3D Tetris Deluxe

Michael Stambler, Uri Schwartz, and Sophie Kader

Abstract

This project is a 3D Tetris game with interesting graphics, power-ups, and multiplayer options. Graphics include standard block options as well as using spherical blocks, and changing the texture to brick or marble, as well as a neon effect using postprocessing. Power-ups include bombs, speeding up or slowing down the game, and shuffling the blocks in each row.

1. Introduction

1.1. Goal

In this project, we set out to build a 3D representation of the game Tetris. We wanted to both enhance the game using graphics and changing the appearance of the playing blocks as well as develop power-ups that would make the game more exciting. The game was designed for people who enjoy Tetris and who want to play against people they are close with, using the same computer.

1.2. Related Work

This project was inspired by the original game of Tetris as well as a 3D version we found online [1]. We decided to build out a 3D version of 2D Tetris because we felt it allowed us to explore the graphics component of the game while maintaining the integrity and fun of it. The versions with a 3D floor grid that we saw online were complicated and not intuitive to play so we felt we could experiment more with graphics and powerup components in the version we created.

1.3. Approach

We utilized the ThreeJS library along with NodeJS (using the provided starter code as a building block) to make our game. We also relied on many ThreeJS examples to help guide our work, such as one that demonstrates how to split a screen [2] and one that shows how to selectively apply bloom postprocessing [3].

2. Methodology

In this section we will describe the different components of the game and reasoning for the decisions we made.

2.1. Block

In `Block.js` we built the framework for the block object of the game. Each block has a starting position and a state which contains block and game properties. The `makeBlock()` function creates the shape of the block by randomly selecting one of the seven shapes modeled off regular Tetris. Four meshes are created using `BoxBufferGeometries`, and are translated accordingly in order to make the overall block have the randomly chosen shape. The offsets in both the x and y directions for each of the four individual cubes is stored in `this.state.offsets`.

Using these offsets, we were able to implement wall, floor, and block collisions. In any situation that the block is moved (either naturally as it falls, or as a result of a keydown event), we iterate through all of the offsets of the block, calculate the x and y positions of that individual offset cube, and check for collisions. With this, we are able to know where a block can move, rotate towards, and whether or not it has reached a resting position.

The material selected in the GUI is created by using functions from `Three.js` such as `LineDashedMaterial`, `MeshPhongMaterial`, and the `TextureLoader`.

2.1.1. Shapes We implemented two possible shapes to make up the individual segments of the blocks. These can be either cubes or spheres. This is chosen in the GUI. Upon creating a new block, we check what value is selected in the GUI, and use the appropriate geometry to build up our block.

2.1.2. Shadows A standard part of Tetris is being able to see the "shadow" of the block – an outline of its ending position if you let it fall naturally. In `makeBlock()` we create the shadow, using the geometry of the block to make an `EdgesGeometry` that we use to visualize the shadow. Then, in `updateShadow()`, we calculate the shadow's position. We do this by calculating the minimum number of cubes the given block could drop down before intersecting another block or the floor, which gives us the shadow's position. We call `updateShadow()` whenever there is a keydown event, as well as in the block's `update()` function.

2.1.3. Textures On top of the standard appearance of the Tetris cubes, we allow the player to choose two other appearances of the blocks. These are chosen by the player in the GUI. Using the appropriate `Three.js Texture`, the player can choose for the block's appearance to be `Brick` or `Marble`. When one of those options is selected, we load the appropriate texture, and apply it to each new block, while leaving the existing blocks on the board as they were. This gives the player the ability to have a more "realistic" looking grid as it gets filled up by bricks or marble.

2.2. Floor

In `Floor.js` we create the floor of the Tetris grid. We use the `PlaneBufferGeometry` to create the geometry of the floor, and use `MeshBasicMaterial` to choose the color, and then create a mesh from the geometry and material. We add the mesh and set the `y` position to -10 so that it appears at the bottom of the grid.

2.3. Grid

In `Grid.js` we create the grid for the Tetris blocks to fall through and populate. We create an array of points to then use the `BufferGeometry` function to draw lines and create the grid. We then create the material using `LineDashedMaterial` and create the lines using `LineSegments` from the grid and the material. We add this to the mesh, and set its y position to -10, so that it is right atop the floor.

