Tetris in Haskell - PKD Spring 2020

Robert Martinis, Niclas Gärds, Jakob Paulsson, Ludvig Westerholm

February & March 2020

Contents

1 Introduction

1.1 Tetris

Tetris is a very well known game founded in the year of 1984 by Aleksej Pazjitnov in Russia, then known as the Soviet Union. Since its release, it has sold over 170 million copies [4].

Tetris has 7 different tetrominoes (playing blocks) that each consists of 4 squares "glued" together. All 7 tetrominoes have different shapes and different color. Together, they make up all different combinations of shapes that's possible to make with 4 squares, given that rotation is possible. All tetrominoes have their own names, and they are named after a character in the alphabet they resemble, which can be seen in figure 1.

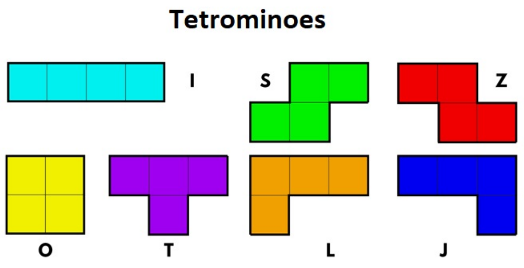


Figure 1: The seven different tetrominoes

The objective of the game is to reach the highest possible score. To do this, the player needs to strategically place falling tetrominoes to build full rows, which clears those rows and grants the player a certain score amount for each row cleared. If the rows stack up to the top of the playing field, the game is over and the score is shown.

The player can always see which three tetrominoes are coming after the current tetromino, and every time a tetromino is placed, the next tetromino comes into play. In modern versions of Tetris, there's also a feature in the game that allows the player swap the current tetromino for the supposed next one and replacing the stored block with the current block in a "hold" window. On startup, the hold window is empty and the first swap simply places the current block in the hold window and the next block comes into play. The player can see the tetromino that is currently held and every time the player swaps block, the stored tetromino comes into play instead of the current tetromino.

1.2 Tetris in Haskell

This implementation is made using Haskell which is a functional programming language. The Gloss library [2] is used to render the game with graphics. This version was developed during a time period of approximately three weeks in a group of four people. Github [1] was used to document changes in the code as soon as any group member introduced new code into the game.

1.3 Why Tetris?

The reasons for choosing this idea as the project is simple. The group enjoys playing the game and found it sufficiently challenging to make and a project of this size and difficulty leads to a great learning experience. One could argue Tetris is too hard, but since this group is made of four people, it makes sense to pick a larger project than those of groups of three.

2 Summary

The game will let the user play a regular game of Tetris with a few differences. User-inputs from the keyboard are used to specify what moves to be made. The validity of a move will be checked before being called upon, which will prevent cheating and crashing the game. The game will be rendered to the screen from values of the current state of the game.

Using the Gloss library in Haskell, the program is able to render the game using colors and shapes, which is essential to Tetris. The play field will be rendered from a list of lists containing information about the placed tetrominoes, as well as where the play field is open. The current falling tetromino will also be rendered based on its coordinates and block type.

Every user input will call for a function checking the validity of the input, and if the input is invalid the program will just keep going, meaning that the current block will keep falling, acting as if no input was made.

3 Use cases

In this section, the user inputs and software requirements will be presented.

3.1 User inputs

The game has 7 different user inputs:

- **Arrow key up**
  Pressing the up arrow makes the current falling tetromino rotate 90 degrees clockwise.
- **Arrow key down**
  Pressing the down arrow makes the current falling tetromino fall one step further down.
- **Arrow key left/right**
  Pressing the right or left arrow moves the current falling tetromino one step to the left using the the left arrow, and the same for right, but in the opposite direction.
- **'R' key**
  Pressing the 'R' key resets the game, meaning all current tetrominoes on the board are cleared as well as resetting the current score, line score and level.

- **'C' key**
  Pressing the 'C' key swaps the current falling block with the next one in the queue. It's only possible to swap a tetromino once until the current block is placed.

- **Spacebar**
  Pressing space bar instantly moves the block down to where it would naturally fall if the user didn't do anything.

