Tagger - The Game

1st Michel de Buren
dept. of Finance
HEC, University of Lausanne
Lausanne, Switzerland
michel.deburen@unil.ch

2nd Eloïse Robyr
dept. of Finance
HEC, University of Lausanne
Lausanne, Switzerland
eloise.robyr@unil.ch

3rd Paolo Trifoni
dept. of Finance
HEC, University of Lausanne
Lausanne, Switzerland
paolo.trifoni@unil.ch

***Abstract*—This project is a chasing and catching game, similar to a tag game. Two players switch roles each round, the first whose role is to catch the other player, and the second whose role is avoiding being touched for the round time. The final winner is the first player who wins the maximum rounds set up. Players, style and features of the game can be personalized in the options menu.**

I. Introduction

In this project, we used the Pygame package to program a game. Pygame is a python module designed to program video games. It's based on SDL, which allows to program the multimedia parts of the game, often the most restrictive ones, more easily. It therefore simplifies the setup of graphical and sound features .

In the Tagger game, 2 players face each other in a closed arena. Both players play on the same keyboard. One player is the chaser, while the other must avoid being caught. Should the players' positions coincide, the chasing player's score counter will go up as long as both players have common coordinates. Once the score reaches the maximum score or the maximum time has passed, the round ends and the roles of the players are switched.

There are four different powers that can be used, if caught, by both players. The invisibility power makes the player disappear, the speed power make the player's circle move faster, the size power makes the circle bigger and the infinity power allows the player to go through the arena's borders.

Default settings can be changed and personalized in the game's menu. There are three settings menus. One for the players, which enables players' specific customization. The second for gameplay settings and the last for style and theme changes.

Once one of the player gets to maximum number of points, a victory point is given. Victories are accumulated and are displayed on the game menu, on both sides of the menu tabs. When the user quits the game (that is, closes the game window), the total victories are set back at 0.

II. Description of the research question and relevant literature

In this section, we will describe the basic unfolding of the game and the aspects we wanted to implement.

A. Default settings

If the users start playing the game without going to the option menu, the options are set such that the players' names are respectively "Blue" and "Red" for player one and two. The circles are blue and red. In the gameplay menu, all powers are on. The round time is 30 seconds. A player wins a round once their score reaches 100. The winning number of rounds is set at 3, and players' default speed is 0.8 pixels/millisecond. In the style menu, the background is pale beige, the details are pale green and the font is dark grey.

All of these settings can be customized in the respective menus, without having to change the code.

B. Game play

Once the game is started, it is player one's turn to try and catch player two before the maximum time is reached. Once they reach the maximum score, the round is stopped, the chasing player wins the round. Then the roles are reversed and it is player two's turn to try and catch player one.

If the player that is trying to catch the other does not manage to do so before the maximum time is reached, the round is won by the other player, i.e. the one that was chased.

During the round, some powers appear on the background of the arena. The player that catches one of them before it disappears benefits from it for a few seconds, then gets back to its original settings.

If one player catches the invisibility power, its circle disappears for two seconds. If a player catches the size power, their circle's size changes for 5 seconds, as its radius increases. After 5 seconds, the size gets back to its default radius. If a player catches the speed power, their circle moves faster. Its temporary movement size is increased and gets back to normal after 5 seconds. If a player catches the

infinity power, their circle can go through the borders of the playground. If it goes further up than the top border, it appears back at the bottom of the arena. The same effect happens if it goes down, to the left or to the right in the corresponding opposite side. After 2 seconds, the power turns off and the player can't go through borders anymore.

Please refer to the Code implementation section for details about code's operation.

C. Scoring

The score starts at zero at the beginning of each round and resets when the round is finished. The round finishes when the chasing player manages to catch the other player long enough under the maximum time to get the maximum points, or that the maximum time elapsed before they did so.

Each time the players' circles overlap, points are generated. The function that increases the score is defined in the game's mechanics file in and described in the Code implementation section.

D. Relevant literature

As stated previously, we used many features from the Pygame's library. However, we decided to code ourselves some features of this project to have more flexibility. That is the case for the menu implementation.

III. Methodology

In this section, we describe the timeline we followed for the game configuration.

A. Core Design

The first step of this project was to construct the core of the game. In the core, the game window, the playground and the players were defined.

B. Players & Movement

Player's movement's were implemented so that they move in the corresponding direction once the user presses the movement keys.

C. Time & Score

Time was designed so that rounds have a time limit. The score is implemented to account for how long players have been in contact with each other.

D. Rounds & Points

Rounds were introduced to enable players to switch roles and to set an objective number of points which would result in a victory. Points are attributed at the end of each round to the winning player.

