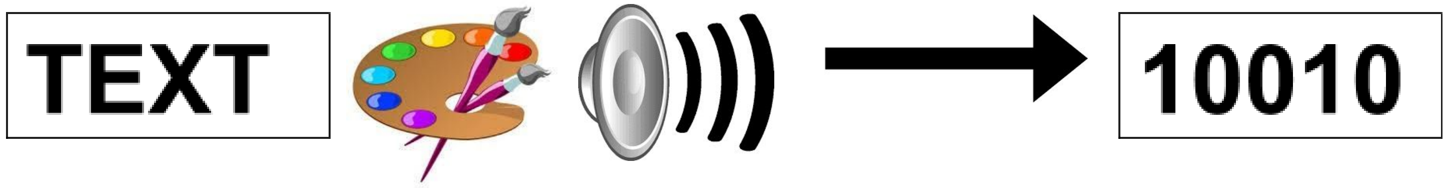
Lesson: 1. Play the "A Journey for Peace" game. Make your own version of the game.		**Comic: All comics in the series.**
Overview of Key Skills Concepts – 2D mapping, N/S/E/W directions, Skills and process - logical thought and planning/strategising		**Cross-curricular links** English: reading and writing, creative writing Maths: coordinates History: famous people PSHE: **Health and Wellbeing**
Learning Objectives:	● To understand the fundamentals of text-based adventure game design ● To understand how a physical area can be represented virtually as a 2D grid ● To understand how the number of elements in a grid is determined by (number of columns x number of rows) ● To learn by practice – how to use strategy and forethought in order to successfully complete the game ● To develop imagination/creative thought by setting up a copy of the game from scratch	
Key Teaching Points / Research Opportunities	Introduce the unit and the learning objectives attached to it. Discuss how computers cannot think for themselves….they need to be told what to do and how to do it! "Stand up! Sit down! Turn your head to the left! Turn your head to the right!"… just as you can follow commands or instructions, a computer can too! When we provide a computer with a list of commands/instructions to follow, this list is called a program. Discuss how computer program instructions are in a mathematical language called **binary** which is made up only of 0s and 1s. Use Sheet 1 – "All about binary". Discuss the good news we don't need to know how to program in a **low-level language** like binary…we can use Visual Basic instead! Explain how the "Talking Adventure Game Maker" has been programmed using Visual Basic and that we have a game based on the "A Journey to Peace" comics, by Dr. Patterson and his crew, and Eevee Fox. Read through the comics and discuss the content, with particular reference to **Health and Wellbeing.** Explain how the game is controlled and how it works. Play the game.	
Independent Work	Think about how you can design your own version of the game. What do you need to know? How can you plan how to go about this? Think about place descriptions, objects and sound effects that will be needed to design the game. Take notes on your thoughts to help you to produce a design plan for your game.	

Plenary	● How did you find the game? Did you like playing it? What did you like/dislike about it? How could it be improved? ● How would you rate your understanding of binary? ● What do you feel that you have learned during this lesson? Now let's try the End-Of-Lesson Assessment.
Resources, including ICT	Appendix 1: Sheet 1 – "All about binary". The "Talking Text Based Adventure Game Maker" software. Planning sheets End-Of-Lesson Assessment Online quiz
Key Questions	● If we use a 4x4 grid, how many locations will we have? ● What language do computers understand? ● What type of programming language can we use to make programming easier? Can you give some examples?
Vocabulary	Binary, low-level language, high-level language, Visual Basic, C#, coordinates, instructions, commands, text, adventure, objects, sound effects.
Success Criteria	● Able to understand the fundamentals of text-based adventure game design ● Able to understand how a physical area can be represented virtually as a 2D grid ● Able to understand how the number of elements in a grid is determined by (number of columns x number of rows) ● Able to use strategy and forethought in order to successfully complete the game
Assessment Opportunities	● Questions on Sheet 1 – "All about binary" ● Completed design sheets and accompanying notes ● Completion of own game ● Post-Lesson Assessment sheet and online quiz

