Embedding Mobile Games in QR Codes

Finlay Laidlaw

BSc (Hons) Computer Games Technology, 4

School of Design and Informatics
Abertay University

Table of Contents

1 Table of Tables 3

2 Acknowledgements 4

3 Abstract 5

4 Introduction 6

4.1 Background 6

4.1.1 Game Distribution 6

4.1.2 QR Codes 6

4.1.3 Game Distribution Through QR Codes 7

4.2 Research Question 8

4.3 Aim 8

4.4 Objectives 8

4.5 Overview 8

5 Literature Review 10

5.1 Embedding Assets in Barcodes 10

5.2 Compression 11

5.2.1 Code Compression 12

5.2.2 Level Compression 13

5.3 Summary 15

6 Methodology 16

6.1 Application 16

6.1.1 Platform 16

6.1.2 QR Code 17

6.1.3 Scripting 18

6.1.4 Graphics 18

6.1.5 Levels 19

6.1.6 Entities 20
6.1.7 Physics 21
6.1.8 Audio 21
6.1.9 Documentation 22
6.2 Compression 22
6.2.1 Scripts 22
6.2.2 Levels 23
6.2.3 Serialisation 23
6.3 Evaluation Metrics 23
6.3.1 Features 24
6.3.2 Performance 24
6.3.3 Compression 25
7 Results 26
7.1 Features and Performance 26
7.2 Compression 28
8 Discussion 30
8.1 Features 30
8.1.1 Categories of Game Development Applications 30
8.2 Performance 32
8.3 Compression 32
8.4 Development Practices 34
8.5 Limitations 35
8.6 Practical Applications 35
9 Conclusion and Future Work 38
9.1 Implications 38
9.2 Future Work 39
10 List of References 41
11 Appendices 44

1 Table of Tables

Table 1 Features to be compared between applications 24
Table 2 Feature matrix 26
Table 3 Install size (megabytes, lower is better) 28
Table 4 The demos that were created in the game engine 28
Table 5 Compressed size of demos with binary serialisation 29
Table 6 Compressed size of demos in with JSON serialisation 29

2 Acknowledgements

I would like to thank my project supervisor Dr Andrei Boiko for providing guidance, feedback, and encouraging my problem solving and creativity throughout this research project.

3 Abstract

Since their invention, video games have been distributed using a variety of formats and technologies, new formats bringing new opportunities and trade-offs. QR codes are a technology used pervasively in the modern world - from mobile payment systems to restaurant menus - and they are supported by nearly every mobile device. Most often, they store URLs for web pages, but they also have the potential to store any kind of data directly, making them a physical data storage format that can be easily read by any mobile device. While the amount of data that can be stored on a QR code is limited, it is enough that they may be viable as a format for games.

This project aims to determine the extent to which QR codes can be used as a medium for game distribution. To accomplish this, an application is developed and evaluated that allows for the creation and playing of games stored in a QR code.

A cross-platform progressive web app (PWA) is created; it has a player and editor and the ability to export or import games from a QR code. A 2D game engine is developed with support for physics, sprite rendering, touch screen input, and sound effects. Game scripts and level data are compressed with PPMd. Game data is further reduced by using the Unifont bitmap font as a source of sprites, generating audio procedurally in code, and building tilemap levels out of reusable chunks.

The application was evaluated and found to support features on par with existing game engines, suitable for the development of a variety of games. The install size of the application was compared with alternatives and found to be lighter weight, ensuring that the application can be installed or loaded quickly. The implemented compression algorithm was tested against alternatives and found to be the most effective, however there is potential for improved compression ratios with further work. Overall, QR codes have the potential to be deployed as medium for distribution in a variety of commercial and non-commercial contexts.

4 Introduction

4.1 Background

4.1.1 Game Distribution

Formats for distributing media have evolved continuously through the information age. Video games specifically have used cassettes, cartridges, flash-memory, CDs, DVDs, Blu-rays and online distribution. Online distribution has become the primary area of growth and development; in the second quarter of 2020, digital sales on the PlayStation 4 exceeded physical sales by nearly a factor of three (Good, 2020). New avenues of digital distribution are also being explored, such as cloud gaming (Google Stadia, GeForce Now) and game subscriptions (Xbox Game Pass, Netflix).

4.1.2 QR Codes

QR codes are a type of 2D barcode, created in 1994 by Denso Wave for tracking parts in automotive manufacturing. In comparison to conventional barcodes, they have the advantage of high data capacity, being scannable from any angle, and including error correction such that even damaged or defaced QR codes can often be read without issue. Due to their high data capacity and ability to be scanned by a digital camera at any angle, they began to see adoption by mobile phones in the 2000s, and by the 2010s they had become a widely supported standard.

A 2015 study found that 86% of mobile QR code scans are for URLs (Lerner *et al.*, 2015) but there are numerous other use cases: boarding passes, e-tickets, Wi-Fi login, two-factor authentication, cryptocurrency, business cards, and more. During the COVID-19 pandemic, QR codes saw further proliferation for uses such as restaurant menus, mobile payment, and contact tracking (*BBC News*, 2021).

In general terms, QR codes allow up to three kilobytes of text data to be printed onto a surface and read by any device with a camera. Though they are most often used to store URLs which link to the internet, there is nothing preventing sufficiently small media from being stored directly.

4.1.3 Game Distribution Through QR Codes

As a distribution format, QR codes have unique characteristics. In comparison to other physical media, they cost less to produce, they are already widely supported by existing mobile devices, and like other physical media they do not require the internet. Furthermore, they can be scanned quickly through the camera, without needing to physically connect with the device like other formats.

QR codes could be used to distribute games when relying on the internet is undesirable, or a developer wants to create a connection between a game and a physical location. They could also be used in an educational setting, their physical nature encouraging real-world creativity, collaboration and communication.

The major limiting factor is their capacity; at only three kilobytes, it is insufficient for any modern game. Weighed down by high quality assets and complex software stacks, a typical modern console game has a size in gigabytes, and even the simplest mobile games are still a few megabytes. To adapt to this size limitation, a dedicated software platform would need to be created that supports the creation and playing of games within this constraint.

The concept of embedding games into barcodes has been explored previously, notably by Nintendo with the *e-Reader* peripheral for the *Game Boy Advance*. Released in 2001, it allows game-data to be loaded from a proprietary barcode format delivered on paper cards. Cards could store minigames, text, NES games (though these needed to be spread over multiple cards), and addon data for another game, such as new characters and music (Harris, 2002).

However, the e-reader was not very successful, and wasn't even released in Europe after failing to gain traction in America (Former, 2004). It has several disadvantages that the proposed system would not: it requires a dedicated addon in addition to the base device, cards can only be produced by licensed developers, one of the key features – adding content to existing

GBA games – can only be used if two consoles are connected by a link cable.

4.2 Research Question

To what extent can QR codes be used as a medium for offline mobile game distribution?

4.3 Aim

QR codes are a promising avenue for game distribution, but it is not clear to what extent their use is technically feasible. The aim of this project is to evaluate the technical feasibility of QR codes as a distribution medium for games by creating a cross-platform mobile application which enables the creation and playing of games that are embedded in QR codes.

4.4 Objectives

- Conduct a literature review on methods of compression applicable to game data.
- Create a cross platform mobile application with a 2D game engine and editor, it must support scripting, physics, audio, sprites, and the QR code format.
- Implement the chosen compression techniques into the application.
- Evaluate the features and performance of the application in comparison to similar game engines
- Evaluate the chosen compression techniques compared to alternatives.
- Analyse results and identify practical applications for QR code-based game distribution.