E. Powers

Powers were implemented to add an extra layer of complexity. There are 4 powers: Invisibility, Infinity, Speed and Size. The implementation consisted in constructing a generation algorithm, a way for the player to acquire the said powers, making them disappear if they are not caught, and the effect of obtaining a power.

F. Menus

Menus are introduced to give possibilities to the players before the game starts and to avoid the automatic start of the game as soon as the code is run. There are three types of menus with different functionalities.

IV. Code implementation

A. main.py

The program runs in main.py, which contains the initialization of the game.

Once the game is launched, we get into the menu, where we set up parameters. When the menu function is finished, it can return two values, either False or None. If value is False, the main function will also return False. This means that, once the game is running and the window open, if the user clicks on the top-left arrow (on Mac) or on the Quit button, the function finishes to execute and return False. If not, the function, thus the game, keeps running, launches the following classes and enter the while running loop.

Fig. 1. Main menu

The functioning of the game is based on a set of nested while loops. These run continuously looking for the score, for contacts and will adjust players' position according to pressed keys.

Inside the *while running* loop, the other loop, *while current_round*, first sets the round start time, which is used to calculate the time that is left in each round. The loop time represents the milliseconds it took the computer to go through the entire *while current_round* loop. It is calculated at the end of this loop, with the start time set up at the beginning

of the loop and the current time, taken at the end. The exact current time is taken with the *pygame.time.get_ticks* function.

In the *while current_round* loop, we check for events with the function:

pygame.event.get().

If the event is quitting the game, we set the function such that it returns None and quits the game. If the event is pressing the escape key, the current round finishes and returns None, and the window of the game menu appears and a sound is played with the function *fame.menu_back.play*, which is inherited from the Game class defined in game_mechanics.py, and plays the sound_menu_back.mp3 sound.

Still in the *current_round* loop, functions set up the Arena. The blit function places elements on the playground but do not display them. The set-up playground is displayed only when all necessary elements have been blitted to the screen.

The function *call_keys*, defined in the game_mechanics.py file, detects and translates the pressed keys into player movements.

The increase of the score is defined as a function of touching surface and time, and not of the game's speed. The *distance* function takes into account the location of the two players' circles and their size. If the distance is smaller than their size, the score increases (as told before, as a function of loop time). When there is a collision, the sound_game_touch.mp3 sound is played. The function:

pygame.mixer.get_busy() == 0

detects if a sound is already being played. If so, the sound that was supposed to be played is cancelled, and the next is played if there's no sound already playing. This allows to get clear sounds each time, instead of a continuous and overlapped one.

The players' circles are placed such that the chasing one superposed the chased one with this function:

```
# Blitting Players & Moving Score
if Game.current_turn % 2 == 0:
    player_one.draw_player()
    player_two.draw_player()
    blit_score_player(player_one,player_two)
else:
    player_two.draw_player()
    player_one.draw_player()
    blit_score_player(player_one,player_two)
```

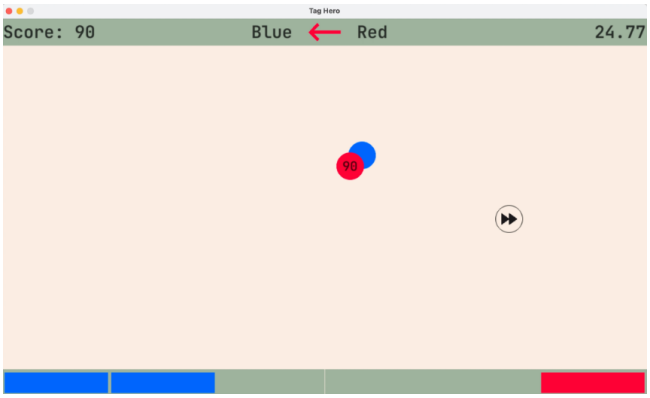

Fig. 2. Chasing player's circle overlaps the other player's

For example, in Figure 2, it is Red's time to chase Blue. When it is touching it, Red's circle is over Blue's. Would it be Blue's turn to be the chaser, Red's circle would be under Blue's if it was touched.

B. Item blitting

The *blit_status_bar* function displays the bars at the top of the playground.

The *power_cool_cycle* function draws the power timer on the top bar that runs lick a watch, for both players.

The arrow displayed on the arena that shows which player is chasing the other is defined by *who_text_blit* function.

Then the bottom bar, i.e., point bar, is "blitted" with the function *blit_point_bar*.

Finally, the score and time left are blitted by the function *blit_time_bar*.

The function *pygame.display.update()* displays everything that was set up as blit previously, all at once. That is, we can see the arrow showing who's chasing who, the remaining time of the round etc. appear at the same time as the bottom bar and the two players' circles.

