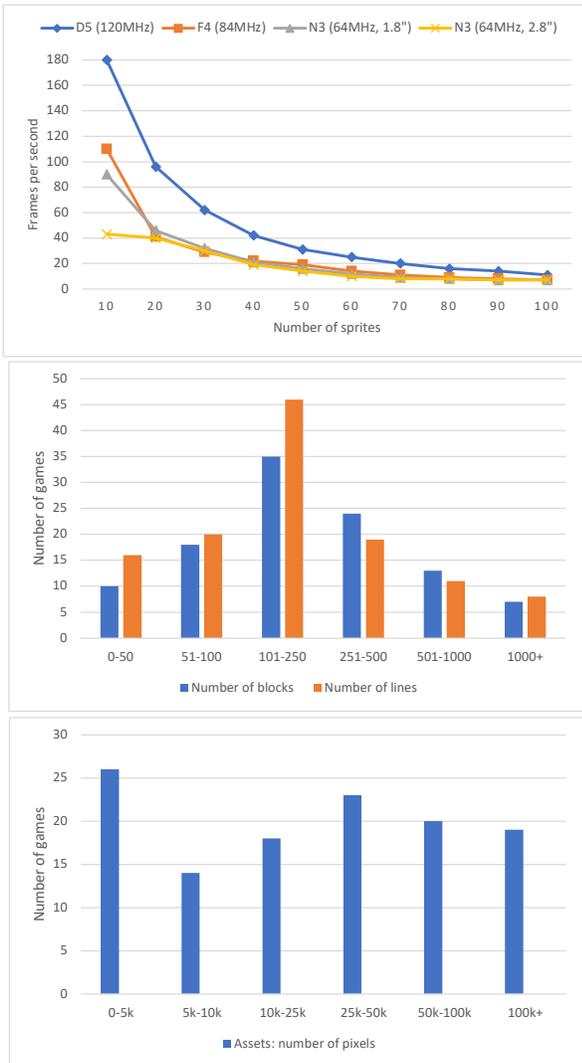

Figure 7: Top: result of running synthetic benchmark on the three MCU families supported by Arcade (D5, F4, N3). Middle and bottom: Static measures of 120 games from Arcade community forum and submitted to game jams.

error in the web browser, most of which had frame rates exceeding several hundred FPS. We tried to compile and run each of the games on the lowest-powered gaming handheld (an F4 with 96 KB of RAM), with the following results:

- Twenty (20) of the games achieved sustained FPS of over 24 (median of 35 FPS); in a few of these games, FPS dropped lower over time as the programs didn't clean up sprites; these games allocated from 3 to 19 sprites, with a median of 12 sprites.
- For another twelve games, six had an FPS of 20 or higher, two had an FPS of around 18, two had an FPS of around 12, and two had an FPS around 3. Not surprisingly, these games tended to allocate many more sprites.

---

provides an extensible plug-and-play experience without compromising microcontroller efficiency and price. Through Jacdac accessories, users will be able to combine retro games programming and the physical world in new and creative ways.