4.5 Overview

- Literature Review: reviews relevant research in compression and existing precedents in the applications of QR codes.
- Methodology: covers the architecture of the application, the chosen compression techniques and the methods of evaluation.
- Results: shows the results of the evaluation.

- Discussion: analyses the results with respect to to project aims, methodology, and broader research context, identifying limitations and practical applications.
- Conclusion: summarises the findings and identifies avenues for future research.

5 Literature Review

This chapter provides an overview of the existing research into the usage of QR codes for storing data and compression methods that may be applicable to game data. This will inform the design of the application and compression methods used to minimize game data size.

5.1 Embedding Assets in Barcodes

Barcodes are primarily used for identifying physical objects in digital systems, for example: scanning barcodes at the checkout identifies for the till's computer what item is being purchased.

For QR codes specifically, their higher data capacity and ubiquitous support on mobile phones means they can be used for broader applications than just carrying a digital identifier. In practice they are mostly used to convey web URLs; for example, a restaurant may display a QR code which links to a website for ordering. They provide a general solution for connecting physical locations to online locations.

Alternatively, the relatively high data capacity of QR codes also presents the possibility of storing media or assets directly in the QR codes, without requiring an external system. At roughly three kilobytes, the capacity of QR codes is high compared to one-dimensional barcodes, but still low compared to alternative physical media; for comparison, a typical NES cartridge has at least 32 kilobytes of ROM chips. Any media that is stored on QR codes must either be compact in nature or highly compressed.

QRscript (2022) explores the process of embedding a program into a QR code, with a focus on enabling "smart" objects without requiring an internet connection. To cope with the size limitations, they create a specialised bytecode which is as compact as possible. Unlike other bytecodes, it works at the bit-level, with three bits for each opcode and a variable number of bits for parameters. Text data is stored using only the seven bits per character needed for ASCII, rather than the typical full byte.

Though its bytecode is tightly designed, QRscript leaves potential space savings on the table because it doesn't make use of compression

algorithms. It is difficult to estimate how much the bytecode could be compressed, but English text has an entropy upper bound of roughly 1.75 bits per character (Brown *et al.*, 1992), which is four times smaller than the seven bits used for ASCII data in QRscript. Though this theoretical limit is difficult to reach in practice, the virtual machine could at minimum use Huffman coding with a predefined code table tuned for English text; this would reach roughly 4 bits per character.

Work has also been done on embedding other media in QR codes, for example 3D models. Ertugrul et al. (2020) lower the data footprint of the models by converting them to voxels and compressing them as voxel octrees. To further reduce the size, they train a neural network to upscale lower resolution voxel models. Their methods are intrinsically lossy, but they produce recognisable models with a low data footprint.

Compression can only be taken so far; there are fundamental limits to lossless compression, and lossy compression algorithms must guess at what features of the data should be preserved or discarded. An aesthetically superior result could be achieved if, instead of trying to compress existing models, they created their own models in their own format from the outset. It depends on if their goal is explicitly to compress existing models, or just to create any 3D representation of an object that can fit on a QR code.

QRscript and the work by Ertugrul et al. establish a precedent for embedding different types of media in QR codes. QRscript deals with the size constraint by designing a minimal bytecode, forgoing typical data compression algorithms, whereas the research by Ertugrul et al. focuses on using lossy data compression techniques. Both sources reinforce the key motivations for embedding assets in QR codes: access without relying on the internet and the ability to create physical-to-digital connections.

5.2 Compression

Compression is the act of processing data to be more compact. The level of compression is the compression ratio, calculated as the decompressed size divided by the compressed size. Compression is lossless when after

compression and decompression the data is the same, alternatively, lossy compression causes the decompressed data to be degraded, trading accuracy for higher a compression ratio. There are many different compression algorithms for different data types, and all compression algorithms have trade-offs between the compression ratio, encoding time, and decoding time.

This project requires lossless compression with as high a compression ratio as possible. Game data includes script and level data; these are symbolic data types and must be compressed with no degradation because even a small change to a script or level could change its behaviour. Compression speed is not a concern because the size of the data for this project is much lower than what compression algorithms typically deal with. Even if the capacity of a QR code is filled with a very high compression ratio of 10-to-1, that is still only 30 kilobytes. Looking to empirical data, the Squash Compression Benchmark (Nemerson, 2018) shows that it is rare for a high compression ratio algorithm to take more than a second to compress that much data, which is acceptable.

5.2.1 Code Compression

Hindle et al. (2016) shows that the semantic structures of source code are susceptible to the same processing methods as natural language. They exploit this to improve code completion features in the Eclipse IDE, but it also means that methods that work well for natural language compression are likely to also work for source code compression.

There is a significant amount of research into natural language text compression, the Large Text Compression Benchmark (Mahoney, 2026) compares over 200 lossless compression programs by compressing Wikipedia text and finds that the best compression ratios come from context-mixing and neural network based models.

However, the highest performing programs on the benchmark are optimised for the conditions of the benchmark, limiting their applicability elsewhere. They have very high memory consumption - not suitable for a mobile application - and they are also optimised for text data on the scale

of the enwik8 and enwik9 datasets used in the benchmark, which means millions of characters. This is a particular concern for the neural network model, which may not have enough data in smaller file to learn adequately.

Gupta et al. (2017) benchmarks five different modern compression programs, they find that the prediction by partial matching (PPM) based algorithms have the best compression ratios on English natural language data. PPMd and PPMonstr (Shkarin, 2010) implement prediction by partial matching with information inheritance (PPMII) in conjunction with a range coder. As the byte sequence is processed, the PPM algorithm builds a statistical context model of which byte is likely based on previous bytes. The context model is used to calculate a probability for each possible byte; these probabilities can then be used by a range coder to encode the byte. The higher the probability of the byte that is encoded, the lower the number of bits the range coder must use to encode it.

While much research has been done on compressing English text, there is less work on effectively compressing non-English text. A particular issue is texts which have a larger character set and make use of multi-byte encodings such as UTF-8. However, by assuming that characters of the same Unicode block are likely to be used together, it is possible to make a full Unicode based text compressor work efficiently (Gleave and Steinruecken, 2017). The researchers show that their method has superior compression ratios on UTF-8 text that uses non-ASCII characters heavily, however on ASCII texts it is out-matched by PPMII. It may be possible to integrate their method into PPMd, given that PPMd is open source, but how difficult this would be is difficult to estimate without a detailed understanding of PPMd's construction.

5.2.2 Level Compression

Many 2D games use tilemaps to construct their levels; a tilemap is a square grid where each cell contains a tile, drawn from a predefined list (called a tile set or tile atlas). Tilemaps are space efficient because they reuse assets heavily.

Because tilemaps are relatively niche and already quite space efficient, there has been no direct academic research into compressing them further. However, they can be viewed as having some characteristics of both image and text compression: like images, they have a two-dimensional grid structure, and like text, they use discrete symbols (tiles from a tile atlas).

A format which appears similar to tilemaps in its data characteristics is the palette image. Like tilemaps, they are both discrete and two-dimensional. Forchhammer and Salinas (2002) note that the commonly used GIF and JPEG image formats do not compress palette images efficiently because they do not use a model suited to the data. They adapt the PPM (prediction by partial matching) compression algorithm by changing it from using a one-dimensional sequential context to using a two-dimensional context, including pixels from the line above.

Images have many structures and patterns which can be exploited by compression algorithms and will likely be shared by tile-maps. PNGs employ the Paeth predictor (*PNG Specification: Filter Algorithms*, 1996) which attempts to continue horizontal or vertical lines when it finds them. It is not the most comprehensive algorithm but could be used as a pre-processing step to augment an otherwise one-dimensional compression algorithm.

