Luca Bruni

Mr. O’Connor

TEJ2O1 – Culminating Activity

29 January 2016

**Part 1: RESEARCH**

*Brainstorming*: My culminating activity will be based on the programming language “Turing”. The idea that I proposed was to make a working game. After scanning through ideas on my phone, I decided to create a replica of the popular game “Tetris”.

*Skills*: The skills I will exert for this project include all knowledge regarding the programming language “Turing”. I will demonstrate these skills through the commands in the program that I tell the computer what to output. In this case, it will output a working game of Tetris.

*Identification*: Turing program; Working computer game; Replica of Tetris

*Explanation*: How will my program work? Once you open the software “Turing”, open the file “culminatingTaskCode.t”. After you run the program, you will have the option to begin the game, or view a manual on how to play and the controls for the game. Once you begin the game, a classic Tetris-based map will display. Your first piece (chosen at random) will begin to fall down the screen. You may move the piece left using the ‘A’ key, and right using the ‘D’ key. You may also rotate the piece using the ‘Space Bar.’ You may press the ‘S’ key to increase the speed at which the piece falls if you are impatient. When you create a line using the Tetris pieces, the line will clear, causing all above pieces to fall if there are any. In turn, this increases your score. Once a piece reaches the top of the map, it is game over. This game is particularly useful since it may be a source of entertainment. It also has been proven to train your brain.

*Commercial* *Potential*: Despite commercial success of the original Tetris, many online and application remakes of the game have made many other versions available. There will not be any commercial potential for my game since it costs nothing to create, and there are many other games of competition available.

*Knowledge* *required*: To create a game like I made, you must have thorough knowledge of the “Turing” language. This means you must know commands such as: put, cls, drawing commands, values, variables, procedures, arrays and loops. You must be able to combine these aspects of code, along with others, to allow the code to create something. You must understand how code runs from top to bottom, and you must configure the loops accordingly.

*My* *experience*: I do have experience needed for the project. To begin, I’ve had Turing experience beginning in the summer with the basics, learning how outputs using commands in the language. Upon taking interest in Turing, I decided to take Computer Technology in school. Aching to learn more about programming, I continued practicing by making more complex programs along the way. Going into the programming unit, I learned more extensive knowledge about the program, remaking game mechanics for games such as snake, pong, and pac-man. However, despite my experience with the basics, I will have to do further reading to make my code as efficient as possible, and easy to make any kind of change. I will also have to learn how to organize my code, not only for the good of the eye, but for additional code to be inserted in the program.

**Part 2: PLANNING**

*Supplies*: Since this is a computer program, the only supplies you would need would be a working version of the programming language, “Turing”, and of course a working computer.

*Schedule*: The project was given on 1/18/2016, and is due on 1/29/2016. I began the project on 1/19/2016 (a Tuesday)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Date | Jan.  19 | Jan.  20 | Jan.  21 | Jan.  22 | Jan.  23 | Jan. 24 | Jan. 25 | Jan. 26 | Jan.  27 | Jan.  28 | Jan. 29 |
| Class | Receive Project  Identify project idea  Type up part 1+2 | Continue to buff up part 1+2  Attempt map creation | Finish part 1+2 complet-  ly  Attempt to make pieces | Finalize piece images  Finalize piece move-  ment | Continue to work on line making and ending the game | Add interface for scores and levels to map | Make sure the game is working properly  Fix any issues | Add potential code up-grades  Finalize bug fixes | Begin Part 3 typing  Acquire screen-shots | Prepare for in-class presen-tation | N/A |
| Home | Begin initial code | Finalize variables and map | Make pieces rotate  Make pieces move down upon a time counter | Finish proced-  ures for  movement  Begin to work on the mecha-  nics for  line making | Finalize how the game works generally  Attempt to add: scores, manual page, | Finalize variables for scores and level system | Change code for more ergonomics  Fix further issues  Beta test game | Last test for issues or last minute changes | Finish part 3 | Practice for the class | N/A |

*Interacting* *Features*: Press ‘A’ to move piece to the left. Press ‘D’ to move piece to the right. Press ‘Space Bar’ to rotate the piece clockwise. Press and hold ‘S’ to move the piece down quickly. Press ‘P’ to pause the game. You may also choose to look at the manual for instructions and controls, or to begin the game.

