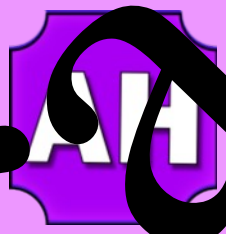
Computing Science

Software Design and Development

Advanced Higher Programming

(using Python and Pygame)



Project 1 - Balloon Burst

Version 1

Contents

Project 1 - Balloon Burst

Coding Balloon Burst

Extension Work

Improving Your Programming

How to use each booklet

There are four booklets in this series:

- Project 1 - Balloon Burst
- Project 2 -
- Project 3 -
- Project 4 - Galaxians

The four booklets have been written to cover the following content in Advanced Higher Computing.

	Advanced Higher AH
Languages and Environments Programming Paradigms	Object Orientated • object • encapsulation • method • property • class • inheritance Imperative • variables • sequence • selection • iteration • modularity
Computational Constructs and Principles (for software and information systems)	Explain and Implement the following constructs: • reading and writing data from sequential files • reading and writing data to and from databases
Data Types and Structures	• records, linked lists • 2-D arrays • queues, stacks • arrays of records and/or array of objects
Standard Algorithms	• linear and binary search • selection with two lists • sort algorithms (insertion, bubble, quicksort)

This booklet contains object orientated programming theory and practical work. Make sure you read both carefully to ensure a full understanding of the code.

There is an expectation at Advanced Higher that pupils work independantly. Ensure that you have spent a significant amount of time trying to overcomes issues in your code before you ask for help.

AH

Introduction

As many pupils who program have an interest in computer games, this unit will cover the programming requirements of the Advanced Higher course by teaching the basics of game coding. Before you start work ensure that you have the following installed:

- Python 3 - the programming language used in this unit.
- A suitable Python Code Editor - PyScripter was used to write the projects in this unit but any Python development environment may be used.
- Pygame - a module library of procedures and functions used to create and manipulate game sprites.

As you progress through this unit you will experience a series of increasingly complex projects in the form of simple games. Each project will include detailed explanations of the code which should be used as a reference throughout your course. On the completion of each project you will be expected to design and create a game of a similar complexity using the programming paradigms, constructs, data structures and standard algorithms covered in the projects.

AH Programming in Summary

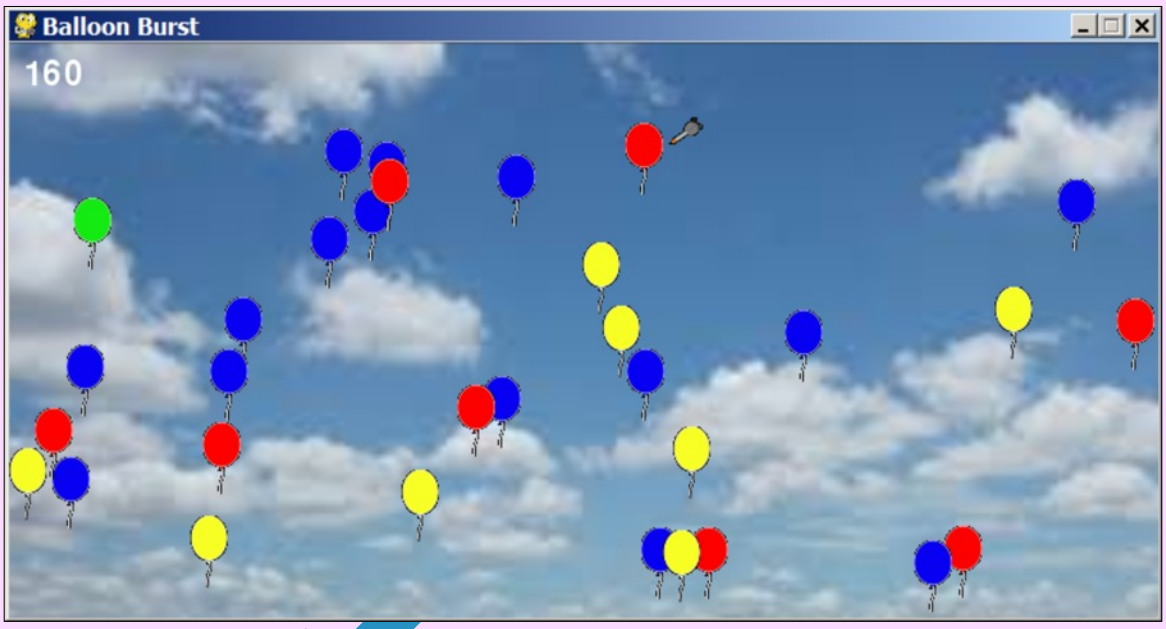
In the Advanced Higher course you are expected to learn, understand and be proficient in the following.

- object-orientated programming
- several new data structures including linked lists, dictionaries, records, queues, stacks and 2D arrays
- file handing
- reading and writing to/from databases
- standard algorithms including binary search, selection with two lists, insertion sort, bubble sort and quick sort.

Projects Covered in Unit

Balloon Burst

This game will be used to introduce the concept of objects, instances and methods. The game will generate different coloured balloons approximately once every second. These will drift left and right across the game window. The user has to click on the balloons to burst them. Points will be award for each balloon burst. The blue balloons should be avoided as they will end the game.



Project 2 - Game to be decided - Preview for staff below

Project will include: animated sprites, 2D array use, high score stored in text file (keyboard input required). Objects will now include concepts of inheritance an encapsulation.

Projects

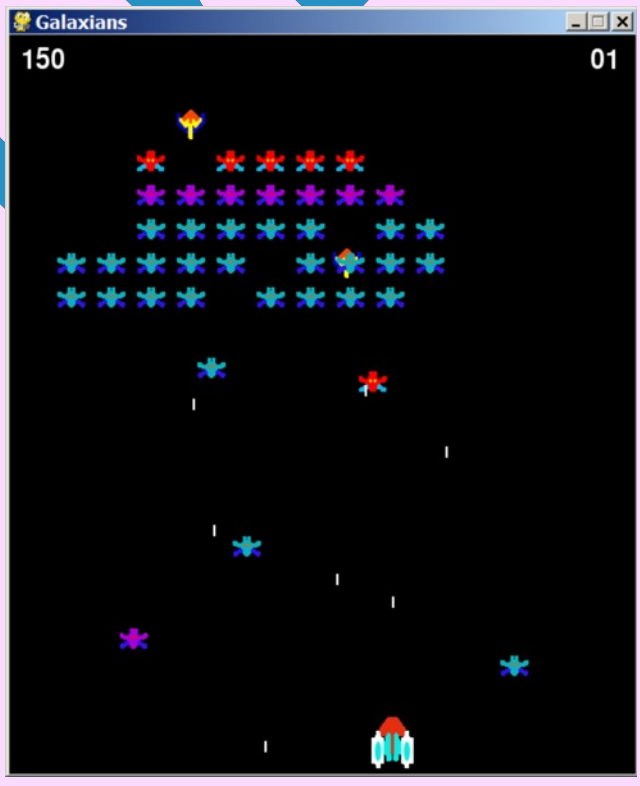
AH

Project 3 - Centipede - Preview for staff below

Project 3 will be used to review and practice the concepts learned in project 2. The amount of help given to pupils will be reduced. High scores will be read from text file, sorted and written back to empty file. Concept of linked list will be used.

Project 4 - Galaxians - Preview for staff below.