In practice, video games which use tile-maps and have limited storage space, such as games released for the Nintendo Entertainment System, compose their levels out of higher-level building blocks (NovaSquirrel, 2023). In *Super Mario Bros.* levels are stored as a list of objects, each with a position and type. These objects can be individual coins or enemies, or they can be larger elements such as staircases or flagpoles. Alternatively, in *The Legend of Zelda* each room selects from a predefined set of vertical columns, so instead of storing a full grid of tiles for each room, it only needs to store a list of column indices.

These approaches demonstrate that compression can be achieved not only after the level has been designed, but by applying constraints during the level design process.

5.3 Summary

This chapter has covered existing projects that establish a precedent for embedding media into QR codes and high-level approaches for dealing with the size constraint. PPMd was identified as an ideal algorithm for script compression. Level compression can make use of higher-level building blocks (to reuse level data) and apply palette image compression techniques.

6 Methodology

This chapter covers the implementation of the software application and the metrics used to evaluate it. It includes details on the game engine implementation, the editor, and data compression techniques. An attempt is made to provide a rational for all significant design decisions.

6.1 Application

6.1.1 Platform

The aim is to create an application that supports both major mobile platforms. The typical method of creating a cross-platform mobile application is to use a framework that can output an app for both platforms from one source. However, this has disadvantages that make an alternative approach preferred.

Native apps built with a cross-platform framework are installed through the app store and have access to all device features. However, using a framework introduces a sizeable overhead to their install size and because they are on the app store, they must go through an external review process to become publicly accessible.

Apple app store guidelines (Apple Inc, 2026) place restrictions on both dynamically executed code (section 2.5.2) and user generated content (section 1.2) which may apply to this project. It would likely still be possible to publish, but the review process may be slowed, especially because the nature of this project is unconventional.

Alternatively, progressive web apps or PWAs (*Progressive web apps | MDN*, 2025) are websites which can be saved to a device to be accessed from the home-screen and ran offline. Because iOS and Android browsers support common web standards, the same PWA can be used on both platforms. PWAs have small install sizes because they are powered by the web browser already on the device. PWAs can also be distributed freely without needing to apply to app stores or satisfy their policies, which simplifies the application’s deployment.

This application does not require access to all device features and being accessible as a website allows for a quicker first-time user experience, even if the goal is that the user will ideally download the website as a PWA so they can use it offline. It would also be ideal to avoid a risk of deployment being blocked or slowed in the review process. For these reasons, a PWA is the right choice.

6.1.2 QR Code

The game data is formatted into a URL so that an external QR code scanner can be used and automatically redirect to the application. Installed apps and PWAs can intercept links, so the user will be directed to the app if it installed or the website if it is not.

QR codes support multiple “modes” of encoding data: numeric, alphanumeric, binary, and kanji. Each of these modes supports a specific set of characters, URLs are encoded using binary mode because numeric and alphanumeric lack the necessary characters. Software engineer Huon Wilson demonstrates on his blog (2024) that by encoding a URL with multiple data segments, the required size of the QR code can be reduced. However, he finds that despite this method being theoretically supported, the QR code scanner on iOS reads an incorrect URL.

To ensure proper support by scanners, the most straightforward encoding method is used for this project. The data is encoded as a Base64 string inside the query parameter of the URL. Binary mode QR codes have a maximum capacity of 2953 bytes; after subtracting 30 bytes for the URLs header and multiplying by 75% (the efficiency of Base64), the maximum game data size is 2192 bytes.

It is not necessary to include a QR code scanner in the app; the ubiquity of QR codes in the modern world means that all mobile phones with a camera can be expected to already have some way of scanning QR codes, either with the built-in camera app or a dedicated QR code scanner app. Not including a QR code scanner also reduces complexity and means that the app does not need to request camera access.

6.1.3 Scripting

To program game logic, a scripting language is required. The scripting language must fulfil the following requirements:

- It must be executable from source code in the browser.
- The required libraries and runtime must not be too large, so they do not increase the load-time of the app significantly.
- It must be compressible or have a small data footprint.

While many different programming languages can be compiled for the web ahead-of-time, there are far fewer that fully support compiling user-created scripts at runtime. WebAssembly does allow for desktop applications to be ran in the browser, but the compilers and interpreters for most languages are too cumbersome for a project that is aiming for a short loading time.

After these considerations, the remaining options are running JavaScript in a sandbox or, using the Wasmoon (Francisco, 2026) library to run Lua. The Lua interpreter is lightweight enough that it is not a major concern for page load time, and JavaScript is already supported by all web browsers.

Both Lua and JavaScript have a history of being used as scripting languages in game engines. However, Lua has the advantage of being carefully designed and easier to pick up, whereas JavaScript has accumulated many caveats and additions due to its use as a web core technology. So, for the sake of ease-of-use, Lua has been chosen as the scripting language.

CodeMirror (Marijn Haverbeke, 2025) is used to provide a script editor with syntax highlighting for Lua. Though CodeMirror does not support more sophisticated options such as auto complete, it is the best immediate option and more advanced code editing features are not a priority.

6.1.4 Graphics

Image data requires a large amount of space in comparison to text data. Reducing the amount of image data included in a game would increase the scope of games possible by allowing for more level and script data.

To achieve this, a predefined sprite sheet is included in the engine, such that games only need to reference the sprites that they want to use. To create this sprite sheet, a subset of Unicode characters is extracted from an open-source bitmap font (Hardy, 2025). A predefined set of sprites might be expected to limit the kinds of games possible, but Unicode provides a wide and unopinionated set of iconography including letters, faces, animals, food, shapes, and patterns that will ensure games are not lacking in options.

The sprite sheet is generated by a Python script which iterates all the characters in Unifont. It selects logographic characters and Emoji, along with basic Latin characters, with a filter to remove offensive symbols. The characters are arranged into a square sprite sheet with a power-of-two width (which is optimal for textures). A PNG optimiser (Louvrier, 2026) is used to minimise the sprite sheets size.

To allow for colour while keeping the required data low, a limited colour palette is used. Each sprite or tile is painted with one of eight colours, with the background being transparent. There is also a flag to swap the background and foreground of each sprite or tile, allowing for more flexibility in how characters are used without requiring a separate background colour to be stored. Eight colours and one flag require only 4 bits in total.

Initially, a rendering engine was created using the Canvas API, but poor performance meant it was necessary to switch to a WebGL based batch renderer. Each frame, all the sprites and tiles are added to a buffer, each with a position, character ID, foreground colour, and background colour. At the end of the frame, the sprites can all be drawn from the buffer in one batch. This could be optimised further by only adding tiles and sprites that are in the cameras view to the buffer, but such optimisation was ultimately not needed.

6.1.5 Levels

The levels use a tilemap, an N by M grid of square tiles where each tile is a Unicode character, coloured with the same method as the sprites. On top

of this, there is also a higher level of structure. Much like how *The Legend of Zelda* reuses columns, levels are built using tile “patches” of user-defined size. The patches are arranged into a “layout”, like how tiles are arranged into a tilemap.

Because the game engine has a physics engine, there must be a way to add physics collision to the tilemap. Which tiles have collision is determined by a text field, where the user enters all the Unicode characters that they wish to act as solid. This method is ideal because it requires minimal additional data to be included the game, only a short string.

