10.014 Computational Thinking for Design 1D Project - Term 1 2023 (SUTDent Ultimate Test Domain)

Cohort Number/Team: F01/ Team 1B

Game Description:

1. Background

The software's origins began from a project team member's daily coffee intake, which was necessary to cope with the demands of university life, especially with SUTD being known for the culture of 'Staying Up to Dawn'. Upon looking into the other reasons why students stay up till dawn, we realised that stress management is important to prevent SUTDents from staying up till dawn. These observations were the foundation for the software's inspiration, which eventually became a solution for stress among SUTDents. As such, our game name 'SUTDent Ultimate Test Domain' is in response to address this very problem we find interesting in SUTD.

2. Scenario and rationale behind our game:

This game design is focused on freshmores in general in tackling stress management as they adjust to university life, which is the reason behind the context in the game surrounding the expectations in Term 1 of SUTD Curriculum. After SUTDents have played the game, they will have a review of their performance based on what power-ups they have collected which would affect their health, and management of studies, and the main variable in question is stress. We hope that through this fun game, students will be able to balance their stress levels while maintaining good physical and mental health, and good study management.

3. How the game is to be played:

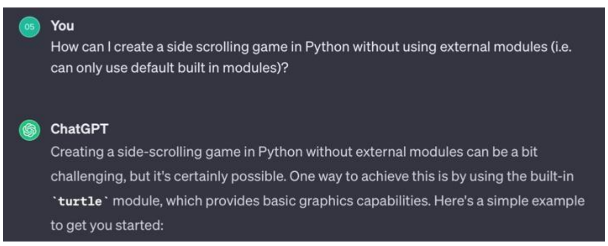
The game is a single-player side scroller with the stress, health, and grades bars shown until the end of the game. The game is meant to mimic the player's real-life decisions based on our Term 1. The player will control an icon on the screen and the background will move to the left in a loop. The powerups will positively or negatively impact the health, stress, and grades of the player's character. The result will only be revealed at the end of the game, which will help the player evaluate his health and stress levels and provide advice accordingly.

4. Main features in the game:

The main features of the software include the picturesque custom-made background as well as the advice that the user can get at the end of the game.

5. Citations/References to sources used when planning the game:

As none of us knew if it was possible to create a game in Python without external libraries, we used ChatGPT to test the feasibility of the project.



```python
import turtle
import random

# Set up the screen
screen = turtle.Screen()
screen.title("Side-Scrolling Game")
screen.bgcolor("white")
screen.setup(width=800, height=600)

# Create the player
player = turtle.Turtle()
player.shape("square")
player.color("blue")
player.shapesize(stretch_wid=1, stretch_len=2)
player.penup()
player.goto(-300, 0)

# Set up the player speed
player_speed = 15

# Create a list to store the obstacles
obstacles = []

# Set up the obstacle speed
obstacle_speed = 5

# Function to move the player up
def move_up():
    y = player.ycor()
    y += player_speed
    player.sety(y)

# Function to move the player down
def move_down():
    y = player.ycor()
    y -= player_speed
    player.sety(y)
```

```python
# Main game loop
while True:
    # Move the obstacles
    for obstacle in obstacles:
        x = obstacle.xcor()
        x -= obstacle_speed
        obstacle.setx(x)

        # Check for collision with the player
        if player.distance(obstacle) < 20:
            player.color("red")
            player.write("Game Over", align="center", font=("Courier", 24, "normal"))
            screen.update()
            screen.bye()

    # Create a new obstacle at random intervals
    if random.randint(1, 10) == 1:
        obstacle = turtle.Turtle()
        obstacle.shape("square")
        obstacle.color("red")
        obstacle.shapesize(stretch_wid=1, stretch_len=2)
        obstacle.penup()
        obstacle.goto(400, random.randint(-280, 280))
        obstacles.append(obstacle)

    # Update the screen
    screen.update()
```

We found out that it seemed to be possible to do so, hence we continued with the idea of making a 2D side scrolling game. ChatGPT's code was very barebone and had quite a few bugs. We ended up referring to the documentation of the Turtle module and built our game from scratch.

We also used ChatGPT sparingly to debug and obtain more complicated code required for our game.

You
is there a way to use python subprocessing to use powershell to play a wave file without opening any application?

ChatGPT
Yes, you can use Python's `subprocess` module to execute PowerShell commands to play a WAV file without opening any application. Here's an example:

```python
import subprocess

wav_file_path = r'C:\path\to\your\file.wav'

powershell_command = f'[System.Media.SoundPlayer]::new("{wav_file_path}

subprocess.run(['powershell', '-Command', powershell_command])
```

Replace `C:\path\to\your\file.wav` with the path to your WAV file.