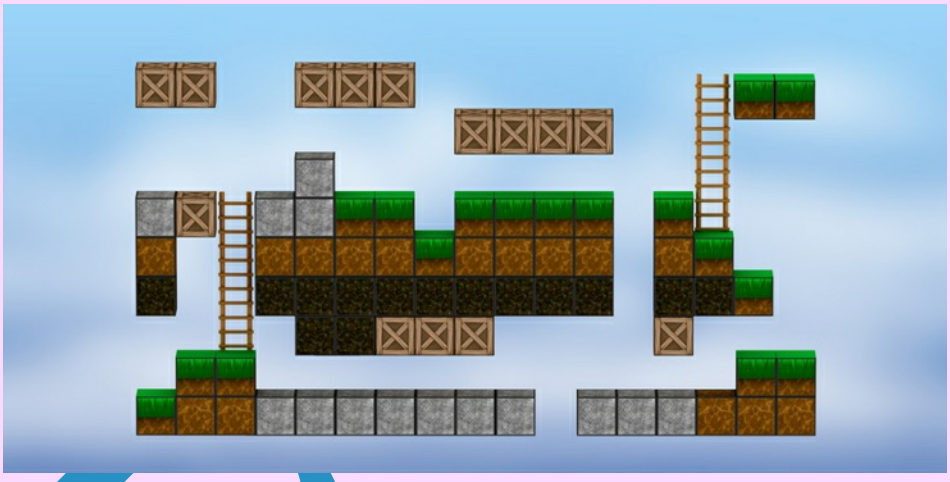
Project 4 will focus on design in preparation for coursework task. The final project will keep usernames, passwords and high scores in a database. The game will use SQL to store each player's score and provide the player with top scores and personal scores, both sorted. The game will also demand a level of problem solving not seen in the previous game. (trial version was 500 lines of code without the database linking)



AH

Object Orientated Programming Theory

When programming computer games you quickly realise that duplicates of multiple objects are a regular occurrence. Look at the screen shot below from a simple platform game design.



Even in a simple game we have to create multiple objects.

 x 14  x 17  x 15  x 14  x 14  x 2 = 76 objects

For each of the above 76 objects let's imagine that we are required to store the following simple data:

X Coordinate - real

Y Coordinate - real

Image - graphic

Visible - boolean

A **programming paradigm** is a fundamental style of computer programming, serving as a way of building the structure and elements of computer programs. In Higher computing you learned to program using an *imperative* style of programming. Imperative programming uses variables and arrays to store data in memory and then uses procedures (containing assignments, conditional statements, loops and arithmetic operations) and functions to change the state of the stored data.

If we used an imperative style of programming to write the above program we could store the required data using variables or arrays.

If simple variables were used to store the game objects above, 304 variables would have to be created, named and assigned values.

XCoord	68
YCoord	152
Image	grass.png
Visible	Yes

} x 76

We can quickly surmise that a solution involving individual variables is unmanageable and would lead to thousands of lines of unnecessary code.

AH

If arrays were used instead to store the object data then 4 arrays of 76 elements would be required. This solution, although better, also has flaws.

array index	XCoord	YCoord	Image	Visible
0	76	256	crate.png	Yes
1	99	256	crate.png	Yes
2	102	256	grass.png	Yes
3	115	256	grass.png	Yes
4	128	256	grass.png	Yes
5	154	269	crate.png	Yes
6	167	269	crate.png	No
7	180	282	stone.png	Yes
8	…	…	…	…

for each of the 76 elements (one for each object)

To examine the flaws of the imperative array solution let's imagine that some of our objects (blocks) move down slightly when they are stepped on and some other objects change to a different type of object when they are touched. New arrays would have to be created to store this additional information for the game objects.

array index	XCoord	YCoord	Image	Visible	OffsetDown	AlternativeImage
0	76	256	crate.png	Yes		smallCoin.png
1	99	256	crate.png	Yes	3	largeCoin.png
2	102	256	grass.png	Yes		
3	115	256	grass.png	Yes	6	
4	128	256	grass.png	Yes	3	
5	154	269	crate.png	Yes		star.png
6	167	269	crate.png	No		heart.png
7	180	282	stone.png	Yes	6	

From the tables it is easy to see the flaw in the array solution. If some of our objects require the additional data and some do not it is inevitable that we will create arrays that are only partially filled. This is inefficient as array elements would be created but then would never be used.

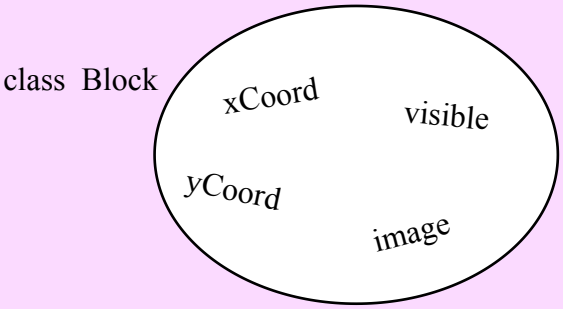
A solution to the issues above is to change our programming paradigm from imperative to object-orientated.

A program written in an object-orientated paradigm uses *classes*, *objects*, *methods* and *instances* to define and manipulate objects.

Classes

A *class* contains a set of properties (or attributes) and methods that define an object's behaviour.

We decided earlier that the Block objects in our game require the basic set of attributes visualised in the diagram to the right, the x and y coordinates, whether we can see the block or not and the background image of the block.



Constructors and Instances

Each time we create an object using a defined class we create an *instance* of the object. Python uses the function def __init__ to create *instances* of objects. For each object created the program passes in a list of parameters into the 'init' function, allowing each object to be created with different attributes. In object orientated programming a function that creates an object is known as a *constructor*.

The Python code below shows:

- a class called 'Block' being defined
- the def __init__ function receiving 4 parameters and assigning the to the attributes of the class
- three objects being created called 'grass1', 'stone1' and 'grass2'.

```
class Block:

    def __init__(self,xCoord,yCoord,image,visible):
        self.XCoordinate = xCoord
        self.YCoordinate = yCoord
        self.Image = image
        self.Visible = visible


grass1 = Block(64,123,"grass.png",True)
stone1 = Block(77,123,"stone.png",True)
grass2 = Block(64,123,"grass.png",True)
```

The use of the word 'self' in the above function is one of the key concepts of object orientated programming. In simple terms it means "for THIS object".

When the instance of the Block class, 'grass1', is created the 4 *actual* parameters passed (64,123,"grass.png",True) are then stored as attributes "for this new object" using the *formal* parameters (xCoord,yCoord,image,visible).

self.Xcoordinate = xCoord

Would be implemented as:

thisObject.Xcoordinate = 64

AH

An object-orientated style of coding suits game programming extremely well. Game objects can be created where the attributes of the objects are grouped and handled together. New objects can be created and then modified or deleted as required by the game.

Attributes

To use the stored object attributes in code we refer to the object and then the attribute, 'stone1.Image'.

Some examples of how to use object attributes are shown on the right.

```
#Assigning values to an object's attributes
grass1.XCoordinate = 74
stone1.Visible = False

#Using an object's attributes in a statement
if grass2.YCoordinate < 120:
    grass2.YCoordinate += 2

#Displaying the attributes of an object
print(grass1.Visible)
print("Current position =",grass2.XCoordinate,grass2.YCoordinate)
```

The dot **.** notation used to access the attributes of an object is another key concept of object orientated programming.

Class Attributes

Note that classes may also contain *class attributes*. These attributes are the same for each object of that class.

Class attributes can be set to an initial value when the object is created rather than requiring their values to passed as parameters.

Let's say that every block in our game is 20x26 pixels . This could be coded as follows.

```
1  class Block:
2
3      #Class attributes for Length and Height
4      Length = 20
5      Height = 26
6
7      def __init__(self,xCoord,yCoord,image,visible):
8          self.XCoordinate = xCoord
9          self.YCoordinate = yCoord
10         self.Image = image
11         self.Visible = visible
12
13
14 grass1 = Block(64,123,"grass.png",True)
15
16 print(grass1.Length)
17 print(grass1.Height)
```

```
*** Remote Interpreter Reinitialized  ***
>>>
20
26
>>>
```

Line 14 - creates a new object and assigns values to the two coordinates, image and visibility attributes.

However, we can see from the above output that the object now has additional values for length and height.