In addition to the tilemap, games also often need to define the locations of sprites, such as enemies or collectables. To allow for this, there is a “marker” system. On top of the level layout the user can add markers to any tile position. Each marker has a Unicode character which acts as an ID, these markers can then be iterated in the script to spawn all the enemies at the start of the game. For example, the user could add markers to the level with an ID of “c” and then iterate all the markers with ID “c” to spawn coins at their locations.

6.1.6 Entities

Entities make up the dynamic and interactive components elements of a game, separate from the tilemap. Each entity has three components: sprite, physics, and input. Components can be enabled or disabled and have fields for controlling each aspect of its functionality.

Initially, a component structure was not used, but as development progressed it became necessary. The complexity of entities grew as functionality was added such that it was too much to manage in one class. The components also have common functionality for positioning and setting their size which can be implemented in a base class and inherited by each component, taking advantage of object oriented design.

The scripting API provides functions for creating, removing, and copying entities. The copying function is useful because it allows the setup process (configuring its components by setting their fields) to be skipped if a similar entity already exists, reducing the size of the script.

6.1.7 Physics

Matter.js (2026) is integrated into the game engine to support rigid body physics. While a more limited, custom programmed physics engine would have had a smaller package size, using a fully featured 2D physics engine like Matter.js allows for more flexibility in what games can be made. Matter.js was chosen against alternative engines because it has been used in popular Phaser game engine.

The physics component exposes fields which control the parameters of the rigid body, including friction, restitution, whether the body is static or simulated, and whether the body just detects overlaps or collides with other bodies.

To react to collisions or overlaps in scripts (e.g. the player touches an enemy and should take damage) there are two functions on the physics component, “onOverlapBegin” and “onOverlapEnd”. These functions are called to allow the developer to execute code whenever another entity starts or stops overlapping with a physics body.

6.1.8 Audio

Modern games typically use high quality audio formats for their music, but large file sizes mean that utilising any form of prerecorded audio is infeasible for this project. Fortunately, the Web Audio API has functionality for generating audio waves and applying effects and filters to them, which can be used to produce sound effects procedurally on the web platform.

To extend the audio feature-set further, a JavaScript port of Software Automatic Mouth (Schiffler, 2026) is included, enabling text-to-speech functionality. SAM was first released for home computers in 1982, and while it has been exceeded by other software in terms of realism, it’s small program size and retro character make it ideal for this project.

Support for the Music Macro Language is also included. MML has been used in various software programs to allow for the generation of simple melodies. It takes the form of sequence of commands packed into a human-readable string. The commands do things such as set the tempo, play a note, increment or decrement the octave, play some notes

simultaneously, or loop through a sequence of notes. An existing JavaScript library (Starzak, 2024) is used to parse the MML syntax.

6.1.9 Documentation

The application includes a built-in documentation tab which explains all the included engine functions and how to use the editor.

6.2 Compression

6.2.1 Scripts

The Lua scripts will be included in the game data as source code. It is possible to compile Lua to bytecode and – in theory – this could reduce the size of the script data by removing human readable information (such as comments and variable names). QRScript took advantage of this when by creating a custom bytecode which stripped unneeded information. However, when tested, Lua's bytecode was not more compact or more compressible than the corresponding source code.

It may have been possible to improve the compression of Lua's bytecode, but the lack of existing research in the area makes it a more difficult problem to approach than text compression. There is also the additional benefit that including source code unaltered allows for it to be read, understood, and modified by anyone loading the game.

The Lua source code is compressed using PPMd. Popular compression libraries such as gzip (Jean-Loup Gailly and Mark Adler, 2025) and Brotli (Jyrki Alakuijala *et al.*, 2025) focus on performing well on a broad range of data types and being able to compress large files quickly. In comparison, PPMd is specifically designed for text compression and prioritises high compression ratios at the cost of longer execution times. However, because the amount of data being compressed for each game is very small, PPMd's longer execution time does not make a noticeable difference.

PPMd is an implementation of the prediction by partial matching data compression algorithm written in C++. To be deployed on the web, the application is compiled to WebAssembly with Emscripten (Zakai, 2026).

Another advantage of using PPMd over alternatives is that it is very lightweight, only 82 kilobytes, whereas Brotli is more than a megabyte.

6.2.2 Levels

The level data can be broken into 4 parts: the tile codepoints (Unicode characters), the tile colours, the patch IDs in the layout, the patch transforms in the layout (though included in the format, transform data ultimately went unused). Each of these parts is compressed with PPMd independently because they are each different types of data with different distributions; the model that PPMd builds for one datatype will not be useful when compressing a different datatype.

6.2.3 Serialisation

The game data is serialised into a binary format; this format includes the compressed script and level data as well as other metadata needed for the game. JavaScript natively supports the serialisation of any object to JSON, but it is not particularly compact. A JavaScript class is created which supports packing and unpacking the data with functions for converting each data type into bytes and adding it to a JavaScript DataView.

To allow for both flexibility and space efficiency in the format, integers are stored as variable length quantities. To encode an unsigned integer, it is divided into 7-bit sections, and for each section a byte is written, with 7 bits containing the data and 1 bit indicating whether there are more sections for this integer. With this method, an unsigned integer less than 128 (2^7) is encoded as 1 byte, an unsigned integer less than 16384 (2^{14}) is encoded as two bytes, and so on.

The exact operations used to serialise and deserialise the game data can be seen in Appendix A and B.

6.3 Evaluation Metrics

The software artefact will be evaluated on three factors relevant to its use as a platform for games. A flexible game development application should have a selection of features on par with other game engines and platforms,

it should have good performance, and for QR code games it should maximise the compression ratio.

6.3.1 Features

To make a variety of games, the engine and editor must be equipped with a comparable set of features to other 2D game development applications.

Table 1 Features to be compared between applications

Category	**Features**
Player	Pause, Share, Fullscreen, Mobile Support
Editor	Tilemap, Script, Documentation, Entity Placement
Graphics	Import Images, Sprite Editor, Sprites, Animation, Camera, Particles, Layers, Shaders
Audio	Import Sound Effects, Generate Audio in Code, Sound Effect Editor, Music Editor, Text-to-speech
Physics	Collision, Rigid Body, Overlap, Ray Cast
Input	Touch, Keyboard & Mouse, Controller, Accelerometer, Microphone

Applications for comparison will be selected with the following criteria: they must support a player and editor on the web platform, they must be popular, and they must be distinct in some way from other compared applications (to enable insightful comparison).

6.3.2 Performance

There are many possible metrics that can be used to evaluate the performance of a game engine, website, or application. For game engines and games, performance typically equates to frame time (the amount of time required to calculate one frame). However, this project is focusing on 2D games, for which low frame times are well within the capabilities of modern hardware without any exceptional optimisation.

A more relevant performance metric is the install size of the app. If a QR code with an embedded game is encountered by a new user, the process of downloading the app or loading the website should be as quick as possible. Google research (An, 2018) has found that as page load time increases from 1 second to 5 seconds, the probability of the user abandoning the website roughly doubles. This demonstrates the importance of short load times as a prerequisite for a good user experience.

The install size of the application is compared to other similar applications. Because website architecture allows for different parts of the application to be loaded separately, the install size of the editor and player will be measured separately.

6.3.3 Compression

The data compression solution is compared to off-the-shelf compression libraries in terms of compressed size in bytes on a set of test games. Each test game is compressed with each compression method to create a table. Compression ratio is the focus because it is the most important metric given the tight space constraints of QR codes.

The compression libraries are selected from the npm Registry to be representative of the most popular compression algorithms available on the web platform. For libraries which have configuration options, they are configured to maximise compression ratio at the cost of speed.

The process is repeated with JSON serialisation instead of binary serialisation to show what difference it makes across the different compression algorithms.