This Python script constructs a PowerShell command that uses the `System.Media.SoundPlayer` class to play the specified WAV file. Then, it uses `subprocess.run()` to execute this PowerShell command within Python.

The `PlaySync()` method used in this example will play the sound synchronously, meaning it will wait until the sound finishes playing before continuing with the script. If you want it to play asynchronously (while the script continues), you can use `Play()` instead of `PlaySync()`.

Documentation of our code:

1. Overview of our code

Our approach for coding the game was to use Object-Oriented Programming (OOP). OOP is a programming paradigm we define object classes in our code, with each class having its own set of attributes and procedures. The following table shows the classes we defined in our game code and the role of the class:

Class Name	**Purpose of the Class**
GameInitialisation	Contains and initialises all game and game window attributes. Also contains all common variables that may be changed.
TitleScreen[1]	Handles the graphics and logic of the title screen and character select screen
GameController[1]	Handles the graphics and logic of the main gameplay
EndScreen[1]	Handles the graphics and logic of the end screen
Sprite	Contains all basic attributes and methods related to sprites (moving images) in the game. Serves as the backbone for the implementation of `Background`, `Player` and `Item` classes
Background[2]	Defines attributes and behaviour of the game's background image
Player[2]	Defines attributes and behaviour of the player character in the game
Item[2]	Defines attributes and behaviour of items in the game, such as obstacles and power-ups
Display	A 'command' object used by `GameController` to update the game display
Delay	A 'command' object used by `GameController` to standardise game execution speed between different computers ensures game is run at relatively constant speed

[1] Inherits from `GameInitialisation` to access game and graphics related attributes / methods
[2] Inherits from `Sprite` to access basic sprite related attributes / methods

2. Importance of using OOP in our program

- Using OOP is important for creating large programs, like the game we created. It allows us make the code modular and have each class handle a crucial aspect of our program, as shown in the table above.
- OOP also promotes code reusability as classes with similar functionality can inherit attributes and methods from a superclass, such as how `Background`, `Player` and `Item` inherits from `Sprite`.
- Finally, OOP allows for code generalisation by changing how methods work in different subclasses while keeping the same name of the method. This is known as polymorphism. For example, even though `Player` and `Item` has different logic for moving in the game, we are able to keep the same name for the method `move`, which aids in code readability as well.

3. Modules used in our program

The following built-in modules were used in our game:

Module Name	How the module is used in our code
turtle	Used to implement game user interface and graphics
queue	Used to implement the running of commands in the game
random	Used in functions that require randomness (e.g. to randomise the speed of objects in the game)
csv	Used to read game data from .csv files
time	Used to control execution speed of the game
winsound	Used to play game music

4. Key components in our program

As we have a lot of code in our program, we will focus on showcasing the most crucial aspects of our code that allows the game to work the way it is.

a. Graphics Implementation

We used Python's Turtle module to implement the graphics for our program.

```
def initialise_screen(self):
    self._screen = turtle.Screen()
    turtle.tracer(0, 0)
    self._screen.title(self._title)
    self._screen.setup(width=self._screen_width, height=self._screen_height)
```

In the method `initialise_screen` in `GameInitialisaion`, we use `turtle.Screen()` to set up a game window and defined the size of the window by using the `setup` method.

```
class Sprite:
    def __init_(self, controller):
        self._controller = controller
        self._alive = True

        # Turtle initialisations
        self._obj = turtle.Turtle()
        self._obj.speed(0)
        self._obj.penup()
```

To implement sprites in our game we used the `turtle.Turtle()` object.

```
# Turtle initialisations
self._obj.shape(f"{self._controller._player_sprite}0.gif")
self._obj.goto(self._controller._player_start_xcor, 0)
```

For each sprite in the game, we created one instance of `turtle.Turtle()` and used methods like `shape`, `goto` to modify attributes of the object, such as the image used for the sprite and it's location on the screen.

b. The GameInitialisation Class

We have grouped all variables that may be changed, item spawn rates, player padding, etc. in the `__init` method of the `GameInitialisation` for ease of access since the class definition is located at the top of the code file.

