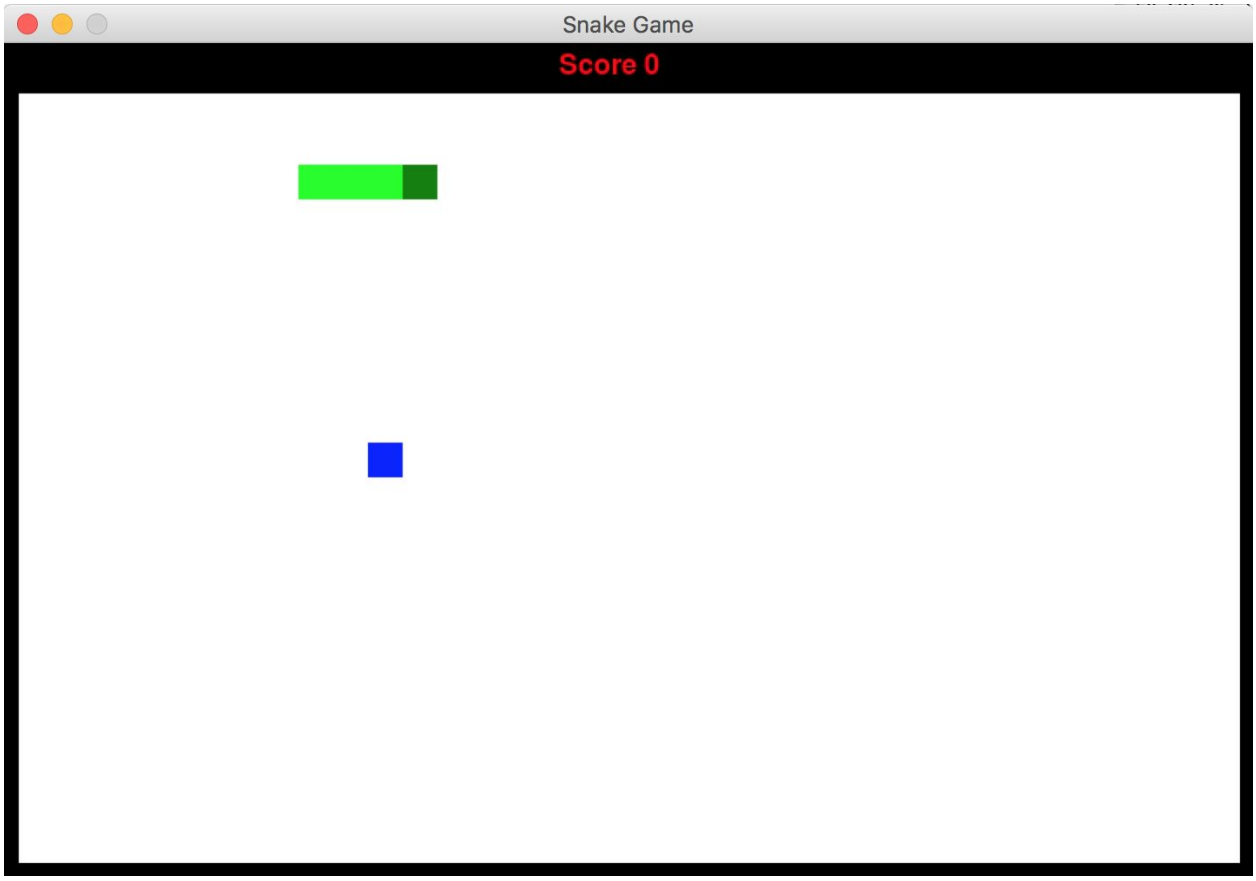


Project Case Study: Snake Game!

Year level band: 9-10

Description:

This project creates the Snake Game seen in the Worked Example for the 9/10 MOOC, using a Makey Makey controller to build skill in the use of digital systems, Object-Oriented programming design and implementation, and data representation. Students will learn about Object-Oriented programming using existing Classes and Objects, as well in designing and creating multiple new Classes, making this a good intermediate project.

Resources:

- MakeyMakey kit
- Mac or PC with Python 3 and pygame library installed
- Free USB port to plug MakeyMakey into

Prior Student Learning:

A basic understanding of circuits is useful.

