**PIC 16 Project Write-Up**


| **Name:** | **UID:** |
|---|---|
| Yi Zu Tan | 004559249 |
| Ryan Grgurich | 103634923 |
| Erick Hernandez | 304737995 |


## Introduction

### The Problem

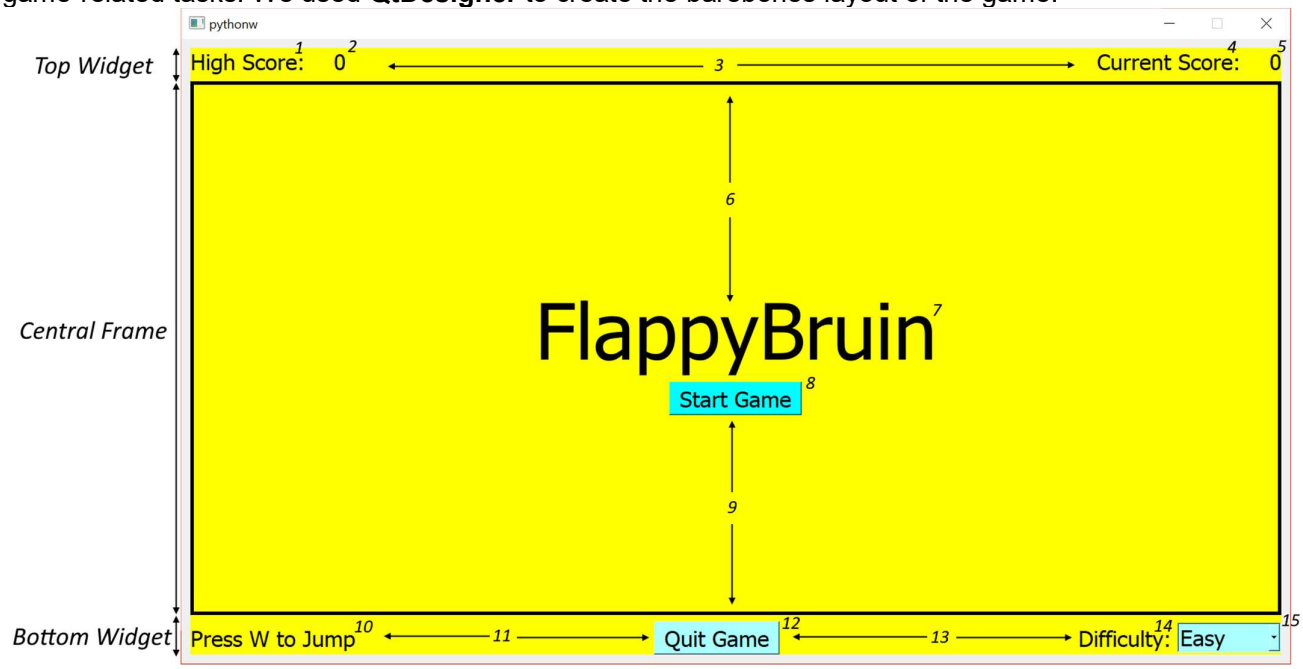
This is a single tap or key press game design. As the player you are Joe Bruin and you must magically **jump** your way through an endless barrage of dorm buildings, by **pressing the W key to jump**. Each oncoming dorm has an opening through which Joe can pass. If Joe can adroitly **jump** through the dorm opening in the building while listening to the Bruin clap, he gets a point. Joe will die if he **collides with either a building or the edges of the frame**. The number of points received for passing each building depends on the difficulty chosen, with higher difficulties earning more points. However, increasing the difficulty also increases the number of floors the buildings have and makes the passage through each building narrower.

### The Solution

The code for our game is composed of three main files, **flappybruin**.py, **building**.py and **bruin**.py, and supporting files (**lasky**.png, **intro**.mp3, **bruinsfight**.mp3, **upper0**.png, **inner0**.png, **inner1**.png, **lower0**.png, **jump**.ogg, **Joe_Bruin**.png).

## Interface Implementation

**flappybruin.py** defines the overall interface of the game and is responsible for a variety of miscellaneous game-related tasks. We used **QtDesigner** to create the barebones layout of the game.




First, we create a **background widget** with a **vertical layout**. We have three sections, the **top widget** (scores), the **central frame** (game) and **bottom widget** (options). The first frame has a **horizontal layout**, from left to right, **two labels (1,2)** a **horizontal spacer (3)** and another **two labels (4,5)**. The center frame has a **vertical layout**, from top to bottom, a **vertical spacer (6)**, a **label (7)**, a **tool button (8)** and a **vertical spacer (9)**. The bottom frame has a **horizontal layout**, from left to right, a **label (10)**, a **horizontal spacer (11)**, a **tool button (12)**, a **horizontal spacer (13)**, a **label (14)** and a **combo box (15)**.

