Munching Squares

Teacher Notes to support Tenderfoot Unit 6: Bits AND Chips – Boolean logic and machine architecture

Practical coding and supplementary exercises using software familiar to many teachers – Game Maker. The programming challenge is probably too advanced for KS3 pupils but is used as a basis for teacher development and as an exemplar of problem analysis. The challenge combines binary, logical bitwise operations, two dimensional arrays and a surprising animated display.

Preparation required:
Game Maker installed on all computers.
Munching Squares templates (exemplars for each attendee).
Munching Squares implementation handout (and sample solution code).

Logical Bitwise Operations

This is a practical coding activity designed to reinforce the ideas of basic logic. It has nothing to do with crisps! It is a famous 'display hack'. Display hacks originated as simple code designed to manipulate pixels on computer screens to produce pretty patterns. They developed into a genre of 'computer demos', many becoming famous screensavers. Often these were kaleidoscopic in effect. The presentation gives and example of a famous early display hack known as 'smoking clover'.

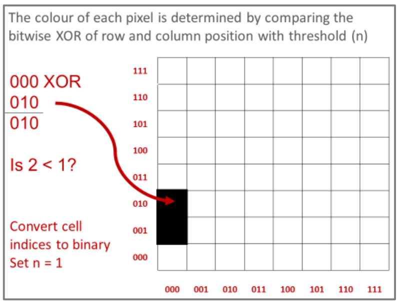
Munching Squares is thought to have been discovered by Jackson Wright on the PDP-1 (shown) around 1962. The formal definition is:

A plot of the cells on a grid satisfying bitwise XOR (x, y) < n for consecutive values of n.

The term n, refers to a threshold value that will increase by 1, up to the size of the grid. The definition may not make much sense yet, but we unpick it in the slides.

We use a grid 8 squares wide and 8 high. The cells are identified by their row and column position, labelled from the bottom left, starting from 0. The colour of each square (representing a pixel) is determined by comparing the result of a bitwise XOR of the row and column position with threshold value (n). A 'bitwise XOR' requires the cell indices to be converted to binary first.

So here the positions are given in their binary equivalent. Let's now consider the first cell; position 000, 000. Each of these bits are considered as input to an XOR in turn, starting with the least significant bit. 0 XOR 0 gives an output of 0 so in this case the output of each bitwise operation is 0. Finally we consider the binary number generated (000) and ask if it is less than the threshold value, which we have set to 1 initially. Because it is true (000 is less than 1) we leave the cell white.


The colour of each pixel is determined by comparing the bitwise XOR of row and column position with threshold (n)

We now move up one cell and XOR 000 with 001. Work through the bitwise operation with students. In this case the output is 001. This isn't less than 1 (false), so we will colour the cell black.

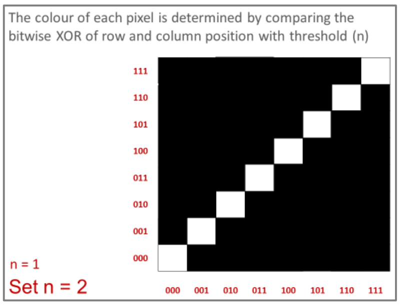
You could ask students what values are compared next. What does the bitwise XOR of 000 and 010 give? In this case the answer is 010, which is 2 in base 10. Is 2 < 1? Clearly not, so the cell is filled black.

We continue up the column doing a similar bitwise comparison. Students can be challenged at this stage to work out what colour each of the cells in that column will be. Every further cell will be black.

We continue the comparisons into the next column. A sequence of clicks walks through the calculation for the next two cells. It's worth seeing if the students can do the process without any prompts.

By now, they may be able to spot a pattern developing. Ask them to predict what the completed grid will look like, before revealing it. Point out that this is the pattern when the threshold value (n) is set to 1. Once complete we increment the threshold value and start again.