Methods

In game programming certain events are repeated over and over again. These events are dealt with in object-orientated programs by adding functions to classes to manipulate the object's behaviour. These functions may move characters, react to collisions, increment a score etc. A function attached to a class is called a *method* (or member function).
The example below shows the use of a method which moves a Block object. By passing different values into the function the object could be moved by differing amounts.

```
1  class Block:
2
3      Length = 20
4      Height = 26
5
6      def __init__(self,xCoord,yCoord,image,visible):
7          self.XCoordinate = xCoord
8          self.YCoordinate = yCoord
9          self.Image = image
10         self.Visible = visible
11
12     #A method 'move' which increments the coordinate attributes
13     #by the values passed in as parameters.
14
15     def move(self,xCoord,yCoord):
16         self.XCoordinate += xCoord
17         self.YCoordinate += yCoord
18
19 #Create an object called grass1
20 grass1 = Block(64,123,"grass.png",True)
21
22 #Call the move method for the object grass1
23 grass1.move(0,6)
```

Line 23 - The method is called using the parameters (0,6). This would increment the x coordinate by 0 and the y coordinate by 6.

Note that the move method is called for a specific object, 'grass1', using the dot notation.

The actual parameters of 0 and 6 are then passed into move method for that specific instance of the Block class. This ensures that we only change the coordinates of the 'grass1' instance.

Destructors

When an object is no longer required it can be deleted using a method known as a *destructor*. In python objects may be deleted using the del command.

```
1  class Block:
2
3      def __init__(self,xCoord,yCoord,image,visible):
4          self.XCoordinate = xCoord
5          self.YCoordinate = yCoord
6          self.Image = image
7          self.Visible = visible
8
9
10 #Create three objects called grass1, stone1 & grass2
11 grass1 = Block(64,123,"grass.png",True)
12 stone1 = Block(77,123,"stone.png",True)
13 grass2 = Block(64,123,"grass.png",True)
14
15 #Delete the object 'grass2' just created in the above line
16 del grass2
17
18 #Display the Image attribute of the three objects
19 print (grass1.Image)
20 print (stone1.Image)
21 print (grass2.Image)
```

The object 'grass2' is constructed on line 13 and then deconstructed on line 16.

When this code is executed the Image attributes for the first two objects are displayed. The third print command creates an error message saying the object doesn't exist.

```
*** Remote Interpreter Reinitialized  ***
>>>
grass.png
stone.png
Traceback (most recent call last):
  File "<string>", line 420, in run_nodebug
  File "E:\Advanced Higher (New)\SDD\Code\OOP
 Delete Object Example.py", line 28, in <module>
    print (grass2.Image)
NameError: name 'grass2' is not defined
>>>
```

AH

Summary of Object Orientated Program

Hopefully you are now gaining an understanding of the benefits of object orientated programming and its suitability to games programming.

A summary of these and other benefits of object orientated programming are listed below:

- The data structures and methods used in OOP relate to real life attributes and actions.
- Each object controls its own actions and how other objects interact with it.
- Objects can be deleted, reclaiming resources such as the memory they used.
- Code, once designed and implemented can be reused/recycled.
- The attributes of the object can be hidden, only accessible accessor methods.
- The attributes of the object can be hidden and only changed through mutator methods.
- Program maintenance is easier as objects can be updated/modified independently of other code.

Note - *Inheritance* will be discussed later in this unit.

Pygame Explained

Every program written using the pygame library should have a fairly similar structure.

Import libraries

Define classes

Initialise pygame

Set up the game window

Define additional functions and procedures

Start the main game loop

    Check for events

    Update sprites

    Redraw window

End the main game loop

You will start each project with a Pygame template file, supplied by your teacher. This file is shown and explained on the next page.

The previous structure is shown and explained in the code below:

```
# Basic Pygame Structure

import pygame                               # Imports pygame and other libraries

# Define Classes (sprites) here

pygame.init()                               # Pygame is initialised (starts running)

screen = pygame.display.set_mode([700,500]) # Set the width and height of the screen [width,height]
pygame.display.set_caption("My Game")       # Name your window
done = False                                # Loop until the user clicks the close button.
clock = pygame.time.Clock()                 # Used to manage how fast the screen updates
black    = (   0,   0,   0)                # Define some colors using rgb values.  These can be
white    = ( 255, 255, 255)                 # used throughout the game instead of using rgb values.

# Define additional Functions and Procedures here

# -------- Main Program Loop -----------
while done == False:

    for event in pygame.event.get():        # Check for an event (mouse click, key press)
        if event.type == pygame.QUIT:       # If user clicked close window
            done = True                     # Flag that we are done so we exit this loop

    # Update sprites here

    pygame.display.flip()                   # Go ahead and update the screen with what we've drawn.
    clock.tick(20)                          # Limit to 20 frames per second

pygame.quit()                               # Close the window and quit.
```

Read the program comment lines carefully as they explain each line of code. The above file will be supplied to you as a template for all your Pygame projects.

Once running the program continually repeats the code in the 'Main Program Loop'. Each time the program checks for events like a key being pressed or the mouse being clicked. It will then execute the game code, updating sprites, checking for collisions, updating scores etc.

Once every object, attribute and value has been updated, the objects on the screen are redrawn and sent (flipped) to the monitor. If this is done faster than the specified clock tick the program will pause before looping again. This effectively creates a frame rate for the game.

Remember the Frames Concept

The concept of Pygame code running as individual frames is very important. When programming in Pygame constantly keep in mind that your code is generating the next frame image (or snapshot) of your game.

Movement between frames will be relatively small. For example if you write code within the main program loop to make a character jump into the air and fall again this entire action will take place in a fraction of a second. When you run the code your character will look like they've not moved. Instead think of how far your character will move up each frame, when will they stop and how fast will they will fall again. If your whole jumping movement lasts 1.5 seconds and your game is running at 20fps the screen will have been drawn 30 times during that process.

Project 1 - Balloon Burst

AH

Project Outline

Description - The game will start in a landscape window of 800 by 400 pixels. Within the window small balloons of different colours will appear at the left hand edge. The balloons will drift right and then left across the window. The speed at which the balloons move will be matched with their colour.

Purpose - The user is required to move a dart round the screen with a mouse. They will click on the balloons at which point they will burst and disappear. The user will gain points for each balloon they burst. During the game blue balloons will appear which the user should avoid clicking on as the game will end when a blue balloon is burst.

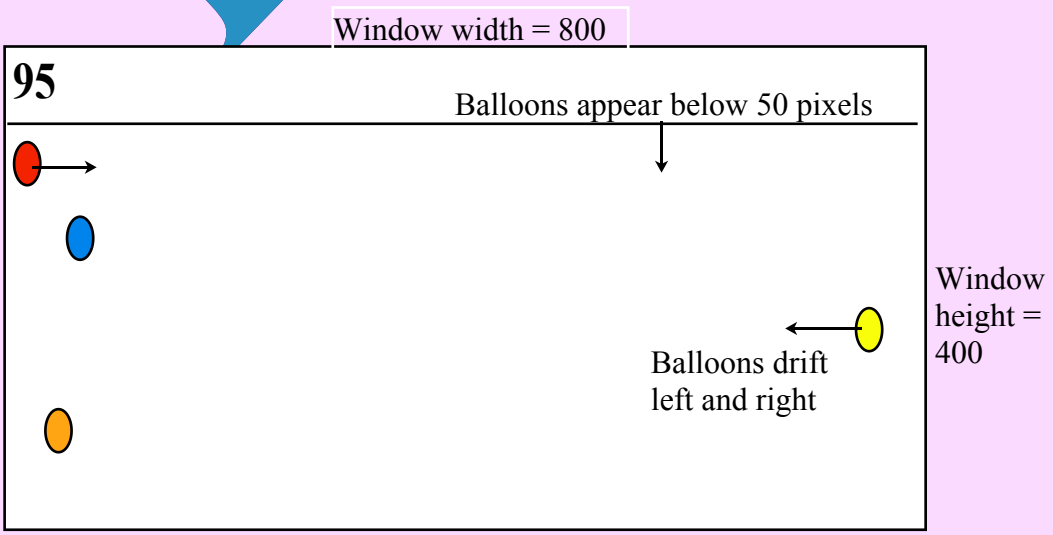
Objects, Attributes and Methods

The tables below are examples of *UML (Unified Modelling Language) class diagrams*, part of a design methodology used in object orientated programming. These note from top to bottom: class, attributes and methods.