In addition, it contains the initialise_screen() method (previously mentioned) as well as the `register_sprite_images()` and `register_other_images()` which serve to register gif files to Turtle and import item-related data saved as a dictionary.

```python
def register_sprite_images(self):
    filename_list = [self._normal_items_stats, self._stress_items_stats, self._bonus_items_stats]
    dict_list = [self._normal_item_dict, self._stress_item_dict, self._bonus_item_dict]

    for filename, item_dict in zip(filename_list, dict_list):
        f = open(filename, 'r')
        csv_reader = list(csv.reader(f, delimiter=','))
        for row in csv_reader[1:]:
            turtle.register_shape(f"{row[0]}.gif")
            item_dict[row[0]] = [int(stat) for stat in row[1:]]
        f.close()
```

Included in the zip folder is a list of 3 CSV files that contain the names, x_padding, y_padding, speed across the x-axis, and the stress, health, and grades values each game item has.

Using the built-in csv reader module, for every file, the values corresponding to one game item is consolidated into one list. These lists will form a 'list of lists' of game item characteristics (csv_reader). The lists in csv_reader would then form a dictionary key-value pair; where the key would be the name of the object, and the value is the list of the remaining value characteristics the game item has.

The `register_other_images()` works similarly by reading a txt file containing the filename of the non-item gifs to register to Turtle.

c. Game Logic Implementation

To understand the gameplay logic, we must first introduce you to the `execute()` methods. In addition to methods like getters, setters, and other functions and procedures, most classes have its own a method called `execute()` which is called when operations relating to the class needs to be performed. The following table summarises what the `execute()` methods does.

Class Name	**Purpose of the execute() method**
GameInitialisation	Initialises the turtle screen and registers the gif files used in turtle.
TitleScreen	Displays the title screen background and switch to character selection after “space” is pressed.
GameController	Initialisation Phase: Adds initial objects to the queue, creates turtles for UI purposes, starts listening for keyboard input to control the player Game Loop Phase: Gets and calls the execute method of the next object in queue, updates the player animation, switches event stage based number of seconds passed, checks for player collision and updates the stats accordingly.
Sprite	No execute() method
Background	Moves the background to the left and teleports it back to the right once fully out of screen. Returns True.
Player	Updates the player’s speed based on what keys are pressed and moves the player accordingly (restricted to within the screen). Returns True.
Item	Checks if the items is out of the screen and returns False, otherwise moves the item to the left and returns True
Display	Updates the display of the turtle screen. Returns True
Delay	Sleeps for the remaining duration of frame/queue iteration. This is to standardise the time taken for each queue iteration. Returns True
EndScreen	Gets the results description and displays it.

There needs to be a way to run all of the different `execute()` methods for the different objects. We could maintain a list of objects that are currently being utilised for the game and repeatedly iterate through each one, calling its execute method to run it. However, if the object is no longer needed, we would then need to remove it from the list while still iterating through the list, which may cause issues.

To solve this issue, we utilised a queue. A queue is a data structure that allows us to perform operations in a First In First Out order. Items can be added to the queue (enqueued), and await being executed. One by one, items can them be removed from the front of the queue (dequeued), executed, and then if needed, be enqueued for execution again later on.

This gameplay execution logic is handled by the `GameController` class by utilising a queue of objects such as the player, background, and other items.

The following is an example of what our queue typically contains sometime during the gameplay.

```
Display, Delay, Background1, Background2, Player, Item1, Item2, Item3
```

During this iteration of the Game Loop Phase of the `GameController`, the `GameController` gets the next object, in this case, it is the `Display` object, and runs the object’s execute method, the result of which will be the turtle screen being updated. If the execute method returns `True`, the object is added back into the queue and the queue now looks like this.

Delay, Background1, Background2, Player, Item1, Item2, Item3, Display

If the `execute()` method returns `False` (only possible if the object is of the `Item` class), the `Item` will forever be removed from the queue and the `kill()` method is called to free-up memory.

d. Movement of Sprites

Movement of the various sprites is implemented using Turtle's `setx()` and `sety()` method. The logic for the object movement is encapsulated within its own `move()` method. Below is a snippet for the `move()` method of `Item` class.

```
# Functions / Procedures
def move(self):
    # Shift obstacle x coordinate by speed units
    self._obj.setx(self._obj.xcor() - self._speed)
```

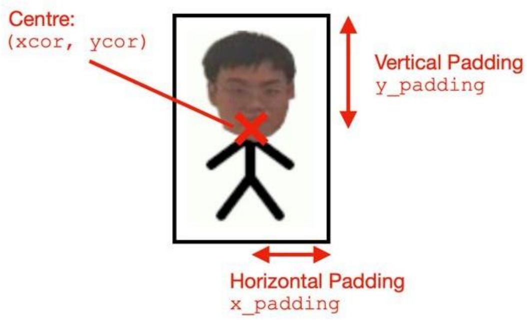
By setting the new x or y coordinate as the current one minus the object's speed attribute, the object is essentially teleported to the left by speed units. Using a combination of small values () for speed and a high frame rate (around 120 frames per second), the rapid teleportation creates the illusion of the object moving smoothly across the screen.

