Python Alchemy: Crafting The Future of Education

Yianni Culmone, Omid Hemmati, Neyl Nasr, Benjamin Gavriely

April 16, 2024

1 Introduction

Many video games have crafting systems. Vastly popular games like Minecraft, Terraria, No Man's Sky, Subnautica, and Stardew Valley contain different mechanics, motivations, and outcomes. Regardless of all their differences in implementation, games with crafting systems appeal to players interested in creativity. (Grow et al.)

For games with crafting capabilities, the crafting system is often a keystone of depth and cognitive challenge. Crafting systems allow players to create new items/elements, advance in the game, and pursue creative endeavors. The appeal of crafting lies in its satisfaction of discovering new recipes, solving puzzles of resource harvesting and management, and tackling problems through unique methods (such as different items/powers/recipes). Either way, an appropriate inclusion of a crafting system in a video game would result in an enhanced depth and replayability of the game. (Grow et al.)

Crafting, due to its inherent roots in creativity, can be used as an effective learning method. It is a multi-sensory experience that can cater to various learning styles. Visual learners benefit from seeing and understanding how different components fit together, and tactile learners benefit from solidifying abstract concepts into concrete examples. There is a positive correlation between creativity and academic abilities, particularly in areas like reading, comprehension, and writing tasks. Fostering creativity within an education system can improve academic performance (Tzachrista et al.), so promoting creativity within a video game's crafting system can be seen as beneficial from most standpoints.

The goal of this project is **to implement a crafting system using graphs to foster creativity, and turn learning into a tactile gaming experience.** This problem matters because understanding the underlying mechanics of crafting systems can allow for the creation of a more engaging, creative, and educational gaming experience. This is done by cleverly converting the input JSON file into graphs, and using functions to make combinations from the ingredients being represented as vertices. By focusing on a simple model using graphs (associations between vertices), we can optimize and enhance the components that make crafting appealing to a player's creative side, such as a lack of limitations/rules on what a player can craft. This model can then be used in other games, itemsets, and applications, resulting in a multitude of use cases.

2 Data Sets

2.1 Main Data Set: Recipes.json

This data set contained elements and recipes for the crafting system of our program. It followed the format:


```
[
 {
   "NAME": "Acid Rain",
   "RECIPES": "rain, smoke / rain, smog"
 },
 {
   "NAME": "Air",
   "RECIPES": "$DEFAULT"
 },
 {
   "NAME": "Airplane",
   "RECIPES": "bird, steel / bird, metal"
 },
```

```
{
  "NAME": "Alarm Clock",
  "RECIPES": "clock, sound"
},
.
.
.
```

NAME: The name of the item
RECIPES: The items that you need to combine to create this compound. In cases where there are multiple recipes to craft the item, they are separated by a forward slash. Default items are indicated by the recipe "$DEFAULT"

This dataset was sourced from IGN.com (Claiborn, Samuel, et al.). It was extracted as an HTML table, and then converted into a JSON file. We decided to use JSON over csv/txt due to the ability to store recipes as dictionaries, which we go more in-depth into later.

2.2 Other Data sets: Chemistry.json

This set includes data for the "chemistry" game mode of our program. It follows the format:

```
{
    "NAME": "Sulfur",
    "RECIPES": "$DEFAULT"
},
{
    "NAME": "Hydrogen Gas",
    "RECIPES": "hydrogen, hydrogen"
},
.
.
.
```

NAME: The name of the chemical compound
RECIPES: The compounds/elements that you need to combine to create this compound. If this is a chemical element, that means its given to the player from the start and cannot be created by crafting. In this case, it will be tagged as '$DEFAULT'.
This set was generated by hand and did not use any external sources.

3 Computational Overview:

3.1 Data Representations and Computations:

In our implementation, we created a modified version of a graph. The following representation invariants for the Graph and _Vertex classes are as follows:

```
Class _Vertex:
    item: Any                       # This is the name of the element
    neighbours: dict[Any, _Vertex] # Map of the element this specific vertex can combine with,
                                      and the element created {elementcombine : elementcreated}
Class Graph:
    _vertices: dict[any, _Vertex]  # Map of the elements item, and its specific _vertex class
    discovered: list[_Vertex]      # A list of the successful combination created
```

Designing the graph in this manner is crucial for the ease of traversing. Especially, editing the _Vertex class in which the neighbours are a map. This allows for extremely easy traversing, and thus when combining 2 items within the game, we can efficiently produce the combination. The extra implementation of the discovered invariant within the Graph class allows us to easily display the 'found' elements on the left of the screen.