Appendix 1: ALL ABOUT BINARY

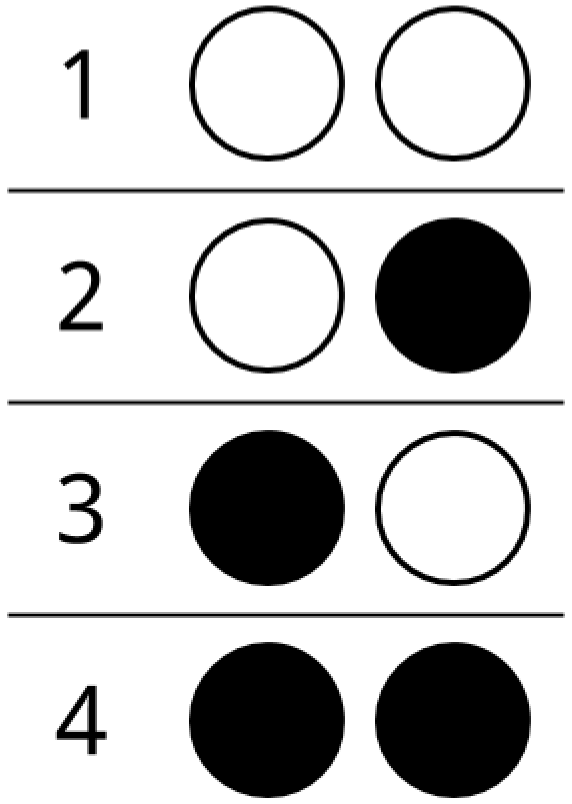
Representing Text, Images and Sound in Binary



More than 200 years ago a 15-year-old French boy invented a system for representing text using combinations of flat and raised dots on paper so that they could be read by touch. The system became very popular with people who had visual impairment as it provided a relatively fast and reliable way to “read” text without seeing it. Louis Braille’s system is an early example of a “binary” representation of data — there are only two symbols (raised and flat), and yet combinations of them can be used to represent reference books and works of literature. Each character in braille is represented with a cell of 6 dots. Each dot can either be raised or not raised. Different numbers and letters can be made by using different patterns of raised and not raised dots.

Let’s work out how many different patterns can be made using the 6 dots in a Braille character. When working through the material in this section, a good way to draw braille on paper without having to actually make raised dots is to draw a rectangle with 6 small circles in it, and to colour in the circles that are raised, and not colour in the ones that aren’t raised.

If braille used only 2 dots, there would be 4 patterns.



NODOT	NODOT
NODOT	DOT
DOT	NODOT
DOT	DOT

And with 3 dots there would be 8 patterns

1	○	○	○
2	○	○	●
3	○	●	○
4	○	●	●
5	●	○	○
6	●	○	●
7	●	●	○

NODOT	NODOT	NODOT
NODOT	NODOT	DOT
NODOT	DOT	NODOT
NODOT	DOT	DOT
DOT	NODOT	NODOT
DOT	NODOT	DOT
DOT	DOT	NODOT
DOT	DOT	DOT

You may have noticed that there are twice as many patterns with 3 dots as there are with 2 dots. It turns out that every time you add an extra dot, that gives twice as many patterns (why?), so with 4 dots there are 16 patterns, 5 dots has 32 patterns, and 6 dots has 64 patterns.

So, Braille can make 64 patterns. That’s enough for all the letters of the alphabet, and other symbols too, such as digits and punctuation.

Braille also illustrates why binary representation is so popular. It would be possible to have three kinds of dot: flat, half raised, and raised. A skilled braille reader could distinguish them, and with three values per dot, you would only need 4 dots to represent 64 patterns. The trouble is that you would need more accurate devices to create the dots, and people would need to be more accurate at sensing them. If a page was squashed, even very slightly, it could leave the information unreadable.