An understanding of general and Object-Oriented programming concepts.

Digital Technologies Summary	This activity introduces students to: ● complex digital systems, including the use of MakeyMakey controller and transmission in networked environments. ● Object-Oriented design and implementation, including the analysis, tracing and understanding and existing Object-Oriented software components and the design and creation of new Object-Oriented software components. ● The lesson introduces the students to the idea of designing interactive programs, developing systems thinking, focusing on how different aspects of a program interact with each other..
Band	**Content Descriptors**
9-10	● Investigate the role of hardware and software in managing, controlling and securing the movement of and access to data in networked digital systems (ACTDIK034) ● Analyse and visualise data to create information and address complex problems, and model processes, entities and their relationships using structured data (ACTDIP037) ● Design the user experience of a digital system by evaluating alternative designs against criteria including functionality, accessibility, usability, and aesthetics (ACTDIP039) ● Design algorithms represented diagrammatically and in structured English and validate algorithms and programs through tracing and test cases (ACTDIP040) ● Implement modular programs, applying selected algorithms and data structures including using an object-oriented programming language (ACTDIP041) ● Plan and manage projects using an iterative and collaborative approach, identifying risks and considering safety and sustainability (ACTDIP044)
Achievement Standards	● Students plan and manage digital projects using an iterative approach. ● They define and decompose complex problems in terms of functional and non-functional requirements. ● Students design and evaluate user experiences and algorithms. ● They design and implement modular programs, including an object-oriented program, using algorithms and data structures involving modular functions that reflect the relationships of real-world data and data entities. ● They share and collaborate online, establishing protocols for the use, transmission and maintenance of data and projects.

Project Outline

In this project, students will work through the problem solving and project design process to create a Snake game, using a MakeyMakey controller to send directions to the snake, controlling its movement on the screen.

Defining the Problem	To create a **Snake game** that allows users to control the movement of a snake on a screen, to get points for eating food and avoiding running into the walls or the growing tail of the snake itself. In this problem, we want to write a game where a graphical representation of a snake moves across the screen. When it encounters a piece of food, the snake grows longer and we gain a point. If it hits the wall or runs into itself, we die. The snake is controlled by a MakeyMakey controller. To write this program we are going to need: ● A way of representing the snake ● A way of representing the food ● A way to display the score, ● a way for our instructions to reach the snake, ● and a way to know when we've run into something and died Our system is going to involve working with both hardware and software, and so we will need to understand what we have available in hardware that can assist us. The MakeyMakey board produces information that can be interpreted by our computers as keystrokes, as if they were typed on a keyboard. The 6 options available on the board face are the four directional arrows (UP, DOWN, LEFT, RIGHT), the space bar, and a click. If we build our software so that the snake is controlled by directional arrows on the keyboard, then it will work the same way when we hook up the MakeyMakey. The only difference is that we can build our own controller board for the snake game, rather than squeezing our fingers into the space that holds the arrow keys on a standard keyboard. Now that understand how our hardware will work in the design of our system, let's move on to starting the design of our software system. We are going to use an object-oriented approach and provide some detail here. The 9/10 MOOC course contains a much more detailed exploration of how to build this. We have to think about the Classes that we want to build, with the associated variables and functions that will make sense for the development. Let's start by looking at the snake itself, the hero of the game. The snake has a location on the screen, and contains multiple visual elements, as it can

grow, and the snake's head is connected to the rest of the snake and the snake's body follows it around the screen. If the snake "eats" food, it grows. The snake also keeps track of which way it's going, using the heading variable. This gives us a high level Class design of:

Class: Snake	Variables	Functions
	Head position Body locations Heading	Move Grow Eat

Here are the requirements (functional requirements) for how the snake moves.