While our classes may initially appear more intricate compared to the original implementation, they were purposefully crafted to facilitate the seamless integration of future implementations. For example, the implementation of loading and saving merely necessitates a straightforward mutation of `graph.discovered`. Throughout later portions of the report, the seamless transitions and implementations can be seen.

A major computational portion of our program is building the graph from the dataset. This involves a slightly more complex version of the graph methods commonly seen throughout the course. This is most important function is graph.add_edge(), which is found within recipeloader.py:

```
def add_edge(self, item_created: str, item1: str, item2: str) -> None:
"""
Add an edge between two vertices, representing a combination that creates a new element.
"""
if item_created not in self._vertices:
    self.add_vertex(item_created)
if item1 not in self._vertices:
    self.add_vertex(item1)
if item2 not in self._vertices:
    self.add_vertex(item2)

# Add the neighbours
v0, v1, v2 = self._vertices[item_created], self._vertices[item1], self._vertices[item2]
v1.neighbours.update({item2: v0})
v2.neighbours.update({item1: v0})
```

This code snippet adds any of the three items that are not currently stored within the graph. It then mutates the neighbours of all the vertices with the correct format. This allows for our JSON reader to be very simplified, as it does not need to confirm the existence of a vertex before adding an edge. This implementation allows the reader to work "by all means", meaning that the order of data points in the given JSON file is irrelevant, and the representation invariants will never be violated, either. Within the intializer of the Graph class, we call a function in recipeloader.py called 'load_vertices()'. This reads all the data within the JSON file and updates the graph structure accordingly. An example of this can be seen as follows:

```
data = json.load(file)
    item_created, recipes = '', []
    for row in data:
        for key in row:
            if key == 'NAME':
                item_created = row[key].lower()
            else:  # key == 'RECIPES'
                recipes = split_text(row[key].lower())

        if not recipes:
            self.add_vertex(item_created)
            self.discovered.append(self._vertices[item_created])
        for combo in recipes:
            item1, item2 = combo
            self.add_edge(item_created, item1, item2)
```

This code reads through each data point, separates whether it is the name or recipe of an item, then adds each of the created recipes using the 'add_edge()' function from recipeloader.py.

Combinations are one of the essential computations for our graph implementation. Due to our implementation of the vertex class, the combinations are able to be completed with constant runtime.

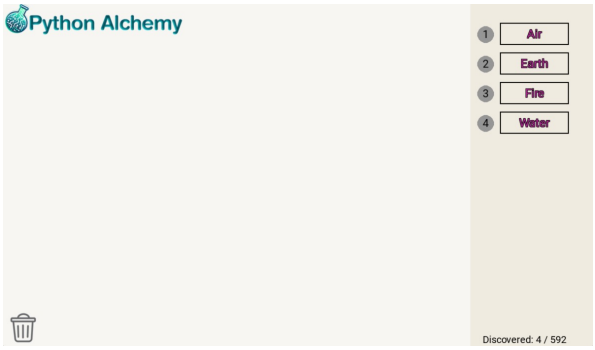
In order to compute a combination, the first ingredient is used as a key for the dictionary of possible crafting recipes of the second ingredient, or vice versa. The value of this dictionary lookup is the result of the craft, if it is a valid combination. Using a dictionary lookup (which has a constant runtime) is the most effective way to represent a combination within two items, which goes to show our flawless implementation of vertices in the context of a crafting system.

3.2 Visual Output:

Our program uses Pygame to display the results of the computations in a game format on a popup window. Upon running main.py, the user is greeted with a main menu screen (baraltech). They are met with three options:

1. Play

When the user chooses to play, they enter the gamespace. There is a blank "crafting area", in which users are able to craft new combinations. Their ingredients are found on the sidebar to the right, which will always contain the default items. Users select items from the sidebar by clicking on them, in which they will appear in the main crafting area, within a specified zone (this is done to prevent items from appearing on top of each other). Upon creating a new combination, the newly made item is added to the sidebar on the right. Every item is numbered in the sidebar, and a total count appears at the bottom. Once more than 10 items are in the sidebar, arrows appear to scroll through the list of discovered items. The scroll can be initiated through either the up/down arrow keys, or by left-clicking on the arrows. Whenever the player's crafting area gets too cluttered, they can clear it by pressing on the trashcan in the bottom left. This will only clear the crafting space, and will not affect the discovered items in the sidebar. When the player wants to return to the main menu, they can click on the logo in the top left.