3.2 Software requirements

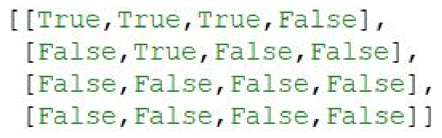
The group chose to compile the game into an .exe file for Windows users and an executable for Ubuntu. This means that the only requirement for running the game is having either one of those operative systems. The game is also available in a .hs file for documentation purposes and any Haskell compiler will be able to run the code and play the game if the Gloss library is available on the computer.

4 Program documentation

4.1 Description of data structures

The data structures consist of six types and one data type:

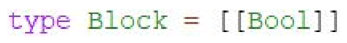
- **type Block**
  The Block type is a list of list of Booleans. A Block represents a tetromino built inside a 4x4 grid. The Block type is meant to have four lists inside the main list to build a 4x4 square. Shown in figure 2 is an example of a tetromino that was created using the Block type.



```haskell
[[True,True,True,False],
 [False,True,False,False],
 [False,False,False,False],
 [False,False,False,False]]
```

Figure 2: Example of a T-tetromino

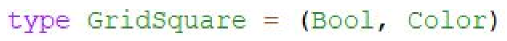
  In earlier versions of the groups game, the group tried to create the Block type using a four-tuple where each tuple is also a four-tuple. This allowed for strictly defining the amount of elements that the type could handle but ultimately it was decided to stick with lists as it is easier to work with recursion using lists. Declaration shown in figure 3.



```haskell
type Block = [[Bool]]
```

Figure 3: Block type declaration

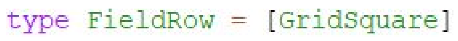
- **type GridSquare**
  The GridSquare type is a tuple where the first element in the tuple is a Boolean and the other is Color (imported from the Gloss library). One GridSquare represents one square in the playing field. The Boolean shows whether there is a block or not, and the color is either black if there's not a block in the square or the color of the block. Declaration shown in figure 4.



```
type GridSquare = (Bool, Color)
```

Figure 4: GridSquare type declaration

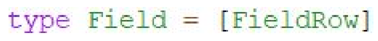
- **type FieldRow**
  The FieldRow type is a list of GridSquare and represents one row of squares in the playing field. Declaration shown in figure 5.



```
type FieldRow = [GridSquare]
```

Figure 5: FieldRow type declaration

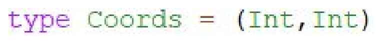
- **type Field**
  The Field type is a list of FieldRow and represents all rows of squares in the playing field. Declaration shown in figure 6.



```
type Field = [FieldRow]
```

Figure 6: Field type declaration

- **type Coords**
  The Coords type is a tuple where both elements are Integers. They represent coordinates in the playing field. The origin (0,0) is placed in the upper left corner of the playing field. Declaration shown in figure 7.

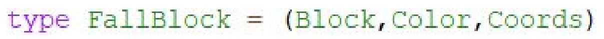


```
type Coords = (Int,Int)
```

Figure 7: Coords type declaration

- **type FallBlock**
  The FallBlock type is a three-tuple where the first element is of the type Block, the second is of the type Color and the third is of the type Coords. FallBlock represents a falling block in the playing field where Block represents the shape of the tetromino, Color represents the color of the tetromino and Coords represent where the tetromino is on the playing field. Declaration shown in figure 8.

- **data GameState**
  The GameState data type is a dictionary that contain different values. Those values are things that the game constantly needs to change or update as the game progresses.

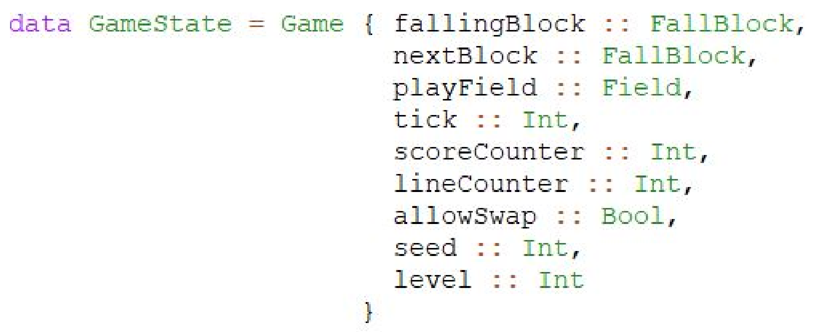


```
type FallBlock = (Block,Color,Coords)
```