Certain classes, like the `Background` and `Player` class, have added constraints/conditions in their `move()` method.

e. Collision Detection Algorithm

Collision detection algorithms are essential in a 2D game. A good collision detection algorithm be accurate in detecting collision between objects and be fast to execute. Our game requires checking if the `Player` collides with any Item.

We decided to implement collision in our game by using bounding box method. In this method, a rectangle is defined around each sprite of the object.



Each `Sprite` contains attributes that define its current coordinate of the sprite's centre, the horizontal padding and vertical padding.

```python
def check_collision(self, object1, object2):
    # Get the range of x, y coordinates of the two objects
    obj_1_x_range = range(int(object1.get_xcor() - object1.get_x_padding()), int(object1.get_xcor() + object1.get_x_padding() + 1))
    obj_2_x_range = range(int(object2.get_xcor() - object2.get_x_padding()), int(object2.get_xcor() + object2.get_x_padding() + 1))
    obj_1_y_range = range(int(object1.get_ycor() - object1.get_y_padding()), int(object1.get_ycor() + object1.get_y_padding() + 1))
    obj_2_y_range = range(int(object2.get_ycor() - object2.get_y_padding()), int(object2.get_ycor() + object2.get_y_padding() + 1))
    x_range_same = None
    y_range_same = None

    # Check if any of the x, y coordinate overlap
    for i in obj_1_x_range:
        if i in obj_2_x_range:
            x_range_same = True
            break
    for j in obj_1_y_range:
        if j in obj_2_y_range:
            y_range_same = True
            break

    # If the x and y coordinate overlaps the object collides
    return x_range_same and y_range_same
```

Given the coordinates, `x_padding` and `y_padding` values, the range of coordinates the object covers at any point in the game can be mapped into 2 lists: based on the x-coordinates and y-coordinates. The ranges for both player and other game objects are obtained.

As the player is the constant game icon within the game, the values in the range of x-coordinates and y-coordinates are used to check if they overlap with the respective ranges of values of x-coordinates and y-coordinates for the other game items. If there is an overlap between the ranges of the respective coordinates for both player and game item for both the x-coordinates and y-coordinates, there is a collision between the player and the game object.

f. The Background Class

To create the effect of an endlessly scrolling background, we created a `Background` class and create two `Background` object instances in the Initialisation Phase of the `GameController`. At the beginning, the first background object is placed fully within the window and the second is placed out of screen to the right of the window.

Once the Game Loop Phase starts and the `execute()` method is called, the `Background` object is moved to the left. However, a condition is added such that when the `Background` object is out of screen to the left of the window, it gets placed back to the right of the window to create a seamless scrolling animation.

g. The Player Class

In regards to movement, the player's position is being constrained to within the track area of the game background. For each of the directions the player can move (top, bottom, left, right), if statements are used to check whether the player's padding coordinate is within the constrains mentioned (for the top boundary, the maximum limit is where his feet are on the track surface), and if not, the player is placed back to the edge of the constrained area.

Unlike the `Background` and `Item` classes whose movement is restricted leftwards, the player is able move in multiple directions by listening for key presses. Turtle's `onkeypress()` and `onkeyrelease()` methods are threaded, so when all these methods are called during the `GameController`'s Initialisation Phase, Turtle continues to listen for key presses and releases.

```python
def listen_for_keypress(self):
    # Listen for keypress/release; used to move the player
    self._screen.onkeypress(self._player.up_pressed, "Up")
    self._screen.onkeypress(self._player.down_pressed, "Down")
    self._screen.onkeypress(self._player.right_pressed, "Right")
    self._screen.onkeypress(self._player.left_pressed, "Left")
    self._screen.onkeyrelease(self._player.up_released, "Up")
    self._screen.onkeyrelease(self._player.down_released, "Down")
    self._screen.onkeyrelease(self._player.right_released, "Right")
    self._screen.onkeyrelease(self._player.left_released, "Left")
    self._screen.listen()
```

The `player.up_pressed` and `player.up_released` functions sets a Boolean attribute of the `Player` class to `True` or `False` to keep track whether or not that respective key is being pressed. These Boolean attributes are used in the following method which updates the players speed.

```python
def update_speed(self):
    """ Updates the speed based on what key is pressed. Allows for multiple keys to be pressed at the same time """
    # Resets speed to 0
    self._speed_x = 0
    self._speed_y = 0

    # Set speed according to what keys are pressed
    if self._up:
        self._speed_y += self._speed
    if self._down:
        self._speed_y -= self._speed
    if self._right:
        self._speed_x += self._speed
    if self._left:
        self._speed_x -= self._speed
```

This `update_speed()` method does the following.

