# SE 3XA3: Test Plan
# Tetrileet

Team 15, AAA Solutions
Abdallah Taha, tahaa8
Ali Tabar, sahraeia
Andrew Carvalino, carvalia

April 11, 2021

# Contents

# List of Tables



# List of Figures

Table 1: **Revision History**

| Date | Version | Notes |
|---|---|---|
| March 2nd, 2021 | 1.0 | Added section 1 |
| March 3rd, 2021 | 1.1 | Added section 2 |
| March 4th, 2021 | 1.2 | Added sections 3 and 5 |
| March 4th, 2021 | 1.3 | Added section 6 |
| March 5th, 2021 | 1.4 | Added section 4 |
| March 5th, 2021 | 1.5 | Final edit and review |

# 1 General Information

## 1.1 Purpose

The purpose of this document is to describe the testing tools, plan and test cases that will be used to verify Tetrileet's UI and back-end system.

## 1.2 Scope

This test plan document will justify our intentions for testing the Tetrileet program. Our main objective is to encounter any existing game glitches or other defects, and later fix the program as necessary once they have been identified.
The test plan document is an approach to documenting all activities related to testing. Aspects of the software to be tested, testing tools used, and summaries of testing methods will all be presented.

## 1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

| **Abbreviation** | **Definition** |
|---|---|
| UI | User Interface |
| DOM | Document Object Model |

Table 3: **Table of Definitions**

| **Term** | **Definition** |
|---|---|
| Back-end | The code which handles hidden events in the program (not visible to the user) |
| Front-end | The code which handles the visual aspects of the program (visible to the user) |

### 1.4 Overview of Document

This document will cover our testing plan which will include the software description, test team, automated testing approach, testing tools, and testing schedule. It will also cover the system test description which will include the tests for functional and nonfunctional requirements. Furthermore, the tests for the proof of concept pertaining to Tetrileet will be provided. Finally, comparison tests of Tetrileet with another existing implementation of Tetris will be discussed, as well as a unit testing plan.

## 2 Plan

### 2.1 Software Description

The software is a simple remake of the classic video game known as *Tetris*. Randomized, preset shapes made up of blocks will fall from the top of a grid, able to be moved left or right in the grid by the player's input. The player can also rotate the blocks by 90, 180, 270, or 360 degrees. The blocks will keep falling until they reach the bottom of the grid, or land on top of another shape. If the player can arrange a full row of blocks by placing the shapes accordingly in the grid, that row of blocks will disappear, causing all blocks above to fall down one row. The player will be rewarded and gain points whenever this happens. If the top of the grid is reached by any blocks (the grid "overflows"), then the game is over.
Our software will feature a main menu in additional to the base game, as well as a difficulty adjusting feature - allowing users to change the speed of which the blocks fall.

### 2.2 Test Team

The test team for Tetrileet will consist of Abdallah Taha, Ali Tabar, and Andrew Carvalino.

### 2.3 Automated Testing Approach

The Jest testing library will be used to write automated tests for both the front-end and back-end of our program. For our UI, we will use the JavaScript testing libraries Jest and jsdom. These two libraries provide the functionality

to run tests on a mock DOM tree. They also allow us to select and manipulate DOM elements in order to verify some properties of our UI, such as if an element has the correct height or width.

## 2.4 Testing Tools

The testing tools that we will use are the JavaScript testing libraries Jest and jsdom. Additionally, manual user testing will be used.

## 2.5 Testing Schedule

https://gitlab.cas.mcmaster.ca/sahraeia/3xa3-L01-group-15/-/blob/master/ProjectSchedule/group115.pdf

# 3 System Test Description

## 3.1 Tests for Functional Requirements

### 3.1.1 Hit Detection

**Test for Block Hit Detection**

1. HD-1

   Type: Functional, Dynamic, Manual, Unit

   Initial State: Tetris grid with no blocks

   Input: One of every possible Tetris block.

   Output: Tetris grid, with each shape landed in place in each testing instance.

   How test will be performed: The blocks will be manually moved to the left and right as they fall into several different areas at the bottom of the grid, in different orientations. This will ensure that no blocks will exceed their designated boundaries set by the walls of the grid.

2. HD-2

   Type: Functional, Dynamic, Manual, Unit

Initial State: Tetris grid with a variety of blocks already placed randomly.

Input: One of every possible Tetris block.

Output: Tetris grid, with input block in the grid.