### Imports

**sys**: For starting and closing the game safely
**PyQt5**: **QtCore**, **QtGui**, **QtWidgets** for the game elements
**bruin**: **Bruin** for the implementation of the bruin player
**building**: **Building** for the implementation of the buildings
**pygame**: Only for sound (because PyQt's QtMultimedia was glitchy on one of our systems)

**FlappyBruinGame Functions**

**__init__**: Initializes the class and all of its member variables, also calls setupUi
- Creates QTimer object used to repeatedly call **animate**
- Creates variables names **player** and **buildings** but sets them to None first

**setupAudio**: Sets up the audio input for the game
- Uses **pygame.mixer** functions to load music and play in a loop

**setupUi**: Sets up all the GUI elements of the game, is called by __init__
- Draws a border around the center frame
- Links the Start Game and Quit Game tool buttons to **gameStart** and **quitgame** respectively
- Links the combo box to **setMode**
- Disables close window button on main window using **setWindowFlags**

**setMode**: Updates the difficulty mode when user interacts with the combo box
- Calls the player and buildlings' **setDiff**

**gameStart**: Defines actions to start the game
- Stops the menu music, and starts playing the active game music
- Hides the center label and start game button
- Disables resizing of window using **setFixedSize** and **setWindowFlags**
- Disables the difficulty combo box
- Calls the player and buildings' **reset**
- Resets current score to 0

**gameEnd**: Defines actions to end the game
- Stops the active game music, and starts playing menu music
- Displays game over with the center label and restart on the start game button
- Reenables the difficulty combo box
- Reenables resizing of window using **setMaximumSize, setMinimumSize** and **setWindowFlags**

**framePress**: Debugging tool used to output to the console the mouse position when clicking on the GUI

**keyPressEvent**: Defines what happens when user presses the 'W' key
- Calls player.**jump** when W is pressed

**framePaint**: Constructs a QPainter for the central frame and calls the player and building's paint functions.
- Instantiates player and buildings when called the first time
- Sets focus on itself when mouse is not hovering over combo box

**eventFilter**: Allows game to switch event focus to the combo box
- Sets focus to the combo box when mouse is hovering over it

**updateScore**: Updates the score and high scores as the game progresses
- If score exceeds high score, update high score as well in real time

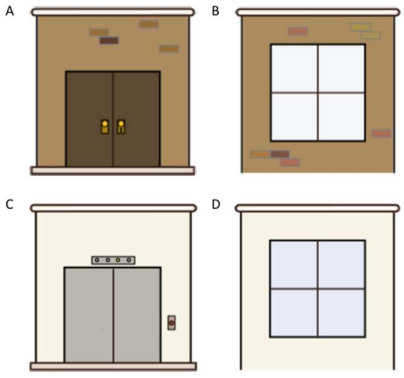
**quitgame**: Quits the game safely using **close**

**animate**: Calls **update()** for Bruin, Building and the center frame, also calls updateScore
- If player is dead, calls **gameEnd**
- Calls **updateScore** while player is not dead

## Obstacle Implementation

**building.py** defines the building objects in the game and through them the game difficulty is also set.

We used **Adobe Illustrator** to draw images representing each section of the building. The building is made up of a combination of images depending on where it is that the player can pass through and the difficulty level of the game. These are illustrated here:



A and B represent the external components of the building, with A being the ground floor and B representing all other floors. A and B are objects that if hit by the player will result in player losing and the game ending. C and D are the internal components of the building with C represent the lobby and D the inside of a dorm room, through which a player can pass to receive points.

Buildings are constructed in three ways depending on the difficulty. Below is an illustration of each case:

A  B  C 

A represents the configuration for easy, B for medium and C for hard. Each result is achieved through function in the **Building** class, specifically the **setDiff** function. Depending on whether the **mode** variable is set to easy, medium or hard the **numblocks** variable is respectively set to 3, 4 or 5. This in turn allows the function **paintBuilding** to know how many **floors** to paint, using the **paintBlock** function. **paintBlock** allows us to catch the **gapIndex** variable to draw where the opening for Joe to flap through should be located. **gapIndex** is randomly set so that each building has a randomly set opening. Furthermore, there are always exactly three buildings drawn to the screen at one time. The buildings move from right to left at constant velocity and are redrawn with a new randomly selected gap location when they exit the left hand side of the screen.

**Imports**
**PyQt5**: **QtGui** for game elements, specifically the **QImage** class.
**random:** For generating random integers using **randint**.