Digital devices almost always use two values (binary) for similar reasons: computer disks and memory can be made cheaper and smaller if they only need to be able to distinguish between two extreme values (such as a high and low voltage), rather than fine-grained distinctions between very subtle differences in voltages. Arithmetic is also easy with binary values; if you have only two digits (0 and 1), then there aren't many rules to learn - adding digits only requires circuits to calculate 0+0, 0+1, 1+0 and 1+1. You might like to work out how many combinations of decimal digits you need to be able to add if you're doing conventional arithmetic!

In fact, every kind of file on a computer is represented using just a whole lot of binary digits — text, pictures, spreadsheets, web pages, songs — *everything* is stored using just two values. Even the programs (apps) that you run use binary representation — sometimes a program file that the computer can run is referred to as a "binary file", which is a bit odd since every file on a computer is binary!

REPRESENTING TEXT WITH BITS

We saw above that 64 unique patterns can be made using 6 dots in Braille. Count how many different upper-case letters, lower-case letters, numbers, and symbols that you could insert into a text editor using your keyboard. (Don't forget to count both of the symbols that share the number keys, and the symbols to the side that are for punctuation!) The collective name for these is *characters* e.g. a, D, 1, h, 6, *,], and ~ are all characters.

Would 6 dots (which can represent 64 patterns) be enough to represent all these characters? If you counted correctly, you should find that there were more than 64 characters! How many bits would you need to be able to represent all the characters you counted on your keyboard?

It turns out that 7 dots is enough as this gives 128 possible patterns, and this is exactly what the ASCII code for text does. ASCII is one of the main systems that computers use to represent English text. It was first used commercially in 1963, and despite the big changes in computers since then, it is still the basis of how English text is stored on computers.

ASCII assigned a different pattern of bits to each of the characters, along with a few other “control” characters that you don’t need to worry about yet. For reasons that we will get to later, each pattern in ASCII is usually stored in 8 bits, with one wasted bit, rather than 7 bits. However, the first bit in each 8-bit pattern is a 0, meaning there are still only 128 possible patterns.

Below is a table that shows the patterns of bits that ASCII uses for each of the characters.

(See the separate ASCII table)

For example, the letter c (lower-case) in the table has the pattern “01100011” (the 0 at the front is just extra padding to make it up to 8 bits). The letter o has the pattern “01101111”. You could write a word out using this code, and if you give it to someone else, they should be able to decode it exactly.

Computers can represent pieces of text with sequences of these patterns, much like Braille does. For example, the word “computers” (all lower-case) would be 01100011 01101111 01101101 01110000 01110101 01110100 01100101 01110010 01110011.

How would you represent the word “science” in ASCII? What about “Wellington” (note that it starts with an upper-case “W”)? How would you represent “358” in ASCII (it is three characters, even though it looks like a number)? What about the sentence “Hello, how are you?” (look for the comma, question mark, and space characters in the ASCII table).

REPRESENTING IMAGES WITH BITS

When a black and white picture, photograph or image is scanned into a computer, the scanner scans in the picture one very thin line at a time. For a large image, this might be around 500 lines.

Along each line, a laser shines a thin beam of light onto the paper many times, moving from one side of the line to the other e.g. from left to right. For a large image it might also shine the beam of light about 500 times. If the beam of light hits a black part of the image, it will bounce back weakly and this might be stored in binary as a zero. If the beam of light hits a white part of the image, it will bounce back strongly and this might be stored in binary as a one. When the scan is finished there will be 500 lines and each line will scan 500 black or white dots into the computer as zeros or ones. Add then all together and you will get 500 lines multiplied by 500 dots which makes 250,000 dots! Of course this means there will be 250,000 zeroes or ones. Remember that a binary zero or one is called a bit. So that means that 250,000 bits have been scanned for the image.

