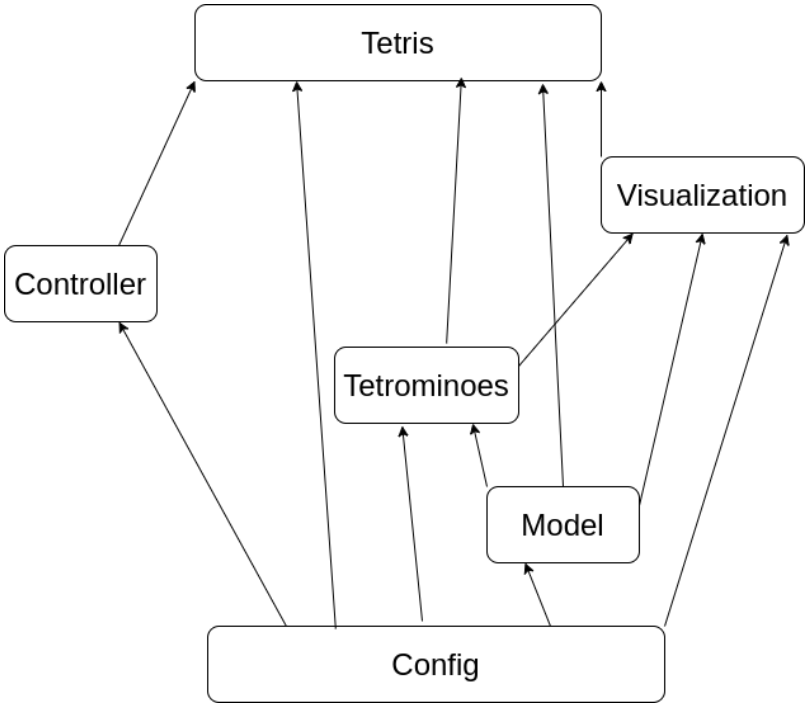
OCaml Tetris Writeup

Our Tetris implementation was primarily divided using the model–view–controller design. The “viewer” is written in `visualization.ml`, while the controller is written in `controller.ml`. Visualization encompasses the functions used to visualize the game, while Controller defines the functionality used for users to interact with it.

Due to the complexity of the game’s model, we further divided our model across three files: `tetris.ml`, `model.ml`, and `tetromino.ml`. Tetris includes the main game loops; Model defines the `model` type and associated functions; and Tetromino defines both the tetromino class and the square class of which tetrominoes are composed of.

A final Config file stores the initial configuration values and type definitions that the rest of the game draws from.

This overall structure of our project is visualized below



Model (model.ml)

One of the most important concepts in our implementation is the model, which represents the playfield or “matrix”. At any given time, our model stores the state of the board in a two dimensional matrix of integers using the default OCaml `Array` type. This form was decided upon in order to most directly represent the different colors of displayed squares. Thus, the elements of the array are initially set to 0, the value representing an empty square, and as the game progresses, these 0’s are replaced by hex values of the color corresponding to the tetromino placed.

Critically, the model does not store the current active tetromino under the player’s control. Instead, the model stores the environment while the active tetromino is treated as an independent entity. In this way,

manipulation of the active tetromino is streamlined and the model, which remains static for the duration of a single tetromino's "lifespan", is left unmodified for the said duration.

Another design decision in the implementation of the model was to define a model as an array of "rows". The model was defined in such a way in order to facilitate the clearing of lines, which folds over each row.

Additional functions

The `sq_full` function defined in `model.ml` simply takes in an `x`, `y` position and a model and returns whether the position is "occupied" in the model or is out of bounds. The only exception is if the piece is within the `x` boundaries but above the upper `y` boundary of the model. This was done in order to allow the piece to spawn on the 20th and 21st rows as per the standard tetris model - as well as a possible future implementation of a "vanish/buffer zone" (described here: https://harddrop.com/wiki/Playfield).

The `clear_lines` function takes a model as an argument and destructively clears lines (rows) that are fully occupied. After several attempts, we concluded that the most coherent and efficient way of clearing lines was by checking each row for the existence of an empty square, and if no empty squares exist, incrementing a `shift` counter (initially set to 0). The current and subsequent rows would then adopt rows `shift` positions up. Once this is complete, the top `shift` rows of the model are duplicated and thus the function finishes by setting the top `shift` rows to new arrays of empty squares. We lastly had the function return the final value of `shift` in order to have a basic scoring system. We are quite happy with this algorithm as it has complexity O(n), when originally we thought we would have to use something $O(n^2)$.

Tetrominoes (tetromino.ml)

In our definitions of tetrominoes, we decided to use object-oriented programming in order to more concretely encompass and group the structure and functionality of tetrominoes. Since we were creating many different types of tetrominoes that would all share the same functionality, through sufficient abstraction we would be (and were) able to efficiently define 7 subclasses of tetrominoes (the standard pieces) concisely from a robust tetromino superclass.