1. The snake must appear to move around the screen
2. The snake must turn in response to user input
3. The snake will increase in length if it eats food
4. The snake will die if it runs over itself
5. The snake will die if it runs into the walls
6. The snake never stops moving

To keep the game interesting,

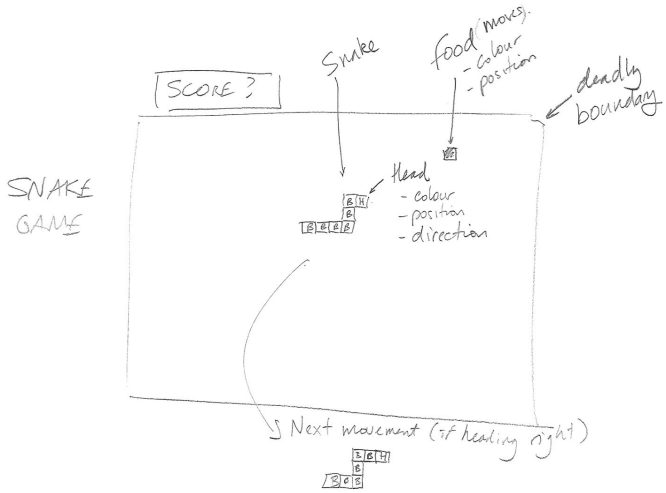
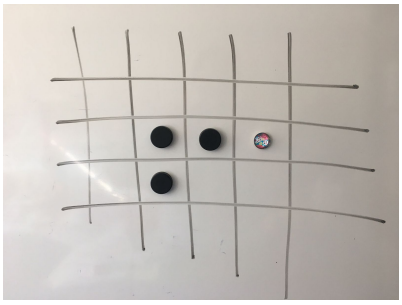
1. The snake must move at a speed fast enough to be interesting but slow enough to control.
2. The code must be object-oriented (curriculum requirement)
3. The game should still be playable with a really, really long snake.

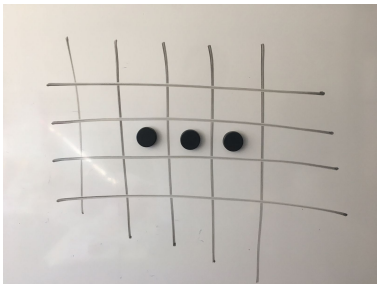
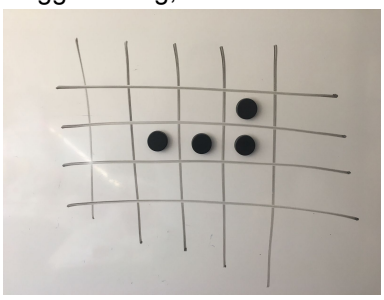
The snake moves in a very precise way. Based on what the user types, the snake will move in a given direction. Every time the snake moves, the head will go in the new direction, and every piece of the snake will move up, by occupying the space that was formerly occupied by the piece in front of it.

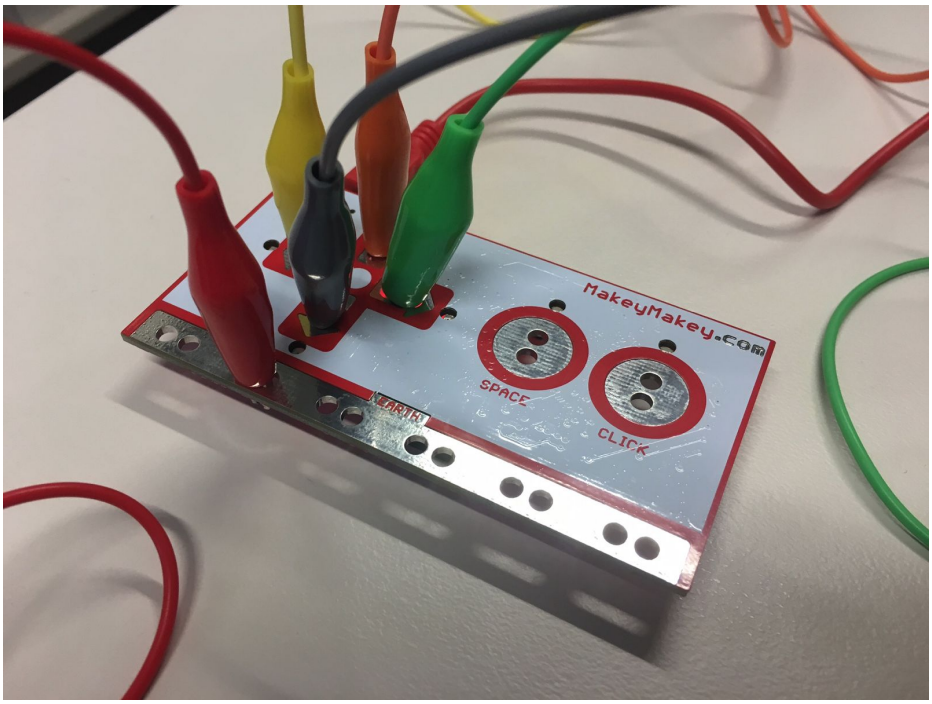
To grow in size, the snake has to eat food. How can we show the snake eating? The simplest answer is that if the head of the snake and the food are in the same place, we consider that the snake eats the food. This means that we have to know where the food is. When it's eaten, it disappears, the snake grows, and the food shows up somewhere else.