The following table shows a simple image of a cross:

⬛	⬜	⬛
⬜	⬜	⬜
⬛	⬜	⬛

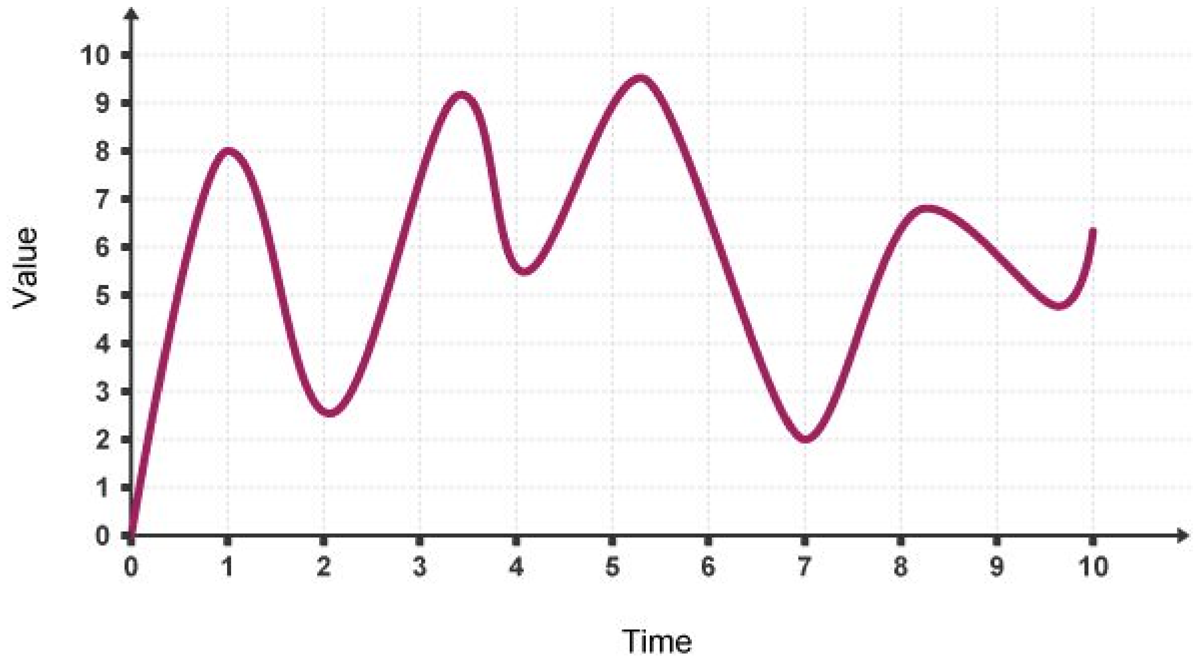
How would this be stored in binary?

Colour images are a little more complicated because we need more bits for each dot! Perhaps we’ll look at how this works at a later stage!

Representing sound

Sound needs to be converted into **binary** for computers to be able to process it. To do this, sound is captured - usually by a microphone - and then converted into a **digital** signal.