In our initial development, we started with basic single block "squares" which could only translate and not rotate. In this class we originally included only a method for motion and an auxiliary method which returned whether an action was possible. However, when considering the development of larger tetrominoes, we realized that characterizing tetrominoes by a list of `square`s would greatly simplify the movement and rotation of them.

Square class

In the final result, each square object is initialized with `x` and `y` coordinates. Alongside functions for retrieving and setting the position as well as "adding" a square to a model, the `square` class additionally contains a `move` method which takes a center position and an action. For translations the method simply returns shifted coordinates; however, for rotations the method rotates the position about the given center by 90 degrees in either direction. Thus the rotation of a square gains significance and the `move` method may simply be called upon by the tetromino class without having to distinguish rotations from translations.

Tetromino class

The `tetromino` class implementation always defines a center square halfway along the model horizontally (rounded down) and one square above the model. The class, moreover, takes as a parameter a list of positions of the other squares relative to the center (we decided to make these positions relative in order to make the tetrominoes less static and easier to modify). This list is then used to construct new squares, which alongside the center is compiled into a list of squares composing the tetromino in the initializer. We defined the `center` square as a value outside of the initializer, however, in order to more securely hold it and more easily access it later on (as compared to simply defining the center as the first square in the list).

For the `move` method, we had originally attempted to test whether an action was possible in the `move` method of each individual square and simply iterate over the squares. However we quickly discovered that doing so resulted in the pieces falling apart. Thus, (for actions other than `Drop` and `NoAction`) the method calculates the hypothetical shifted positions of *all* the squares and checks whether the corresponding squares in the model are occupied before moving the whole tetromino. This change from the original implementation likewise resulted in the `square` class's `move` method going from modifying the square to simply returning a new position. Of particular note is how the tetromino class's `move` method returns a boolean. This output returns whether an action was actually made in order to simplify the implementation of the actual game in `tetris.ml`.

From there, the seven distinct tetrominoes are simply defined as subclasses of the `tetromino` class and are characterized by a list of relative positions (as described earlier) and a unique color.

View (visualization.ml)

This is where all the gritty work happens drawing the board. Mostly self-explanatory, the functions here all basically do as they're named. We initialized our graph to contain a grid of the playfield and a surrounding rectangle, as well as "displays" above and to the right. The above display is used to view the controls, level (speed of gravity), and score (how many lines cleared). The display to the right houses the queued pieces, which at the present moment there is only one of since the generally slow movement speed makes having any more uninteresting for the game. The `render_model` and `render_text` functions were separated in order to more clearly encompass what each does.

Controller (controller.ml)

While the current implementation of the controller appears simple, a great deal of thought and time had actually gone into it. Due to the way the `Graphics` module implements `read_key`, there exists a delay between reading keypresses and thus the speed with which inputs can be taken is capped. The only possible solution we were able to come up with was using the `wait_next_event` function and including the `Poll` event which skips the wait and "returns immediately". However, again as a result of the `Graphics` implementation, doing so resulted in the keypresses being rapidly queued in a queue inaccessible to us. This had the effect that only the first keypress was read and constantly repeated for an extremely extended duration. As a result of this, we have used the current implementation and thus the current game has a somewhat limited movement speed.

Bringing it all together (tetris.ml)

In our implementation of the game, we decided to run everything in a statically true `while` loop. The game ends when a “GameOver” error is raised and later caught, outputting the final score in the terminal. This was done in order to keep the scope of the current piece limited to inside the loop, thus slightly simplifying the code. Inside this overarching loop, we implemented two more nested `while` loops, the surrounding one for “gravity” and the inner loop for user interaction. The outer loop continues shifting the current piece down until the piece can no longer do so. On each iteration, it renders the entire game (model, tetromino, and queued tetrominoes) and runs a nested loop that checks for and responds to key inputs from the user for a set duration. This duration was implemented using the system time rather than a delay in order to continue receiving user input. We also included a case for a “Drop” input such that once dropped, the piece wouldn’t have a delay at the bottom before locking.

When receiving keyboard input, we use a polling technique using the `on_capture ()` function. A while loop runs the `on_capture ()` function constantly in between ticks. Because of the ticking nature of the game, if we used a blocking technique, we would still need to run a timer of some sort in the background in order to reset the blocking call. Thus, we found that the polling technique works well for our purposes.

Config (config.ml)

Beyond the basic configuration values for the model and gameplay, we additionally included in our `configuration` file values for the different colors used. While this could be hardcoded into the tetromino definitions, we felt putting values in config.ml would make the code more readable and additionally make it easier to reconfigure the colors should the need arise. (Ya seriously edward why did you pick such a bright yellow)

Tetris Gameplay

Enjoy! We had a lot of fun procrastinating other finals by “playtesting.” Move with WASD and rotate with “<” and “>”. The game speeds up pretty quickly. Our highscores are 36 and 56 (Felix and Edward respectively).

Controls: “W” : hard drop (drops and locks immediately) “S” : soft drop (moves down one square) “A” : left “D” : right “<” : counterclockwise rotation “>” : clockwise rotation