How test will be performed: The shape will be manually moved to land with at least one of their blocks touching a preexisting shape on the grid. This will ensure that newly generated shapes will successfully detect preexisting shapes as taken portions of the grid, and collide with them correctly as such.

3. HD-3

Type: Functional, Static, Manual, Unit

Initial State: A grid with blocks spread out all over the grid.

Input: One of every possible Tetris block.

Output: Tetris grid, with input block in the grid

How test will be performed: Blocks will not be affected by the game's "gravity" in this test - the user will be able to move an input shape in all four cardinal directions in the grid, and attempt to rotate it while close to others. This is to test if the rotating function will be restricted in certain cases it needs to be, such as if rotating a shape would cause it to overlay over other shapes.

### 3.1.2 Executable HTML test

**Testing that game runs using a executable HTML file**

1. EH-1

Type: Functional, Dynamic, Manual

Initial State: Unopened HTML file.

Input: Double clicking the HTML file

Output: HTML file containing the program running within a web browser.

How test will be performed: The HTML file will be double-clicked and we will check that the game launches through a web browser. Once it is launched, we will test that the game mechanics and UI are as expected.

### 3.1.3 Initial Condition

**Test for checking that the initial condition of the game is a start menu, and that the program will stay in a standby state until user input is received.**

1. IC-1

   Type: Functional, Dynamic, Manual

   Initial State: Game is executed from HTML file.

   Input: None (waiting)

   Output: Program remains in start menu.

   How test will be performed: We will stand by for five minutes to ensure that the program is staying in standby mode. Finally, we will click the "start game" button to show that the program successfully leaves stand-by mode once it receives user input, and begins the game.

### 3.1.4 Initial game condition

**Test for initial conditions of the game when it starts**

1. IGC-1

   Type: Functional, Dynamic, Manual

   Initial State: Game start menu

   Input: Clicking on the "start game" button.

   Output: Empty game grid and one block falling from the top.

   How test will be performed: This test will be performed by running the game and visually ensuring that there are no other blocks on the grid present, disregarding the initial block at the top.

2. IGC-2

   Type: Functional, Dynamic, Automated

   Initial State: Game start menu.

   Input: Clicking on the start game button.

   Output: True/false

   How test will be performed: We will run a test using Jest to check that once the start button is pressed, all blocks of the grid minus the starting Tetris block are marked as "not taken". Once it parses through all the grid blocks, it will return either true or false showing if the test passed.

3. IGC-3

   Type: Functional, Dynamic, Manual

   Initial State: Game start menu.

   Input: Clicking on the start game button.

   Output: Game score should be set to zero.

   How test will be performed: This test will be performed by running the game and visually ensuring that the scoreboard reads zero when the game has begun.

4. IGC-4

   Type: Functional, Dynamic, Automated

   Initial State: Game start menu.

   Input: Clicking on the difficulty adjustment button, setting a difficulty, and then clicking on the button to start the game

   Output: Game should change the speed at which blocks fall to the bottom of the screen, depending on which difficulty was selected.

   How test will be performed: Using Jest, the variable that sets the difficulty of the game will be automatically verified.

### 3.1.5 Block Movement

**Test for showing that blocks have left, right, and down movement as well as 90 degree rotation with user input, and ensuring that the user cannot move blocks at an inappropriate time**

1. BM-1

   Type: Functional, Dynamic, Manual

   Initial State: Shape is in initial state at top of grid

   Input: Pressing the A key

   Output: Shape moves one grid block to the left.

   How test will be performed: The game will be run and the user will press the A key and visually check that the shape has moved by one grid block to the left.

2. BM-2

   Type: Functional, Dynamic, Manual

   Initial State: Shape is in initial state at top of grid

   Input: Pressing the S key

   Output: Shape moves one grid block down.

   How test will be performed: The game will be run and the user will press the S key and visually check that the shape has moved by one grid block downward.

3. BM-3

   Type: Functional, Dynamic, Manual

   Initial State: Shape is in initial state at top of grid

   Input: Pressing the D key

   Output: Shape moves one grid block to the right.

   How test will be performed: The game will be run and the user will press the D key and visually check that the Shape has moved by one grid block to the right.

4. BM-4

   Type: Functional, Dynamic, Manual

   Initial State: Shape is in initial state at top of grid

   Input: Pressing the W key

   Output: Shape rotates 90 degrees clockwise.

   How test will be performed: The game will be run and the user will press the W key and visually check that the Shape has rotated 90 degrees clockwise.

