Tetris Game Report

Contents

Introduction

The Tetris game tests players' ability to arrange falling Tetrimino blocks in order to form whole lines and score points. This paper will examine the TypeScript code that was used to build a Tetris game utilising the RxJS library and functional reactive programming (FRP) principles. The code seeks to animate the Tetrimino blocks, add graphics to an SVG element, and make the game interactive.

Code Structure

The code is structured into several sections, each with specific responsibilities:

Constants and Utility Functions

Defines constants for canvas dimensions, tick rate, grid size, and block dimensions.

Provides utility functions for state processing, collision detection, and Tetrimino rotating.

```
// Constants used throughout the game

const Viewport = { ... } as const;

const Constants = { ... } as const;

const Block = { ... };
```

User Input Handling

We define types for handling user input, including key codes and actions to be applied to the game state based on user input.

```
// User input handling

type Key = ...;

type Event = ...;

type ActionForKey = { apply: (s: State) => State };

// Utility functions

// ... (also)


// User input observables

const key$ = fromEvent<KeyboardEvent>(document, "keydown");

const fromKey = (keyCode: Key) => key$.pipe(filter(({ code }) => code === keyCode));
```

State Management

Sets the State interface's parameters to represent the game state, including the grid itself, current Tetrimino, user score, level, and more.

```
Initializes the game // State processing

type State = { ... };

const initialState: State = { ... };


// Functions for updating game state

const tick = (s: State): State => { ... };

// ...'s initial state, including grid and initial Tetrimino.
```

Tetrimino Definitions

We define the shape of the gold Tetrimino used in the game, as an exclusive bonus pointer kind of tetromino.

```
// Tetrimino definitions

const goldTetrimino: Tetrimino = [ ... ];
```

Actions and Action Handling

Defines classes for various game actions, such as moving Tetriminos, rotating, dropping, restarting, and pausing.

```
Handles collision detection and updates the game state accordingly.

// Action classes for user input

class moveTetriminoLeft implements ActionForKey { ... }

class moveTetriminoRight implements ActionForKey { ... }

class moveTetriminoDown implements ActionForKey { ... }

class Rotate implements ActionForKey { ... }

class instantDROP implements ActionForKey { ... }

class Restart implements ActionForKey { ... }

class TogglePauseResume implements ActionForKey { ... }


// ...
```

Grid Operations

They process, as in, handle collision detection and processing of collisions when a Tetrimino reaches the bottom or collides with existing blocks.

```
// Grid operations

function isCollisionDetected(movedTetrimino: Tetrimino, grid: boolean[][]): boolean { ... }

function processCollision(s: State): State { ... }
```

Random Number Generation (RNG)

Implements an RNG class for generating random numbers used in the game and a function for creating a stream of random numbers from an observable source.

```
// Random number generation

abstract class RNG { ... }
```

```
export function createRngStreamFromSource<T>(source$: Observable<T>) { ... }
```

Tetrimino Generation

It's meant to generate a new Tetrimino, including the possibility of a gold Tetrimino.// Tetrimino generation

```
function generateANewTetrimino(): Tetrimino { ... }
```

Rendering Functions

These functions are used for displaying and hiding SVG elements on the canvas and creating SVG elements with specific properties.

```
// Rendering functions

const show = (elem: SVGGraphicsElement) => { ... };

const hide = (elem: SVGGraphicsElement) => { ... };

const createSvgElement = (namespace: string | null, name: string, props: Record<string, string> = {})
=> { ... };
```

Main Function

Common main function that initializes the game and sets up event handling, rendering, and state management.

```
// Main function

export function main() { ... }
```

Observables and Streams

Utilizes RxJS to create observables for user inputs, game ticks, and state updates.

Defines observables for keyboard inputs, tick rate calculation, and merging all user actions.

```
// Game-related observables

const tick$ = interval(initialTickRate).pipe(map(elapsed => new Tick(elapsed)));

// ...
```

Rendering and DOM Manipulation

Contains functions to display SVG elements on the canvas and hide them.

Creates SVG elements for Tetrimino blocks and renders them on the canvas.

Updates game-related text fields, such as the score and level.