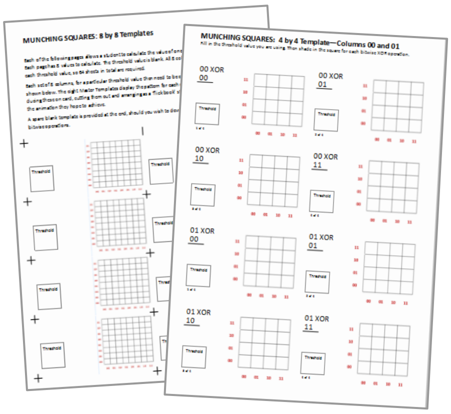
This might be a good time to get students to try to work out the pattern when the threshold is set to 2. The first column is completed to help get them going. The answer might surprise them, and will be revealed on a click.


The colour of each pixel is determined by comparing the bitwise XOR of row and column position with threshold (n)

n = 1

Set n = 2

It may be helpful to take particular squares and work through the calculation so students see how the pattern is derived.

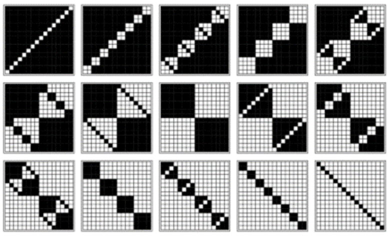

MUNCHING SQUARES: 8 by 8 Templates
MUNCHING SQUARES: 4 by 4 Template—Columns 00 and 01

Again, we increment the threshold value (to 3). Students can either calculate the new values or try to predict the pattern. If predicting, the result may well surprise them again. Once again a click will reveal it. A good group activity might be to work out the rest of the sequence, with each child in a group taking a different threshold value. There are templates provided in the resources to allow students to calculate the values themselves.

An 8 by 8 grid makes a good group activity, with a lot of individual calculations (64 per threshold value). If an 8 by 8 grid is too hard, a 4 by 4 template is also included. The smaller 4 by 4 grid is also provided which could be used as a (homework) exercise to establish understanding.

If you use the group activity to calculate the pattern for each threshold value, the individual patterns can be completed on the 'master templates' provided for each threshold value. By reproducing the 'master templates' on card and cutting them out students can see the animation effect by creating a 'flick book' of their answers. Ensure exemplars are available and attendees understand how to use each sheet.

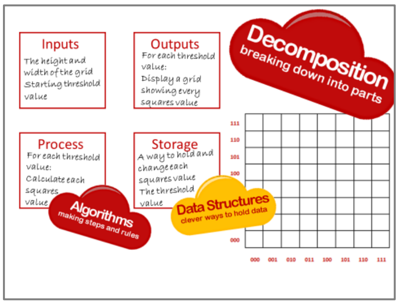
The process can be applied to any sized grid, but numbers divisible by 4 give a nice regular pattern. An example shows the sequence of patterns generated by a 16 by 16 grid. The beauty of the pattern becomes more apparent when you cycle through the threshold values, creating an animation. This animation included came from Wolfram MathWorld: http://mathworld.wolfram.com/MunchingSquares.html



Decomposing the Problem

Our task is to try to code an application to do something similar to the animation shown. Before we worry about the language we will use, let's try to decompose the problem. The notes that follow should be worked through in conjunction with the slides, before delivering the presentation.

Discuss with the group the steps required to develop a solution. A good starting point is to identify answers to the 4 areas outlined (Input / Output / Process / Storage) which are based on a basic model of a computer. The best starting point is usually to consider the output first. This often informs what data needs to be input. A series of two clicks will reveal suggestions for Output and Inputs. From this we can usually get some idea of what needs to be stored and the processes involved. These latter two boxes equate roughly to the algorithms and data structures required in our program.


Inputs
The height and width of the grid
Starting threshold value

Outputs
For each threshold value:
Display a grid showing every squares value

Process
For each threshold value:
Calculate each squares value

Storage
A way to hold and change each squares value
The threshold value

Decomposition
breaking down into parts

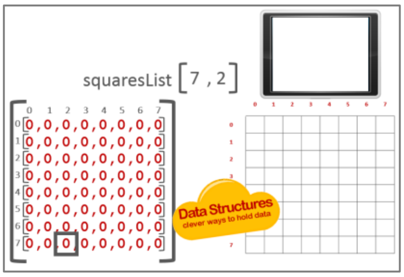
Algorithms
making steps and rules

Data Structures
clever ways to hold data

Let’s now consider the data structures in more detail. The threshold value is a single entity, so a single variable will be fine, but what about a structure to hold the value for each square. What data structure do we know that can store lots of similar items? A list of course (sometimes called an array).