It is important to highlight that every detail is set up to that the game works as correctly as possible. For example, the top bar is "blitted" before the power timer watch, otherwise the watch would be behind the bar and wouldn't be displayed and visible. Order matters!

A function checks if the game's score is equal or higher than the maximum score (set up at 100 by default but can be changed by the user in the menu). If so, another function checks if it was the first or second's player turn to chase the other and gives the point to the chasing one. If the first player is the chasing one, they are given a point with the method

player_one.give_point(),
otherwise player two is given the point with the method player_two.give_point().

The *Game.last_round_winner* is set with the name of the winner of the round.

The same outcome occurs if the game's time passes before the chasing player makes the maximum score set up.

As soon as the score is higher than the maximum one set up, the *current_round* loop will break and go back to the *while running* loop. First thing, the loop will do the game victory handling. That will check whether a player has a number of points larger or equal to the maximum number of points set up. If so, it will display the name of the concerned player followed by the text "Wins the Game", at the center of the playground as configured with the function centered message, add one victory to the total number of victories displayed in the game's menu and run the function *space_bar_initiation()*.

One step of the loop checks whether the last round winner is different from " ", that is it checks if there already was a round. If so, it displays the name of the last round's winner, updates the point bar at the bottom and displays the message in order to continue or quit the game. After the message is displayed, if the user pressed the spacebar, the start time is reset and the games continues. If not, that is if the user presses escape or quit the game, the function returns None and the game quits.

C. game_mechanics.py

1) Game:

In the game_mechanics.py file, the **Game class** is defined, with the layout of the window, the movement size (distance by units of time), players' circles size and the touch distance, which triggers an event when one circle touches the other. *sleep_time* is set up so that messages appear 1 second after the event occurred and one second from each other.

Dark factor is used later to keep the same "base" color in circles, but darker.

The bottom bar, which shows point won by winner, is split exactly at the middle by a thin line, which is the same color as the arena's background.

Game.current_turn is set at 0 by default and increases at each round, so that "Round 1!" is displayed on the arena before the start of round one, "Round 2!" before starting round number two, etc. Once the game is over, *current_round* loop is done and the *Game.current_round* is reset to 0.

The default settings are such that the round time is 30 seconds, the maximum score is 100 points, the maximum points (i.e., winning rounds per game) is 3 (see Figure 3).

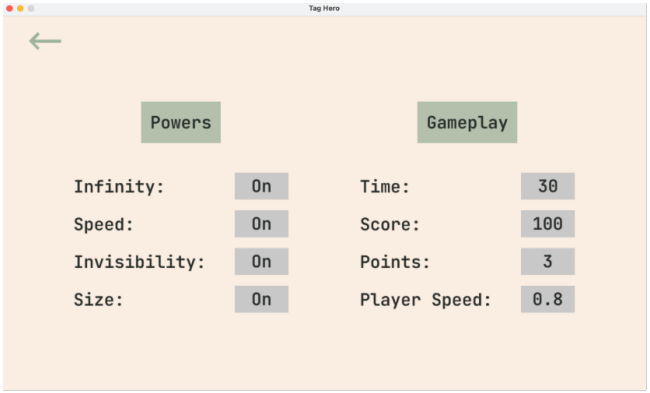

Fig. 3. Gameplay menu - default settings

The bars at the top and bottom of the arena window are set such as their height is the same as the players' circles diameter, and their width is the same as the arena's.

In the Game class, almost everything that is used later for the game itself, even main.py file or menu.py file, is defined. main.py is the dynamic of the game, the instructions, and game_mechanics.py is the file where main.py looks for info in when it's executed. To sum it, game_mechanics.py is the file the defines and explains how to play the game, and main.py gives the sequence of what happens when.

Point bar paddings are surfaces filled with the same color as the top and bottom bars. They are used to visually separate players' point bar from the border of the bar and from each other. Their width is calculated by taking the total width of the playground divided by 2 and also divided but the maximum number of points defined.

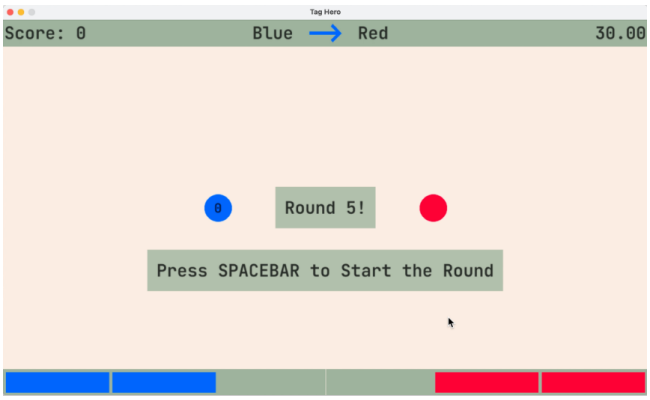

Fig. 4. Bar paddings make a frame around point color bars

Algorithmically, we created surfaces with players' colors which increase with points. Those surfaces are placed beneath

horizontal and vertical bars (paddings), which create an impression of frame of the same color as the point bar. The frame is built with very thin bars and its display position depends on the defined maximum point number.