Figure 8: FallBlock type declaration

1. **fallingBlock**
   fallingBlock is the current tetromino that is falling in the playing field and is of the type FallBlock.
2. **nextBlock**
   nextBlock is the next tetromino in the queue that comes in play as the current tetromino is placed in the playing field or if the user swaps tetrominoes. It is of the type FallBlock.
3. **playField**
   playField is the current playing field and is of the type Field.
4. **tick**
   tick keeps track of when the game should progress and is of the type Int.
5. **scoreCounter**
   scoreCounter updates the score as the user places the tetrominoes in an order that creates one or more full rows in the playing field and is of the type Int.
6. **lineCounter**
   lineCounter updates a value that represents how many rows the user has cleared during the game and is of the type Int.
7. **allowSwap**
   allowSwap shows whether the user is allowed to swap blocks or not and is of the type Boolean.
8. **seed**
   seed is used to create a pseudo-random number to generate tetrominoes. Every user input updates the seed to simulate randomness and every frame increases the tick. It is of the type Int.
9. **level**
   level updates a value that represents how far the user has progressed in the game. The level increases for every fifth row that the user clears. Level is of the type Int.

Declaration depicting the GameState datatype can be seen in figure 9.



```
data GameState = Game { fallingBlock :: FallBlock,
                        nextBlock :: FallBlock,
                        playField :: Field,
                        tick :: Int,
                        scoreCounter :: Int,
                        lineCounter :: Int,
                        allowSwap :: Bool,
                        seed :: Int,
                        level :: Int
                      }
```

Figure 9: GameState data declaration

4.2 Description of algorithms and functions

All of the main functions works with the data type GameState as the input and most of them outputs GameState aswell.

A rough summarisation of this Tetris can be seen in figure 10, with all functions of Tetris being used.

- **randomBlock**
  The randomBlock function takes a seed of type Int as an input, which is used generate a block, then returns a FallBlock. The randomBlock functions generates the next falling block.

- **rotateBlock**
  The rotateBlock function takes GameState as an input and outputs an updated GameState. The function extracts a block from fallingBlock inside of GameState and uses a help function to rearrange the elements inside of the block to match one rotation.

  One type of tetromino is excluded from this function using another help function because rotating that block wouldn't change anything about the blocks properties since it's a square. Originally, the block was placed in the middle of the 4x4 grid which meant that rotation of that tetromino wouldn't change it's position if the tetromino were to be rotated. It was eventually realized that all tetrominoes had to have at least one square in the top of the 4x4 grid for the gameOver function to work properly so that's why an exception was made in this case.

- **fallStep**
  The fallStep function takes GameState as an input and outputs GameState. The function extracts the coordinates from fallingBlock inside of GameState and changes the value of the y coordinate by 1.

- **tryMoveLeft and tryMoveRight**
  The tryMoveLeft and tryMoveRight functions takes GameState as inputs and outputs GameState. The functions extracts the coordinates from fallingBlock inside of GameState and changes the value of the x coordinate by 1 (right) or -1 (left).