Each square will hold a value of True or False. This will be the result of evaluating whether the result of the bitwise XOR is less than the threshold value, but we can start with them all holding nothing (or zero). We could hold all these in a long list of 64 values but we would have to keep working out which position in the list corresponded to which square in the grid.

A much neater way of storing the values is to think of each row as a separate list. We could then store these lists in a big ‘list of the lists’. Such a structure is sometimes known as a two dimensional list or array. Each list still has 8 index positions (0 to 7). But we now put those into a big list so each small list also has an index position. Let’s call our large list squaresList.



Any individual square can now be referred to by:

- Which small list it is in. This is the first index inside the brackets.
- And then by which position in that list. Which becomes a second index.

To ensure that is clear, four examples are given to test student’s understanding, with progressively less prompts. Further questions can be asked if needed, but please note one key thing. We’ve flipped our grid, so the position 0,0 is now the top left, rather than the bottom left of the grid. This reflects the fact that computer screens and graphic windows usually reference the location of a pixel from the top left.

We’ve considered the data structures needed in our program. Now let’s consider the algorithm;

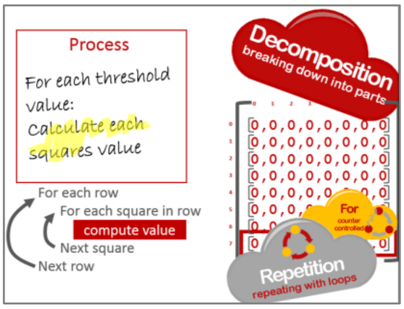
```
For each threshold value:
        Calculate each squares value
```

Again, we can decompose the idea further. First, let’s consider how we might implement the second statement: calculating each squares value. We can break this down into a sequence of steps:

- Convert the two index values to binary
- Perform a bitwise XOR
- Convert the result back
- Compare with threshold

Once we can compute the value for one square what do we need to do? We need to do the same thing for the next one, and the next one, and so on. There are only 3 constructs in algorithms: Sequence, selection and repetition. Which would we use for this? Repetition. More specifically, a counter controlled loop will allow us to move along the row.

Remember, the row is really the first small list in our 2D ‘list of lists’. Having computed the value in the first row, what next? We need to do the same thing for every row (or small list).


Process

For each threshold value:
Calculate each squares value

```
For each row
    For each square in row
        compute value
    Next square
Next row
```

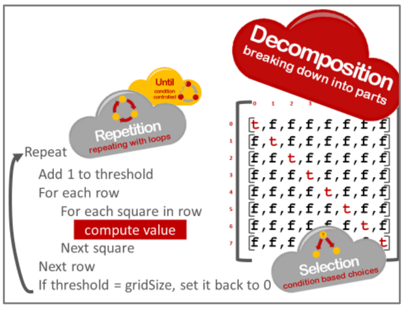
Notice again that we will use a counter controlled loop to do this. We can now see what will happen as this is executed when the threshold value is 1. Each squares value is computed in turn, and true or false replaces the 0 in each position.

Once that is clear, we need to consider how to iterate through each threshold value up to the size of the grid. Remember we want our animation to continue to repeat. How can we do this? Will a counter controlled loop work in this case?

If we start with the threshold set to 0, what needs to be done once we enter the main loop? Add 1 to the threshold to get our starting value. What needs to be done at the end, before repeating? We need to check if it has reached the size of the grid, and if so, set it back to zero.

Continual reinforcement of basic ideas and concepts is important. There are 3 constructs in algorithms: sequence, selection and repetition. What is the last line an example of? It's a selection structure: If. If threshold = gridSize, set it back to 0.