5. BM-5

   Type: Functional, Dynamic, Automatic

   Initial State: Blocks have been placed all the way to the second highest row of the grid.

   Input: Placing a block on top of tower so that it falls outside of grid.

   Output: Program shall disable all keyboard input.

   How test will be performed: Jest will check if the boolean variable allowing for keyboard input is disabled, not allowing for the user to move around any tetris blocks on the grid.

#### 3.1.6 Testing Score / Row Clearing

**Test for a row being successfully cleared after being completely filled with blocks, and the player gaining points as a result.**

1. CR-1

   Type: Functional, Dynamic, Manual, Unit

   Initial State: Shapes have been placed in a manner on the grid that with one placement of preset shape, one or more row(s) of blocks should be completed on the grid and cleared

   Input: Placing a shape in a manner that will result in one or more rows being cleared

Output: Program shall clear all completed rows, removing those blocks off the screen, causing any blocks above to fall down the number of rows below them that were cleared.

How test will be performed: The tester will visually check to make sure all blocks are where they need to be - additionally, visually checking if the score displayed has been appropriately updated.

#### 3.1.7 Testing Game Over Condition

**Test for a running game ending if any block exceeds the top boundary of the grid.**

1. GO-1

   Type: Structural, Dynamic, Manual

   Initial State: Blocks have been placed all the way to the second highest row of the grid.

   Input: Place block on top of tower so that it falls outside of grid

   Output: Program shall display a "Game Over" message and end the game.

   How test will be performed: The game will be played and blocks will be stacked until they reach the top of the grid, and then one last block is placed so part of it is outside the grid. Tester will visually check that the game stops, "Game Over" is displayed, a button giving an option to start a new game appears, and all keyboard input is disabled.

### 3.2 Tests for Nonfunctional Requirements

#### 3.2.1 Appearance Requirements

**Testing that the appearance of the game matches the appearance requirements**

1. AR-1

   Type: Functional, Dynamic, Manual

   Initial State: Game screen

Input/Condition: Game has not started yet

Output/Result: Grid is separated and centered while the scoreboard is above the grid.

How test will be performed: Jsdom will be used to verify that the width and colors of the grid and scoreboard are up to standard, and visual testing will be done on numerous web browsers to test for correct placement.

2. AR-2

   Type: Structural, Dynamic, Manual

   Initial State: Game start menu

   Input: Execute HTML file

   Output: Start menu of the game

   How test will be performed: Game will be run, and a visual test will be done by the programmers to ensure that the company logo is on the main screen, and distinguishable.

3. AR-3

   Type: Structural, Dynamic, Manual, Static etc.

   Initial State: Game screen

   Input: Clicking start button

   Output: Different blocks will fall onto grid

   How test will be performed: Game will be run and a visual test will be done by the programmers to ensure that the different shapes in the game are all a unique color from one another.

### 3.2.2 Style Requirements

**Testing that the style of the game matches the style requirements**

1. SR-1

   Type: Structural, Dynamic, Manual

Initial State: Game screen

Input: Clicking start button

Output: UI of the game

How test will be performed: Game will be run and a visual test will be done by the programmers to ensure that all of the fonts throughout the program are consistent.

2. SR-2

Type: Structural, Dynamic, Automated

Initial State: Game Menu

Input: Clicking start button

Output: Blocks begin falling

How test will be performed: Game will be run and Jsdom will be used to ensure that the blocks are falling and that the screen changes the colors within the grid seamlessly.

#### 3.2.3 Usability and Humanity Requirements

**Testing the usability and humanity requirements throughout the software.**

1. UH-1

Type: Structural, Dynamic, Manual

Initial State: Game start menu

Input: Clicking start button

Output: Game begins

How test will be performed: Game will be played by multiple users, and we as a development team will take their written comments and ensure that the game has basic understandability based on their feedback.

2. UH-2

Type: Structural, Dynamic, Manual

Initial State: Game Menu

Input: Click start button

Output: Blocks begin falling

How test will be performed: Game will be played by multiple users under a five minute timer, and after the timer ends, we will assess if they could get a full understandability of the game within the time frame, based on feedback.

3. UH-3

   Type: Structural, Dynamic, Manual

   Initial State: Game Menu

   Input: Click start button

   Output: Blocks begin falling

   How test will be performed: Game will be played by multiple users that can operate a mouse and keyboard, and we will take their feedback to ensure that they had full capability to play the game, with only the understanding of using those tools.