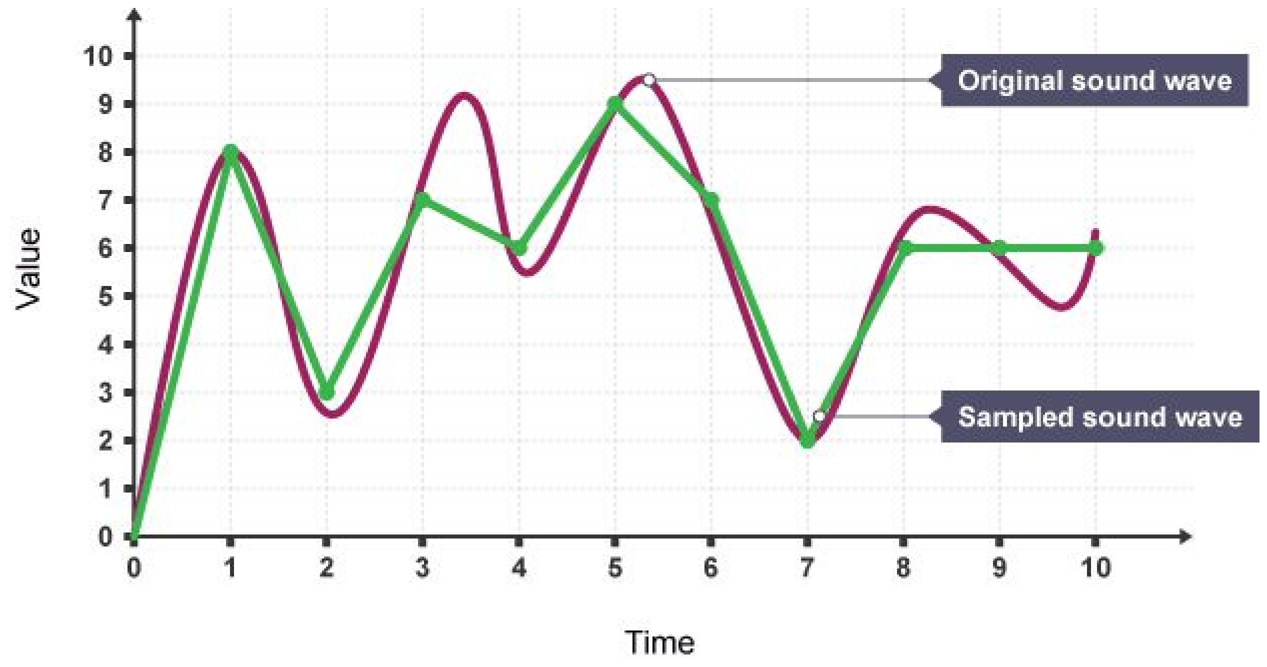
An **analogue** to digital converter will sample a sound wave at regular time intervals. For example, a sound wave like this can be sampled at each time sample point:



The samples can then be converted to binary. They will be recorded to the nearest whole number.

Time sample	1	2	3	4	5	6	7	8	9	10
Denary	8	3	7	6	9	7	2	6	6	6
Binary	1000	0011	0111	0110	1001	0111	0010	0100	0110	0110

If the time samples are then plotted back onto the same graph, it can be seen that the sound wave now looks different. This is because sampling does not take into account what the sound wave is doing in between each time sample.



This means that the sound loses quality as data has been lost between the time samples. The way to increase the quality and store the sound at a quality closer to the original, is to have more time samples that are closer together. This way, more detail about the sound can be collected, so when it’s converted to digital and back to analogue again it does not lose as much quality.
The frequency at which samples are taken is called the **sample rate**, and is measured in Hertz (Hz). 1 Hz is one sample per second. Most CD-quality audio is sampled at 44 100 or 48 000 KHz.

Binary (Alternative)

10101010111110000
10101010111110000
10101010111110000

Computers don’t really understand the English language. They understand their own special kind of mathematical language, known as machine code. Machine code is made with what are known as binary numbers. When we normally count from zero to nine, we count like this:

0
1
2
3
4
5
6

7
8
9

When we get to 10, we are using digits that we have already used before (1 and 0) so there are never any new digits after 9.

In binary, there are only two digits: 0 and 1.

Binary does not use the digits 2-9. We count in binary like this:

0
01
10
11
100
101
110
111
1000
1001

As you can see, when we get as far as nine, in binary it is written as 1001.

When a number is stored inside the computer, it is translated into binary. So if we want the computer to store the number 9, it will be stored as 1001.