2. Options

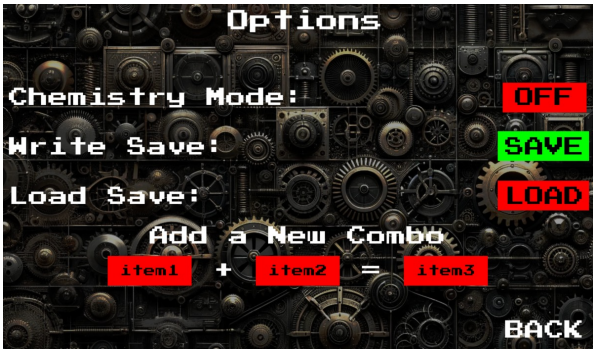
In the options menu, the player has access to 4 options: Chemistry Mode, Write Save, Load Save, and Add a New Combo.

'Chemistry Mode' is an option with a reactive button that can be used to enable the chemistry mode. In chemistry mode, the items in the sidebar are replaced with elements from the periodic table. The user can combine these elements together to form reactions, and the products are added to the sidebar. This mode is just a proof-of-concept to demonstrate the educative properties and potential of Python Alchemy.

'Write Save' allows the user to save their progress. Upon pressing the save button, their discovered items are stored to a csv file. It also indicates the mode that the user is in, to ensure that the user does not accidentally load a chemistry save while not in chemistry mode (and vice versa).

'Load Save' allows the user to load their progress, reading from the save file. As mentioned previously, the save will only be loaded if the user is in the same game mode as the save file.

'Add a New Combo' allows the user to create custom items with custom crafting combinations. The user is able to select two items which currently exist within the current items, and create a new item (or an existing item) that they combine to.



3. Quit

This simply quits the game. Unsaved progress will be lost, but any progress that has been saved will remain indefinitely for the user to load up again.

3.3 Libraries:

The following libraries are used in Python Alchemist:

1. __future__

Uses postponed annotations for our Graph and _Vertex classes.

2. typing

Uses Optional[Any] return types for many of our functions

3. json

json.load(file) is used throughout Python Alchemy. Graph.load_vertices() in recipeloader.py uses json.load(file) to parse the JSON file containing item data and recipes. JSON is a helpful file format to use because it is easy to read and write for humans, and easy to parse for computers. This allows Python Alchemy to act as a game engine with no hardcoded datasets. This enables easy updates and modifications to the dataset without the requirement of real coding knowledge. A snippet of json.load() in action within load_vertices() can be observed as follows:

```
data = json.load(file)
item_created, recipes = '', []
for row in data:
    for key in row:
        if key == 'NAME':
            item_created = row[key].lower()
        else:  # key == 'RECIPES'
            recipes = split_text(row[key].lower())
```

Since JSON uses dictionaries, it synergizes perfectly with our implementation of the Graph class. Each item and its combinations are stored in a dictionary with item names as keys and its components as values. This mapping allows for efficient access and updates to the graph's vertices, which makes JSON a fitting choice for our data structure's needs.

4. sys

The sys.exit() function is called in response to the pygame.QUIT event. This is triggered when the user clicks the window's close button, or whenever they press the QUIT button found in the main menu. This ensures that the game ends gracefully, flushing all resources and closing the window without leaving the Python interpreter idling. This is called within the function options() in main.py:

```
for event in pygame.event.get():
if event.type == pygame.QUIT:
    pygame.quit()
    sys.exit()
```

5. pygame

Pygame is used throughout Python Alchemy and it is central to its development and function. It provides the necessary tools and functionalities for developing the GUI, handling events, rendering graphics, and managing the gamestate. Following is a detailed breakdown of how Pygame is used throughout Python Alchemy, within the main.py file. None of the backend (recipeloader.py) makes use of Pygame, since Pygame serves the purpose of user-interaction display and handling.