1. HTML Structure

```
<!DOCTYPE html>
<html>
<head>
  <title>Tetris Game</title>
  <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <div id="game-container">
    <svg id="game-canvas" width="300" height="600"></svg>
    <div id="score">Score: 0</div>
    <div id="level">Level: 1</div>
  </div>
  <script src="tetris.js"></script>
</body>
</html>
```

2. CSS Styling (style.css)

```
#goldTetriminoMessage {
 font-size: 1.5em;
 font-weight: bolder;
 color: gold;
 text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.5);
 background-color: rgba(0, 0, 0, 0.8); /* Add a semi-transparent background */
 border-radius: 5px;
 padding: 0.2em 0.5em; /* Add padding for better readability */
 margin: 0.5em; /* Add margin for spacing */
 border: 2px solid gold; /* Add a gold border */
 box-shadow: 0 0 10px rgba(0, 0, 0, 0.3);
 text-align: center; /* Center-align text */
 transition: background-color 0.3s ease; /* Add a smooth background color transition */
}

/* Hover effect: Lighten the background color on hover */

#goldTetriminoMessage:hover {
 background-color: rgba(0, 0, 0, 0.6); /* Lighter background on hover */
}

/* Additional style for the x2 message (if needed) */

#goldTetriminoMessage.x2 {
 font-size: 2.0em;
}
```

3. JavaScript (main.ts)

```
const createSvgElement = (
  namespace: string | null,
  name: string,
  props: Record<string, string> = {}
) => {
  const elem = document.createElementNS(namespace, name) as SVGElement;
  Object.entries(props).forEach(([k, v]) => elem.setAttribute(k, v));
  return elem;
};
```

Main Function

Canvas Setup:

It initializes the game canvas and preview canvas by selecting the SVG elements from the HTML document using querySelector.

It sets the height and width of the SVG elements to match the predefined Viewport constants.

User Input Handling:

It listens to keyboard events using the fromEvent function from RxJS.

It creates observables for different keyboard inputs such as moving left, right, down, rotating, dropping, restarting, and pausing/resuming the game.

Observable Creation:

It creates the tick$ observable using interval to control the game's tick rate. The tick rate depends on the player's level, which is calculated using the calculateTickRate function.

It combines all the user input observables into a single actionForKey$ observable using merge.

Game State Management:

It defines the initial game state (initialState) and initializes it with values such as the game grid, the current and next Tetriminos, user score, user level, high score, and game pause state.

State Observable:

It defines the state$ observable by applying a scan operator to actionForKey$. The scan operator processes user input actions and updates the game state accordingly.

Rendering:

It subscribes to the state$ observable, so whenever the game state changes, it triggers a rendering function (render) to update the game canvas.

Rendering Preview:

It calls the renderPreview function to display the preview of the next Tetrimino.

Game Over Handling:

It defines a function (handleGameOver) to handle game over conditions, such as displaying the "game over" screen and resetting the game state after a delay.

Background Music:

It includes code to handle background music (not shown in the provided code snippet).

Gameplay Mechanics

Tetris should adhere to the standard Tetris rules, which include the following elements:

Tetriminos:

Tetriminos should fall from the top of the canvas.

Movement:

Players should be able to move Tetriminos left, right, down, and rotate them. Additionally, there should be an option to drop them instantly.

Line Clearing:

Lines should be cleared when they are filled with blocks, and players should earn points for cleared lines.

Special "Gold" Tetriminos:

These should provide extra points when used in line clears.

Scoring:

The game should keep track of the user's score, level, and high score.

Pause/Resume:

Players should be able to restart the game or pause/resume gameplay.

Visuals and Interactivity

This section focuses on how you use SVG elements to represent the game's visuals, provide feedback to players through colorful blocks and informative text fields, and display messages when specific in-game events occur. Prepare to digest the whole flow textually.

VISUALS

SVG Elements:

Using SVG (Scalable Vector Graphics) elements is a great choice for rendering the game's graphics. SVG allows for scalable and visually appealing graphics that can adapt to different screen sizes without loss of quality. Here are some considerations for implementing SVG elements:

Tetriminos Representation:

Ensure that Tetriminos are accurately represented using SVG shapes (typically rectangles) that can be easily manipulated and animated on the game grid.

Grid Representation:

Represent the game grid using SVG rectangles or lines. This grid helps players visualize the play area and make strategic decisions.

Colors:

Utilize colors effectively to make Tetriminos and the game grid visually distinct and appealing. You can use different colors for different Tetrimino shapes, and perhaps even incorporate special effects or transitions for visual flair.

Feedback with Colorful Blocks:

Visual feedback is essential for player engagement. Here's how you can provide feedback:

Highlighting:

Implement visual cues to highlight active Tetriminos, making it clear which Tetrimino the player is currently controlling.

Row Clearing Animation:

When lines are cleared, consider adding animations or visual effects to make the event more satisfying. For instance, you can briefly highlight the cleared row before it disappears.

Special Points:

When players earn special points with gold Tetriminos, make these events visually distinct. You can use animations, color changes, or special effects to emphasize the achievement.