The coordinates of the food are part of its variables and it has a function *move*, which will move it to a different place. To make the game interesting, we probably want this to be a random location, which means that we'll have to make sure that our program can generate random numbers. I also made a note that the food will keep track of its own colour, that would be another variable.

	<table><tr><td rowspan="2">Class: Food</td><td>Variables</td><td>Functions</td></tr><tr><td>Positon Colour</td><td>Move</td></tr></table> But we also wanted to show a score, so we need a variable to keep track of that as well. In this case, we'll create a Scoreboard class where we can increase the counter value and display it. <table><tr><td rowspan="2">Class: Scoreboard</td><td>Variables</td><td>Functions</td></tr><tr><td>Counter</td><td>Display Increase</td></tr></table> Let's look at how a program to run the whole game might look: 1. Draw the playing area with bounding rectangle, set the counter to zero and display it. 2. Draw the snake in a starting position. 3. Draw the food in a starting location. 4. On user input, change snake direction. 5. Move the snake one move 6. If the snake is over food, eat it, increase the score, grow, move the food, 7. else if the snake is over itself or in a wall, die. 8. Go back to 4. 9. Until the snake dies We didn't talk about what makes the snake die. The snake eats when the food and the snake's head are in the same place. In a game, we refer to that as collision detection. Although the objects didn't really collide, in a real world sense, they are in the same place and we can perform actions based on this. We're going to use collision detection to see when the snake runs into its own body, which means that we're going to have to keep track of every piece of the body in variables in our program. Just because something is drawn on the screen doesn't mean that it's still stored in a variable somewhere.
Understanding our Data	Most of the data representation in the program is straightforward. The score is a number. The position of the food is given by an (x,y) coordinate pair but the food's move function needs to know the size of the canvas if it's going to randomly appear in a place where we can eat it. Our Scoreboard class needs to be told when the score has increased and this function will be called whenever the snake runs over food.

	The Snake is more complex as it needs to know the bounds of the canvas, the location of the food, and the location of all pieces of itself. Thus, the Snake class keeps track of where the head of the Snake is but stores a list of Body pieces. By providing a collide function in the Snake class, the Snake objects can search their internal set of pieces to see if any of them match the position of the head. The result of this search is returned to the calling program. The Snake's eat method, which was not originally implemented in the Worked Example, should take a food as a parameter and check to see if the food and the Snake are in the same place. If they are, the Snake should grow, the counter should increase, and the food should move.
Prototyping and User Interface Design	Sketch prototypes are very useful to get a feel for how the game will work. These can be actual sketches or small examples using a whiteboard or post-it notes.  And here are the whiteboard examples, showing a snake moving, eating food, and then growing after eating.  Here the snake is approaching the food from the right.