pygame.init() is used to initialize all of the modules used within Python Alchemy. It prepares the software for running the program. This is rain within main() as follows:

```
def main():
"""
The main function that gets ran upon starting the file. This starts the program through Pygame.
"""
pygame.init()
pygame.display.set_caption("Menu")
main_menu(False, [], [])
```

pygame.display.set_mode() sets up the window's size/resolution and returns a pygame.Surface object representing the screen space. This is where all of the graphical elements (backgrounds, text, buttons, etc) are drawn.

pygame.Surface() is an object that represents the screenspace, where all graphical elements are rendered. We can see this being used in render_with_outline(), which is used to render text with outlines:

```
base = font.render(text, True, text_color)
outline_size = (base.get_width() + 2 * outline_width, base.get_height() + 2 * outline_width)
outline = pygame.Surface(outline_size, pygame.SRCALPHA)
```

pygame.Rect() is used to define the position and dimension of surfaces, enabling collision detection (collision/overlap between two items indicates a request to combine the items together), and defines the positioning of GUI elements like buttons. For example, within play(), the sidebar is a rect:

```
# initializing the sidebar
sidebar_width = 250
sidebar = pygame.Rect(screen_width - sidebar_width, 0, sidebar_width, screen_height)
```

pygame.event.get() is crucial for handling the interactions within the game. It retrieves events from the queue, such as keyboard presses and mouse clicks. This allows the program to respond to user actions like clicking buttons, dragging elements, pressing MOUSEDOWN or MOUSEUP to scroll the sidebar, or typing in an input to create custom recipes. For example, it is used to identify whenever an arrow key is inputted:

```
for event in pygame.event.get():
    # adding scroll when down arrow is clicked
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_DOWN:
            if k + 10 < len(g.discovered):
                k = k + 1
                pygame.mixer.Sound.play(click_sound)
        if event.key == pygame.K_UP:
            if k - 1 >= 0:
                k = k - 1
                pygame.mixer.Sound.play(click_sound)
```

pygame.image.load() loads images from files. These are used for sprites and backgrounds. Examples include the main menu/options menu backgrounds, our game logo which also serves as a home button, and the trash bin which is used to clear the crafting area. The usage of visual images enhances the appeal and complexity of the user experience for Python Alchemy. The implementation for creating a background image within main_menu() can be seen as follows:

```
background = pygame.image.load("assets/background.png")
```

pygame.font.Font() allows for the usage of custom fonts within text file. The font we use is provided within the as a .ttf file, and is used alongside varying font sizes to represent all the textual content in the game (element labels, menu titles, button labels, etc).

pygame.draw is a collection of functions that are used within Python Alchemy for drawing basic shapes. Within Python Alchemy, it is used to draw rectangles around elements and buttons, which also determines the hitboxes/boundaries of such elements and buttons.

pygame.mixer.Sound() loads and plays sound effects, such as clicking sounds when interacting with buttons or the sounds played when elements are combined. The sound effects in Python Alchemy were sourced from Minecraft, in which Python Alchemy sourced inspiration. The auditory feedback enhances the user experience, and provides another level of response to the game's output. Within play(), many sounds are initialized:

```
# initializing the sounds
combine_sound1 = pygame.mixer.Sound("assets/combine1.wav")
combine_sound2 = pygame.mixer.Sound("assets/combine2.wav")
combine_sound3 = pygame.mixer.Sound("assets/combine3.wav")
combine_sounds = [combine_sound1, combine_sound2, combine_sound3]
click_sound = pygame.mixer.Sound("assets/click.wav")
```

pygame.display.flip() is used to update the full display Surface, and make changes visible to the user.

Element is a class which uses pygame to represent interactive items within Python Alchemy that can be moved, clicked, and combined. Pygame surfaces are used for rendering rectangles and for collision detection within Elements. This is used within play() to create the individual items:

```
# Create and draw element
element = Element(screen_width - 190, 40 + j, 140, 45,
value.item.title(), font, text_color, item_color)
discovered.append(element)
if element.text_surface.get_width() + 10 >= 140:
    element.rect.w = element.text_surface.get_width() + 10
    element.text_rect = element.text_surface.get_rect(center=element.rect.center)
element.draw(screen)
j += 60
```