**Part 3: Building/Prototyping**

*Steps on how to replicate my program*

Set the screen graphics size to 270x440. setscreen ("graphics:270;440") or any other size that fits you.

As you will need the mouse, include variables for the x and y of the mouse, its buttons, and characters of the keyboard. (var x, y, button : int)(var chars : array char of boolean).

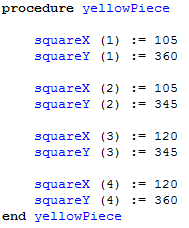
If you want to use fonts, make sure to create variables for different fonts you plan to use in the program. (var font1 := Font.New (“*fontname:fontsize”).*

Create an array for the map at a reasonable size for guiding your piece.

Using the map array, draw the grid accordingly with an appropriate size. Here’s an example:

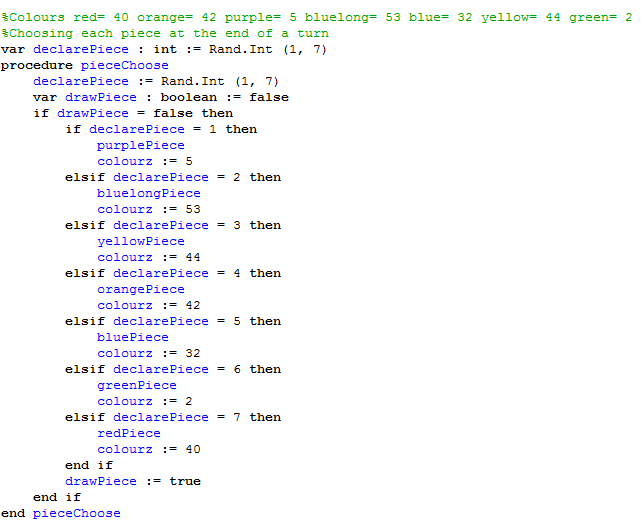
https://i.gyazo.com/12a33d651764d38e0710ecb3504bec94.png

Create 2 arrays (1 .. 4) for the X and Y location of each cube of each piece. Note that there are 4 cubes per piece, and you may want to organize them as left, right, top, and bottom, or anything else that’s applicable. (var squarex, squarey : array 1 .. 4 of int)

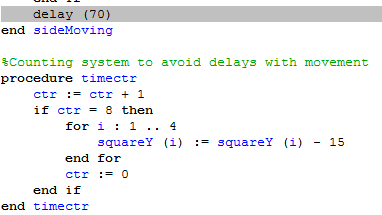
Draw the pieces’ start locations. There will be 8 parts to each drawing. Each cube of the piece has an X and a Y, and there are 4 cubes. Shape each piece accordingly using locations of each X and Y for the individual cubes. Put this into a procedure for later. Here is an example of the yellow cube.

Now you must colour each piece. Create a variable as the data type of int. Each colour on Turing has its own integer value, so you may use a variable to declare each colour and assign it to the corresponding piece. For example: (var colours : int)

Once you have finished creating the map and the pieces, you will now work on the mechanics of the pieces. First, we need something to declare a piece in order to select at random, as well as assigning the colour.

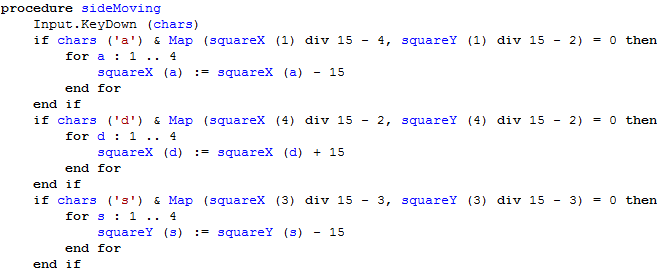


Previously was an example on how to create a system which will: choose 1 of the 7 pieces at random, and once done, will call on the procedure of which you drew the corresponding picture, assigning it its proper colour.