The algorithm will loop back and cycle through the threshold values. What sort of repetition is this an example of? There are two types of loops, counter controlled and condition controlled (Perhaps encourage chanting of these answers).



```
Repeat
  Add 1 to threshold
  For each row
      For each square in row
          compute value
      Next square
  Next row
  If threshold = gridSize, set it back to 0
```

```
[t,f,f,f,f,f,f,f]
[f,t,f,f,f,f,f,f]
[f,f,t,f,f,f,f,f]
[f,f,f,t,f,f,f,f]
[f,f,f,f,t,f,f,f]
[f,f,f,f,f,t,f,f]
[f,f,f,f      f,t,f]
[f,f,f            t]
```

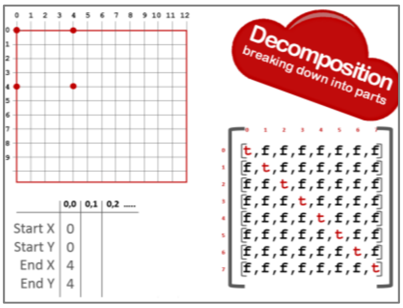
This time we need a condition controlled loop, but what is the condition that will stop it? That can be discussed – maybe a key press can stop the algorithm? As an aside, there are two ways of phrasing a condition controlled loop, either repeating until a condition stops the loop (as in this case), or repeating while a condition remains true. We could use either.

Now that we have an understanding of the key process, let's not worry too much about how we implement it. Instead, let's abstract away the detail and just call it 'compute all squares'. Until we come to implement this, we want to work at a higher level of abstraction, so we can plan and understand the structure of our program.

Let's take stock again, in our planning. We've considered what inputs we need, and how to store any data. We've also considered the process to work out the values of the squares. All that remains is to consider how the result will be displayed. Again, we can decompose the idea further. We already know how to get the value of each square, all we need to do is consider how to draw it. Ask the students what steps are required for each value. Again, clicks reveal the steps, and the pseudocode is outlined below:

```
For each value in list
    If value is true:
        Then draw a white square
    Else draw a black square
Next value in list
```

Let's now consider how to draw a square. We'll assume we want to draw from the top left corner of a screen or window and we want each square to be 4 pixels high / wide. A chart has been started for the first square in our list [0,0] listing the starting and ending pixel values on both the X and Y axis. As a class you can complete the values for the next two squares [0,1] and [0,2] using the presentation. Another click reveals the co-ordinates for [0,2]. These would continue along the top row. Encourage students to see if they can spot a pattern in the way the start and end co-ordinates advance.



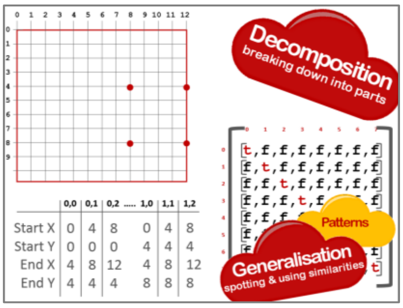
	0,0	0,1	0,2 .....
Start X	0		
Start Y	0		
End X	4		
End Y	4		

```
    0 1 2 3 4 5 6 7
0  [t,f,f,f,f,f,f,f]
1  [f,t,f,f,f,f,f,f]
2  [f,f,t,f,f,f,f,f]
3  [f,f,f,t,f,f,f,f]
4  [f,f,f,f,t,f,f,f]
5  [f,f,f,f,f,t,f,f]
6  [f,f,f,f,f,f,t,f]
7  [f,f,f,f,f,f,f,t]
```

Pattern recognition is an important part of computational thinking. It allows us to generalise a sequence, so we should be able to code it for any number of squares (or size of square).

If participants only come up with fixed relationships i.e. that X always increases by 4 and Y remains the same, encourage them to see how these values relate to the index positions of the value in the list.

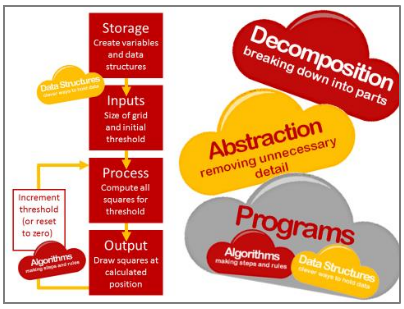
The starting X co-ordinate is given by multiplying the second index position by 4. The finishing X co-ordinate is the starting X co-ordinate plus 4. Similarly, the starting Y co-ordinate is derived by multiplying the first index position by 4, and the end co-ordinate calculated by adding 4 to that value. We can show this more clearly by considering values in the second row.