7 Results

This chapter will provide results for the three different metrics of evaluation.

7.1 Features and Performance

The following applications were selected based on the criteria for comparison:

- **Scratch** (Scratch Foundation, 2019) - A free web-based game platform created by MIT Media Lab in 2007. It uses a visual programming language to encourage creativity and teach programming.
- **Godot** (Juan Linietsky and Ariel Manzur, 2026) - A 2D and 3D open-source game engine first released in 2014. Although web is not the primary platform, there is a web-based version of the editor and support for making HTML5 games.
- **PICO-8** (Lexaloffle Games, 2024) - A game engine which emulates the characteristics of a retro game console. It supports programming in Lua and runs on web, mobile, and desktop.

The install size/page weight data is collected using the Firefox network debugger.

Table 2 Feature matrix

Category	Feature	QR	Scratch	PICO-8	Godot
Player	Pause	✓		✓	
	Share	✓	✓		
	Fullscreen		✓		✓
	Mobile Support	✓	✓	✓	✓
Editor	Tilemap	✓		✓	✓
	Script	✓	✓	✓	✓
	Documentation	✓			✓
	Entity Placement		✓		✓
	Mobile Support	✓			

Category	Feature	QR	Scratch	PICO-8	Godot
Graphics	Import Images		✓		✓
	Sprite Editor		✓	✓	
	Sprites	✓	✓	✓	✓
	Animation		✓		✓
	Camera	✓	✓	✓	✓
	Particles				✓
	Layers		✓	✓	✓
	Shaders				✓
	Immediate Mode			✓	
Audio	Import Sound Effects		✓		✓
	Generate Audio in Code	✓			
	Sound Effect Editor		✓	✓	
	Music Editor			✓	
	Text-to-speech	✓	✓		✓
Physics	Collision	✓			✓
	Rigid Body	✓			✓
	Overlap		✓		✓
	Ray Cast				✓
Input	Touch	✓	✓	✓	✓
	Keyboard & Mouse	✓	✓	✓	✓
	Controller		✓	✓	✓
	Accelerometer				✓
	Microphone		✓		✓

Table 3 Install size (megabytes, lower is better)

Platform	Install Size (MB)	
	Player	**Editor**
QR	0.701	1.12
Scratch	23.06	23.91
Godot	34.42	92.29
PICO-8	1.84	2.14

7.2 Compression

The following three compression algorithms were selected to compare against:

- **Deflate** (Barett, 2024) - A combination of Huffman coding and LZ77 (a dictionary compression scheme).
- **LZMA** (Rugg, 2016) - Used in the 7-Zip archiver, it combines a bit-based range coder and a dictionary compression scheme like LZ77.
- **Brotli** (Tim Perry, 2024) - Much like Deflate, uses Huffman coding and LZ77, however it implements more optimisations to get the most out of them.

Table 4 The demos that were created in the game engine

Name	Description
Platformer	A platformer game that demonstrates all engine features.
Air Hockey	A 2-player air hockey game, showing how bouncy physics can be used.
Sound Effects	A soundboard showcasing the sound effect composition system.
Daisy Bell	A simple demo showing text-to-speech and sprite manipulation.
Infinity	A simple demo showing camera movement and the tile patch system.
Empty Game	An empty game.

Table 5 Compressed size of demos with binary serialisation

(bytes, lower is better)

Demo	Initial Size	Compression Algorithm			
		Deflate	LZMA	Brotli	PPMII
Platformer	4694	1758	1853	1580	1494
Air Hockey	1436	682	810	620	573
Sound Effects	3229	783	859	659	614
Daisy Bell	1201	593	722	528	519
Infinity	422	274	395	247	226
Empty Game	425	109	231	77	81
Total	11407	4199	4870	3711	3507

Table 6 Compressed size of demos in with JSON serialisation

(bytes, lower is better)

Demo	Initial Size	Compression Algorithm			
		Deflate	LZMA	Brotli	PPMII
Platformer	6445	1939	1873	1672	1647
Air Hockey	1778	741	767	671	647
Sound Effects	4114	864	828	723	693
Daisy Bell	2037	657	683	579	587
Infinity	1026	361	379	313	327
Empty Game	1226	203	216	178	163
Total	16626	4765	4746	4136	4064

8 Discussion

This project endeavoured to determine the extent to which QR codes can be used for mobile game distribution. To achieve this, an application was made to test the technical feasibility and evaluate it quantitatively. The application is evaluated in terms of features, performance, and the compression ratio achieved on game data.

8.1 Features

This section will review the features implemented in comparison to other similar applications, analysing whether features were justifiably included or excluded, and what changes might be made in a revised version of the project.

Scaling back the feature set would make the application much easier to develop, but it would also reduce its flexibility and viability as a game making tool. Therefore, the feature set of the application should be compared to existing popular engines to determine if it is fit-for-purpose as game making tool.

8.1.1 Categories of Game Development Applications

The specific aims of the application should be given consideration when comparing it to other similar programs. More features are not strictly better if they don't make sense for the goals of the project.

The three applications chosen for comparison all support 2D game development on the web. However, they each also fall into slightly more specific categories:

- **Godot** is a production ready game engine. It is built to with the expectation of being used by professionals in concert with other professional tools. It doesn't attempt to provide asset creation functionality for sound effects, sprites, or music. It also doesn't provide any immediate publishing or sharing solution for games, leaving it to the developer.
- **PICO-8** is a fantasy console. It is an attempt to create an idealised developer experience, emulating the aesthetics and limitations of

retro video game consoles for fun. It includes music, sound effects and sprite editors to provide a more self-contained development environment.

- **Scratch** is a platform for creating and sharing games, primarily developed as an educational tool to teach children creativity and programming.

Comparing to these categories, this project is more in-line with fantasy consoles or platforms. If compared with Godot, it lacks the ability to import assets or export the game for publishing externally, because both features don't make sense in the context of using QR codes as a medium. Like PICO-8 and Scratch, it is a self-contained development tool.

While all the applications support the mobile platform, only this project supports a full editor experience on a phone screen. It can be speculated that this is because phones are not ergonomic for longer sessions of game development work and resources it would take to develop a portrait mobile UI are not worth it for the application developers. However, in the context of this project it made sense to support full development on phone screens. It places less constraints on how the application can be used, therefore supporting the use of QR code games to a greater extent.

A physics engine is often a core feature of a game engine, but it can add significant complexity. Real physics is inherently complicated, and the bugs and quirks of the implementation will only add to that complexity. This is likely the reason why Scratch and PICO-8 forgo a built-in physics engine. 2D games rarely need fully simulated rigid bodies, they can get away with much simpler handwritten collision and response logic, which also has the benefit of giving the developer greater control. This project includes a physics engine because while it may be more conceptually in line with Scratch and PICO-8, that handwritten logic would extend the script size, which needs to be minimised.

Although the project supports basic features, it could do more to take advantage of the platform it is on. Phones have accelerometers and microphones, which Scratch supports as inputs. This project could have

also supported these unique input methods, which would have encouraged creativity and opened new possibilities for games. Overall, though there are some gaps, the applications feature set is on par with existing engines.

8.2 Performance

Performance is evaluated to establish that the application has a good foundation. Performance is a key way by which the technical implementation of an application directly affects the user experience. The performance metric chosen for this project is the install size (or page weight). A large page weight would be detrimental, dramatically slowing down the first-time user experience.