	 The snake moves to enter the spot, with the last element of its tail being dragged along, as this snake is only three long.  But now it has eaten, so it increases in length, and we have steered it up to get away from the boundary. Because we are using the MakeyMakey, students are free to develop their own control systems for the snake game. What will these look like? The MakeyMakey connection allows us to discuss the representation of data in computer systems, as the electrical circuit initiates a signal that is interpreted by the MakeyMakey board as if you had pressed the corresponding key on the keyboard. This keyboard signal is then sent via USB to the computer, where it is picked up just as if it had been typed in the usual fashion.
Implementation	The full Python3 code for this is shown below. This will automatically connect to a MakeyMakey hooked in and using the directional arrows. The MakeyMakey will need to be connected to the USB port and the earth lead held by (or attached to) the person who'll be playing the game. Each of the directional controller leads will need to be attached and either touched directly by the player or connected to another object that will act as the arrow. The board should look like this:



The following code shows the entire implementation. This is one way to solve the problem but there are many others.

```
import sys, pygame
# This gives us access to the Pygame library
from random import randint
# We need this to generate random numbers

pygame.init()                # Set up pygame
# Set up the font we'll use for the score
font = pygame.font.SysFont("None", 24)
# Set up the window that we'll use
main = pygame.display.set_mode((720, 480))
# Fill the window with white
main.fill((255, 255, 255))

class Body:  # This is the Body of the snake
    x=0           # x position of this piece of the body
    y=0           # y position of this piece of the body

    # An __init__ function sets up the objects built from
    # this class
    def __init__(self,x,y):
        self.x=x      # Setting the x and y variables
        self.y=y

# This is the main Snake class, the heart of the game
class Snake:
    # This variable is a Python list
    # that will keep track of the body pieces
```

```
    bodylist=[]
    heading=0      # Which way the snake is going to move
    x=0            # x position of the snake's head
    y=0            # y position of the snake's head

    def __init__(self,x,y):   # initialistion function

        # Always start the snake pointing right.
        self.heading=pygame.K_RIGHT
        self.x=x               # Set the x,y position
        self.y=y

        # Now add three pieces of Body to the head
        body = Body(x-20,y)
        self.bodylist.append(body)

        # We create new body pieces and append them to
        # the end of the bodylist.

        bod2 = Body(x-40,y)
        self.bodylist.append(bod2)

        # They are all staggered to trail away to the
        # left because the snake is going right.

        bod3 = Body(x-60,y)
        self.bodylist.append(bod3)

# The draw function steps through every element of body,
# looks at their x,y coordinates and
# draws a matching square on the screen.
# The head of the snake is drawn last, in a darker
# colour, to make it clear which way it's going.

    def draw(self):
        for element in self.bodylist:

pygame.draw.rect(main,(0,255,0),((element.x-10,element.y-10
),(20,20)),0)

pygame.draw.rect(main,(10,128,0),((self.x-10,self.y-10),(20
,20)),0)

# The setheading function will tell the snake's head
# which way to move next and is set
# by passing keystrokes (or MakeyMakey commands) to the
# program

    def setheading(self,keystroke):
        self.heading=keystroke

# The collide function is used to see if the snake has
# run into itself.
```

```
# We search through every part of the body and look at
# the x and y values to see if they
# match the head position. If they do, we've crashed,
# game over.

    def collide(self):
        for bd in self.bodylist:
            if ((self.x,self.y)==(bd.x,bd.y)):
                return True
        return False

# The grow function adds an element to the end of the
# snake. This means that, over time as the snake eats,
# it's going to get longer and longer

    def grow(self):
        tmpbody=self.bodylist[len(self.bodylist)-1]
        self.bodylist.append(tmpbody)

# The move function takes advantage of the way that the
# snake moves.
# Because the snake follows its own path, we can add a
# copy of the head to the front of the body and throw
# away the old tail of the list.
# This way, the real head moves to a new position, but
# everything appears to move up one.
# We then draw a white square where the old end used to
# be to erase it.
# Move is also where we handle the keystrokes
# pygame.K_RIGHT refers to the right arrow
# The adjustments to x and y reflect the direction given
# by the arrow and the orientation of the axes on
# the canvas.
# If the user types a 'q', we exit.

    def move(self):
