Gavin Leising, Drew Curran, Aidan Walsh

COS 426 Final Project: 3D Tetris

Abstract: https://github.com/harveyw24/Glider/blob/main/writeup.pdf

3-D Tetris is an augmented, recreated version of the classic Tetris game. In our game, the user is presented with a 3D 6x6 box that they are to efficiently fill with falling objects. These falling objects are 3D versions of the classic Tetris objects, such as the L-shaped and T-shaped blocks. Like classic Tetris, the objective of the game is clear as many sheets of blocks (a “line” in the 2D version) to accrue as many points as possible. To clear a sheet, the player must rotate and organize the falling shapes such that an entire sheet in the box is filled with blocks. The game ends when the top of the highest landed shape exceeds the height of the box. To keep track of the player’s points, we create a point system that varies based on difficulty. This difficulty can be tuned by the player and automatically increases as the game goes on. Furthermore, we develop a sandbox mode that lets the player configure what kinds of blocks can fall. Our final version of the game puts the player’s Tetris ability to the test, while also giving them the opportunity to practice and adjust to the demanding 3D Tetris environment.

1. Introduction

1.1. Background

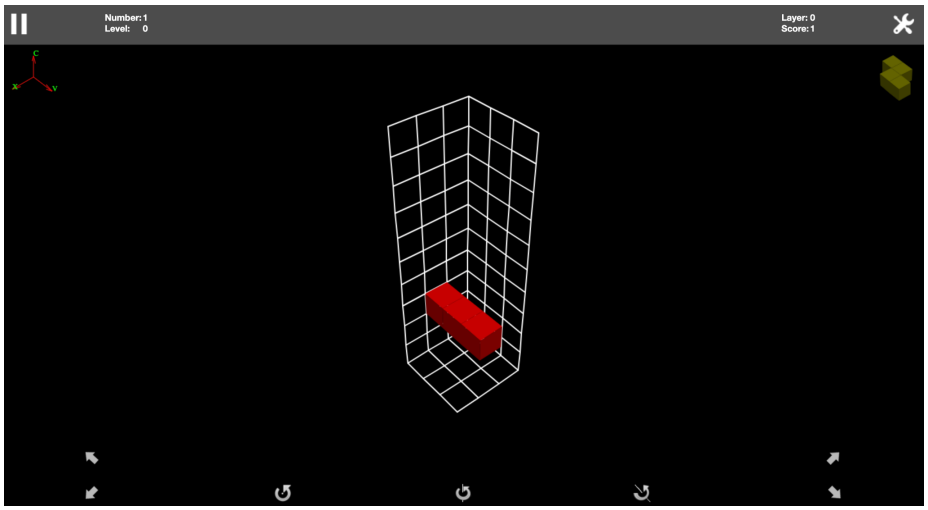
Tetris was first created in 1985 by Russian developer Alexey Pajitnov. Soon after it was released, it became a very popular game that was developed for almost every computer system. It went on to be developed for Atari, Gameboy, and many different versions went on to be developed. All versions of Tetris have a similar objective: score as many points as possible by clearing as many lines as possible (to clear a line, the player must fill an entire line with blocks). These latter versions, however, add difficulty levels, multipliers, adjusted scoring systems, and newer shapes. Like our project, other work has also attempted to put Tetris in a 3D environment.

1.2 Motivation

There exists 3D Tetris on various websites: 3dtetris.com, ebhasin.com. 3D Tetris was also developed in 1996 by T&E Soft and released by Nintendo. All of these versions are unique in their UI and manipulation, but their shortcomings motivate us to create a better version.

1.2.1 3dtetris.com

The version on 3dtetris.com uses a 3x3x10 box. This 3x3 is far too small to have enough room to toggle the rotations of the shapes and make the most of the 3D component of the game. There is also no notion of shading or highlighting, so a player does not know where their block will land or may be confused by the depth. Positively, the game does enable rotations with respect to the x, y, and z axis, and lets the camera toggle with mouse clicks. It also involves multiple difficulty levels, but does not let the player toggle them, nor the blocks that may fall.



1.2.2 ebhasin.com

ebhasin.com uses a circular 3D object, which gives the game an interesting twist, but this technically makes it 2.5D. This is because it is in 3 dimensions but has only a depth of 1. Thus, it can only rotate with respect to one axis. The game does include shadows that show where the block will exactly land, and the game includes 3 different game modes.

1.2.3 1995 3D Tetris

T&E Soft's Tetris included multiple game modes, configurable difficulty levels, shadows, but had its limitations: it could only rotate blocks in the x and y direction, and the UI made it very difficult to understand what was going on. Almost everything was red, the depth was incredibly difficult to perceive, and the game lacks appealing visuals and colors.