Say the player's speed is set to 5 and the up key is pressed. The method updates the value of `self._speed_y` to 5.

If both the up and down key are pressed, `self._speed_y` will be added by 5 then subtracted by 5 which results it to remain as 0 (therefore no vertical movement as one would expect)

If both the up and right key are pressed, then `self._speed_y` and `self._speed_x` will both be 5 resulting in diagonal movement in the up-right direction.

Without the `self._up` Boolean attribute and the `update_speed()` method, if a method like `move_up()` (for example) is called directly after the up key is pressed, the player will only be able to move in one direction at a time without any diagonal movement.

h. The Item Class

In addition to the `move()` method mentioned above, the Item class also has an `is_out()` method to check whether it is out of the screen and a `kill()` method to free-up memory if the item is indeed no longer in the screen.

i. The Delay Class

As the time it takes for one queue iteration may vary depending on the number of items in the queue and the processor speed during that iteration, we have created a `Delay` object whose purpose is to pause the program execution for the remaining duration of the intended duration for one queue iteration.

```
class Delay:
    """ Normalises execution speed of game across different devices and ensures game is run at relatively constant speed """
    def __init__(self, controller):
        self._controller = controller
        self._prev_time = time.time()

    def execute(self):
        # Sleeps for remaining duration of frame
        # If one frame is set to 10 seconds and all other operations are completed in 7 seconds, delay sleeps for 3 seconds
        now = time.time()
        sleep_time = 1/self._controller.get_game_fps() - (now - self._prev_time)

        if sleep_time > 0:
            time.sleep(sleep_time)
            self._prev_time = now + sleep_time
        else:
            self._prev_time = now
        return True
```

j. The GameController Class

The `GameController` class contains all functions pertaining the game logic (collision detection, spawning items, updating the player's stats values) and the flow of the game (event stages).

After the Initialisation Phase (adding objects to the queue, creating Turtles for UI purposes, start listening for key presses), the `GameController execute()` method enters the Game Loop Phase which uses a while loop. The following is an oversimplified pseudocode for the while loop.

1. Get the next object from the queue

2. If object is of type `Display`, continue to 3 else jump to 7 # This `Display` object is made to appear only once every queue iteration.

3. Show ending screen if game has ended any no more Item objects left in queue

4. Update the player animation

5. Check the number of seconds passed and switch game stages accordingly

6. Spawn random item every `spawn_rate` frames

7. Call the object's execute method

8. If object is of type `Item`, check for collision and update player's stats according

9. Put object back in queue if execute returns `True`, else kill the object

Because the `Display` object appears once every iteration and it is the object which will update the Turtle screen, it can be considered that every time the `Display` object appears one frame has passed. The reason steps 3 to 6 are nested within this the `Display` object condition is so that these steps are repeated only once each frame rather than once each time an object is dequeued which improves the efficiency of the code.

Below is the code to update the player animation.

```
# One cycle animation cycle for player has 4 frames/stages
if frames % (self._player_animation_rate // 4) == 0:
    self._player.update_frame((frames // (self._player_animation_rate // 4)) % 4)
```

For example, if the `player_animation_rate` is 48 frames, the if statement will be true for every 12, 24, 36, 48, 60, 72 etc. frames. Then the value being passed to the `update_frame()` method will be 0, 1, 2, 3, 0, 1, etc. respectively. The `update_frame()` method within the `Player` class sets the `Player` object shape according to the frame number.

Since there are 4 frames in each animation cycle, this ensures the duration of the full cycle is equal to the animation rate and that the `update_frame` is only called 4 time each cycle.