Install size was measured for the editor and player of each of the applications. The results (Table 3) show that this project has the smallest install size, less than half of PICO-8 which is the next smallest. The Ofcom Mobile Matters Report (2025) surveyed mobile network speeds and found that 4G still dominates with 70% of connections, 28% of those 4G connections are at 10 Mbit/s or lower. At these speeds, Godot would take 27 seconds to load the player and 73 seconds to load the editor. In comparison, this project would take 0.6 seconds and 0.9 seconds respectively. Furthermore, 5% of 4G connections fall below 2 Mbit/s, which would quintuple these load times. By working well with poor connections, a QR code game distribution system provides a more reliable user experience and gains the flexibility to be deployed outside and specifically in rural areas.

8.3 Compression

Before discussing the compression results, it should be noted that much of the reduced size of game data stems from engine design decisions rather than compression algorithms. The game engine uses Unicode characters for graphics avoiding including graphics assets, audio is generated with code to avoid including audio assets, and the levels use a layout system to reuse patches of tiles.

With that in mind, the compression results on the demos (Table 5) show that the algorithm that has the best compression ratio out of those tested is PPMII. Deflate, LZMA, and Brotli are all dictionary-based compression methods, whereas PPMII is a statistical context modelling-based method.

A clear difference between binary serialisation (Table 5) and JSON serialisation (Table 6) can also be seen. The total size of the demos compressed with PPMII is 4064 bytes for JSON serialisation and 3507 for binary serialisation, a 13.7% decrease in size. Partly, this is because the JSON serialisation contains additional information in the form of field names; it is also because different compression algorithms will perform better or worse depending on how data is formatted.

In this case, it can be speculated that PPMII performs better with the binary serialisation because it is able to compress the different data types with different context models. The binary serialisation method executes the compression algorithm on each data type (level codepoints, level colours, script, description, etc.) independently, whereas the JSON serialisation converts all the data to a string which is then compressed together.

While Brotli is competitive with PPMII in terms of compression ratio, Brotli's package size is more than one megabyte, whereas PPMd is only eighty kilobytes. Brotli has an inflated package size because it has a very complex design, it has been designed to optimise for both execution speed and compression ratio.

In relation to QRscript, this project presents an alternative approach to embedding executable programs in QR codes. QRscript focused on designing a highly constrained and highly compact bytecode, whereas this project applies text compression to an existing scripting language. The QRscript bytecode only supports decision trees, stating that other applications will need to create a new sub-dialect for whatever features they need. This seems like a significant development cost, compressed scripts may not be as compact as QRscript bytecode, but if they are compact enough, using them would allow QRscript to support more applications more easily. The "Air Hockey" demo has 50 lines of Lua script

(see Appendix C) which compresses to 573 bytes, at this ratio a QR code could contain 257 lines of script at maximum (191 lines if it is encoded as a URL). This seems like it would be sufficient for many use cases.

While the application successfully demonstrates how QR codes can be used to distribute games, the implementation of compression techniques is somewhat lacking. PPMd was ported to the web platform and performs well in comparison, but due to difficulty working with the source code, it is not modified or improved beyond that.

PPMd compression ratio on the scripts perhaps could have been improved by “training” the context model before compressing. Lua scripts could be collected from open-source projects and compiled to create a small dataset which would be included in the application and therefore known by the compressor and decompressor. PPMd would be modified so that it runs through the primer before compression or decompression, improving the context model and therefore the compression ratio.

Tile data is compressed with PPMd, it could have benefited from a more specialised compression algorithm. The literature review covered how the PPM algorithm can be adapted for palette images with a 2D context, the same method could be applied tiles in a tilemap. Furthermore, PPMd could be modified to work at the Unicode codepoint level rather than the byte level.

8.4 Development Practices

During development, it was found that the many components of the game engine became difficult to manage, and the development process was slowed by needing to repeatedly run the application to find minor typos or logic errors. To ameliorate this, the decision was made to convert the JavaScript project to TypeScript. TypeScript is a popular extension of JavaScript that provides support for static typing (JavaScript is normally dynamically typed). Once type information is specified, the TypeScript compiler can analyse the code and indicate errors and potential issues before the program is ever run, saving developer time. Type information also allows for better auto-complete functionality because the IDE can also

use the type information. Though the transition required some effort to add type information across the project, the improvement in development efficiency was worth it.

8.5 Limitations

In the version of the artefact that was submitted, there is a cache misconfiguration that causes the PWA to not be able to load new games when the device is offline. This issue has been fixed in newer versions.

There are likely unknown bugs when the application is used on iOS/Safari because an Apple device was not consistently available during development for testing purposes. Even so, enough testing has been conducted that the most important features should work.

Overall, the PWA approach has some significant flaws. The PWA process is unfamiliar for many users, so installing the application to work offline may require guidance which the application doesn't provide. When using Android and Chrome, a scanned QR code will direct the user to the PWA if it is installed. However, with Safari links will continue to direct to user to the website in the Safari browser. It is possible that this issue will be fixed by Apple in the future, but that shouldn't be relied upon.

An alternative approach to packaging the application is to wrap the PWA inside of a native app, allowing it to capture links like a native app. This method has the advantage of small install sizes like regular PWAs, whilst receiving the better platform support and simpler installation process that native apps enjoy. This packaging option could be deployed alongside the existing PWA, so that the application remains easily accessible by URL and there is backup for if the app does not pass review for the app store.

8.6 Practical Applications

Unlike other physical methods of game distribution, QR code games do not need to be removed from where they are situated to be played. They can be placed in the environment and function for anyone that wishes to use them indefinitely. In contrast with the Nintendo e-Reader, the ease with which they can be copied means they cannot be sold or monetised directly,

instead any effort to use them commercially should focus on integrating them into an existing experience.

QR code games could be deployed in situations where an organisation wants to create a real connection between a physical location and a virtual game. If this application were released on app stores, an organisation could make use of it to make games for any purpose without needing to host them on a website or go through the process of releasing and maintaining an app. For example, a museum could create small games to add an interactive edutainment element to exhibits.

Furthermore, because they do not rely on the internet, they can be deployed in outdoor environments where a connection is not guaranteed. Geocaching is a hobby where people use GPS and other clues to find containers – called geocaches – hidden in the environment. QR code games could be integrated into such an experience, allowing the user to grow a library of games on their mobile device as they find them in their environment.

In an educational context, platforms like Scratch are already used to teach programming and encourage creativity; QR code games could fill the same role while introducing a physical element. A lesson could involve having students creating a project and then design a “box” or card for the game to presented in, which could then be cheaply printed. On top of this, students can easily share their work to collaborate by scanning their peers QR codes from their screens.

It is also possible that QR code games could be produced without commercial value, purely for novelty of the constraints it imposes. Fantasy consoles like PICO-8 demonstrate how artificial constraints can encourage creativity by making game development more approachable. Compared to other fantasy consoles, a QR code-based system has the unique characteristic of using physical media, which draws it closer to the retro console experience that these engines often emulate, while still not requiring dedicated hardware. Furthermore, QR code games can be

created and released with complete anonymity, which may be appealing to some users.

9 Conclusion and Future Work

The ability to distribute games through QR codes is intriguing because its enables games to be distributed at a low cost and without depending on the internet. QR code games can be installed into the physical environment to tie a game to real world location. They could be used commercially to provide edutainment at museums, integrated into activities such as geocaching, or bring a physical and collaborative element to computer science education.

The application created for this project allows games such as platformers and air hockey to be created within the size constraints of a QR code. The application supports physics, text-based graphics, tilemaps, procedurally generated audio, and Lua scripting. It is deployed as a progressive web app (PWA) to support use as both an offline app and a website. The size of the application is kept minimal, smaller than all other tested game development applications, ensuring that it can be loaded and installed within 5 seconds even on very poor connections.