The arena is then defined as the total playground, or window, minus the top and bottom bars. The color is taken from predefined in *background_color* and can be modified in the game's menu.

We decided to use JetBrains' monospaced font for the whole game. This allows to always display the countdown at the top right of the arena window at the same distance from the border, even though the size of the characters changes. For example, a "1" does not take the same space as a "9", and the countdown would normally appear blurry each time a second passes as the displayed text changes, but with the monospaced font the frame in which the countdown appears always keeps the same size. The same effect occurred and was solved by this font with the number on the chasing player's circle.

The *pygame.mixer.init* function initializes the Pygame sound module.

Powers settings are also made in the Game class. *power_despawn_time* defines the duration a power is displayed on the screen during the round and is set at 3 seconds. The *power_cool*_cycle function set up the dial (its dimensions and position) that appears on t the top bar when a player catches a power.

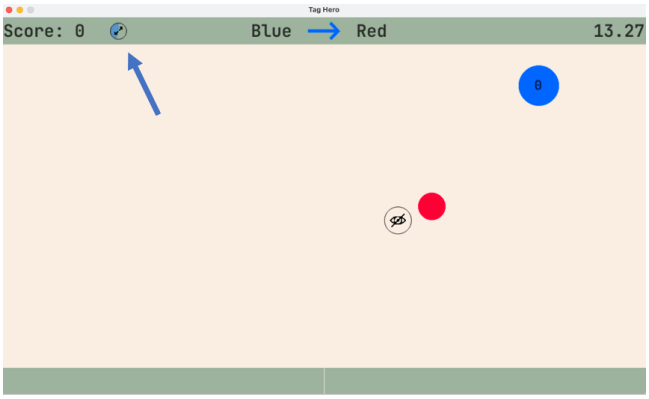

Fig. 5. Power timer appears when a player catches a power

The powers are activated by default by can be deactivated in the menu.

The titles of the menu also use the monospaced font of JetBrains and are anti-aliased so that the displayed text isn't distorted. Then the texts are centered on the window by 1. Getting the text's size with the function *title_main_menu_surface.get_size* and 2. Removing half of this size from half of the window's size horizontally. The text is displayed at the middle of the window horizontally and at 1/3 of the window vertically

After the game Class, the anti-aliased players' circles are defined with the functions:

gfxdraw.aacircle(surface, round(x), round(y), radius, color)
gfxdraw.filled_circle(surface,round(x),round(y),radius,color)

2) *Players:*

The **class Player** inherits from the Game class, and takes values name, initial position with point x and y, rgb (color), and keys to go left, right, up and down. The function *who_text_blit* will display the players' names on the top bar, and the names will be dynamically placed at the same distance of the arrow on the top bar.

The colors are set up such that the circles have a specified color, and the number displayed on them have them same color but darkened (with the previously stated dark factor).

The *self.rgb_alpha = self.rgb + (self.icon_alpha,)* function will use the color set for the player who gets the power, and display it in transparency in the power dial on the top bar.

The players' default score is set to 0.

The *self.player_powers* dictionary takes the activation status for each power and the activation time, which is set at 0.

The *get_position()* method, which is used in the main.py file to award points and to check for powers' pick-up, is defined and return players' positions according to their coordinates (x and y).

The movements are defined with the respective "*move_*" functions, which are then used in the move function. The move function defines the direction and distance the player's circle will move according to the pressed keys. This function is used in the main.py file to translate the pressed key(s) into a movement.

The reset function resets the player's default attributes and is used in main.py while running loop, so it launched when the user quits either the game or a round.

The *draw_player* function displays the player's circle if the invisibility power in inactive.

The type of playground is defined and depends on whether the infinity power is activated or not, and is used in the *player_physics* function. If the power is active, the player benefiting from it can go "through" the borders of the arena.

For example, if they go further right than the right border, their circle will appear back on the left border.

The *distance* function, later used in power activation and score calculation, returns the squared root of the sum of squared differences of both objects' coordinates (Pythagoras)

3) *Powers:*

Like the Player class, the **class Powers** inherits from class Game.