6. random

random.randint() is used within Python Alchemy to randomize positions of newly spawned item. Whenever a user selects an item from the sidebar, it is spawned onto the crafting area. To prevent new elements from always appearing at the same location (consequently on top of each other), random.randint() randomizes the location of the item spawning. This random spawning implementation within play() can be seen as follows:

```
for discover in discovered:
    if discover.is_clicked(event.pos):
        pygame.mixer.Sound.play(click_sound)
        elements.append(Element(400 + random.randint(-100, 100), 300 +
        random.randint(-100, 100), 140, 45, discover.text.title(), font, text_color, item_color))
```

Additionally, random.randint() is used to randomize the sound effects played whenever successfully combining items. This enhances the diversity of the auditory feedback, since listening to the same sound effect repeatedly would get boring. This variation adds a surprising amount of depth, and shows our group's attention to detail. This is done within the function play():

```
pygame.mixer.Sound.play(combine_sounds[random.randint(0, 2)])
```

4 Running Instructions

Although we have zipped the data in our submission (contained in compressed datasets.zip), it is not necessary for running Python Alchemy, as the datasets are pre-included within the assets folder.

The game is ran as follows:

1. Run the file main.py.
2. Press 'Play'.
3. Try crafting! Here is an example combination:

*Note that you will need to scroll the sidebar by pressing the arrow keys (up/down), or by clicking on the triangles that appear on the sidebar whenever you are able to scroll in a certain direction.

```
air + water = rain
earth + water = mud
earth + fire = lava
air + fire = energy
earth + rain = plant
air + lava = stone
```

```
air + stone = sand
mud + plant = swamp
fire + sand = glass
energy + swamp = life
air + life = bird
glass + sand = time
bird + bird = egg
egg + swamp = lizard
lizard + time = dinosaur
```

4. Clear the crafting area by pressing the trash can icon in the bottom left.

5. Press the Python Alchemy logo in the top left to return to the main menu.

6. Enter 'Options', click 'Save', then restart the game. Notice that your progress is gone, until pressing 'Load' in the options menu which restores all saved progress.

7. Enter 'Options', turn on Chemistry Mode, press 'Back', then 'Play' again.

8. Try a chemistry combination, such as this example:

```
oxygen + oxygen = oxygen gas
hydrogen + hydrogen = hydrogen gas
oxygen gas + hydrogen gas = water
hydrogen gas + carbon = methane
methane + hydrogen gas = ethane
ethane + hydrogen gas = propane
```

9. Return to the 'Options' screen (return to the main menu by pressing the logo in the top left), turn OFF chemistry mode, then add a custom item!

a) This is done by selecting two ingredients for item1 and item2, then typing the name of the result in item3. Click on item1, type in an item name, then press enter (or click on another item's box). Repeat the same for item2 and item3, then click on 'Add a New Combo'.

b) Make sure the ingredients item1 and item2 already exist! We suggest using the elementary items (fire, water, earth, air), since they are always accessible. For example, you could add 'fire + fire = big fire'. Make sure you click on 'Add a New Combo' after typing out the recipe.

c) Now, try it out! See if you can create 'big fire'.

10. You have successfully explored Python Alchemy! Feel free to quit the game by pressing the 'Quit' button in the main menu, or keep playing, if you'd like. Can you discover all 590+ items?

5 Changes Since Proposal

Initially, we intended to have a 'Flashcard Mode' which randomly selects a recipe and allows the user to guess the result. This was meant to use word association to promote learning through the crafting system. This was not prioritized within our project as the implementation was quite simple and rather unrelated to the main goal of the project, which was to create an interactive representation of a crafting system. As a result, Flashcard Mode was left out of the project and is left as a potential addition to Python Alchemy in future updates.

Our original idea for 'Chemistry Mode' was to allow the user to use more than three elements, and modify the chemistry environment (temperature, atmosphere pressure, etc). Chemistry Mode's final implementation was far simpler than we intended and is nothing but a proof-of-concept at the moment. It shows that Python Alchemy is able to modularly interpret different datasets for a wide variety of purposes, and is not arbitrarily limited to only alchemy. We were limited as a group by our lack of understanding of chemistry, which is why our implementation was left as a proof of concept.

In our proposal, our plan was to use csv files to represent the various datasets that Python Alchemy is able to read and manipulate. After beginning the implementation, we quickly realized that the JSON file format was far better for our needs. This change is further detailed in section 3.3 under '5. pygame'.

6 Discussion

6.1 Successful Implementation

Our goal for the project was to **implement a crafting system using graphs to foster creativity, and turn learning into a tactile gaming experience**. The results of our computations did help achieve this goal, as we were successfully able to represent a crafting system using graphs. Everything works as intended: you are able to craft freely, errors/crashes do not occur from incorrect user inputs, and progress can be saved at any time.