Now, you must create the movement for the pieces. This includes the automatic movement going down, and the manual movements: side-to-side, and down quickly. Creating a *counter* for the program is very useful. It allows you to use the current program delay to create even more of a delay for something else. For example, you want to be able to manually move faster than your piece automatically moves down, so you feel free. Here is an example as to how you can use a counter system to *delay the delay*. 

Above, I created a variable for the counter, in which at the speed of the delay, will count up to 8. Once it reaches 8, it will move the Y location of all the cubes in a piece down by 15, which is the length between a cube in the map to the next. After this, the counter resets back to 0, and the process will repeat. This is responsible for the automatic downward movement.

Now, you must create the manual movement of the pieces. In my program, I used A to move left, D to move right, S to quickly move down, and SPACE to rotate, which I’ll get into later. Use Input.KeyDown (chars) to initiate the keyboard’s ability.



This example shows how I played with many values to properly work my side movements to correspond to the cube variables (squareX and squareY) to move accordingly.

We will get into rotating the pieces later. For now, I’d like to show you how to make the game more organized.

If you want music to be played, download a soundtrack and place it in the same location/folder as your program. You can make a procedure to call on the music.

https://i.gyazo.com/c829c612992a5f9bffccb1572115c339.png

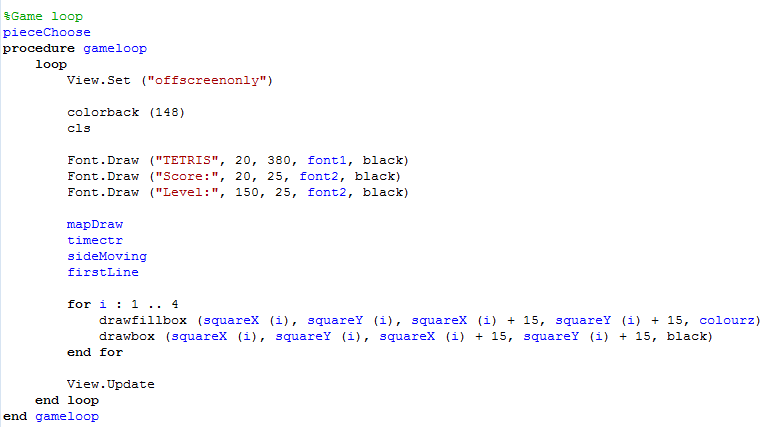
Make sure the file name in quotations matches the name in the folder.

Now you must make a lobby system. Assuming you are experienced with the basics of Turing, I won’t go into immense detail. You must make an organized lobby system. What I mean by this is that you should have buttons in which you can begin the game, view the controls and instructions. You may create a GUI button, or simply create a makeshift button in which if the coordinates of the “button” are matched with the mouse’s x and y, and RMB is pressed, then something will happen.



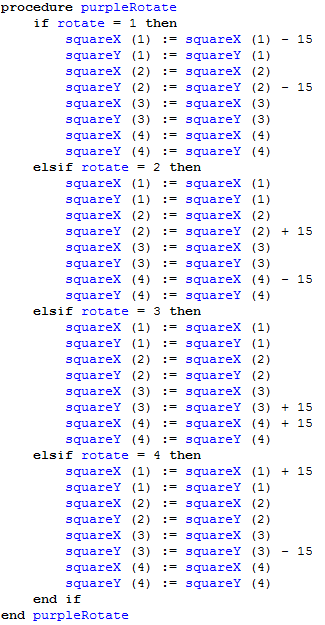
In my case, I have 3 buttons: 1 for start, 1 to view instructions, and 1 to view controls. Clicking one will activate something. It is up to you to decide what activation takes place.

The most important step is to make the master game loop. This loop will include the procedures and drawings. You will need to visualize movement. Type View.Set ("offscreenonly"), followed by ALL procedures which include drawings (in the correct order at your discretion, of course) and additional drawings. After this, View.Update must be typed in the loop.

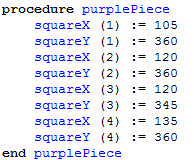


The example above shows a master game loop, which includes the View.Set process. Below all of the needed procedures, I have included the drawings of the pieces which will loop over.