The PPMd compression program (which implements prediction by partial matching with information inheritance) is used in the application to compress game scripts and level data. It was ported to the web platform by using Emscripten to compile it to WebAssembly. The compression ratio achieved by this program is superior Deflate, LZMA, and Brotli, while maintaining a small package size. It is used in conjunction with binary serialisation to further reduce the game data size.

9.1 Implications

This project establishes that technical foundation for the use of QR codes as a distribution medium for games. Due to their unique characteristics, they do not directly compete with any existing methods. Instead, they open new possibilities for how games can be deployed both commercially and non-commercially.

An application such as this could be deployed as a non-commercial open-source project, to support the practical applications previously mentioned. Because the system does not require any server-side databases or

authentication, it can be deployed cheaply or entirely for free on a service such as GitHub Pages. However, for users to rely on an external website it would need to support some method of versioning or backwards compatibility, so that games do not break when the platform updates.

Now that QR codes have been proven able to support games, existing game development tools such as Scratch, PICO-8 or other fantasy consoles could use this research to guide their own integration of QR code functionality, augmenting their existing experience with the benefits of QR codes as a medium for game distribution.

The work on compressing game data establishes that PPMd can be used to effectively compress source code and proposes how compression ratios could be improved even further. This should be considered as an option in the further development of platforms like QRscript and potentially in other space constrained environments such as embedded software.

9.2 Future Work

The application created successfully demonstrates the technical feasibility of using QR codes for game distribution, the user experience has not been thoroughly explored. An attempt was made with this project to support a comfortable game development experience on a mobile phone, the only example that could be found of an existing game engine that supports this is GDevelop (Rival, 2026), so it is an area where further research could be valuable.

One avenue for increasing the scope of games possible with the application is procedural generation. Procedural generation is the process of generating content through code rather than by hand. To make this possible, the tilemap could be exposed through the scripting API, allowing the level to be edited or constructed from the games script. Alternatively, a procedural generation algorithm such as Wave Function Collapse (Gumin, 2016) could be included in the engine. Wave Function Collapse can generate new tilemaps that are “locally similar” to an example tilemap; this means that tiles which are never adjacent in the example tilemap are also never adjacent in the generated tilemap. Such an algorithm could also take

advantage of the layout system already in the game engine, allowing levels to be generated at the patch-level rather than the tile-level.

An alternative medium to QR codes with similar characteristics could be explored. For example, NFC stickers are similarly low cost, can be read by phones, and can be installed in the physical environment. In comparison to QR codes, they can have higher data capacity and can be rewritable. This could allow for features such as score boards stored on the sticker and updated by players. It could also be used to support asynchronous collaboration, allowing players to make changes and store them back into the sticker.

10 List of References

An, D. (2018) ‘Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed’, *Think with Google*, 20 February. Available at: https://business.google.com/ca-en/think/marketing-strategies/mobile-page-speed-new-industry-benchmarks/ (Accessed: 30 April 2026).

Apple Inc (2026) *App Review Guidelines*, *Apple Developer*. Available at: https://developer.apple.com/app-store/review/guidelines/ (Accessed: 26 April 2026).

Barett, A. (2024) ‘fflate’. Available at: https://www.npmjs.com/package/fflate (Accessed: 6 May 2026).

BBC News (2021) ‘How Covid turbocharged the QR revolution’, 22 January. Available at: https://www.bbc.co.uk/news/business-55579480 (Accessed: 19 March 2026).

Brown, P.F. *et al.* (1992) ‘An estimate of an upper bound for the entropy of English’, *Computational Linguistics*, 18(1), pp. 31–40. Available at: https://doi.org/10.5555/146680.146685.

Ertugrul, E. *et al.* (2020) ‘Embedding 3D models in offline physical environments’, *Computer Animation and Virtual Worlds*, 31(4–5), p. e1959. Available at: https://doi.org/10.1002/cav.1959.

Forchhammer, S. and Salinas, J.M. (2002) ‘Progressive coding of palette images and digital maps’, *Proceedings DCC 2002. Data Compression Conference*. *DCC 2002. Data Compression Conference*, pp. 362–371. Available at: https://doi.org/10.1109/DCC.2002.999974.

Former, T.B. (2004) *Nintendo will not release E-Reader in Europe*, *Eurogamer.net*. Available at: https://www.eurogamer.net/news080704ereader (Accessed: 22 April 2026).

Francisco, G. (2026) ‘wasmoon’. Available at: https://github.com/ceifa/wasmoon (Accessed: 29 January 2026).

Gleave, A. and Steinruecken, C. (2017) ‘Making Compression Algorithms for Unicode Text’, *2017 Data Compression Conference (DCC)*. *2017 Data Compression Conference (DCC)*, pp. 441–441. Available at: https://doi.org/10.1109/DCC.2017.58.

Good, O.S. (2020) *COVID-19 pandemic turns console gamers to digital sales in record numbers*, *Polygon.com*. Available at: https://www.polygon.com/2020/8/11/21363951/video-game-sales-digital-online-xbox-one-ps4-publisher-figures-ea-activision-2k-games/ (Accessed: 18 March 2026).

Gumin, M. (2016) ‘Wave Function Collapse Algorithm’. Available at: https://github.com/mxgmn/WaveFunctionCollapse (Accessed: 4 May 2026).

Gupta, A., Bansal, A. and Khanduja, V. (2017) ‘Modern lossless compression techniques: Review, comparison and analysis’, *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pp. 1–8. Available at: https://doi.org/10.1109/ICECCT.2017.8117850.

Hardy, P. (2025) *GNU Unifont Glyphs*. Available at: https://www.unifoundry.com/unifont/index.html (Accessed: 29 January 2026).

Hindle, A. *et al.* (2016) ‘On the naturalness of software’, *Commun. ACM*, 59(5), pp. 122–131. Available at: https://doi.org/10.1145/2902362.

Jean-Loup Gailly and Mark Adler (2025) ‘gzip’. Available at: https://savannah.gnu.org/projects/gzip (Accessed: 6 May 2026).

Juan Linietsky and Ariel Manzur (2026) ‘Godot’. Godot Foundation. Available at: https://github.com/godotengine/godot (Accessed: 6 May 2026).

Jyrki Alakuijala *et al.* (2025) ‘Brotli’. Google. Available at: https://github.com/google/brotli (Accessed: 6 May 2026).

Lerner, A. *et al.* (2015) ‘Analyzing the Use of Quick Response Codes in the Wild’, *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. New York, NY, USA: Association for Computing Machinery (MobiSys ’15), pp. 359–374. Available at: https://doi.org/10.1145/2742647.2742650.

Lexaloffle Games (2024) ‘PICO-8’.

liabru (2026) ‘Matter.js’. Available at: https://github.com/liabru/matter-js (Accessed: 27 April 2026).

Louvrier, C. (2026) ‘pingo, image optimizer for web’. Available at: https://css-ig.net/pingo (Accessed: 6 May 2026).

Mahoney, M. (2026) *Large Text Compression Benchmark*, *Large Text Compression Benchmark*. Available at: https://mattmahoney.net/dc/text.html (Accessed: 13 February 2026).

Marijn Haverbeke (2025) ‘CodeMirror’. Available at: http://codemirror.net/ (Accessed: 6 May 2026).

Nemerson, E. (2018) *Squash Compression Benchmark*. Available at: https://quixdb.github.io/squash-benchmark/ (Accessed: 5 May 2026).

NovaSquirrel (2023) *Level Compression*, *NESdev Wiki*. Available at: https://www.nesdev.org/wiki/Level_compression (Accessed: 12 February 2026).