The following part of the code is used to control the flow of the game.

```
if frames % self._game_fps == 0:
    # The following runs once every second
    match seconds % 30:
        # Event cycle repeats every 30 seconds
        case 0:
            instructions_turtle.clear()

            # End game after finals or start the normal stage
            if self._curr_event == "Finals":
                ws.PlaySound(None,ws.SND_PURGE)
                ws.PlaySound('Ending.wav', ws.SND_ASYNC + ws.SND_LOOP + ws.SND_FILENAME)
                self._game_ending = True

            else:
                self._curr_stage = "Normal"
                self._item_spawn_rate = self._normal_spawn_rate
                ws.PlaySound(None,ws.SND_PURGE)
                ws.PlaySound('Sakura.wav', ws.SND_ASYNC + ws.SND_FILENAME)
```

The if statement above become true in essence after every second has passed. The stage cycle of our game repeats every 30 seconds, where the current stage is set to “Normal”, “Normal Transition”, “Event” and “Event Transition” at the 0th, 15th, 18th, and 28th second of that stage cycle. A match statement is used instead of an if statement to improve readability since we are only checking against multiple specific values.

The current stage above determines the item to be spawned according to the code below. The conditional statements are used to choose a random item from their respective lists (based on the stage and event) and obtain data from the corresponding item dictionaries. Then a new `Item` object is instantiated and placed into the queue.

```python
def spawn_random_item(self):
    # Spawns random items based on the current stage of the game
    if self._curr_stage == "Normal":
        item_list = list(self._normal_item_dict.keys())
        item_name = random.choice(item_list)
        item_data = self._normal_item_dict[item_name]

    elif self._curr_stage == "Event":
        match self._curr_event:
            case "Rhino":
                item_name = "Rhino"
                item_data = self._stress_item_dict[item_name]

            case "Mid Terms" | "Finals":
                item_name = "Exam"
                item_data = self._stress_item_dict[item_name]

            case "Recess":
                item_list = list(self._bonus_item_dict.keys())
                item_name = random.choice(item_list)
                item_data = self._bonus_item_dict[item_name]

            case "Projects":
                item_name = random.choice(["1DProject", "2DProject"])
                item_data = self._stress_item_dict[item_name]

            case _:
                return
    else:
        return

    # Instantiate the item and add it to the queue
    self._queue.put(Item(self, self._item_start_xcor, item_name, item_data))
```

Next, the object's `execute()` method is called, which runs different code depending on the type of the object. If the type of the object is Item, the `GameController` will also check if it has collided with the `Player` after it has been executed.

```python
# Execute code for next object
result = nxt_obj.execute()

# If type of object is Item
if type(nxt_obj) == Item:

    # Check if Item collides with Player
    collide_result = self.check_collision(self._player, nxt_obj)
```

If the `Item` is found to have collided with the `Player`, a suitable sound is played depending on the type of `Item` it collided with. The relevant stats of the game and its displayed values will also be updated, also depending on the type of `Item`.

```python
# If item collides with Player
if collide_result == True:
    # print(f"Player collided with {type(nxt_obj)} {nxt_obj} with name {nxt_obj.get_name()} at {nxt_obj.get_location()}.")

    if nxt_obj.get_name() in ["Crate1", "Crate2", "Crate3", "Rhino"]:
        play_thread = threading.Thread(target=self.play_breaking_sound)
        play_thread.start()
    else:
        play_thread = threading.Thread(target=self.play_pickup_sound)
        play_thread.start()

    # Update game stats
    if 0 <= self._game_stats["Stress"] + nxt_obj.get_stress() <= self._max_stat_value:
        self._game_stats["Stress"] += nxt_obj.get_stress()

    if 0 <= self._game_stats["Health"] + nxt_obj.get_health() <= self._max_stat_value:
        self._game_stats["Health"] += nxt_obj.get_health()

    if 0 <= self._game_stats["Grades"] + nxt_obj.get_grades() <= self._max_stat_value:
        self._game_stats["Grades"] += nxt_obj.get_grades()

    # print(self._game_stats)

    # Updates the stat display
    self.update_stats_display(stats_turtle)
```

All `Items` the `Player` collided with will be killed, hence result is set to `False` to signify that the Item should not be added back into the queue. If result is `False`, either from collision with `Player` or from the `execute()` function of the item, the `Item` instance is terminated and will not be put back into the queue.

```python
            # Kill Item instance
            result = False

    # If execute function returns True, object will be added back to queue
    if result:
        self._queue.put(nxt_obj)
    else:
        nxt_obj.kill()
```