Balloon
Integer: Xcoordinate Integer: Ycoordinate Integer: Speed String: Direction Integer: BalloonType Integer: Score Graphic: BalloonImage
Constructor (XCoordinate,YCoordinate,Direction,BalloonType) Move Balloon ( )

Dart
Integer: Xcoordinate Integer: Ycoordinate Graphic: DartImage
Constructor (XCoordinate,YCoordinate) Move Dart (mouseX, mouseY)

Screen Design



Coding Balloon Burst

The Game Window

All PyGame projects should start with setting up your game window. For Balloon Burst we shall create the window shown below with the following attributes:

- a resolution of 800 x 400
- window title - "Balloon Burst"
- a background image of the sky

Open up the PyGame template file and make the changes shown below to the window setup. Note that each time you are shown code there will be reference to the line numbers. These line numbers will not match your file exactly. Concentrate of finding the correct place to add the new code and ensure that you add every line required. Test your program thoroughly at each stage.

```
1  # Basic Pygame Structure
2
3  import pygame                                  # Imports pygame and other
4
5  # Define Classes (sprites) here
6
7  pygame.init()                                  # Pygame is initialised (s
8
9  screen = pygame.display.set_mode([800,400]) # Set the width and height
10 pygame.display.set_caption("Balloon Burst")       # Name your window
11 background_image = pygame.image.load("SkyBackground.png").convert()
12 pygame.mouse.set_visible(False)
13 done = False                                   # Loop until the user clic
14 clock = pygame.time.Clock()                    # Used to manage how fast
15 black    = (   0,   0,   0)                   # Define some colors using
16 white    = ( 255, 255, 255)                   # used throughout the game
```

- edit the window resolution (line 9)
- edit the window title (line 10)
- add code to load in the background image (line 11)
- add code to hide the mouse cursor while the game is running (line 12)
- create a new folder and save your file

AH

You've now used your first two Pygame library functions!

pygame.display.set_mode( )

This creates a display Surface to a given resolution. A Surface is a Pygame object used to represent images. Our code creates a new image object called "screen".

pygame.image.load( )

This function loads a new image from a file ("SkyBackground.png") and creates a new Surface. We have assigned this image to the Surface "background".

Note that the file name may be proceeded by a path name to the file. As ours does not, the graphic must be stored in the same folder as your saved Balloon Burst program code. Make sure the background graphic supplied by your teacher is copied into the correct folder now.

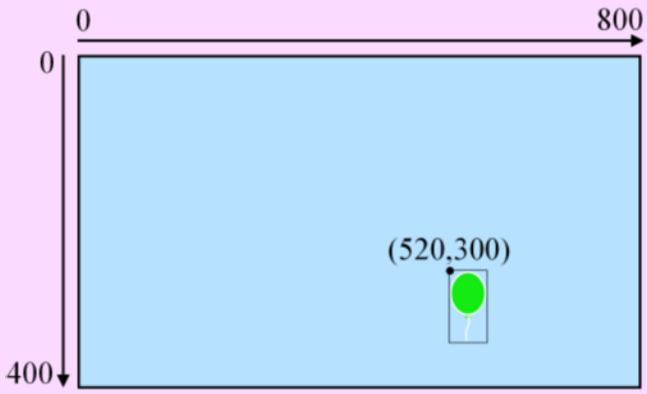
When you create your own games you will have to prepare all your own graphics. This can be done using a suitable graphic editing application and exporting the files at the resolution you require and in a suitable standard file format.

The Blit Command

The blit command is used to draw one Surface onto another. As the main program loop runs we will use this command to create a single video frame by drawing all our objects onto the screen object.

- add the blit command to draw the "background_image" Surface on top of the "screen" Surface (line 30)

```
20 # -------- Main Program Loop -----------
21 while done == False:
22
23     for event in pygame.event.get():        # Check for an ev
24         if event.type == pygame.QUIT:       # If user clicked
25             done = True                     # Flag that we ar
26
27     # Update sprites here
28
29
30     screen.blit(background_image, [0,0])
31     pygame.display.flip()                   # Go ahead and up
32     clock.tick(20)                          # Limit to 20 fra
33
34 pygame.quit()                              # Close the windo
```



The blit command contains coordinates which are used to position one image over the other.

The "screen" surface has a resolution of 800x400. The "background_image" also has a resolution of 800x400. To place one surface over another of the same size we would position it using the coordinates [0,0]. We will use this technique later to draw our balloons at specific coordinates.

Run your program to test your game window.

Creating the Balloons

Time to create our first object using object orientated programming.

To create a Balloon Class add the code below to the area of the template marked "# Define Classes (sprites) here".

```
5  # Define Classes (sprites) here
6  class Balloon(pygame.sprite.Sprite):
7
8      def __init__(self,x,y,direction,balloonType):
9          pygame.sprite.Sprite.__init__(self)
10
11         self.Direction = direction
12         self.BalloonType = balloonType
13
14         if balloonType == 1:
15             balloonImage = pygame.image.load("RedBalloon.png")
16             self.Speed = 3
17             self.Score = 5
18         if balloonType == 2:
19             balloonImage = pygame.image.load("YellowBalloon.png")
20             self.Speed = 7
21             self.Score = 15
22         if balloonType == 3:
23             balloonImage = pygame.image.load("GreenBalloon.png")
24             self.Speed = 5
25             self.Score = 10
26         if balloonType == 4:
27             balloonImage = pygame.image.load("BlueBalloon.png")
28             self.Speed = 10
29             self.Score = 0
30
31         self.image = pygame.Surface([26,50])
32         self.image.set_colorkey(black)
33         self.image.blit(balloonImage,(0,0))
34         self.rect = self.image.get_rect()
35         self.rect.x = x
36         self.rect.y = y
```

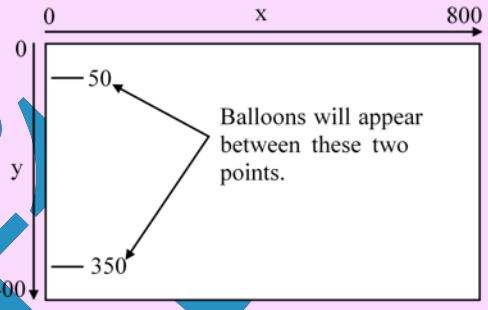
Line 6	Here we declare a new class called 'Balloon'. Note that class names always start with a capital letter. This class will inherit the attributes of a Pygame sprite (line 9).
Line 8	The constructor method def __init__ is used to create a new instance of the Balloon class. When we call this method we will pass in values for: ○ x,y - these will become the coordinates of the balloon on the "screen" Surface. ○ direction - whether the balloon is currently moving "left" or "right". ○ balloonType - this integer, with values 1-4, will be used to control the colour of balloon that is created.
Line 11	This line creates a class property called 'Direction' and assigns it the value that was passed into the init method.
Line 12	This creates a BalloonType property to store the type of balloon as an integer. The game will have 4 types of balloon - red, blue, green and yellow.
Lines 14-29	Each type of balloon is assigned a different image, speed and score.
Line 31	A new Surface is created for the balloon object.
Line 32	The Colorkey( ) method is used to ensure that the transparent areas of the balloon png image files are actually transparent.
Line 33	The chosen balloon image is copied onto the new Surface created in line 31.
Line 34	Pygame uses Rect objects to store and manipulate rectangular areas. This line creates a new Rect object from our image created in the above three lines
Line 35-36	We can now assign the properties of the Rect object (rect.x and rect.y) to the x and y coordinates that were passed into the init method. This has the effect of positioning the Rect where we want it and can be used later to move our new Balloon object.

To create instances of the Balloon class we now need to call the init method. Where we call instance methods in our code will determine how often objects are created.