4. UH-3

   Type: Structural, Dynamic, Manual

   Initial State: Game screen with blocks stacked until the top grid row

   Input: Dropping block so that it falls outside the grid

   Output: Game ends and "play again" button pops up.

   How test will be performed: Game will be played by the user to create the scenario, and Jest will be used to ensure that the "play again" button appears using an assert statement.

#### 3.2.4 Performance Requirements

**Testing the performance of the game based on predetermined requirements.**

1. Speed and Latency Test - 1

   Type: Structural, Dynamic, Manual

   Initial State: Game start menu

   Input: Click start button

   Output: Game begins

   How test will be performed: User will click the button to start the game, and will manually time that the first block begins falling onto the grid within five seconds of the button being pressed.

2. Speed and Latency Test - 2

   Type: Structural, Dynamic, Manual

   Initial State: Game screen with blocks stacked until the top grid row

   Input: Dropping a block so that it falls outside the grid

   Output: Game ends and "game over" screen appears, final score is calculated and displayed

   How test will be performed: User will place the final block in order to end the game, and will manually time that the "game over" screen appears, and the final score is calculated within three seconds of game ending.

## 3.3 Traceability Between Test Cases and Requirements

All requirement section numbers are based off of the SRS document.

| Requirement | Test case Id |
|---|---|
| 2.2.3 | IGC-1 |
| 2.2.5 | IGC-2 |
| 2.2.4 | IGC-3 |
| 2.2.6 | GO-1 |
| 3.1.1 | AR1-AR3 |
| 3.1.2 | SR-1,SR-2 |
| 3.2.1 | UH-1 |
| 3.2.3 | UH-2 |
| 3.2.5 | UH-3 |
| 3.3.1 | Speed and Latency Test - 1 |
| 3.3.1 | Speed and Latency Test - 2 |

# 4 Tests for Proof of Concept

## 4.1 Block Rotation Change

**Testing the rotation function of current blocks**

1. Basic Rotation Test

   Type: Functional, Dynamic, Manual

   Initial State: Shape will be placed in the middle of the grid without any shapes blocks already set.

   Input: The user will press the 'W' key four times

   Output: Shapes will take on their proper orientation with respect to how many times 'W' was pressed - with every press of the 'W' key, the shape should rotate 90 degrees clockwise.

   How test will be performed: The user will start the game and test the rotation of each type of shape by inputting 'W' to see if it takes on the proper rotation orientation.

2. Boundary Rotation Test

   Type: Functional, Dynamic, Manual

Initial State: Shape will be placed to the right or left border of the grid.

Input: The user will enter the 'W' command four times

Output: The shape should rotate with respect to the grid boundaries - if rotation is not possible with respect to the grid boundaries, the shape should not rotate.

How test will be performed: The user will start the game and test the rotation of each type of shape by moving it to either side of the grid and inputting 'W' to see if it takes on the proper rotation position, or does not rotate if it is not possible.

3. Block Overlap Test

   Type: Functional, Dynamic, Manual

   Initial State: Current shape is positioned horizontally or vertically adjacent to another previously placed shape without landing (has at least one block-width in between itself and the shape beneath it).

   Input: The user will input 'W' multiple times in order to rotate it and account for all possible rotation positions.

   Output: Shapes with rotate with respect to the boundaries of other shapes already on the grid - if rotation is not possible, the program should not allow the rotation to happen.

   How test will be performed: The user tests all possible rotations for all possible shapes after moving the current block to be horizontally or vertically adjacent to another shape that may possibly be overlapped by the rotation.

## 4.2 Block Position Change

**Testing the movement functions of current blocks**

1. Move Down

   Type: Functional, Dynamic, Manual

Initial State: Current shape has at least one unfilled block space beneath each block element the shape is comprised of.

Input: The user will enter the 'S' command.

Output: The series of blocks which comprise the current shape in the HTML file are "shifted" down, with each grid space being cleared and the space directly beneath it being filled.

How test will be performed: The user will start the game and test the functionality of each block to move down by inputting 'S'.

2. Move Right and Left (Unrestricted)

   Type: Functional, Dynamic, Manual

   Initial State: Current shape has at least one unfilled series of blocks to its left or right (depending on whether the moveLeft() or moveRight() functionality is being tested).

   Input: The user will enter the 'A' command to move left and enter the 'D' command to move right.