**Building Functions**
**__init__:** Initializes the Building class, loading in image files, setting the building count and initial drawing coordinates.
-Uses **QImage** to load png files representing the different building segments
-Uses **reset** to call other function that set initial coordinates
**frame_width:** Captures the current frame width so that it can be used to adjust object sizes when window size changes.
-uses the **QtGui** function **frame.geometry().width()**
**frame_height:** Captures the current frame height so that it can be used to adjust object sizes when window size changes.
-uses the **QtGui** function **frame.geometry().height()**
**randomizeGap:** randomly selects an index value within the range of building floors.
-Uses the **randint** function from the random class
- Prevents a 5th floor gap immediately following a 1st floor gap
- Sets collision coordinates of the upper and lower parts of the building
**update:** Used to update the coordinates of the building for animation
- Updates current position of building based on speed
- Sends building back to the start if it passes left edge
**setDiff:** Sets the game difficulty by changing the number of building sections
**paintBuilding:** paints building in pieces using **paintBlock**.
**paintBlock**: Paints a floor of the building.
-Uses the **drawImage** from the **QPainter** class to draw the image on the frame.
**resize:** Redraws the building coordinates when the window size is changed.
-Uses the **frame_width()** and **frame_height()**
**reset:** Resets the building class attributes

## Player Implementation

**bruin.py** constructs the player object. Updates position, velocity, acceleration and score in the game.

The **__init__()** function takes in a frame argument that is used to track the position of the player with respect to its frame. We use functions **w()** and **h()** to constantly track the frame size as it resizes and **lb()** to update the player's lower bound based on the first two. We will use the frame size to determine the starting location, velocities and acceleration of the player so that the game can be safely resized while still maintaining playability. We keep track of the player's score, position, velocity, acceleration, death status and inside building status. Also, we pass in a list of building objects as an argument into the **update()** function so that the player can check whether or not it has collided with a building or is inside a building. When determining collisions, we also provide about 2 pixels of leeway to make the animation smoother. Also, we use **bulidingCheck**() to update the scores based on the player's change of inside building status.

**Imports**
**PyQt5**: **QtGui** for game elements, specifically the **QImage** classes.
**oygame**: for audio capabilities

**Bruin Functions**
**__init__()**: Initializer
- Calls **reset()**
- Uses QImage to import image of Joe Bruin
- Uses pygame to import sound of a jump

**w()**: Returns frame width
- Uses **frame.geometry().width()** to access frame width

**h()**: Returns frame height
-Uses **frame.geometry().height()** to access frame width

**lb()**: Returns lower bound of playing screen which depends on upper bound and frame height
**reset()**: Resets the player to initial starting attributes.
- Set x position to 1/4th frame width from left of frame
- Set y position to middle of frame height
- Sets velocities vx, vy to 0
- Sets vertical acceleration proportional to frame height
- Sets inBuilding to False
- Sets dead to False

**buildingCheck()**: Used for updating player scores based on its inside building status
- **bool** argument indicates the player's new building status
- If player was in the building but is now outside of the building, increases score
- Otherwise, sets player's building status to bool

**setDiff()**: Sets scaleFactor and scoreFactor depending on the difficulty setting
**jump()**: Implements the jump sound as it adjusts the verticle velocity to simulate a jump
- Plays jump sound on **pygame.mixer.Channel(0)**
- Jump velocity is scaled by the **scaleFactor**

**update()**: Updates the current position, velocity, and checks for collisions.
- **buildings** argument is a list of building objects in the game
- Updates x,y based on vx and vy
- Checks each building in buildings for collisions
- If collision occurs, changes **dead** to True.
- If player is inside any of the buildings, calls **buildingCheck**(True)
- If player is outside all of the buildings, calls **buildingCheck**(False)

**paintBruin()**: Paints graphic onto the player position

## Discussion of Results

It would have been nice to stick to **QtMultimedia** to implement sound so that we only use the **PyQt5** package, but due to technical difficulties with the way python interacts with the sound device on one of our PCs, we had to implement a **pygame** solution to the sound instead. Also, parts of the code such as Building.**paintBlock()** includes a graphic adjustment factor **adj** that was included to remove unsightly gaps in between graphical images, which could be prevented with more image editing work. The most difficult part to code was to ensure that the game remained playable in spite of possible window resizing by the user. We circumvented this somewhat by disabling resizing during gameplay, but there could still be **minor graphical glitches** that occur when resizing the window to a weird aspect ratio. Moving forward, it might be a good idea to set a fixed window size based on the user's current screen resolution so that it would work on most systems. Overall, we are satisfied with how the game plays and it actually is pretty challenging on **Hard mode**.