Ofcom (2025) *Mobile Matters Report 2025*. Ofcom. Available at: https://www.ofcom.org.uk/phones-and-broadband/mobile-phones/mobile-matters-interactive-report-2025 (Accessed: 5 May 2026).

PNG Specification: Filter Algorithms (1996). W3C. Available at: https://www.w3.org/TR/PNG-Filters.html (Accessed: 29 January 2026).

Progressive web apps | MDN (2025) *MDN Web Docs*. Available at: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps (Accessed: 29 January 2026).

Rival, F. (2026) 'GDevelop'. Available at: https://gdevelop.io/ (Accessed: 4 May 2026).

Rugg, N. (2016) 'LZMA-JS'. Available at: https://www.npmjs.com/package/lzma (Accessed: 6 May 2026).

Scanzio, S., Cena, G. and Valenzano, A. (2022) 'QRscript: Embedding a Programming Language in QR codes to support Decision and Management', *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8. Available at: https://doi.org/10.1109/ETFA52439.2022.9921530.

Schiffler, C. (2026) 'SAM: Software Automatic Mouth'. Available at: https://github.com/discordier/sam (Accessed: 27 April 2026).

Scratch Foundation (2019) 'Scratch - Imagine, Program, Share'. Available at: https://scratch.mit.edu/ (Accessed: 6 May 2026).

Shkarin, D. (2010) 'PPMd'. Available at: https://www.compression.ru/ds/.

Starzak, J. (2024) 'Music Macro Language Iterator'. Available at: https://github.com/jstarpl/mml-iterator (Accessed: 27 April 2026).

Tim Perry (2024) 'brotli-wasm'. Available at: https://www.npmjs.com/package/brotli-wasm (Accessed: 6 May 2026).

Wilson, H. (2024) '10 > 64, in QR codes', *Huon On The Internet*, 31 March. Available at: https://huonw.github.io/blog/2024/03/qr-base10-base64/ (Accessed: 30 April 2026).

Zakai, A. (2026) 'Emscripten'. Available at: https://github.com/emscripten-core/emscripten (Accessed: 6 May 2026).

11 Appendices

Appendix A Binary serialisation of the game object (TypeScript)

```
export function unpackGame(data: Uint8Array, decompressor: DataTransform =
PPMd.decompress): Game {
    const unpacker = new Unpacker(data, decompressor);
    // Info
    const infoTitle = unpacker.unpackString();
    const infoDescription = unpacker.unpackString();
    // Script
    const script = unpacker.unpackString();
    // Tilemap
    const tileMapDimW = unpacker.unpackUintVar();
    const tileMapDimH = unpacker.unpackUintVar();
    const tileMapCount = unpacker.unpackUintVar();
    const tileMapCodePoints = stringToCodePoints(unpacker.unpackString());
    const tileMapColours = unpacker.unpackUint8Array();
    const tileMap = new TileMap({w: tileMapDimW, h: tileMapDimH}, tileMapCount);
    const patchSize = tileMapDimW * tileMapDimH;
    for(let p = 0; p < tileMapCount; p++) {
        tileMap.tileData[p].codePoint = tileMapCodePoints.slice(patchSize * p,
patchSize * (p + 1));
        tileMap.tileData[p].colour = [...tileMapColours.slice(patchSize * p, patchSize
* (p + 1))];
    }
    // Patch Map
    const patchMapDimW = unpacker.unpackUintVar();
    const patchMapDimH = unpacker.unpackUintVar();
    const patchMapPatchIds = unpacker.unpackUint8Array();
    const patchMapTransforms = unpacker.unpackUint8Array();
    const patchMap = new PatchMap({w: patchMapDimW, h: patchMapDimH});
    patchMap.tileData.patchId = [...patchMapPatchIds];
    patchMap.tileData.transform = [...patchMapTransforms];
    // Markers
    const markerLength = unpacker.unpackUintVar();
    const markers: Marker[] = [];
    for (let i = 0; i < markerLength; i++) {
        markers.push({
            x: unpacker.unpackUintVar(),
            y: unpacker.unpackUintVar(),
            codePoint: unpacker.unpackUintVar(),
        });
    }
    // Solid tiles
    const tileMapSolidTiles = unpacker.unpackString();
    let game = new Game({title: infoTitle, description: infoDescription}, script,
tileMap, patchMap);
    game.markers = markers;
    game.solidTiles = stringToCodePoints(tileMapSolidTiles);
    return game;
}
```

Appendix B Binary deserialisation of the game object (TypeScript)

```
export function packGame(game: Game, compressor: DataTransform = PPMd.compress):
Uint8Array {
    const packer = new Packer(compressor);
    // Info
    packer.packString(game.metadata.title);
    packer.packString(game.metadata.description);
    // Script
    packer.packString(game.script);
    // Tilemap
    packer.packUintVar(game.tileMap.dim.w);
    packer.packUintVar(game.tileMap.dim.h);
    packer.packUintVar(game.tileMap.count);
    const codePoints = [];
    const colours = [];
    for(const data of game.tileMap.tileData) {
        codePoints.push(...data.codePoint);
        colours.push(...data.colour);
    }
    packer.packString(String.fromCodePoint(...codePoints));
    packer.packUint8Array(Uint8Array.from(colours));
    // Patch Map
    packer.packUintVar(game.patchMap.dim.w);
    packer.packUintVar(game.patchMap.dim.h);
    packer.packUint8Array(Uint8Array.from(game.patchMap.tileData.patchId));
    packer.packUint8Array(Uint8Array.from(game.patchMap.tileData.transform));
    // Markers
    packer.packUintVar(game.markers.length);
    for (const marker of game.markers) {
        packer.packUintVar(marker.x);
        packer.packUintVar(marker.y);
        packer.packUintVar(marker.codePoint);
    }
    // Solid Tiles
    packer.packString(String.fromCodePoint(...game.solidTiles));
    return packer.getUint8Array();
}
```

Appendix C Script for Air Hockey demo (Lua)

```lua
-- Paddles
local top = createEntity('----', 8, 96, 32)
top.physics.enabled = true
top.physics.static = true
top.physics.dim.x = 32
local bottom = copyEntity(top)
bottom.pos.y = 256 - 32
-- Ball
local ball = createEntity('●', 4, 96, 128)
ball.physics.enabled = true
ball.physics.bounce = 1.0
ball.physics.dim.x = 16
-- Control Paddles
function onDrag(pos)
  local x = math.min(math.max(pos.x, 32), 192-32)
  if pos.y < 64 then
    top.pos.x = x
  elseif pos.y > (256-64) then
    bottom.pos.x = x
  end
end
-- Score
top.score = 0
bottom.score = 0
local topScore = createEntity('', 12, 20, 4)
topScore.sprite.pivot = {x = -1, y = -1}
local bottomScore = createEntity('', 12, 192-20, 256-4)
bottomScore.sprite.pivot = {x = 1, y = 1}
-- Update score and reset ball
function newRound()
    topScore.sprite.char = tostring(top.score)
    bottomScore.sprite.char = tostring(bottom.score)
    ball.pos.x = 96
    local dirY = math.random(0, 1)*2-1
    ball.pos.y = 128-dirY*64
    ball.physics.vel.y = dirY*3
    ball.physics.vel.x = (math.random(0, 1)*2-1)*1.5
end
newRound()
-- Update
function onUpdate()
    if ball.pos.y < 0 then
        top.score = top.score + 1
        newRound()
    end
    if ball.pos.y > 256 then
        bottom.score = bottom.score + 1
        newRound()
    end
end
```