**Process of chaotic chess**

**Basic properties of an item**

In order to explain what each item individually does; I will first describe what these items have in common. Because they all have the same underlying code for the generation, visualization, interaction and reappearance of item.

Generating the position of items randomly

Ein Bild, das Schachfigur, Brettspiel, Tabletopspiel, Spiele enthält.

Automatisch generierte BeschreibungIf I wanted chaotic chess to be a balanced game, I had to generate the items equally on both sides. To make this happen I declared two list of buttons. A button represents a field on the chess board. The white buttons only featured buttons from rows 1-4 and the black buttons rows 5-8. With these two lists, I could now tell the program to randomly select a button from each list, and on each side an item would appear. However, the problem would occur that the program selects a button that is already occupied by a chess piece. This way the item would overtake the piece and delete it. Since I only wanted the items to spawn on empty fields, I had to tell the program to make a random selection of the empty fields on each side. Luckily in the original program two lists called “position\_white\_players” and “position\_black \_players” were available. These lists contained the pieces of white and black and most importantly, their positions. This way I could tell the program that it should randomly choose a button to put the item on, but if that button is already occupied by a chess piece, it should randomly choose another button to place the item on.

White button list

Black button list

  while True:

    bomb\_button = random.choice(button\_list)

    bomb\_pos = getButtonPosition(bomb\_button)

    if bomb\_pos not in players\_pos\_list and bomb\_pos != endbutton\_pos:

      break

Embodiment of an item

Once the item has found a button to be placed on it should store the buttons position, since we will need to fall back on it in the future. Afterwards the item should be visually displaying its location to let the user now where it is. For the sake of simplicity and testing I started off by representing the item with a unique letter and if the item was on the white side, it was upper-case if it was on the black side, it was lower-case. Items on the white side were labelled with “\_1” and items on the black side were labelled “\_2” at the end of the item’s name.

With the command button.config(text = x ) I could edit the text that appears on the button. Since we stored the position of the randomly selected button earlier, we can edit the button that the item is placed on.

BOMB\_1 = "O" # white side bomb

# Bomb\_button is the randomly selected button, where we will place our bomb item on

# We visualize our bomb by editing the buttons text to the bomb symbol

bomb\_button.config(text=BOMB\_1)

Picking up an item

Next, we want to let the program now if the user picked up an item. For this to work, we can use our “endbutton\_text” variable again. By saying that if the “endbutton\_text” is equal to the item symbol, it should know that the user stepped on the item. Inside of this condition, we will code what each item will do if it gets picked up.

  # If text of destination field and text of bomb is the same

  if endbutton\_text == BOMB\_1 or endbutton\_text == BOMB\_2:

    # Let the user now that he picked up the bomb

    action\_label.config(text = "YOU PICKED UP A BOMB!", fg = "black")

Regenerating position of items randomly

With the next step I wanted to make the game more random and chaotic, so I decided to change the position of the items after a certain number of rounds. To make this happened I had to delete the original items place if it already existed and choose another position for it. We should also note that if the item has already been picked up it should not delete its former place since the player already removed the item from the board by picking it up. However, this code does not apply to the barrier, since it’s impossible to pick up the barrier and we always have to delete it’s former position. In order to delete the former item, we can make use of button.config(text = “”). The (“”) indicate to the computer that it should display a string, which is simply a text. By not entering anything between the (“”) it tells the computer it should edit the buttons text to nothing.

**Bomb**

When the bomb is picked up it has to color the fields surrounding the bomb black. In order to achieve this 3-by-3 black area I took use of the nested for loop function.

**Coin**

One unique feature about the coin is that it generates with a random choice of text. To get a random piece represented on the coin, I had to use the random module by importing it with “import random”. Now I could tell the program to make a random choice between three different figures including bishop, knight, and rook.

COIN\_LIST = [BB, BN, BR]

coin\_symbol = random.choice(COIN\_LIST)

With this code our coin item will pick a random symbol in coin list and use that symbol to be represented on the coin. Next up the program has to know if the coin is picked up, which symbol is represented on it in order to transform the piece that stepped on the field into the piece on the coin. For this I used “if functions”. At first the program has to determine which player stepped on the coin. In the original chess code, there was already a function that determine which players move it is. So if the coin is picked up when white is on the turn, the code should use the white pieces to transform the piece into the coin. Then when the program nows which color stepped on the coin, it should read what symbol is on the coin. We can use the endbutton\_text again. So if the endbutton\_text is the rook and the turn is white, it should turn the piece that stepped on the coin into a white rook.

  elif player\_color == 'W':

    if endbutton\_text == BB:

      endbutton.config(text = WB)

      figure.name = 'WL'

    if endbutton\_text == BN:

      endbutton.config(text = WN)

      figure.name = 'WS'

    if endbutton\_text == BR:

      endbutton.config(text = WR)

      figure.name = 'WT'

However, there is the same problem that occurs like with the bomb. The program only replaces the look of the piece but not it’s properties. So, although our piece now transformed into the coin piece, it still moves and behaves like before stepping on the coin. Therefore we have to now tell our code that the piece that stepped on the coin should not only take the symbol of the coin piece, but also take the values and properties of coin piece.

**Shield**

**Barrier**

The last item is the barrier. With the help of the code that was already in the original chess program, the coding of this item was simpler. To make the barrier work I had to use the function “check\_if\_move\_legit”, which calculates if a move is possible and the variable “endutton\_text”, which stores the text of the button that the user clicked on the second time. Within the “check\_if\_move\_legit” function, I coded that if the “endbutton\_text” is equal to the barrier’s text, the program should tell the user he can’t step onto a barrier and not allow this move by returning “False”. This then leads to an error and the program ask the user to play another move.

  def check\_if\_move\_legit(self, ps, pe, all\_positions\_enemy, all\_positions\_own\_team, all\_possible\_moves\_enemy, turn):

    # if user chooses barrier as it's destination field,

    if endbutton\_text == barrier\_text:

      # tell the user it cannot step onto the barrier

      action\_label.config(text = "YOU CAN'T STEP ONTO A BARRIER", fg = "red")

      # makes the move impossible to make

      return False