REFERENCES

[1] AADALIE. 2021. Gamebuino. https://gamebuino.com/.
[2] Jonny Austin, Howard Baker, Thomas Ball, James Devine, Joe Finney, Peli De Halleux, Steve Hodges, Michał Moskal, and Gareth Stockdale. 2020. The BBC micro:bit: from the UK to the world. *Commun. ACM* 63, 3 (2020), 62–69.
[3] John Aycock. 2016. *Retrogame Archeology - Exploring Old Computer Games.* Springer. https://doi.org/10.1007/978-3-319-30004-7
[4] Thomas Ball, Peli de Halleux, and Michal Moskal. 2019. Static TypeScript: an implementation of a static compiler for the TypeScript language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*, Antony L. Hosking and Irene Finocchi (Eds.). 105–116.
[5] Mark Claypool and Kajal Claypool. 2009. Perspectives, Frame Rates and Resolutions: It's All in the Game. In *Proceedings of the 4th International Conference on Foundations of Digital Games* (Orlando, Florida) *(FDG '09)*. Association for Computing Machinery, New York, NY, USA, 42–49. https://doi.org/10.1145/1536513.1536530
[6] Stephen Cooper, Wanda Dann, and Randy Pausch. 2003. Teaching objects-first in introductory computer science. In *34th SIGCSE Technical Symposium on Computer Science Education.* ACM, 191–195.
[7] Paul Craven. 2016. *Program Arcade Games: With Python and Pygame.* APress.
[8] James Devine. 2020. *Enabling intuitive and efficient physical computing.* Lancaster University (United Kingdom).
[9] James Devine, Joe Finney, Peli de Halleux, Michal Moskal, Thomas Ball, and Steve Hodges. 2018. MakeCode and CODAL: intuitive and efficient embedded systems programming for education. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2018, Philadelphia, PA, USA, June 19-20, 2018.* 19–30.
[10] Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond).* 49–50. https://doi.org/10.1109/BLOCKS.2015.7369000
[11] Steve Hodges, Sue Sentance, Joe Finney, and Thomas Ball. 2020. Physical computing: A key element of modern computer science education. *Computer* 53, 4 (2020), 20–30.
[12] Caitlin Kelleher, Randy F. Pausch, and Sara B. Kiesler. 2007. Storytelling Alice motivates middle school girls to learn computer programming. In *Conference on Human Factors in Computing Systems, CHI.* ACM, 1455–1464.
[13] Chris Maltby. 2021. GB Studio. https://www.gbstudio.dev/.
[14] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
[15] Filippo Rivato. 2021. TIC80. https://tic80.com/.
[16] Andreas Schrader, Bernhard Jung, and Darren Carlson. 2005. Tangible Interfaces for Pervasive Gaming. In *Digital Games Research Conference 2005, Changing Views: Worlds in Play, June 16-20, 2005, Vancouver, British Columbia, Canada.* http://www.digra.org/digital-library/publications/tangible-interfaces-for-pervasive-gaming/
[17] Sue Sentance, Jane Waite, Steve Hodges, Emily MacLeod, and Lucy Yeomans. 2017. "Creating Cool Stuff": Pupils' Experience of the BBC Micro:Bit. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, 531–536. https://doi.org/10.1145/3017680.3017749
[18] Charles R. Severance. 2014. Massimo Banzi: Building Arduino. *IEEE Computer* 47, 1 (2014), 11–12. https://doi.org/10.1109/MC.2014.19
[19] Teddy Seyed, Peli de Halleux, Michal Moskal, James Devine, Joe Finney, Steve Hodges, and Thomas Ball. 2019. MakerArcade: Using Gaming and Physical Computing for Playful Making, Learning, and Creativity. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, Regan L. Mandryk, Stephen A. Brewster, Mark Hancock, Geraldine Fitzpatrick, Anna L. Cox, Vassilis Kostakos, and Mark Perry (Eds.). ACM. https://doi.org/10.1145/3290607.3312809
[20] Steven Simmons, Betsy DiSalvo, and Mark Guzdial. 2012. Using Game Development to Reveal Programming Competency. In *Proceedings of the International Conference on the Foundations of Digital Games* (Raleigh, North Carolina) *(FDG '12)*. Association for Computing Machinery, New York, NY, USA, 89–96. https://doi.org/10.1145/2282338.2282359
[21] Quang N. Vu and Cor-Paul Bezemer. 2020. An Empirical Study of the Characteristics of Popular Game Jams and Their High-Ranking Submissions on Itch.Io. In *International Conference on the Foundations of Digital Games* (Bugibba, Malta) *(FDG '20)*. Association for Computing Machinery, New York, NY, USA, Article 20, 11 pages. https://doi.org/10.1145/3402942.3402981
[22] Linda Werner, Jill Denner, Michelle Bliesner, and Pat Rex. 2009. Can Middle-Schoolers Use Storytelling Alice to Make Games? Results of a Pilot Study. In *Proceedings of the 4th International Conference on Foundations of Digital Games* (Orlando, Florida) *(FDG '09)*. Association for Computing Machinery, New York, NY, USA, 207–214. https://doi.org/10.1145/1536513.1536552
[23] Joseph White. 2021. PICO-8 Fantasy Console. https://www.lexaloffle.com/pico-8.php.
[24] Quinn Burke Yasmin B. Kafai. 2016. *Connected Gaming: What Making Video Games Can Teach Us about Learning and Literacy.* The MIT Press.