By placing the call statement inside the main program loop we can continually create balloons while the game is running.

```
53 # -------- Main Program Loop -----------
54 while done == False:
55
56     for event in pygame.event.get():        # Check fo
57         if event.type == pygame.QUIT:       # If user
58             done = True                     # Flag tho
59
60     # Update sprites here
61     yCoord = random.randint(50,350)
62     balloonType = random.randint(1,4)
63     balloon = Balloon(0,yCoord,"right",balloonType)
64
65     screen.blit(background_image, [0,0])
66     pygame.display.flip()                   # Go ahead
67     clock.tick(20)                          # Limit to
68
69 pygame.quit()                               # Close th
```

```
1 # Basic Pygame Structure
2
3 import pygame
4 import random
5
```



- add lines 61 to create and store a random y coordinate between 50 and 350 for each balloon
- add line 4 as the random module must be imported before the function will work
- create a random balloon type between 1 and 4
- create a new instance called 'balloon' by calling the class Balloon. Note the actual properties we've given the new object are passed as parameters when we create the object:
  - an x coordinate of 0 to ensure every balloon starts at the left hand edge of the screen
  - a random y coordinate
  - a set direction so that every balloon will start by moving right
  - a random balloon type

No Balloons Appearing!

If you run the program you should see that it executes without crashing but no balloons appear. This is because we have not yet drawn the objects on the display surface. This must happen once during each repetition of the main program loop.

In a game there are always events that occur multiple times. In Balloon Burst this includes:

1. checking to see if the user has clicked on each balloon in turn
2. checking each balloon to see if it has touched the edge of the window and must therefore change direction
3. moving every balloon left or right
4. checking to see if the user has clicked on any blue balloon, which would end the game
5. drawing each balloon in turn on the window's surface

To simplify the handing of these events Pygame allows objects to be grouped together. These groups can then be used to handle the above multiple events, for example, drawing the balloons.

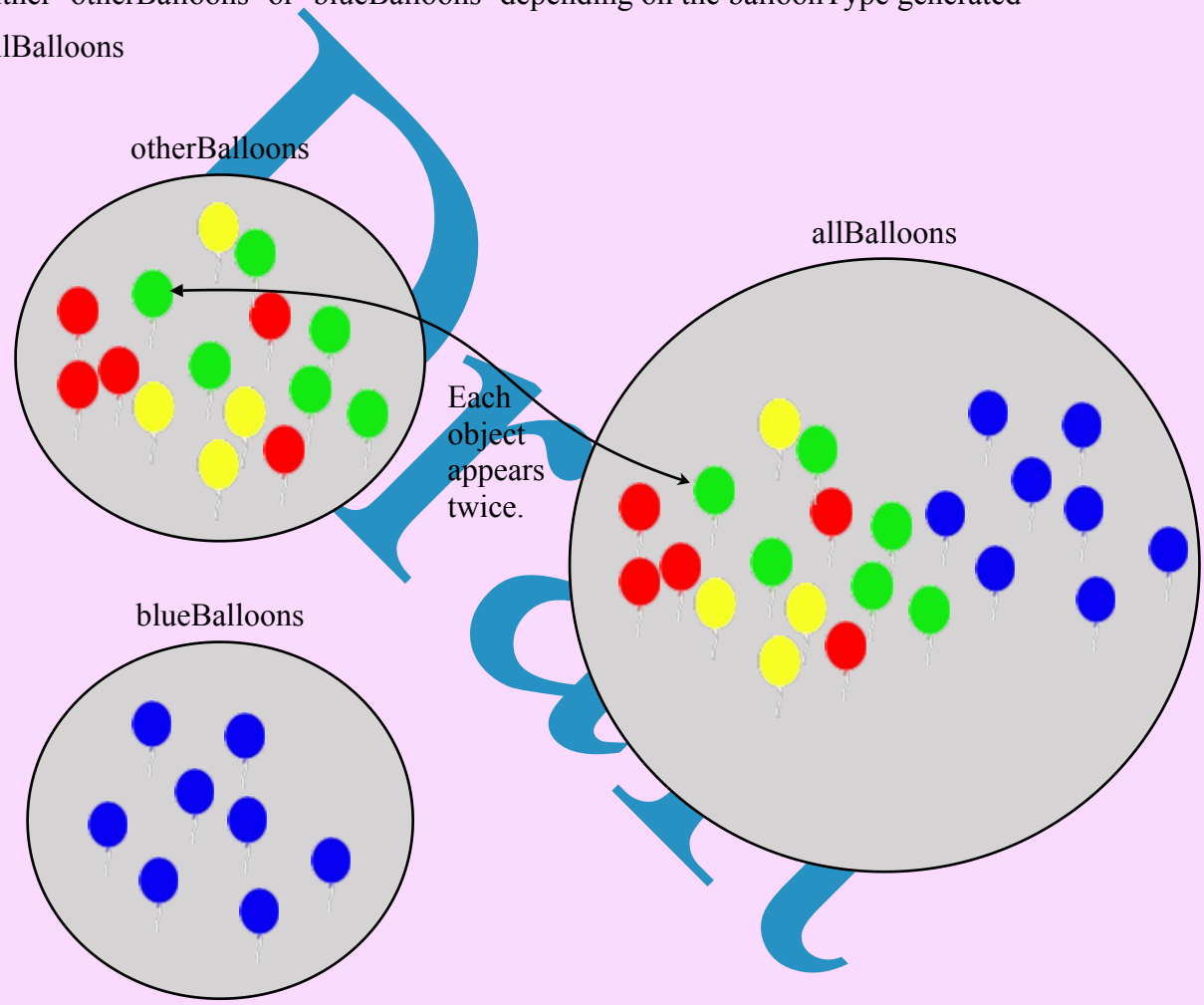
Creating Groups

Designing groups is an important part of programming in Pygame. The simplicity with which you will be able to handle your game objects will depend on the skill with which you design/create groups of objects. Objects in Pygame programs can be added/removed from groups or copied from one group to another.

In Balloon Burst we will create three groups as shown in the diagram below:

Each new instance of a balloon will be added to 2 of the above groups:

- either 'otherBalloons' or 'blueBalloons' depending on the balloonType generated
- allBalloons



Three groups will allow our program code to:

- move every balloon - 'allBalloons'
- check to see if any balloon has reached the left or right hand edge of the window - 'allBalloons'
- check to see if the user has clicked on a red, yellow or green balloon - 'otherBalloons'
- check to see if the user has ended the game by clicking a blue balloon - 'blueBalloons'
- draw all the balloons on the display surface - 'allBalloons'

The following code shows how groups can be used to draw the Balloon objects.

```
48 black     = (   0,   0,   0)                # Def
49 white     = ( 255, 255, 255)                # use
50
51 otherBalloons = pygame.sprite.Group()
52 blueBalloons = pygame.sprite.Group()
53 allBalloons = pygame.sprite.Group()
54
55 # Define additional Functions and Procedures here
56
```

- add lines 51 to 53, in the position shown, to create the three groups

Now add the following code inside the main program loop.

- By adding lines 68-71 and using a simple selection we can determine which of two groups the Balloon objects are added to.
- Adding line 72 ensures that every balloon created is added to the 'allBalloons' group.
- We can then draw every Balloon on the display surface using line 75.

```
64     # Update sprites here
65     yCoord = random.randint(50,350)
66     balloonType = random.randint(1,4)
67     balloon = Balloon(0,yCoord,"right",balloonType)
68     if balloonType >=1 and balloonType <=3:
69         otherBalloons.add(balloon)
70     else:
71         blueBalloons.add(balloon)
72     allBalloons.add(balloon)
73
74     screen.blit(background_image, [0,0])
75     allBalloons.draw(screen)
76
77     pygame.display.flip()                    # Go ahe
78     clock.tick(20)                           # Limit
```

As the game expands we will draw more objects after line 75. Note that the blit command for the background is used before we draw the balloons. If we blit the background last it would be drawn on top of all the other objects.

Stop and Test Regularly!