It takes as inputs classes that define an image, player one, player two, a probability, a duration and an enhancement factor.

Images are loaded and converted into RGB alpha (for transparency), and scaled into smaller images. The color is set defined so that the inside of the circles is transparent, and only the shape of the circle is displayed.

The attributes probability, duration and enhancement factor are created, and we set that the players are equal to the ones defines in the Player class. Initial positions are defined as None as the powers are not displayed at the beginning of the rounds.

The method *generate_position* generates a position for the powers according to an uniform distribution law on the arena. Some elements are subtracted from the width and height so that the powers' circles are entirely on the playground (that is, to avoid not seeing a part of them because they go through the borders).

The method *blit_power* will draw the power's circle with the corresponding image.

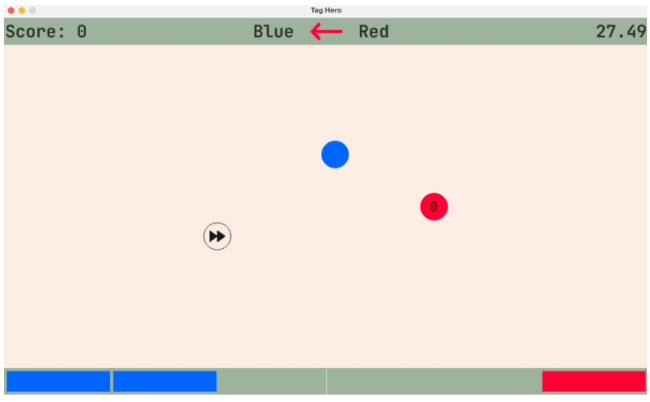

Fig. 6. Speed power is displayed on the arena

The *run_power* method uses the pre-defined poisson probability function.The poisson probability function determines the probability that more than zero event occurs (or that there is no null event). If the power is not displayed, the function will check if the probability of the uniform law is lower than the poisson probability (which uses dt). If so, it will generate a position, blit the power, start the power timer and display. If not, nothing happens. If, however, the power is already displayed, the function will handle the case where a player touches the power's circle (that is, if the position of the player's compared to the power's is smaller than their radius). If there is a contact, a sound is played, the power's circle is not displayed anymore, and the target power "active" is set at True and "time" equalized to *t.time()* for the player who caught it. The power is then activated with the function *activate_power*.

If no player catches the power before the time elapses *(t.time() - self.start_time > Game.power_despawn_time)*, its circle is simply not displayed anymore.

When players have an active power, the function checks whether it has been active for longer than the duration. If so, "active" is set to False and the function *de_activate_power* deactivates the power.

The power subclasses all inherit from the Power class.

Target power, which was set to None, is overwrote with the concerned power name.

4) *Power subclasses:*

For the **Speed_power subclass**, the player who caught the power gets their movement size gets increases but product of enhancement factor and movement size. The movement size (speed) then gets back to normal with the deactivation function.

In the **Size_power subclass**, the activate power function increases the circle's radius of the player who caught it by the rounded product of the enhancement factor and original radius. The deactivate function is to set the circle's size back to normal.

For the infinity and invisibility powers, the activate and deactivate functions are not used as the power activation status is a condition for displaying the players' circles and the type of playground (fixed or continuous).

The powers' parameters (image, player one, player two, probability, duration and enhancement factor) are set up in the main.py file.

5) : Other functions

The function *space_bar_initiation* is an infinite loop checks for specific pressed keys and translates them into:

- Back to menu if escape is pressed
- Continues the game is space bar is pressed
- Quits to game if the X button is clicked

The *centered_message* function is used at multiple times and is defines such that is gives a specific font, color and size to a text. Then it creates a message padding by taking the text size (width and height) plus 0.8 times that size. The alpha factor is again used for transparency in order to see every item (see Figure 7).

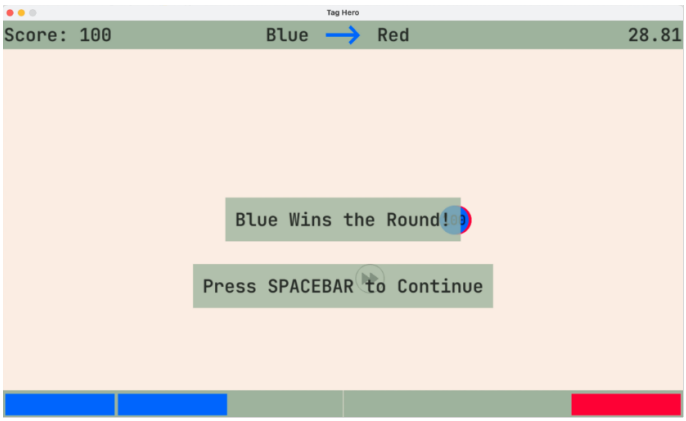

Fig. 7. Transparency allows to see all the elements behind the paddings

message_padding_rect creates a rectangle object using the pygame library and defines each of its angles' coordinates according to the text's coordinates and size.