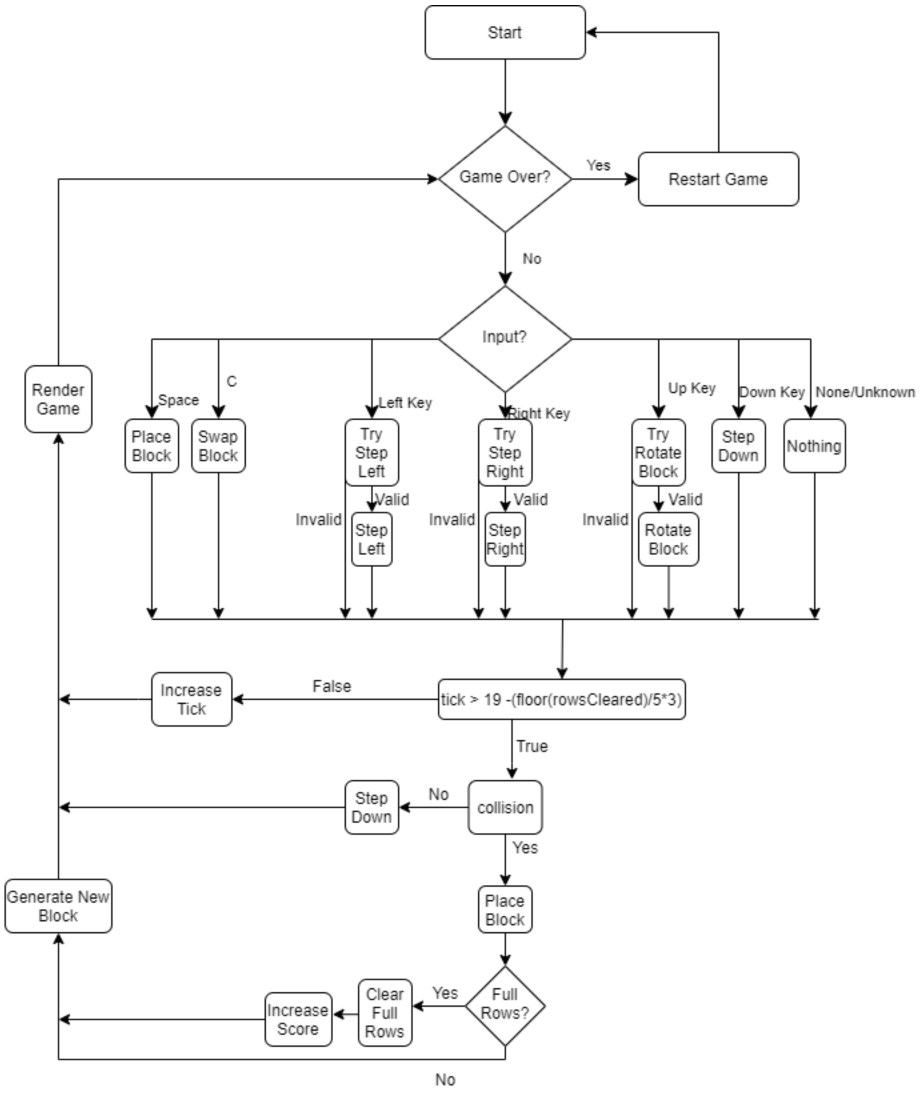


Figure 10: Flowchart of all functions

- **placeBlock**
  the placeBlock function places the currently falling block onto the play field. The coordinates of the falling block is extracted, and recursively going down into the play field, it will stop at the row where it is supposed to be placed. Thereafter it will swap out the next four rows with information about where the new block is placed and its color, then update the play field accordingly.

- **renderGame**
  The renderGame function takes GameState as an input and outputs Picture. The game is rendered using squares which the createSquare function does. It will create a 25x25 pixel square at the given coordinates. renderGame consists of seven different pictures. When combined, the function renders the current GameState to a picture that can be seen by the player. renderGame consists of:

1. **play field**
   The play field is rendered as a 10x20 grid, which is built up from a two-dimensional list of Booleans and Colors. Each row is rendered at a time with a 30-pixel offset between all the rows in vertical direction. The rows themselves will also generate squares using the createSquare function with a 30-pixel offset between them horizontally.

2. **grid lines**
   The grid lines are rendered by creating the vertical lines and horizontal lines separately. Both types of lines are first created using one base line of each type (horizontal and vertical). The base lines are recursively called for to produce new ones, as well as moving the new ones using the translate function.

3. **score counter**
   The score counter is rendered by calling for the updateScore function in conjunction with the show function and translate function to place it in the correct place.

4. **line counter**
   The line counter is rendered by calling for the updateLines function in conjunction with the show function and translate function to place it in the correct place.

5. **level**
   The level is rendered by calling for the lineCounter in the GameState divided by five, because the level increases every five rows cleared. The show function is used to display level as a text, which is then translated to the correct place.

6. **falling block**
   The falling block is rendered using the help function gridFromBlock. gridFromBlock takes a fallBlock as an input and using the blockRow help function that uses the createSquare help function, it returns the block as a Picture.

7. **next block**
   The next block is rendered using the help function gridFromBlock. gridFromBlock takes a fallBlock as an input and using the blockRow help function that uses the createSquare help function, it returns the block as a Picture.

- **collision**
  The collision function takes two types of Block as inputs and outputs a Bool. One of them is from the currently falling block and the other one is a part of the playing field to check for collision. If two elements from both types of Block are true in the same position, the function will return True, otherwise False.