Since the recipes that Python Alchemy operates on can be modified so easily through the JSON file, the creative possibilities are unlimited. Anyone can see the provided recipes.json and immediately understand how it should be modified. Each user can have a unique use case for the program, and each user can find a way to implement Python Alchemy to fulfill its purpose in an adaptive manner. Due to this, we believe that Python Alchemy successfully fosters creativity.

In order to further establish Python Alchemy as an educational tool, we used many different techniques and methods. We allowed the game to be modular in design so that it could adapt to a wide variety of use cases. This means that our program can be applied as an educational tool for usages like chemistry, algebra, culinary studies, etc. Additionally, we added auditory feedback to further enhance the user experience for learning. Audio responses add an extra level of depth to every interaction, and the feedback given for a successful craft can be seen as a form of positive reinforcement. This also enhances the tactility of the game, as it makes your actions feel more impactful. In addition to the existing tactility from the drag-and-click implementation of our crafting system, we successfully turned learning into a tactile gaming experience.

6.2 Limitations

Some limitations that we found were mostly due to the modular aspect of our game. Initially, we wanted to have sprites for every item within the game. This would mean having nearly 600 sprites, but we were determined and willing to make it happen. However, during the implementation of the item editor, we realized that it would be very difficult to allow users to add their own sprites, and then correctly manipulate/edit them within the game so that they are in accordance (in resolution, centering, etc) with the other sprites. This would take away from the modularity of our game, which is a large part of Python Alchemy's identity.

Another area in which we fell short was the 'Chemistry Mode'. We intended to have different tools to control the crafting environment but fell short due to our lack of chemistry knowledge. Our dataset (chemistry.json) is also shorter than we intended, so it only stands to exist as a proof of concept. We also failed to allow crafts with an infinite number of ingredients. This proved to be more difficult than expected, so we decided to stick with binary combinations only.

Currently, adding a custom item recipe, saving a game state which contains that custom item, then loading that save after a game restart will result in the game to crash. This happens because the custom recipes are reset every time the game is restarted, and the only way to prevent this crash (currently) is by re-adding every custom recipe before loading the save. This limitation is due to the way we structured our saving process, and would have to be corrected.

6.3 Next Steps

In the future, we intend to rework how our program implements items so that they can contain sprites/icons. This would further enhance the user experience of the game. The complications regarding item editing by the user would have to be tackled through an in-game image editor/refitter, which would be challenging yet rewarding.

Additionally, with the help of someone more knowledgeable in chemistry, we intend to rework Python Alchemy's chemistry mode. In its current state, as nothing but a proof-of-concept, it is not complete enough for our standards. We intend to allow for multiple-item (greater than 2) recipes, environment control (temperature, air pressure, etc), multiple byproducts during reactions, reaction stages (pre-reaction, during reaction, post-reaction), as well as time-based reactions (progression throughout the length of a long reaction). These changes would further push Python Alchemy's wide applications, and stabilize it as a universally-applicable educational tool.

Currently, adding items does not save to the recipes.json file. This means that you would have to re-add all your custom items upon restarting the game. This could be fixed by adding JSON writing as a functionality to Python Alchemy.

7 References

baraltech. (2022, January 5). HOW TO MAKE a MENU SCREEN IN PYGAME! [Video]. YouTube. https://www.youtube.com/watch?v=GMBqjxcKogA

Claiborn, Samuel, et al. \Little Alchemy Cheats - List of All Combinations - Little Alchemy Guide - IGN." IGN, 17 Feb. 2021, www.ign.com/wikis/little-alchemy/Little_Alchemy_Cheats_-_List_of_All_Combinations.

Grow, April, et al. \Crafting in Games." Digital Humanities Quarterly, 2017, www.digitalhumanities.org/dhq/vol/11/4/000339/000339.html.

Pygame Intro | Pygame v2.6.0 Documentation. www.pygame.org/docs/tut/PygameIntro.html.

Tzachrista, Maria, et al. \Neurocognitive Profile of Creativity in Improving
Academic Performance|A Scoping Review." Education Sciences, vol. 13, no. 11, Nov. 2023, p. 1127.
https://doi.org/10.3390/educsci13111127.