The *arrow_polygon_draw* function draws the arrow at the top of the playground which indicates who is chasing the other player, and colors it with the chasing player's color. Globally, this function computes the arrow's dimensions and how to display it.

The *power_cool_cycle* function displays the dial with the caught power on the top bar. It takes into account the player, the powers, the powers' status and the half of the top bar. The function will calculate the remaining power time and display it on a consuming pie on the top bar. The remaining time is translated into an angle and a loop checks for every point. The back color that fills the dial (corresponding to the transparent player's color) is first blitted, then the circle, and at last the small power's image.

The *blit_score_bar* function displays the score at the top left of the playground.

The *blit_time_bar* function displays the remaining round time with two decimals, with a specified padding.

The *who_text_blit* function displays the players name and the arrow defined in arrow_polygon_draw function, according to the round number, on the top bar of the playground.

The *blit_score_player* function get the score to be displayed, gets the players' positions, checks whose turn it is to chase and displays the score on the chasing player. The color of the score displayed is the same as the circle's but darkened.

The function *blit_total_points* displays the number of victories on the circles besides the options name in the game menu.

The *blit_point_bar* function displays the bar at the bottom of the playground with the previously explained "frames".

The *key_menu_back* function is set such that when the user is in one of the sub-menus, either pressing the escape key or a left key (that is, one of the keys defined to be translated into a left movement), the program quits the sub-menu and goes back to the main menu. The *key_menu_forward* is similar in a way that is goes in the next menu either when the key escape is pressed, the enter key or a left key.

The *menu_blit_all* function will blit the background, the title and the centered messages with the options titles and their respective rectangles. These will allow the program to move the circles beside the titles to move when either the mouse in moved inside the rectangle area, or when a key defined as an up or down movement is pressed. The arrows at the top left of the options pages are also in such a rectangle for the same objective.

The **input_text class** takes a field name, default input, x and y filed left (coordinates), x box right offset (for box alignment on the right) and maximal characters. It first defines fields and boxes' size, color and positions. Il also sets the defaults inputs. The function run_detect of this class checks if a key is pressed or if there is a mouse click. If the backspace key is pressed once in a field box, a character is deleted. Mechanically, it does input[:-1]. If another key is pressed and the maximal character number is not achieved, a new character is added, corresponding to the pressed key. Each time a field in entered, a sound is played. For the fields requiring integers, the function returns the absolute value of the number entered, converted into an integer. For the rgb fields, the function will take the absolute value of lowest value between the number entered and 255, converted into an integer. For the fields requiring floats, it will return the absolute value of the number entered, converted into a float. The blit function turns the selected field box from grey to white, and back to grey when the filed in unselected (see Figure 8). It will also display the updated dynamically-calculated paddings such that the text in the field boxes is always centered, even if modified. This *input_text* class is used in all option menu allowing inputs.

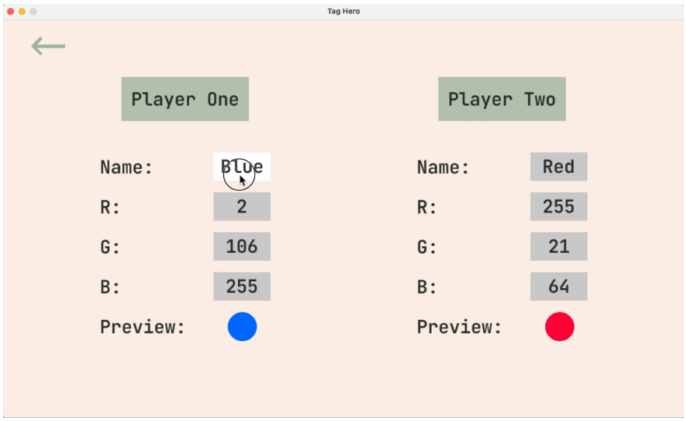

Fig. 8. Field box turns to white when selected

The **input_bool class**, also defined in game_mechanics.py, is very similar to the class *input_text*, except it is applied only to fields with Boolean values. It sets the default value as "On". If there is a mouse click on the field box, the value is switched (from On to Off and the other way around).

D. menu.py

In the menu.py file, several functions are defined with continuous while loops.