One way to imagine how the computer finds it easy to remember binary numbers, is to think about thousands of switches inside the computer. Imagine that each switch can

be switched ON or OFF (just like a light switch). If a switch is ON then it can stand for a 1. If a switch is OFF it can stand for a 0.
So, if we want to store the binary number 1001, a line of switches inside the computer will be switched like this:

ON	OFF	OFF	ON
1	0	0	1

Any other type of information such as text, sound and graphics has to be converted into binary as well, otherwise the computer will not be able to work with it. Even the instructions that are carried out by the CPU are in binary machine code!

Converting from Binary to Decimal

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	1	1	1	1	1	1	1

In the top row of the above table are numbers that can be thought of as column headings. Going from right to left, the column headings are 1, 2, 4, 8, 16, 32, 64 and 128. What do you notice about these numbers? Is there a pattern?

In the bottom row we have our binary number which (from left to right) is 1111111. Notice that it just so happens that there are no zeros in this binary number.

If there is a one under a column heading then the column heading is included in an add sum. Since there is a one under every single column heading in the above table, **all** of the column headings are used in the add sum. This add sum would be as follows:

128+64+32+16+8+4+2+1

The answer to this is 255. This means that the binary number 1111111 is converted to 255 in decimal.

Let’s look at another example:

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	0	1	0	1	0	1	0

In the above table, the bottom row shows the binary number 10101010. Notice that the only column headings that have a one underneath are 128, 32, 8 and 2. So our add sum would be as follows:

128+32+8+2 = 170

This means that the binary number 10101010 is converted to 170 in decimal.

How could we use a spreadsheet to show this?

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
1	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**	
2	1	0	1	0	1	0	1	0	
3									

4	128	0	32	0	8	0	2	1	**170**

How could reproduce the spreadsheet shown in the diagram above using formulas?

Converting from Decimal to Binary

Converting from Decimal to Binary is a little more complicated! The good news is that the spreadsheet has a built-in function to do this for us:

```
=DEC2BIN(cell,digits)
```

Notice that inside the brackets you put the cell reference of the number you want to convert, followed by a comma and then the number of digits that you want such as 8 for an 8-bit binary number.

Try it out on your spreadsheet!

Binary Addition

	A	B	C	D	E
1	1	1	1	0	
2		0	1	1	1
3		1	1	1	0
4	1	0	1	0	1

Look at the above spreadsheet. The red numbers in rows 2 and 3, are two 4-bit binary numbers that will be added together. The first number to add is 0111. What would this be in decimal? The second number to add is 1110. What would this be in decimal? What would be the answer if both decimal numbers were added together?

Let's get back to adding 0111 and 1110. Going from right to left just like in decimal addition, the first two digits to be added (in cells **E2** and **E3**) are 0 and 1. 0+1=1, so that's why a 1 goes underneath in cell **E4**.

The next two digits to add (in cells **D2** and **D3**) are 1 and 1. In binary, 1+1=0, but a 1 is carried over into the next column, that’s why there is a blue 1 in the cell **C1** in the next column to the left.
The next two digits to add (in cells **C2** and **C3**) are 1 and 1, but we must also add the extra blue 1 in cell **C1**, that was carried over from the previous column. This makes 1+1+1. In binary, 1+1+1=1 with 1 carried over to the next column into cell **B1**.
The next two digits to add (in cells **B2** and **B3**) are 0 and 1, but we must also add the extra blue 1 in cell **B1**, that was carried over from the previous column. This makes 0+1+1. In binary, 1+1=0, but a 1 is carried over into the next column, that’s why there is a blue 1 in the cell **A1** in the next column to the left.
There are no more digits to add in cells **A2** and **A3** but a blue 1 has been carried over from the previous column and can be seen in cell **A1**. Nothing in a cell is the same as 0, so this makes 0+0+1. This equals 1 with nothing carried to the next column so now the addition is over! The answer is **10101**. What is this number converted to decimal?

Try this out on the spreadsheet!

APPENDIX 2: DESIGN SHEET

	1	2	3	4
1				
2				
3				
4				