        self.bodylist.insert(0,Body(self.x,self.y))
        removeBlock=self.bodylist.pop()

pygame.draw.rect(main,(255,255,255),((removeBlock.x-10,remo
veBlock.y-10),(20,20)),0)
        if (self.heading==pygame.K_RIGHT):
            self.x+=20
        elif (self.heading==pygame.K_DOWN):
            self.y+=20
        elif (self.heading==pygame.K_UP):
            self.y-=20
        elif (self.heading==pygame.K_LEFT):
            self.x-=20
        elif (self.heading==pygame.K_q):
            sys.exit()
        else:
            return
```

```
# The eat function checks to see if the head of
# the snake and the food are in the same place.
# If they are, return True, else False

    def eat(self,food):
        return ((food.x==self.x) and (food.y==self.y))

# END OF THE SNAKE CLASS

class Food:  # The Food class provides scoring.
    x=0      # Each Food object has an x and y position.
    y=0

    def __init__(self): # But they are set randomly.
        self.x=randint(1,34)*20
        self.y=(randint(1,22)*20)+20
        self.colr=(0,0,255)

# The random numbers chosen for the food have to match up
# to the possible locations of the snake head. We've done
# some calculations of the screen size, taking into
# account that the x,y values of the snake head will
# always be multiples of 20.
# (This is not the simplest way to write this.)

# The Food class has its own drawing instructions
    def draw(self):
pygame.draw.rect(main,self.colr,((self.x-10,self.y-10),(20,
20)),0)

    def move(self): # If Food is eaten, we move.
                    # Using the same algorithm as above.
                    # But first we delete our old self.

pygame.draw.rect(main,(255,255,255),((self.x-10,self.y-10),
(20,20)),0)
        self.x=randint(1,34)*20
        self.y=(randint(1,22)*20)+20

# END OF THE FOOD CLASS

class Scoreboard:  # The Scoreboard class records
    score = 0      # the score

    def __init__(self):  # Initialised to zero.
        self.score=0

    # Increased by calling this function

    def increase(self):
        self.score=self.score+1
```

```
# The display function redraws the black rectangle at the
# top because constant redrawing of the text makes it
# blurry. The text is a rendered string that contains the
# Score information. The blit function call sends the
# rendered text to the screen.

    def display(self):
        pygame.draw.rect(main, (0,0,0), ((0,0),(720,20)),0)
        text=font.render("Score "+ str(self.score),
True,(255,0,0))
        main.blit(text,(320,5))

# END OF THE SCOREBOARD CLASS

# outofbounds is non-class based function that checks to
# see if whatever is being tested is still inside the
# field of play

def outofbounds(x, y):
   if ((x < 20) or (x > 710) or (y < 40) or (y > 470)):
      return True
   return False

# This is the main routine of the program

def Main():
    # Set up all of the objects
    scoreboard = Scoreboard()
    blueberry = Food()
    snake = Snake(100,80)

    # Draw the snake
    snake.draw()
    # Force the display to update
    pygame.display.update()

    # This is the game loop
    while (True):
        # Show the score
        scoreboard.display()

      # Draw the food in its first position
        blueberry.draw()

      # Check that we can still play
      # Are we in bounds?
        if outofbounds(snake.x,snake.y):
            sys.exit()
      # Have we collided with ourselves?
        if (snake.collide()):
            sys.exit()

      # Now see if we can eat anything
```

```
        if snake.eat(blueberry):
          # We can!
          # Increase the score!
            scoreboard.increase()
          # Make the snake bigger!
            snake.grow()
          # Move the fruit!
            blueberry.move()

      # Now move the snake to a new position
        snake.move()

      # And redraw it there.
        snake.draw()

      # Force the display to update
        pygame.display.update()

      # Now look to see if something has
      # happened in the real world.

        for event in pygame.event.get():

      # Did someone hit a key?
      # Pass it to the snake

            if event.type == pygame.KEYDOWN:
                snake.setheading(event.key)

        # Did someone close the window?
      # Quit the game
            if event.type == pygame.QUIT:
                pygame.quit(); sys.exit()

      # This slows the game down enough to make
      # it playable. You can adjust the
      # setting to make the game harder or
      # easier.

        pygame.time.delay(100)

# These actions are taken when the game starts up.
# We set the caption for the window and draw
# the boundin rectangles, then we call
# the Main function

if __name__ == '__main__':
    pygame.display.set_caption("Snake Game")
    pygame.draw.rect(main, (0,0,0), ((0,0),(720,20)),0)
    pygame.draw.rect(main, (0,0,0), ((0,20),(720,460)), 16)

    Main()
```