The condition *if event.type == pygame.MOUSEMOTION* checks whether there is a contact between the user's mouse and one of the rectangles in the menu, and then moves the circles beside accordingly, and plays the sound_menu_select.mp3 sound if it is not already being played. The condition *if event.type == pygame.KEYDOWN* defines that if the selection is already at the top (or at the bottom) of the menu, it will not move if an up (or down) key is pressed, but will move otherwise. It will move the selection up if an up key is pressed, down for down key, display the respective selected option menu when a right key is pressed and launch the game when the selection is on Start Game option.

The *credits_menu* displays the background and the names centered, and the arrow in its rectangle at the top left of the page. A loop check if there is a quit event, in which case the function returns False is the game is quit, if there is a down key pressed, the program quits the credit menu and goes back to the main menu, and if there is a click on the area of the arrow it also goes back to the main menu. The sound_menu_back.mp3 sound is played every time the program goes back to the menu, and the display is updated.

The *main_menu* function starts with a loop that sets the vertical position of the two circles below the option names. An inside loop checks for events. If the event is quitting the game, the program quits the game. If the event is a mouse click or a key pressed, the loop will check on which rectangle (that is, menu option name) the click was made on. For the first one, the program returns the value None and plays the sound_menu_enter.mp3 sound. For the second to the fourth one, the program plays the defined sound and executes a function. If the function executed returns the value False, the program quits the game. The returned value is very important, as in main.py the game starts only when the function *main_menu* returns None.

The function *which_options_menu* blits all the elements of the option menu and the loop will update the display for every event. The same logic applies here as for the previous functions for mouse clicks, pressed keys and sounds. However, in this options menu, the user can change the features of the game, for the player, gameplay and style.

The function *player_options_menu* first places elements on the window, with specific spaces between each element (text and boxes) and uses *input_text* to get input for the different fields and convert it (or default numbers) into a list. It then converts the list into the preview of the selected color for both players and displays it. As a loop is continuously running, the display is updated each time a feature is changed. The *player_options_menu* function will also check for event such as quitting the game (top right arrow) or going back to the previous menu (top left arrow). The previously explained *run_detect* function allows to take event and translate them to keys and updates the display.

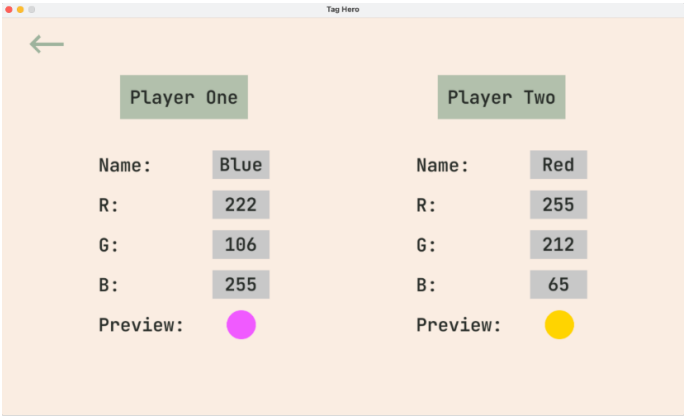

Fig. 9. Circle preview is instantly updated when RGB fields are modified

The function *gameplay_options_menu* is similar to the *player_options_menu* function regarding the display of the window. However, in this menu, the input contains both text and Boolean (Boolean for power activation).

The function *style_options_menu* is also very similar to both previous functions, except that there are three columns instead of two (see Figures 10 & 11). As *player_options_menu*, it only takes text as input (that is, no element of Boolean type). The data modification is at all time dynamic as functions are

in continuous running loops.