2.4. SeedScene

`SeedScene.js` contains the majority of the logic for playing the actual game. We store many components of the state of the game in this.state such as the current block, the next block, the score, the level, the board itself, and others. First we add a floor and grid object. Then we populate the GUI for the game so that the user has the option to set different settings for the game including the shape of the blocks, the material of the blocks, and whether or not they want to use powerups or add a player.

The `startGame()` function starts a game. It sets many of the states of the game such as its level, the rows that have been cleared, the starting speed, etc. This is included here as well as the constructor in case a user plays a game and then starts again. We want the state of the game to be a clean slate if the user restarts.

The `addBlock()` function adds a new block to the board, and updates the current block and next block states of the game.

The `holdBlock()` function allows users to hold a block using the Shift key, and updates the current block to a new block. The `gameOver()` function checks if a player has lost the game by checking the y position of the blocks and whether it has reached above the top of the grid. If a player loses, a large textbox displaying game over is displayed above their score. Once a block has stopped moving, `updateBlock()` is called. First this checks if the game is over. If not, it adds this cube to the current board state. Then, it removes any blocks that need clearing using the `clearBlocks()` function, and executes a little animation to have the board fall if any blocks were removed using the `shiftBlocks()` function. Finally, if any rows were removed, the score is updated.

2.4.1. Scoring We implemented scoring following the rules of basic Tetris. A player starts off at level 1, and increases levels after every 10 rows cleared. A level increase also comes with a speed increase of 0.01.

Clearing 1, 2, 3, and 4 rows yields a score increase of 40, 100, 300, and 1,200 points (times a level multiplier) respectively. Upon ending a game, the current game's score is compared to the session's high score, which is updated if appropriate.

2.4.2. Animations In order to animate the removal of rows/blocks and the falling of blocks, we used TweenJS. This allowed us to change the positions of blocks in a smooth and elegant manner. These animations were done on a cube by cube basis, rather than for the whole block, since it's possible that part of a block gets removed while the rest of it remains. For block removal, we used a combination of 4 tweens to animate it. The first two apply to every cube: linearly change it's color to white and decrease its opacity to 0. Then, if the cube's shape is square, the third tween linearly decreases the opacity of its edges to 0 as well (sphere cubes do not have edges). Finally, if a cube has a powerup attached to it (described below), the fourth tween linear decreases the powerup's opacity to 0. Together, the tweens make the block appear to slowly turn white as it disappears.

Falling was easier to animate, and required just one tween. For each cube that is falling, we calculate how many rows were cleared below it, which is the distance it needs to fall. Then, we animate the fall by creating a tween that subtracts that distance from the cube's position, using a `Quadratic.In` easing to slightly mimic how gravity would cause the cubes to fall (we ignore gravity throughout the game but it makes the animation look nicer to not ignore it here).

The difficult part was chaining the tweens so that all the falling happened after the block removal was done, and then so that a new block would not appear until all the falling was done. The reason this was difficult was because there are potentially many block removal and falling tweens to keep track of at any given point. Thus, we could not just make the tweens and forget about them, but had to keep track of them so that we could set their `onComplete` functions when we had the other tweens made. In order to do this, we keep track of the column each block removal and falling tween is happening in. Then, once the block removal tween is complete for a column, we start the falling tweens for that column, and we also remove that block from the scene fully (since there is no reason to leave it there with an opacity of 0). Finally, once the falling tweens are complete, just once we call the `SeedScene`'s `addBlock()` function to carry on with the game.

2.5. Powerups

We allow users to choose whether or not to enable powerups in the game. If powerups are enabled, new blocks are created with a probability of having one of their cubes contain a powerup. This probability starts off as zero in level 1, and increases up to a max of 1/4 chance as the player rises in levels. The four possible powerups are as follows:

`Flash`: If a row featuring a cube with this powerup is cleared, the speed of the falling blocks is increased by 0.01.

`Snail`: If a row featuring a cube with this powerup is cleared, the speed of the falling blocks is decreased by 0.01 (as long as the current speed is greater than 0.01).

`Bomb`: If a row featuring a cube with this powerup is cleared, the column in which this powerup resides is also cleared (powerups in that column are recursively triggered as well). If a bomb powerup is triggered as the result of a column clearing from a different bomb powerup, then that bomb powerup clears the row it is in instead of the column, and this orientation switching continues through each recursive triggering of a bomb powerup.

`Shuffle`: If a row featuring a cube with this powerup is cleared, the cubes in each row are randomly shuffled. This can obviously be either beneficial or hurtful to the player, depending on the shuffle, so be careful triggering this one!