Run your program to check that it works. It should currently:

- Create a game window sized at 800x400 pixels.
- Display a background image in the window
- Hide the mouse cursor when it is over the window
- Create multiple balloons on the left hand edge of the window (note - the balloons do not move yet)

If your code doesn't execute any of the above, work your way back through the booklet to find what you've missed or typed incorrectly.

AH

Timing the Creation of Balloons

The last line of the main program loop ensures that the Balloon Burst is running at 20 frames per second. As instances of our Balloon objects are being created inside the main loop this means that our balloons are being created at a rate of 20 per second.

```
78      clock.tick(20)
```

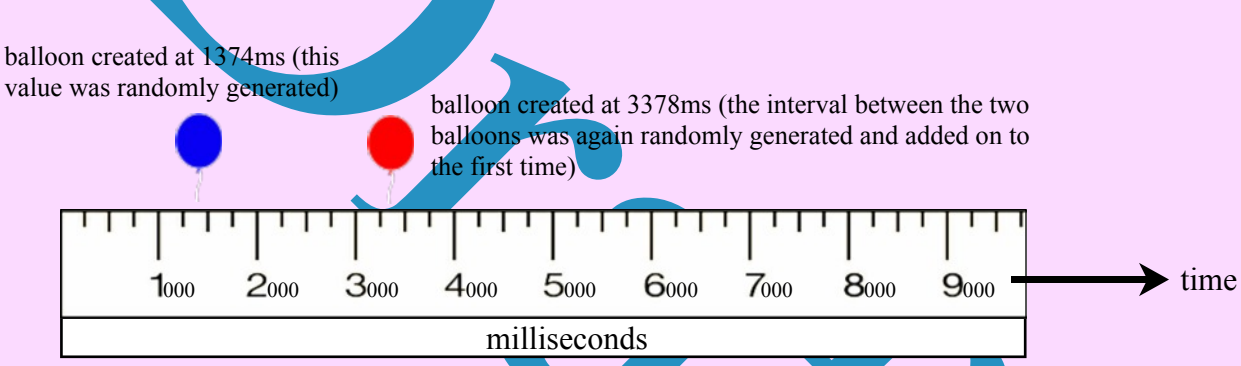
If our game is to be playable we need to slow the balloon generation down and also introduce some random element as to how fast the balloons are being created. To do this we will use the same clock used to control the frame rate.

When Pygame is initialised using the line below a clock starts ticking/counting from 0 in milliseconds.

```
40 pygame.init()
```

To create balloon objects at random intervals we require a variable to control when the balloons are released.

timeTillNextBalloon = random.randint(1000,2000)



This can be programmed by:

```
53 allBalloons = pygame.sprite.Group()
54
55 timeTillNextBalloon = random.randint(1000,2000)
56
57 # Define additional Functions and Procedures here
```

- initialising the timeTillNextBalloon variable (line 55)
- placing our balloon generation code inside a selection statement to ensure the code is only executed after the current time (pygame.time.get.tick()) is greater than timeTillNextBalloon (line 67)
- incrementing the timeTillNextBalloon variable by a random value (line 68)

```
66      # Update sprites here
67      if pygame.time.get_ticks() > timeTillNextBalloon:
68          timeTillNextBalloon += random.randint(300,2500)
69          yCoord = random.randint(50,350)
70          balloonType = random.randint(1,4)
71          balloon = Balloon(0,yCoord,"right",balloonType)
72          if balloonType >=1 and balloonType <=3:
73              otherBalloons.add(balloon)
74          else:
75              blueBalloons.add(balloon)
76          allBalloons.add(balloon)
77
78      screen.blit(background_image, [0,0])
79      allBalloons.draw(screen)
```

AH

Making the Balloons Move

To move the balloons we have to update their position on the screen, currently stored using rect.x and rect.y.

- To move the Balloon objects to the **right** we will **increment** the rect.x property of each balloon by its Speed property.
- To move the Balloon objects to the **left** we will **decrement** the rect.x property of each balloon by its Speed property.

Changing the value of multiple object's properties is usually accomplished using a method. Remember a method is a function attached to a class.

- add the moveBalloons( ) method shown below to the Balloon class
- add lines 85 and 86 to call the moveBalloons( ) method for each balloon object in the allBalloons group

```
36          self.rect.x = x
37          self.rect.y = y
38
39      def moveBalloons(self):
40
41          if self.Direction == "right":
42              self.rect.x += self.Speed
43          if self.Direction == "left":
44              self.rect.x -= self.Speed
```

```
81              blueBalloons.add(balloon)
82          allBalloons.add(balloon)
83
84      # Move each balloon in the allBalloons group
85      for balloon in (allBalloons.sprites()):
86          balloon.moveBalloons()
87
88      screen.blit(background_image, [0,0])
89      allBalloons.draw(screen)
```

When you run the program you'll find that balloons move nicely across the window but do not stop when they reach the right hand edge.

- add lines 85-89 to check the position of each Balloon object in the window. If the rect.x property is beyond the limits given, the Direction property of the Balloon will be changed.

```
84      # Check if balloon sprites have reached edge of screen
85      for balloon in (allBalloons.sprites()):
86          if balloon.rect.x < 0:
87              balloon.Direction = "right"
88          if balloon.rect.x > 774:
89              balloon.Direction = "left"
90
91      # Move each balloon in the allBalloons group
92      for balloon in (allBalloons.sprites()):
93          balloon.moveBalloons()
```

Note that the balloons' movement right (line 88) has been limited to 774 and not 800. This is because the balloon graphics are 26 pixels wide. Since rect.x and rect.y note the top, left hand edge of the graphic we have to allow for the width of the graphic in our code.

Try changing the right hand limit to 800 and observe the difference when the code runs.

The Player's Dart

To pop the balloons the user will be given a small dart  which they can move around the game window with their mouse.

To create the dart we will need another class (with its own init method) and a single instance of that class.

```
39      def moveBalloons(self):
40
41          if self.Direction == "right":
42              self.rect.x += self.Speed
43          if self.Direction == "left":
44              self.rect.x -= self.Speed
45
46  class Dart(pygame.sprite.Sprite):
47
48      def __init__(self):
49          pygame.sprite.Sprite.__init__(self)
50          dartImage = pygame.image.load("Dart.png")
51          self.image = pygame.Surface([24,19])
52          self.image.set_colorkey(black)
53          self.image.blit(dartImage,(0,0))
54          self.rect = self.image.get_rect()
55          self.rect.x = 388
56          self.rect.y = 190
57
58  pygame.init()                                    # Pygame
```

- add line 46 to declare the new class as another Pygame sprite object
- add lines 48 to 56. Note that the constructor __init__ function is much simpler than our Balloon equivalent. We are only require a single image and the x & y coordinates of the sprite.

To create a single instance of the dart object we call the __init__ function before the main program loop.

- add the lines 75 to 77 below to create an instance of the dart and add it to it's own group

```
73  timeTillNextBalloon = random.randint(1000,2000)
74
75  dart = Dart()
76  darts = pygame.sprite.Group()
77  darts.add(dart)
78
79  # Define additional Functions and Procedures here
```

Remember that to see the dart we must add a draw command at the bottom of the main program loop.

```
111     screen.blit(background_image, [0,0])
112     allBalloons.draw(screen)
113     darts.draw(screen)
```

- add line 113 to draw the dart

Again the order of the draw commands is important. By placing darts.draw(screen) after allBalloons.draw(screen) the dart will appear to be in front of the balloons.

AH

To move the dart with the mouse we have to use the events section of the main program loop.

During execution of Pygame code the program stores mouse movements, mouse clicks, keyboard presses and joystick movements in an event queue. Each time the main program loop is executed it begins by checking for events in the queue.