	0,0	0,1	0,2	1,0	1,1	1,2
Start X	0	4	8	0	4	8
Start Y	0	0	0	4	4	4
End X	4	8	12	4	8	12
End Y	4	4	4	8	8	8

Implementing A Solution

Having decomposed the problem, and armed with the analysis as a result, we can now state the problem as a sequence of higher level abstractions.



First we'll create the data structures required and allow a user to input the variables required. Secondly we'll implement the algorithms required to process the data and output it. Remember, we need to keep looping round and performing the algorithms over and over again.

Hopefully the presentation goes some way to illuminating the idea that programs are just ways of combining 'algorithms (a sequence of steps) with data structures (ways to hold the values).

Moreover, it aims to reinforce the two essential techniques we use over and over again to design programs; decomposition (breaking problems into smaller parts) and abstraction (deciding on an appropriate level of detail, allowing us to see the wood from the trees).

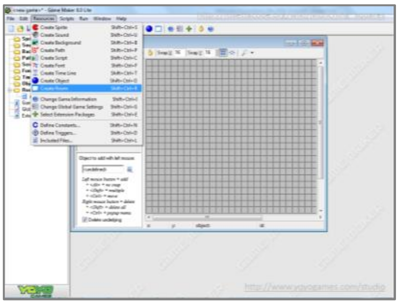
Notice that up to now we haven't considered what programming language to use. Our program can be described independently of any language. Nonetheless, the final challenge is to implement it so we do need to evaluate possible options.

It is worth discussing possible criteria for selecting a language. The criteria table in the presentation suggests two essential elements (drawing capabilities and support for bitwise operations). Direct a discussion towards identifying these, whilst building a list of other desirable factors (the clicks reveal the essential functionality). There may be many more desirable elements – those listed are just some to facilitate debate. Notice though, that there are few of the sort of factors that often pepper the 'language wars' sometimes seen on forums.

Python is a possible contender, as is Greenfoot (Java). Wolfram Mathematica might be a good choice. You'll find implementations in many languages at the Rosetta Code: goo.gl/rCrvKa. We'll use Game Maker, from YoYo Games to develop our solution.

The choice may surprise you but it is an ideal environment. The 'Lite' version is free (Mac and Windows), and is widely used in schools, so children may already be familiar with it. It can be downloaded from goo.gl/BcDvE4. It was developed by Mark Overmaars at Utrecht University to teach game design and programming. It is being continually developed, so check for newer versions at goo.gl/e3AID9

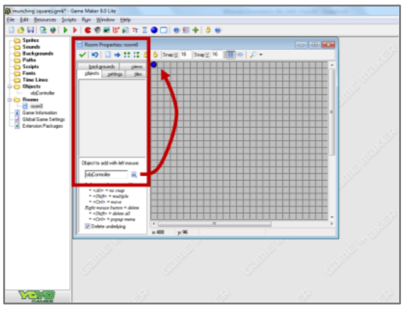
Behind the Graphical User Interface is a powerful programming language. Moreover, it satisfies both of our essential criteria; the language supports bitwise operations and it has drawing capabilities. Although this challenge is probably more appropriate for older students, by familiarising yourselves with this scripting capability, you may be able to introduce simple coding into more traditional 'game design' projects.



For those not familiar with the interface, a Game Maker project is made up of a variety of resources. These are organised in the folders down the left hand side. New resources can be added from the drop down menu, as shown. Notice that one of the resources we can add is Scripts. We'll be adding some Scripts soon.

A game takes place in one or more rooms. We will therefore need a room. We don't need to worry about any of the detailed properties – we just need somewhere for the action to happen.

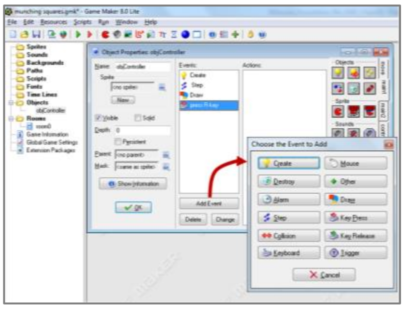
Inside a room, objects can be placed. An object has many properties: what it looks like, its size and so forth. An object can be defined to respond to various Events, such as keypresses, collisions etc. Students normally program the behaviour of an object by dragging a sequence of Actions that would be triggered by a particular Event. In a normal Game Design scenario, children would identify the objects required for their game, and the behaviours they wished to associate with the responses to identified Events.