   Output: The current shape moves to the right or left (depending on whether an 'A' or 'D' is input), with the grid spaces on the opposite side returning to their empty state.

   How test will be performed: The user will start the game and test the functionality of each shape to move right or left by inputting 'D' or 'A' respectively.

3. Move Right and Left (Restricted)

   Type: Functional, Dynamic, Manual

   Initial State: Current shape lies on the left or right grid border (depending on whether the moveLeft() or moveRight() functionality is being tested).

   Input: The user will enter the 'A' command to move left and enter the 'D' command to move right.

   Output: The current block will not move left if against the left border, and will not move right if against the right border.

How test will be performed: The user will start the game and test the functionality of each block to move right or left by inputting 'D' or 'A' respectively.

## 4.3 Start, Pause, and End

**Testing the 'start', 'pause', and 'end' functionality of the game**

1. Start

   Type: Functional, Dynamic, Manual

   Initial State: Game start menu with an empty grid.

   Input: The user will click the "Start Game" button.

   Output: The game will begin with a single shape falling from the top of the grid into the previously empty grid.

   How test will be performed: The user will load the HTML file and click the "Start Game" button, visually verifying that the single shape is falling into an otherwise empty grid.

2. Pause

   Type: Functional, Dynamic, Manual

   Initial State: Game has already been started and has not yet ended.

   Input: The user will click the "Pause Game" button.

   Output: The game will pause the movement of the current shape and prevent further movement inputs until the game is unpaused.

   How test will be performed: The user will begin the game and press the "Pause Game" button, verifying that no movement on the grid, and key inputs are disabled.

3. End Game

   Type: Functional, Dynamic, Manual

   Initial State: Game has already been started and has not yet ended.

Input: The user will stack up blocks until they reach the top border of the grid.

Output: The game will end, and the score will be finalized.

How test will be performed: The user will begin the game and stack up a number of blocks until they reach the top border of the grid, such that the end game condition is met. They will verify that no movement occurs on the grid, shapes are no longer generated, and all key inputs are disabled.

# 5 Comparison to Existing Implementation

Tetrileet was inspired by LoveDaisy Tetris, a console-executable Tetris game. That existing version of the game did not have a user interface to launch it, meaning that none of our existing test cases for our user interface section would apply to that existing implementation. However, the functional requirements testing would be comparable, since the game mechanics should be the same. Those tests include 3.1.1, 3.1.5, 3.1.6, and 3.1.7. Furthermore, the non-functional requirement test of 3.2.4 is comparable since both implementations should have similar performance requirements.

# 6 Unit Testing Plan

The framework that will be used in testing the internal functions in the JavaScript file is Jest, while the output HTML file (altered by the functions of the JavaScript file) is tested by comparing it to what the HTML output is expected to be. All of our code, and therefore, all of our modules, were completed before we began testing. For this reason, no stubs or drivers were needed or used to replace missing modules during testing. The aspects to be covered in our unit test cases are in the main gameplay feature of our program - menu navigating will not be covered. However, aspects to cover include the movement and positioning of the Tetris shapes, pausing the game, or gaining score in the game.
The different types of unit testing used will include boundary testing and control-flow testing. Boundary testing will apply to any functions to do with the positioning and movement of the Tetris shapes. For instance, with rotate(), it should be ensured that rotating a shape while on the rightmost

or leftmost edge of a grid isn't allowed, as well ensuring that rotating a shape just before it touches the bottom of the grid, or land on top of another shape, does not cause any clipping. In regards to control-flow testing, unit tests will be done in a way that ensures that all statements will be reached and executed.

## 6.1 Unit testing of internal functions

The testing of internal functions within the JavaScript source code will be applied to those which either return some value or alter the state variables of the game. These internal functions include: control(e), freezeBlock(), rotate(), and addScore(). These functions will be tested by instantiating the game's variables (current, score, currentPosition, and currentRotation) and checking the resulting value of each to check for the expected change.

## 6.2 Unit testing of output files

In this case, testing of output files refers to examining the functions which are a part of the JavaScript file and whether the respective changes made to the HTML elements match what is the expected output.

# 7 Appendix

## 7.1 Symbolic Parameters

N/A

## 7.2 Usability Survey Questions?

1. Do you feel like you understand the game completely within five minutes of playing it?
2. Would you be able to fully play the game knowing only how to use a keyboard and mouse?
3. Does the game display a "game over" button once you lose the game?
4. With a basic understanding of navigating a web browser, do you feel like you fully understand this game?