The initial Pygame template included lines 84 to 86 below. These lines check the event queue to see if the user has closed the window. The boolean variable done is used to end the main program loop. Leaving the main program loop would cause the program's execution to finish.

```
81 # -------- Main Program Loop -----------
82 while done == False:
83
84     for event in pygame.event.get():        # Check f
85         if event.type == pygame.QUIT:       # If user
86             done = True                     # Flag th
87
88         if event.type == pygame.MOUSEMOTION:
89             mousePosition[:] = list(event.pos)
90             dart.moveDart(mousePosition)
91
```

- add line 88 above to check the event queue for a mouse movement
- add line 89. This copies the x and y coordinates of the mouse from the event queue into an array of two values called mouseposition[ ].
- remember to add line 78 to declare the mouseposition[] array
- line 90 calls the method moveDart( ) and passes the coordinates of the mouse as a parameter.
- add the moveDart( ) method to the Dart class as shown below in lines 58 to 60.

```
77 timeTillNextBalloon = random.randint(1000,2000)
78 mousePosition = [0]*2
```

```
46 class Dart(pygame.sprite.Sprite):
47
48     def __init__(self):
49         pygame.sprite.Sprite.__init__(self)
50         dartImage = pygame.image.load("Dart.png")
51         self.image = pygame.Surface([24,19])
52         self.image.set_colorkey(black)
53         self.image.blit(dartImage,(0,0))
54         self.rect = self.image.get_rect()
55         self.rect.x = 388
56         self.rect.y = 190
57
58     def moveDart(self,mousePosition):
59         self.rect.x = mousePosition[0]
60         self.rect.y = mousePosition[1]
```

The moveDart method simply takes the coordinates of the mouse and assigns them to the rect.x and rect.y properties of the Dart object. As this method is called every time the user moves the mouse this has the effect of making the dart follow the mouse.

Bursting Balloons and Keeping Score

As discussed several times already Pygame programs reply on the use of sprites, groups and the interactions between them. We can sense when two groups collide by using the command below.

pygame.sprite.groupcollide(group1,group2,False, False)

The boolean values at the end are used to kill any sprites from either group that have collided. The ability to delete (kill) an object is one of the defining attributes of object orientated programming.

The code below senses a mouse click event and deletes any balloons that are touching the dart at that time. The code updates the score or ends the game depending on the type of balloon that was clicked.

```
97          if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
98              hitBalloons = pygame.sprite.groupcollide(blueBalloons,darts,False, False)
99              if len(hitBalloons) > 0:
100                 done = True
101             hitBalloons = pygame.sprite.groupcollide(otherBalloons,darts,False, False)
102             for balloon in (hitBalloons):
103                 score += balloon.Score
104             pygame.sprite.spritecollide(dart,allBalloons, True, collided = None)
```

- add a new MOUSEBUTTONDOWN event (line 97) below the MOUSEMOTION event. Note that event.button == 1 means that the mouse button is down (it's been clicked).
- add line 98 to create a list of all objects in the blueBalloon group that are colliding (touching) the darts group when the mouse was clicked.
- add lines 99 & 100. If the list of balloons created is longer than 0 in length a blue balloon has been hit and the game should end by setting the flag variable done to True.
- add lines 101 to 103. These lines create a list of objects that have collided between the groups darts and otherBalloons. We can loop through this list and use the Score property of each Balloon object in the list to update the game score.
- add line 104 to kill any balloons in the balloon group that are colliding with the dart sprite. When objects are killed they are deleted from every group they exist in (blueBalloons, otherBalloons and allBalloons).
- remember to declare the score variable (used in line 103), shown below in line 79

```
77 timeTillNextBalloon = random.randint(1000,2000)
78 mousePosition = [0]*2
79 score = 0
```

Displaying the Score

To display text Pygame creates an image of the text which is then blitted to the screen display at given coordinates. The code for this is shown below.

```
69 clock = pygame.time.Clock()
70 black    = (   0,   0,   0)
71 white    = ( 255, 255, 255)
72 font = pygame.font.Font(None, 36)
```

- add lines 72, 137 and 138 as shown below.

```
133     screen.blit(background_image, [0,0])
134     allBalloons.draw(screen)
135     darts.draw(screen)
136     # Add the score to the screen
137     textImg = font.render(str(score),1,white)
138     screen.blit( textImg, (10,10) )
```

AH

Adding a Pop Sound

Any good game has sound so it would be good if we added a popping sound to our game when we burst a balloon.

As with graphic files a sound is created as an object using the Pygame module library functions.

```
72 font = pygame.font.Font(None, 36)
73
74 popSound = pygame.mixer.Sound("pop.wav")
75
76 otherBalloons = pygame.sprite.Group()
```

- add line 74. You will need to have the sound stored in the same folder as your Python file.
- add line 108 below. The positioning of the line the plays the sound is important. Using the loop below (line 106) makes sense as here the code loops through a list of Balloons objects that have been hit. Adding a sound here will play the pop sound for each object in the hitBalloons list.

```
101         if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
102             hitBalloons = pygame.sprite.groupcollide(blueBalloons,darts,False, False)
103             if len(hitBalloons) > 0:
104                 done = True
105             hitBalloons = pygame.sprite.groupcollide(otherBalloons,darts,False, False)
106             for balloon in (hitBalloons):
107                 score += balloon.Score
108                 popSound.play()
109             pygame.sprite.spritecollide(dart,allBalloons, True, collided = None)
```

That's project 1 finished. Make sure you test the game thoroughly.

Note that the game may run slowly with some lagging if you run it through an Interpreter in your IDE. Try double clicking on the python file in its folder and you should find it runs more smoothly.

Balloon Burst Challenges

Try implementing the following on your own. The suggestions below are listed in increasing order of difficulty.

1. When creating an instance of a balloon make the Balloon objects appear at a random speed.
2. Add a few bonus objects that exhibit the same behaviour as the balloons. These should move faster but have a higher score associated with them.
3. Add a special gold balloon that will kill all the blue balloons when clicked.
4. Make the balloons drift slightly up and down as they move across the game window.
5. Add text to each balloon so that each balloon displays its value in the middle.
6. Add a 'game over' screen to display the final score for 5 seconds.
7. Add an initial screen that allows the user to set the difficulty: easy, medium or hard. The users choice should determine the speed of the balloons and how often they appear.

My First Pygame

Following instructions and explanations like those in the previous pages is a good way to learn but as a programmer you must be able to work and problem solve independently.

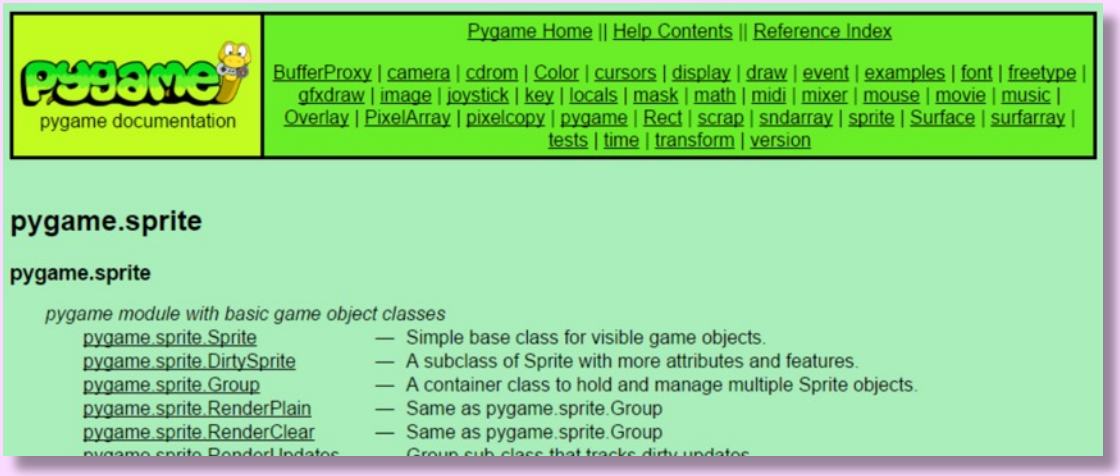
After each of the four projects there will be an expectation that you create a similar program of your own that uses the concepts you have learned in the previous project. Pygame concepts covered up to this point have included:

- understanding the structure of a Pygame program
- creating a Class (including sprites)
- using the constructor method __init__
- creating an instance of an object
- adding images to Surfaces
- bliting one Surface onto another Surface
- moving objects using the sprite rect.x and rect.y properties
- using the clock to control events
- events - sensing mouse movement and mouse clicks
- creating lists of objects that have collided
- killing objects following collisions
- bliting text to the screen surface
- using the done variable to end the game

Now is your chance to create a game of your own. In later projects we will discuss a more formal approach to design and planning. For now a suggested order of stages to follow is detailed below.