2.6. app.js

This file starts the renderer and sets up the camera, controls, and the scene. It also contains logic for the shaders and the multiplayer functionality which will be described in the subsequent sections.

2.6.1. Shaders We implemented two possible shaders for our blocks. These are chosen by the player in the GUI. The first one is the regular shader used by `renderer.render()`, and is used for the game whenever `Standard`, `Brick`, or `Marble` is selected in the GUI's color option.

The second shader we implemented is for the `Neon` option, which is certainly the most complex shading option. If `Neon` is selected, we selectively use an `UnrealBloomPass` postprocessing pass, in order to get a glowing neon effect emitting from the blocks. To accomplish this, we first turn all the non-block meshes in the scene to black, apply the bloom pass, re-add color to the non-block meshes, and then apply a render pass that renders the normal scene, and then applies the bloom elements on top of it. This yields a glowing neon Tetris game.

2.6.2. Multiplayer The multiplayer functionality is largely set up in the `app.js` file. There were many ways we considered implementing this functionality. We could've built two grids within the `SeedScene.js` file, but we would've had to copy and paste a lot of the code we had already written for the single player functionality. Thus, we decided that we would render two scenes side by side.

This originally led to two GUIs, which we thought didn't really make a lot of sense or look good. Thus, instead of creating the GUI in `SeedScene.js`, we create the GUI in `app.js` and pass it into the constructor of `SeedScene`, along with a boolean parameter so that only of the scenes is used to populate the GUI. Then, on each frame update, we make sure that the two scenes are in sync, as dictated by the one GUI.

We created two scenes and two cameras and used the `setScissor()` and `setViewport()` functions to split the screen in half. If the `Multiplayer` functionality was not toggled on in the GUI, we simply set the scissor to the full screen for the main scene.

We created additional arrow key functionality using the "WASD" keys as the arrows, the "x" key as a space-bar to send a block to the bottom of the screen, and the "z" key as the shift key for the second player. On every animation frame, we check if one player has lost, and if so, we display "YOU WIN" and "GAME OVER" texts over the winner's and loser's screens, respectively.

3. Results

We conducted many stress tests to test the edge cases of our game, especially as it relates to block collisions. Even though in normal game play these cases probably wouldn't come up, we wanted to make sure it was never possible for a block to clip through another block or the floor. We ensured that all the basic functionality worked by playing the game ourselves and testing each of the different key events such as arrow keys, space bar, and shift. We tested that the game ended when it was supposed to, and that we were able to change the colors and shapes of the blocks using the GUI. We tested multiplayer functionality and ensured that all the GUI selections were mimicked in the second screen, that the correct player would win when another lost, and also that the second player's keyboard worked as expected. Finally, we played the game for many levels to ensure that powerups and the game worked as planned, even as the levels increased and the game sped up. The end result is a fully functioning Tetris game that you can see in 3D and has fun powerups and graphical elements to enhance the game, just as we had hoped for.

4. Discussion

As a group, we learned a lot about modular graphics programming and working in a group. We became familiar with the tools at our disposal in `Three.js` and enjoyed experimenting with its multitude of features.

5. Conclusion

Our group built a working 3D Tetris game with additional functionality using the `Three.js` library and Node.js. Our game keeps score and has additional graphics components to make the look of the game more exciting, powerups to keep you on your toes, and multiplayer mode to challenge your friends.

5.1. Future Work

Another feature we could implement would be using a web socket to play in multiplayer mode against players around the world. We could also include additional powerups such as one that would remove random blocks around the board.

6. Contributions

Getting the very basics of the game to work was a collaborative effort, since we needed to get a minimal product before being able to split anything up. Michael worked on preventing block collisions, creating the blocks' shadows, creating the `TweenJS` animations, and executing the bomb powerups. Uri worked on creating different shapes and textures for the blocks, the bloom postprocessing pass, the scoring, and the other powerups. Sophie worked on creating the multiplayer functionality, including splitting the screen, making sure the GUIs are in sync, and being able to toggle back and forth between single and multiplayer.

References

[1] http://fridek.github.io/Threejs-Tetris/
[2] https://github.com/mrdoob/three.js/blob/master/examples/webgl_multiple_scenes_comparison.html
[3] https://github.com/mrdoob/three.js/blob/master/examples/webgl_postprocessing_unreal_bloom_selective.html