k. Additional User Interface During Gameplay

The `GameController` class also has 3 additional methods relating to the User Interface(UI), `display_stats_icons()`, `update_stats_display()` and `show_instructions()`.

```python
def display_stats_icons(self):
    # Create turtle for each stat icon
    for stat in self._game_stats_keys:
        icon = turtle.Turtle()
        icon.speed(0)
        icon.penup()
        icon.shape(f"{stat}.gif")
        icon.setx(- (self._screen_width / 2) + self._icon_xcor)
        icon.sety((self._screen_height / 2) - (self._icon_ycor + self._game_stats_keys.index(stat) * self._icon_height))

def update_stats_display(self, stats_turtle):
    # Updates the value displayed for the stats grades, health and stress
    stats_turtle.clear()
    stats_turtle.speed(0)
    stats_turtle.penup()
    stats_turtle.hideturtle()
    stats_turtle.color("navy")
    stats_turtle.setx(- (self._screen_width / 2) + self._icon_xcor * 2)

    for stat in self._game_stats_keys:
        stats_turtle.sety((self._screen_height / 2) - (self._stat_value_ycor + self._game_stats_keys.index(stat) * self._icon_height))
        stats_turtle.write(self._game_stats[stat], False, align='left', font=self._stats_font)
```

The display_stats_icons() method is called during the GameController Initialisation Phase. It creates a new Turtle object which has the shape of its corresponding icon. Here a for loop is used to avoid repetition of code as each icon requires the same steps just with different arguments which can be calculated as shown above.

The update_stats_display() method is called every time a collision is detect and after the player's stats has been updated. Similarly, it makes use of a for loop to update the displayed values of the player's stats by clearing the Turtle and rewriting the new stat values.

The show_instruction() method is called at the start of every event and writes instructions to tell the player whether to dodge or collect the items based on the type of event (stress or bonus event).

I. TitleScreen class

TitleScreen is a class that inherits from GameInitialisation, which is first called in the game's main function and is the first execute() to be called in the program's control flow. When TitleScreen's execute is called, it first displays the title screen graphic using the bgpic() method. The TitleScreen class will contain functions that help the game progress from the starting page to the start of the game. The functions in TitleScreen are executed in the following order once the class is instantiated: execute(), choose_char, Instruction_screen, start_game. Start_game is when the control flow is directed to another class, GameController.

The program is written such that the "spacebar" key is used to progress the game from one page to another for most of the functions in this class. This is done using the onkeypress() method, which allows a function to be executed whenever the binded key is pressed. The onkeypress() method is always coupled with the listen() method, which tells the screen to start listening for keypresses. The very last line of execute is the input function which prevents the turtle window from closing to allow the screen to listen for the keypress. Once "spacebar" key is pressed the control flow is directed to choose_char.

```
def execute(self):
    # Starts turtle screen instance
    super().execute()

    # ADD YOUR CODE HERE
    # Display game title screen
    self._screen.bgpic("Title_Screen.gif")

    # Wait for SPACE key to be pressed then start game
    self._screen.onkeypress(self.choose_char, "space")
    self._screen.listen()

    ws.PlaySound('Main Menu.wav', ws.SND_ASYNC + ws.SND_LOOP + ws.SND_FILENAME)

    # Necessary to prevent Turtle from closing
    input("")
```

[redacted]

The first line of the choose_char function is the clear() method, which will be written every time the control flow is directed from one function to another within the TitleScreen class to prevent screen flashing. Similarly, the bgpic function is used to display a graphic centred in the screen, which will be the character selection graphic in this function. There are now two different programs flows that the player can take on now, which are the male and female character respectively. This is done using the onkeypress method as before and using the "Left" and "Right" arrow keys to select. Once again, listen() method is called to tell the screen to listen for the keypress, which will lead to different functions depending on the input.

In each of the set_male and set_female functions, 2 actions will be executed. The first will be to change the variable of _player_sprite to the strings either "Male" or "Female" respectively, which will eventually need to be passed into the GameController class followed by the Sprite class so that the user selection will be updated into the variable that decides which player graphic to display on the screen. This transferring of the updated variable across classes is done by passing the user input into the game instance execution, which will be mentioned later in start_game. The second action to be executed within set_male or set_female will be to call the instruction_screen method and progress the game automatically.

```
def instruction_screen(self):
    self._screen.clear()
    self._screen.bgpic("Instructions_Page.gif")

    # Schedule the actual game start after the delay
    self._screen.ontimer(self.start_game, self._instruction_screen_duration)

def start_game(self):
    self._screen.clear()
    game = GameController(self._player_sprite) # Create new game instance
    game.execute()
```