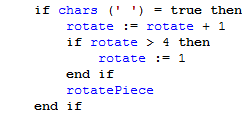
You can rotate pieces by changing values to the individual cubes. It is a lengthy process, but it works well in the end. Here is an example as to how you can play with the values of the singular cubes.



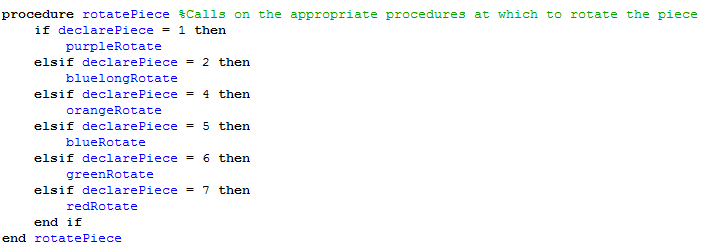
C:\Users\Luca\Pictures\^EC0714D7180CBA860F2CF738692497E68DE7030C2C75F5BAC0^pimgpsh_fullsize_distr.jpgAs you can see, I have the variable “rotate”, and simple if statements confirming if rotate is one value, then the values of squareX/Y change. The way I recognized this was by using Microsoft Paint, creating a replica of the locations of my pieces on a makeshift grid, and moving the individual cubes as I scanned through each rotation period. I looked at the initial coordinates on my main piece make-up procedure as such:



The bracketed numbers indicate the cube. I identified the far left cube as 1, the far right cube as 4, and the alternate cubes of each piece as 2 and 3.



This code example shows that you can use Input.KeyDown (chars) to enable the keyboard, and providing an if-statement that if the space-bar is pressed, a piece will rotate (roughly 90 degrees) based on the locations I set in those procedures. I can call them on using the following code:



This procedure uses if-statements to call on, or enable the procedures at which the cooperative piece rotates. Earlier, I explained how you can use a procedure like declarePiece (in a picture far, far above) to determine which piece comes next. Following this, you can now determine how that specific piece rotates using above information!

These steps will help you as a template to begin your adventure in coding your own Tetris game. Everything listed here is what you need, but it’s really up to you to play with certain values for your own tastes, organization and comfort zone using Turing.

*Changes made as I further tested my design*

* When I first began coding Tetris, I used many different drawbox and drawfillbox commands. I realized that this was EXTREMELY inefficient, even if it DID work. Using arrays with a cooperative map is much easier, and the values are able to be changed quite swiftly. I realized this midway through drawing the boxes of many locations of cubes for the pieces – it just took way too long, and I thought about how much LONGER it would be to move those pieces, as well as rotate them.
* As I tried to implement my music, I used a *process.* Processes in Turing should be avoided, unless they are to do with music. However, in newer versions of Turing like I have, music is much more easily playable using just a 3-line if-statement, as opposed to a complicated process that isn’t cooperative with other lines.
* At the end of the game, I had my program exit completely without even showing the final score, or that you lost – it just abruptly ended. I realized this wasn’t right for a computer game. I quickly messed around with the lobby system and added a loop that will show the final game score when you lost for about 5 seconds, that carries you back to the lobby where you can once again see the manuals, or start a new game.
* Using delays, I quickly found out that in the main loop, if you include a delay, it will effect ALL activities. When I learned that moving side-to-side was the same speed as the automatic movement of the pieces down, it was bugging me to change it. So I implemented a counter system that would cooperate with the delay, and count up using it. If the counter reached a certain limit, an action would occur (in this case, the automatic movement of the pieces down), which allowed me to be able to shorten the delay and make smoother side movement.
* I began using fonts after realizing the inefficiency of putting small text in a certain location. It’s nicer for the eye, too.

*Testing Checklist (Y/N)*

* All pieces made: Y
* All colours assigned: Y
* Map created: Y
* Pieces fall, are able to move sideways, down, and rotate: Y
* Line-making works: Y
* A new piece falls after one stops: Y
* Score system works: Y
* Lobby system works: Y
* Manuals and game-start work: Y
* Game loops back after loss: Y
* Music, pictures: Y
* Fonts, other text: Y
* Other colours: Y