	`# And that's it!`
Testing and Evaluation	How can we test a full Snake game and, assuming it passes that stage, how can we playtest that? The functional requirements that the student developed turn, almost immediately, into a checklist that can be handed to another student. Let's look at an example. To display the snake, the first thing we want to do is to make sure that we can draw the snake and move it around on the screen. So our testing for correct function will be: 1. Can I display the snake's head on the screen? 2. Will it move around as I want it to using keyboard control? 3. Is it displaying correctly? 4. Is the body moving correctly? If we identify an error in the snake, because it's a Class, we will go into the Snake class and fix it there. However, because we've written the Food and Scoreboard as separate classes, whatever we do in the Snake class *shouldn't* break anything in there, unless we accidentally change the code without noticing. The next step for the snake will be checking what happens when the head is detected as colliding with something. Does it grow when it eats food? Does it die when it hits a wall or itself? We'd then continue to test the program until we've tested all of the individual elements and their interactions together. One useful test case is to see if everything is being drawn where you expect. Because we aren't using all the screen, it's possible to draw the food or the snake so that it overlaps the black rectangle that's the boundary. Has the programmer put the correct limits on the ranges where the snake and the food can appear? Testing the non-functional requirements often falls into the realm of playability. We noted before that we can ask students whether the controls are obvious and responsive, and whether the game is fun to play. But they may have to play for a while to get a real feeling for it. What's it like to play the game for an extended period? Many snake games increase the speed of the snake as it gets longer, increasing the difficulty even further. This increased movement speed gives a sense of urgency and can be a way to engage players with the snake. But make it too fast and it becomes unplayable!
Project Extensions	There are many ways this could be extended. Some that we have thought of include: ● How would students go about producing a two-player game? ● Can you make the Snake speed up as it gets longer? ● Can you use a timer to make the Snake starve if it doesn't eat quickly enough?

Project Reflection

In this project, students will undertake an extensive design and implementation process, combining elements of hardware and software. The project is well suited to be taught over a number of sessions, and using group work. While we have not included explicit assessment information here, we have provided a discussion of how these sessions might be structured, and how students might be able to demonstrate their learning.

For a more extensive discussion of assessment possibilities within Digital Technologies at this level, please join us in our online community and course: www.csermoocs.adelaide.edu.au

Learning construction	Students work independently or in teams to design and implement the Python program and the customised MakeyMakey controller. This project is well suited to teamwork, and can be used to build collaboration and communication skills. The activity is about experimenting, trying new solutions, and debugging. Once students have successfully built a working game, ask them to explore what other rules they could add to the game to make it more challenging or more enjoyable. The project can be extended in multiple ways through exploring different variants of the game and customising the components. Encourage students to help each other - and look for help on the internet. Ask a friend. Ask Google. Then ask the teacher.
Learning demo	While students are working in groups, ask questions to give them the opportunity to demonstrate their thinking and understanding: What challenges have you faced in developing this design? How did they decide how to store the data? How do the objects interact with each other?
Learning reflection	Remind students that this is a very simple game but that the program we have been developing has a development process that you would find in real-world game development. The steps that they have taken can be applied to many other larger projects ● Can you think of any exciting games that they want to build, which they haven't managed to find anywhere else? ● How else could we control the Snake? ● Could we build a 'brain' for the Snake that plays the game for us? What would that look like? What other real-world problems could they solve?

Teacher/Student Instructions:

Download the latest version of Python 3 from http://python.org .
When this has been installed, follow the instructions at http://pygame.org/wiki/GettingStarted to install the Pygame library.
IMPORTANT: You must use python3 to run the pygame library on many platforms.

CSER Professional Learning:

This Case Study supports learning from the CSER 9&10 Digital Technologies: Explore! MOOC. You can join us here to learn more about Digital Technologies in Years 9 & 10: www.csermoocs.adelaide.edu.au

Further Resources:

1. Information about Python, including tutorials: http://python.org
2. Information on the pygame library: http://pygame.org
3. Information on the MakeyMakey: http://makeymakey.com/