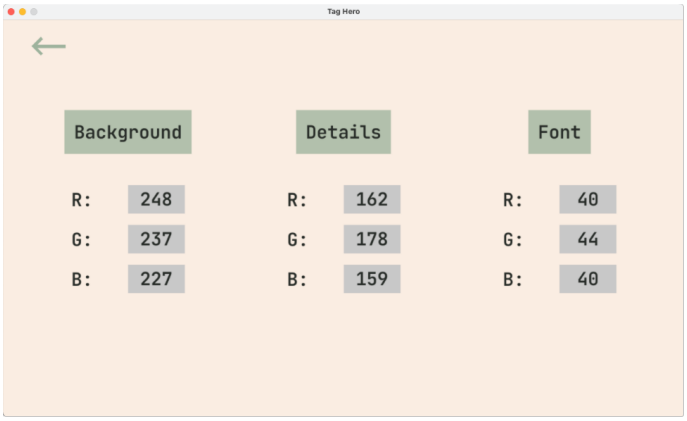

Fig. 10. Style menu with default settings

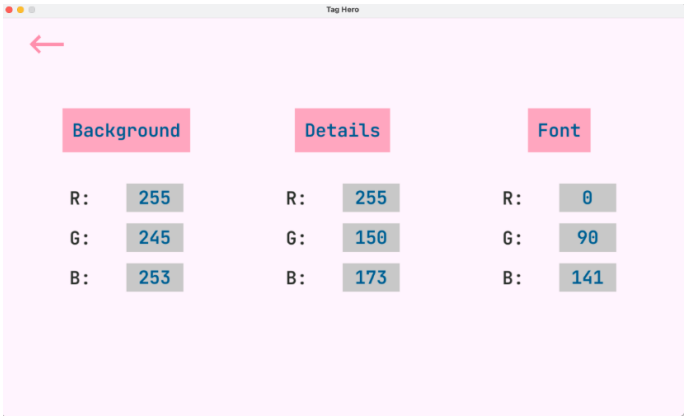

Fig. 11. Style menu with modified settings

Names and colors are set to be a value, which will be set in main.py. This is also the case for keys' specification. Obviously, each time the game is quit and restarted, settings are set back to default and displayed as such if not modified.

V. Results

During the elaboration of this project, we faced several difficulties. The first one we had was with the running of the game. As the game is a continuously running loop, it is affected by the speed of the computer. If the computer is running several applications, the game might be slowed, therefore the players will move slower.

At the beginning of the configuration of the game, the moves were defined as pixels per loop. But as we know, speed is a function of time. So we defined that, in the game, the players' speed would be pixels by milliseconds. Thus the loop time defines the number of pixels the players will move. This is in order to move players according to time and not the computer's speed. This solves also the issue with score increase as it depends on the loop time as well. If the loop was going fast, players would gain more points by catching the other one than if the loop was going slower. The increase of the score is therefore defined as a function of touching surface and time, and not of the game's speed.

We also had some difficulties with the font type. For example, with the countdown displayed at the top right of the playground, we observed that the text moved when the numbers changes, thus was not always centered in its frame. This was due to the fact that all numbers don't take the same space (a "1" does not take as much space as a "9"). After some research, we decided to use the monospaced font of JetBrains. We applied this font to the whole game as the same effect occurred with the number on the chasing player's circle and on the menu options.

Another issue we had was regarding the keys. At the beginning of the project, we tried to implement the keyboard's arrows as keys for the player on the right. However, we observed that the program bugged when several arrows were pressed at the same time as letter keys. The keys dependency being less restrictive among letter keys, we decided to use them for both players, so that all pressed keys can be translated as a movement in the game.

We also recognize that we could have used C++ for this project in order to make the game more versatile and efficient. An another implementation in JavaScript would allow to put the game online, so that people could run it on a web page.

VI. Conclusion

There is room for some improvements we did not implement.

The first one would by an a generalization of keys used. In its current state, the game is to be used by users with a QWERTZ keyboard. If someone wants to play this game with a different keyboard, they would have to change in the code the keys for moving their circle, in the player_one and player_two settings of main.py file. We could have created a menu that enables the user to choose the keys they want to use.

We also could have implemented navigation in the options menu with keys. If the user goes either on Players, Gameplay or Style option menu, they have to click on the field they want to change to access it. An improvement would be to access the fields not only with mouse clicks, but also with keys.

A last improvement we can see would be to configure the game window so it is scalable. The current program defines a window of specific dimensions. We could modify this so that the window can be expanded on the whole screen and adapt to every screen size.

Nonetheless, we are particularly satisfied of the implementation of the time depending loop. The challenge we had with this aspect turned out to be a constructive addition to this project, as the system we put in place allows the game to run perfectly smoothly, whatever the speed of the computer it is played on.

As a final note, we strongly recommend playing this game with two players for an optimal experience (and, indeed, more fun). Please enjoy!

REFERENCES

[1] *Pygame documentation*. Available: https://www.pygame.org/docs/
[2] Wikipedia (2020, September 17). *Rollover (Key)*. Available: https://en.wikipedia.org/wiki/Rollove_(key)
[3] *Pac-Man sounds*. Available: https://www.youtube.com/watch?v=nK71NEluAqA
[4] *Pac-Man sounds*. Available: https://www.classicgaming.cc/classics/pac-man/sounds
[5] broumbroum (2008, March 25) *sf3-sfx-menu-back.wav*. Available: https://freesound.org/people/broumbroum/sounds/50557
[6] pumodi (2008, March 2005) *Menu Select*. Available: https://freesound.org/people/pumodi/sounds/150222
[7] Gaming Sound FX (2019, April 28) *Mouse Click - Sound Effect (HD)*. Available: https://www.youtube.com/watch?v=h6_8SlZZwvQ