- **nextBlockPos**
  The function nextBlockPos is a function that is used in our collision function. It extracts a part of the play field from a given coordinate for use in checking collision

- **moveRows**
  The moveRows function takes GameState as an input and outputs GameState. It works in four steps:

1. **clearRows**
   The clearRows function works by calling for a helper function that checks whether there are any full rows and returns the first full row in the field and removes that field and recursively works it's way through the entire field until it returns an empty list.
2. **rowsMissing**
   The rowsMissing function works by using an accumulator with a starting input as 20 to check how many rows are missing out of 20. It recursively calls for itself and simultaneously and subtracts one from the accumulator for each recursive step and finally returns the accumulator value that represents how many rows are missing.
3. **addRow**
   The addRow function takes an accumulator as an input which represents how many rows are missing to check how many rows the function should add to the field. It recursively adds rows until the accumulator ends up at a value of (-1).
4. **moveRows'**
   The moveRows' is a simple helper function that combines the helper functions to successfully update the field with all full rows cleared and new ones added. The main function moveRows then calls for function.

- **resetBlock**
  The resetBlock function takes GameState as an input and outputs GameState. It's a function which purpose is to call for other functions to update values inside of the GameState. It updates the current block to the next block and makes the next block into a random block using a pseudo-random seed. It also calls for the functions updateScore and updateLines to check whether the score and/or the linescore should be updated. It also sets allowSwap to True as the player is allowed another swap after each tetromino has been placed.

- **updateScore and UpdateLines**
  The updateScore function works using two helper functions. The first one checks if a row is full (isFull) and is used within the other helper function (fullRows). fullRows uses two accumulators that together keeps track of how many lines are cleared and adds ten points for each row cleared. The other accumulator is a multiplier that adds one for each value. Finally it multiplies those two accumulators together to get a final score.

  The main function updateScore calls for newScore that in turn calls for the helper functions to get a score. If there are no full rows, the fore mentioned accumulators will be 0 and no score will be added.

  The updateLines function works very similarly but instead of updating score it updates the linescore and the helper function only uses one accumulator which adds one instead of ten because it only needs to keep track of how many rows are cleared.

- **increaseTick**
  The increaseTick function takes GameState as an input and outputs GameState. the function is called for counting down frames until the next action is supposed to be executed. increaseTick increments tick in GameState by one when it's called for.

- **resetTick**
  The resetTick function takes GameState as an input and outputs GameState. The purpose of resetTick is to reset the ticks in the game so that new events can take place.

- **checkTick**
  The checkTick function takes GameState as an input and outputs a boolean. When the tick reaches a certain amount of ticks, the checkTick functions returns True or false, depending on the level, which is determined by the amount of cleared rows.

- **tryRotate**
  the tryRotate function takes GameState as an input and outputs GameState. It uses the rotateBlock function in conjunction with the collision function to check whether a collision would occur using the current falling block and if it does, it increases the tick, otherwise it rotates the block using the rotateBlock function.

- **tryMoveDown and instaPlace**
  The tryMoveDown function takes GameState as an input and outputs GameState. Similarly to the tryRotate function, is uses the nextPosInField function in conjunction with the collision function to check whether the tetromino if it collides with anything one step below the tetromino. If it does, the function calls for the resetBlock and placeBlock function to place the block and spawn a new block. It also calls for the gameOver function to check whether the game is over and moveRows to check whether any rows should be cleared. If it does not collide, the fallStep function is called for to move the tetromino down one step.

  The instaPlace function works like the tryMoveDown function if a collision is detected. However if a collision is not detected, it recursively calls for itself with the fallStep function which moves down the tetromino one step until a collision is detected.

- **swapBlock**
  The swapBlock function takes GameState as an input and outputs GameState. It extracts fallingBlock, nextBlock and allowSwap from GameState and changes the current block to the next block aswell as changing allowSwap to False until a block is placed.

- **gameOver**
  The gameOver functions takes GameState as an input and outputs GameState. The function uses a helper function that checks whether a row in the playing field has any blocks in it. gameOver calls for the helper function with the head of the playing field which is the first row. If that row has any blocks inside of it, the game is restarted.

- **event**
  The event function takes Event and GameState as inputs and outputs GameState. It

handles all the inputs from the player and calls for other functions that are bound to the inputs.

- **auto**
  The auto function takes Float and GameState as inputs and outputs GameState. The purpose of auto is to make sure that if no input through event is done, something will happen to the game regardless, to prevent the game from "freezing" until an input is made. If no input is made and checkTick has not yet turned into True, it will call increaseTick. If checkTick however is True, auto will call upon tryMoveDown.