Informative Text Fields:

Text fields play a crucial role in conveying essential information to players. Consider the following:

Score Display:

Show the player's current score prominently on the screen so they can easily track their progress.

Level Indicator:

Display the player's current level to provide context for their performance.

High Score:

Include the player's high score as a reference point for their achievements.

Displaying Messages:

Messages can enhance the player's understanding of the game and provide feedback on their actions. For example:

Line Clear Messages:

When lines are cleared, display a message to inform the player how many lines were cleared and the points earned.

Special Points Messages:

When gold Tetriminos are used, show a message that indicates the special points earned to celebrate the achievement.

Game Over Message:

When the game ends, display a message that communicates the result and encourages the player to restart or take other actions.

INTERACTIVITY

In addition to visuals, interactivity is how and the key into keeping players engaged:

Controls:

Ensure that the controls for moving, rotating, and dropping Tetriminos are responsive and intuitive.

Pause/Resume:

Implement a user-friendly pause/resume system that allows players to take breaks or adjust settings during gameplay.

Restart:

Make it easy for players to restart the game if they choose to do so after a game over.

Sound Effects

The code includes a feature for playing background music during gameplay. Players can toggle the music on and off via YouTube's button features directly.

Background Muisc Integration

- Integrating the Lofi Girl livestream as background music is a unique and creative choice. Lofi music is known for its relaxing and ambient qualities, which can complement the gameplay experience nicely, creating a calming atmosphere for players.

- Ensure that the integration is seamless and doesn't interfere with the game's performance. You may want to preload the audio to prevent lag or interruptions during gameplay.

- Consider providing players with the option to toggle the music on and off. This adds an element of player control over their gaming experience and accommodates those who may prefer to play in silence or listen to their own music.

User-Friendly Sound Controls:

Implement a button or menu option that allows players to toggle both background music and sound effects on or off individually. This level of control lets players customize their audio experience to their liking.

Consider Accessibility:

Be mindful of players who may have hearing impairments. Consider adding visual cues or subtitles to complement audio cues, especially for important in-game events like line clears or level changes.

Licensing and Attribution:

Ensure that you have the necessary rights and permissions to use the Lofi Girl livestream in your game. This may involve providing proper attribution or complying with licensing terms. Since it is fair use in this instance.

Conclusion

For what it’s worth the time and effort, my takeaway on this project is that it represents a harmonious blend of meticulous TypeScript coding, thoughtful design, and creative styling, all orchestrated to deliver an exceptional gaming experience. Through the fusion of Functional Reactive Programming (FRP) principles and the RxJS library, this game achieves not only flawless gameplay mechanics but also captivating visuals and immersive soundscapes. I think why not split and explain into three portions for better understanding visually and textually.

Visual Harmony with CSS

The color palette, as defined in the CSS rules, envelops players in an earthy ambiance, setting the stage for an enjoyable gaming session. The gentle background hues of brown and beige create a cozy atmosphere, while contrasting elements draw the eye to essential information.

The use of the H1 element, adorned with a touch of vibrancy, welcomes players with a bold and inviting "Tetris" title, setting the tone for the adventure ahead. This styling choice effectively combines readability with aesthetic appeal.

The layout, structured around the main element, ensures that gameplay remains the central focus. The game grid, characterized by its distinctive rectangular borders and gridlines, beckons players to immerse themselves in the Tetris challenge. The harmonious integration of SVG elements, such as svg and svg rect, provides visual clarity and precision, facilitating an intuitive gaming experience.

Interactive Elements

The CSS styles extend their influence to the interactive elements. The design of buttons, with their rounded edges and shadowing, invites players to engage with confidence. The "Start," "Pause," and "Reset" buttons, situated conveniently in the controls section, allow players to navigate the game seamlessly.

The gameOver text box, discreetly presented upon a game-ending event, maintains the game's aesthetic coherence while conveying important messages to the player. Its carefully chosen styling elements make it an unobtrusive yet informative addition to the user interface.

The messageContainer employs a clever use of CSS positioning and transitions to showcase special messages, such as those celebrating the acquisition of gold Tetriminos. These messages enhance player engagement by providing real-time feedback on their achievements.

The Immersive Soundscape

Beyond aesthetics and interactivity, this Tetris game introduces a unique dimension of immersion through audio. The integration of the Lofi Girl livestream as background music elevates the gaming experience to new heights. The choice of Lofi music, known for its calming and ambient qualities, harmonizes seamlessly with the gameplay, creating a soothing ambiance.

The inclusion of audio controls, not shown in the provided code snippet but implied in the gameplay, demonstrates a commitment to user customization. Players have the power to toggle background music and sound effects individually, ensuring a tailored auditory experience.