This top block of code is meant to display the instruction screen graphic, similarly utilizing the bgpic() method to display the appropriate graphic. The very last line of code in the instruction screen block is using the ontimer() method to automatically redirect control flow to the start_game function after a certain duration has passed.
This value is contained within the variable, instruction_screen_duration and is defined within GameInitialisation. Once the control flow reaches start_game the execute() function within the GameController class is carried out along with the passing of the user selected input of the character gender, which contains the function that initializes the character to display.

m. Endscreen class

EndScreen inherits from GameInitialisation, like TitleScreen. This is because it requires the screen initialisation execute() method from its parent class. The very beginning of this class is the_init method first, which also takes in the parameter of final_game_stats, a data structure that has been accumulating throughout the game into the instantiation of this class. Next the super mechanism is called so that the execute() within game initialisation can be called which will create a new screen for the end screen to then carry out its functions. The bgpic method is used to centre the display of the templated end screen, while the tabulated results will run through if statement logic to decide which text message to display.

```python
class EndScreen(GameInitialisation):
    """ Class containing functions related to end screen """
    def __init__(self, final_game_stats):
        super().__init__()
        self._final_games_stats = final_game_stats

    def execute(self):
        # Starts turtle screen instance
        super().execute()

        # Display end screen
        self._screen.bgpic("Results_Page.gif")
```

An example of the if statement logic for the stress statistic is as follows:

```python
#
if self._final_games_stats["Stress"] > self._stat_high_threshold:
    scale1, stress_msg = 'high,', 'perhaps \nyou should take care of yourself mentally!'
elif self._final_games_stats["Stress"] < self._stat_low_threshold:
    scale1, stress_msg = 'low,', '\ngood job maintaining it this low!'
else:
    scale1, stress_msg = 'neutral,', '\nkeep it up!'
```

The value within the "Stress" key is accessed and compared to the high and low threshold variables – which are defined in GameInitialisation class – to produce the ending messages. This same if statement logic is applied for the 2 other statistics of health and academics. The very last part of the code is making use of the turtle module to produce text on screen using write and coordinate setting functions.

n. Sound and Music Implementation

```python
#Main Menu Music Initialisation
ws.PlaySound('Main Menu.wav', ws.SND_ASYNC + ws.SND_LOOP + ws.SND_FILENAME)
```

To initialise the main menu music, we call the **PlaySound method** from the **winsound library**, which takes in the two variables, the music to play in the form of a .wav file and the flags that affect the interpretation of the music file. The SND_FILENAME flag indicates that the music is the form of a .wav file, the SND_ASYNC flag indicates that the file should be played immediately and that the music can be played asynchronously and lastly, the SND_LOOP flag indicates that the music file should be repeated should the song end.

```python
# Transition to event stage. No items spawning at this point
if self._curr_event == "MidTerm":
    # Current event hasn't been updated at this point;
    # So if the last event event is MidTerm, start playing bonus stage music
   ws.PlaySound(None, ws.SND_PURGE)
    ws.PlaySound('Bonus Event.wav', ws.SND_ASYNC + ws.SND_FILENAME)
```

If the music needs to be changed (e.g. when the game event changes), the currently playing music is first cut, by calling the **PlaySound method** from the **winsound library** and assigning the music variable to be None. The SND_PURGE flag is used, to indicate that all currently playing instances of music are to be cut. Next, the music for the regular gameplay is played, again by calling the PlaySound method. The .wav file is assigned to the music variable while the SND_ASYNC and SND_FILENAME are assigned as flags.

Lastly, implementing the collision sound effect when background music is running was extremely difficult, and required help from ChatGPT, which was further edited to support our needs. This is because the winsound library does not support two instances of wave files playing at the same time.

```
#Get the current file directory
pathname = os.getcwd()
#The respective filenames of the soundfiles
filename = 'ItemBreaking.wav'
#Obtain the true filepath of the soundfiles
filepath = os.path.join(pathname, filename)
#The powershell command to run the soundfile
powershell_command = f'[System.Media.SoundPlayer]::new("{filepath}").PlaySync()'
```

```
def play_breaking_sound(self):
    subprocess.run(['powershell', '-Command', self._pwrshell_break_sound])
```

Implementing the collision sound effects, the command to play the sounds are implemented as individual functions, which are then called later.

```
if nxt_obj.get_name() in ["Crate1", "Crate2", "Crate3", "Rhino"]:
    play_thread = threading.Thread(target=self.play_breaking_sound)
    play_thread.start()
```

When the player character collides with the item, if the item’s name is with the list, a separate thread will be created from the Threading library, which will then play the breaking sound. Threading is used so that the background music will still be playing when this new instance of music is also playing.