- **main**
  The main function is the building block of the game. It uses the play function inside of the Gloss library which allows for building a window, setting the framerate to the game, setting the initial GameState, rendering the game, handling inputs aswell as making sure that the game keeps progressing.

5 Known shortcomings of the program

- **Game menu**
  The idea was to give the player a prompt when the game first ran. Giving the player the options to play, check a highscore and to quit the game.

- **Rotating**
  In the original Tetris, the player can always rotate a tetromino, even if the tetromino is on the side of the field. In this implementation, the player can't rotate the tetromino when it's placed to the far right or far left because there will be a collision With the wall. Another problem is that the t-tetromino doesn't rotate as intended because of a limitation with the gameOver function which requires all tetrominoes to be placed in a 4x4 grid instead of rotating around it's center block which it does in the original game.

- **Storing a block in game**
  Playing the original Tetris, the player can swap the current block with the next up-coming block, or with the already stored block. This function is not fully developed in this implementation as the current swapBlock function removes the current block entirely from the game and swaps it with the next one in queue. Storing is therefore not possible.

- **Seeing where the block will land**
  In the most recent versions of Tetris, the player can see where the tetromino will land and how it will look like as it is placed, by seeing a hollow version (ghost block) of the current tetromino on the bottom of the playing field.

- **Holding down keys**
  A difference from this Tetris and most other Tetris implementations elsewhere, is that in this version, the player can't hold down keys to travel in any given direction. The

player always needs to press the keys for something to happen, in contrast to other versions where the player could hold down the right or left arrow key to move the tetromino to the edges. In most modern versions, the player can also hold down the down arrow key to place the block faster.

- **Scoring system**
  The current scoring system in this version is a simplifed version of most modern scoring systems. For example when playing Tetris at [3], the score is highly dependant on how fast the player place the blocks as well as how many rows are cleared one time. It's also dependant on the level the player is at, meaning if the player is at a higher level, the score would rise more for each row cleared. In this version, the scoring system is only dependant on a base score for each row as well as a multiplier if more than one row is cleared.

- **Show score when game is over** Currently, the game doesn't show the players score when the game is over. Instead, it immediately restarts the game, thus restoring the score to zero.

- **High-score ladder** In most Tetris-games, once the play loses, the player can choose to save the game score for others to see, but this Tetris doesn't have any high-score ladder.

6 Discussion

6.1 Problems during development

There has been a few problems during the development of the game. Most of the problems stems from the short amount of time spent on planning before starting on the project. This lead to an end result that worked somewhat like the group wanted but with a few things not being optimal. This problem could've been solved if we had more time.

The order in which the functions were called for, are not as the group currently desires and leads to unnecessary workload for the computer running the program. Due to a lack of time, we allowed this to stay in the final version, since it does not hinder the program to function. But given more time, an updated could be made, looking cleaner, running smoother as less commands would be running simultaneously.

With more planning, the group would implement all code into the IO monad so that the program could use more convincing randomness when spawning the tetrominoes. Instead, all code were written pure, meaning they don't accept IO datatypes and it was realized too late so the changes that would be needed now would be too big to be implemented in a short amount of time.

The time spent creating functions that would later be scrapped was also greater than desired. The group needed a more concise plan for exactly how the game would work before starting the development. Perhaps by creating the flowchart before development was started would've yielded a better end result. In contrast to creating a plan first, we just created what we needed at the time and then thought about what we might need in the future.

6.2 Conclusion

The groups plan was to first make a very basic version of the game and depending on the amount of time left over until the deadline, the group developed more features to the game with the goal to eventually end up closer and closer to the modern version at Tetris.com. The end result came out well considering that most of the features shown in the goal version were developed in the arguably short time frame that was given.

References

[1] Github. `https://github.com/`. Besökt 2020-03-05.

[2] Gloss. `https://hackage.haskell.org/package/gloss`. Besökt 2020-03-05.

[3] Tetris game. `https://tetris.com/play-tetris`. Besökt 2020-03-05.

[4] Emily Gera. This is how tetris wants you to celebrate for its 30th anniversary. `https://www.polygon.com/2014/5/21/5737488/tetris-turns-30-alexey-pajitnov`. visited on 2020-03-05.