1. On paper draw out a basic design for a game.
2. Create a UML class diagram to note the classes, attributes and methods for each object in your game.
3. Devise a plan of attack by listing the order in which you are going to implement the different parts of the game.
4. Write your code.
5. Test your game yourself.
6. Ask others to verbally evaluate your game for: ease of use, playability, overall look and enjoyment.

For reference, make use of the Documentation section of the Pygame website. Don't be afraid to teach yourself in addition to using what you learned in project 1.



AH

Designing Better Classes (Mistakes made in Balloon Burst)

They say that if you ask 20 programmers to solve the same problem, you'll end up with 20 different programs. So what makes one program better than another?

As this was your introduction to Pygame programming (and maybe your first experience of object orientated programming) the code in balloon burst was simplified to the point that it would be described by an expert as "poorly structured".

To explain…

```
class Balloon(pygame.sprite.Sprite):

    def __init__(self,x,y,direction,balloonType):
        pygame.sprite.Sprite.__init__(self)

        self.Direction = direction
        self.BalloonType = balloonType

        if balloonType == 1:
            balloonImage = pygame.image.load("RedBalloon.png")
            self.Speed = 3
            self.Score = 5
        if balloonType == 2:
            balloonImage = pygame.image.load("YellowBalloon.png")
            self.Speed = 7
            self.Score = 15
        if balloonType == 3:
            balloonImage = pygame.image.load("GreenBalloon.png")
            self.Speed = 5
            self.Score = 10
        if balloonType == 4:
            balloonImage = pygame.image.load("BlueBalloon.png")
            self.Speed = 10
            self.Score = 0

        self.image = pygame.Surface([26,50])
        self.image.set_colorkey(black)
        self.image.blit(balloonImage,(0,0))
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y
```

Discussion Point 1

One important rule of object orientated programming, broken in Balloon Burst, is that "properties and behaviours should remain separate within Classes".

In the constructor method, code has been added to create properties for 4 different types of balloon. By doing this we are modifying the behaviour of the balloons (setting speed, direction and position) at the same time as we are creating properties for the new object.

This issue may be fixed by coding the constructor, as a method that simply creates an object with default values.

No parameters are passed into the constructor as we are creating an instance of a Balloon object without any information on how it is to behave.

```
class Balloon(pygame.sprite.Sprite):

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)

        self.image = pygame.Surface([26,50])
        self.rect = self.image.get_rect()
        self.rect.x = 0
        self.rect.y = 0
        self.direction = ""
        self.type= 0
        self.speed = 0
        self.score = 0
```

The object can then be given values for position, speed, score and direction within a separate method. Parameters are passed into this second method.

It is only at this point that the behaviour of the object is now defined.

This new method could be called again in future if we wished to change the behaviour of a Balloon object.

```
def setBalloon(self,x,y,d,t):
    values = {
        1:{'img':'RedBalloon.png','speed':3,'score':5,'direction':'right'},
        2:{'img':'YellowBalloon.png','speed':7,'score':15,'direction':'right'},
        3:{'img':'GreenBalloon.png','speed':5,'score':10,'direction':'right'},
        4:{'img':'BlueBalloon.png','speed':10,'score':0,'direction':'right'}
    }
    self.rect.x = x
    self.rect.y = y
    self.type= t
    self.direction = d

    attrs = values[self.type]

    self.score = attrs["score"]
    self.speed= attrs["speed"]

    balloonImage = pygame.image.load(attrs["img"])
    self.image.set_colorkey(black)
    self.image.blit(balloonImage,(0,0))
```

The two methods would be called as shown below.

```
balloon = Balloon()
balloon.setBalloon(0,yCoord,"right",balloonType)
```

Discussion Point 2

The lack of efficient organisation of the attributes assigned to each type of balloon is also an issue within program code. There is in fact no organisation of this data at all as the values are simply assigned within the constructor class.

```
if balloonType == 1:
    balloonImage = pygame.image.load("RedBalloon.png")
    self.Speed = 3
    self.Score = 5
if balloonType == 2:
    balloonImage = pygame.image.load("YellowBalloon.png")
    self.Speed = 7
    self.Score = 15
if balloonType == 3:
    balloonImage = pygame.image.load("GreenBalloon.png")
    self.Speed = 5
    self.Score = 10
if balloonType == 4:
    balloonImage = pygame.image.load("BlueBalloon.png")
    self.Speed = 10
    self.Score = 0
```

If more balloon types were added to the game new code (in the form of selection statements) would have to be written. This is poor programming as more balloons should simply lead to more stored data and not an increase in programming constructs.

We could use arrays to store the balloon values for speed, image, score and direction but, as we learned very early on in this unit, that leads to values for a single object being stored in multiple data structures.

A better solution is to use a *record structure*. This structure stores data in organised records, each of which can contain multiple data types. A python record structure to store our balloon values is shown below.

```
values = {
    1:{'img':'RedBalloon.png','speed':3,'score':5,'direction':'right'},
    2:{'img':'YellowBalloon.png','speed':7,'score':15,'direction':'right'},
    3:{'img':'GreenBalloon.png','speed':5,'score':10,'direction':'right'},
    4:{'img':'BlueBalloon.png','speed':10,'score':0,'direction':'right'}
}
```

The name 'record' structure is no coincidence as the data structure bears similarities to a database with its tables, records and fields.

The data is access by copying a record from the table.
For a balloon type of 2, attrs would now store:
'img':'YellowBalloon.png','speed':7,'score':15,'direction':'right'

```
attrs = values[self.type]
```

The different values in the record are then accessed as shown on the right.

```
self.score = attrs["score"]
self.speed= attrs["speed"]
```

If an additional type of Balloon object is required we now simply add another line to the record structure without altering any other code.

```
5:{'img':'GoldBalloon.png','speed':25,'score':1000,'direction':'right'}
```

Discussion Point 3

A third improvement that could be made to Balloon Burst is increased used of use of methods.

An example of this is where the game checks each balloon to see if it has reached the edge of the game window. As this section of code changes the behaviour of an object (direction is changed between 'right' and 'left' as required) this should be handled by a method.

```
# Check if balloon sprites have reached edge of screen
for balloon in (allBalloons.sprites()):
    if balloon.rect.x < 0:
        balloon.Direction = "right"
    if balloon.rect.x > 774:
        balloon.Direction = "left"
```

becomes

```
# Check if balloon sprites have reached
for balloon in (allBalloons.sprites()):
    balloon.checkForEdge()
```

and

```
def checkForEdge(self):
    if self.rect.x < 0:
        self.direction = "right"
    if self.rect.x > 774:
        self.direction = "left"
```

The additional method within the Balloon Class now ensures that the direction properties of the Balloon objects can only be altered within the class.

Discussion Point 4

Within the main program loop the code currently loops through each balloon object twice. While teaching the purpose of each part of the Balloon Burst code in turn it made sense to do this.

```
# Check if balloon sprites have reached edge of screen
for balloon in (allBalloons.sprites()):
    balloon.checkForEdge()

# Move each balloon in the allBalloons group
for balloon in (allBalloons.sprites()):
    balloon.moveBalloon()
```

Now that the code is complete this stands out as a glaring inefficiency. The above code can be rewritten as a single loop in which each balloon's position is checked (checkForEdge( ))and then updated (moveBalloon ( )).

```
# Check if balloon near at edge and update position
for balloon in (allBalloons.sprites()):
    balloon.checkForEdge()
    balloon.moveBalloon()
```

Booklet 2

In booklet 2 we will look at inheritance, encapsulation, sprite animation and file handing.