In our project, we are simply using the room as a canvas (with a helpful grid structure), and we will only need one object to control the start of the script. It is unseen, so no visible appearance is needed.

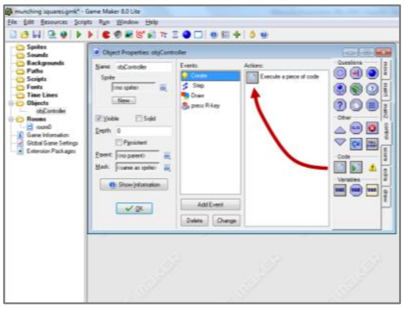
Once the object is created, normally called objController, we drag an instance of it into our room, as shown.

In the Room Properties, under the objects tab, we have selected our object, and clicked the top left square to place an instance in the room.



With the Controller object properties dialogue open, select Add Event. We are going to add 4 Events, which correspond to our program structure as previously outlined:

- The Create event will be used to set up the data structures.
- The Step event will execute the algorithms to calculate each value in the list.
- The Draw event will draw the output, using the room as a canvas grid.
- Finally the Key_Press event is set to a key of your choice (R is used here). When this is pressed it invokes the Action from the Main1 tab, to Restart the Current Room.

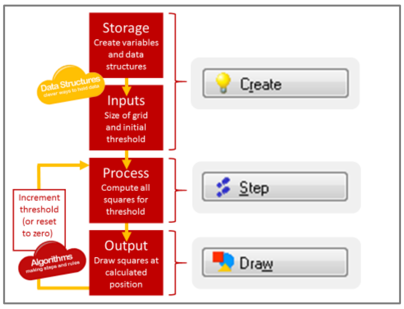


Each of the first 3 Events simply executes a piece of code. When you drag the Code icon from the Control tab into the Actions area, it opens a code editor. This is where you will prepare each of your 3 scripts. You can save them as text files whilst you are working on them, but the Code Editor provides simple syntax checking. That's the challenge.

Slides outline how you might find out more about the syntax of the Game Maker language. Help is available from the menu, or the F1 shortcut. There is a whole section on the Game Maker Language.

As shown, under the GML Overview, the documented Expressions include bitwise OR, AND and XOR. Game Maker uses the ^ symbol for performing a bitwise XOR.

Another way to navigate Help might be through the search facility. A search using 'draw' returns a Drawing Shapes topic, which includes details of the command to draw a rectangle.


Storage
Create variables and data structures
Data Structures
clever ways to hold data
Inputs
Size of grid and initial threshold
Process
Compute all squares for threshold
Increment threshold (or reset to zero)
Algorithms
making steps and rules
Output
Draw squares at calculated position
Create
Step
Draw

A slide outlines the pseudocode for the core algorithm, and can be left on display for reference (or handed out as a prompt sheet). It is strongly recommended you try to solve this yourself, before setting it as a challenge. Warning: the slide that follows contains a solution. A coded solution including comments is also included in the resources for you to study if you get stuck.

The solution should demonstrate how simple the required code is. The key difficulty lies in mastering a nested loop for iterating through the array.

Once coded, there are several logical and bitwise operations you can experiment with. As previously noted, Game Maker supports several bitwise operations. It uses | for a bitwise OR and & for bitwise AND. A ~ is used for NOT. You can experiment with a variety of comparison operators such as >, < and ==. Explore div and mod (floor division and remainder) too. All sorts of patterns are possible and often come as a surprise.

Once implemented this provides a nice environment to encourage experimentation and help reinforce a variety of logical operations. Several suggestions are given in the code comments in the solution included. You'll find them as comments in the Step Event script. Encourage students to work out samples by hand first, rather than just experimenting at random. Blank '4 by 4' grids are included as photocopiable masters in the resources to facilitate this.