These 3 versions of 3D Tetris are only a few of many that exist, but they motivate us to create a version of 3D Tetris that overcomes all of their shortcomings: rotation for all 3 axes, configurable difficulties, a friendly and appealing UI, and a large enough 3D box that gives the player some freedom with rotations. Recreating and augmenting such a popular and original game is also very motivating.

1.2 Approach

We began our project by using the Javascript starter code provided at https://github.com/adamfinkelstein/cos426finalproject using Three JS. Early on, we defined a clear goal and objective for our code: use an object oriented implementation that utilizes modularity and super classes. We also knew what we wanted to include in our project to improve previous versions: a visually appealing scene and manipulation, robust Tetris physics, rotations, and difficulty configurability. After defining these goals, we were able to list out everything that would need to be implemented: block objects, the scene objects, scoring system, rotations, "sheet" clearing, difficulty configurability, event handlers, shadows, and

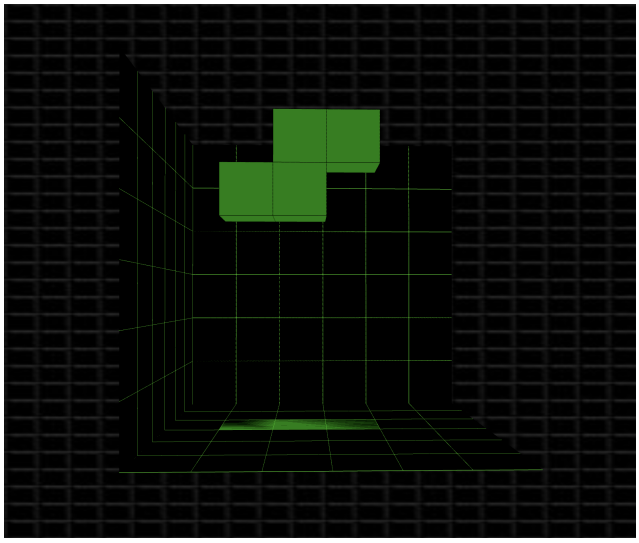
collisions. With this, we were able to assign these objectives to each of the team members. After successfully implementing something, the entire team would go through the finished code to refactor it and make it more modular. For example, we could make subclasses and superclasses out of our levels of difficulty (to define our scene and physics), stages in the game (pause, start, game over, completion), and types of blocks (shapes made up of cubes).

2. Methodology

2.1 Scene Creation

When creating the scene and “starting screen”, it was very important to focus on visually fitting and appealing features such that they could easily be interpreted. Thus, it was decided to create a “starting screen” with a changing-color title that adds to the colorfulness of Tetris. A theme of neon-green for the visuals was also added since it contrasts well with black.
For the starting screen, we just added a couple of functions in app.js that add a canvas to the document. This canvas adds the text, and we add an updating function that changes the color of the title every second. We also added a block of text that describes the controls.

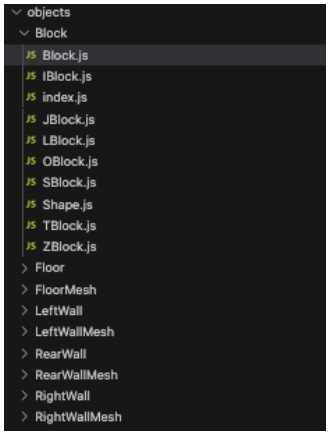
For the Tetris Scene, we add three objects, the left wall, rear wall, and floor. They are all defined to be black, and one difficulty was trying to define all the vertices of these planes using ThreeJS meshes. This is because meshes are defined by triangles, and so we had to define two triangles per plane, and we had to define them such that we created the outline for a box. We also had to ensure that the box was created in a location that was perceivable by the camera (alternatively, we could have drastically changed the location and view of the camera). We also needed to add the neon green lines that run along the planes in both the vertical and horizontal direction. This required just interpolating the vertices of the corners of the planes by a factor of ⅛ (since our box is 6x6x6) and instantiating edges between these interpolated points. We had to do this because Tetris involves blocks, and these blocks are placed in the box in a grid-like fashion. Thus, these grid lines on all planes show the exact location of the player’s blocks.
Finally, we added a background that resembles the classic Tetris background. The image below shows our background and box that the shapes fall into.



2.2 Block Object

Since Tetris uses squares as the base unit of measure, we use cubes as ours. Thus, we define a cube/block object that is simply a cube created by Three.BoxGeometry, and Three.EdgesGeometry. This cube object, however, as our project progressed, required many updates: an update function that defines the falling physics, a collision function that sees if the block will collide in the next timestamp, a shadow function that projects a show onto the closest block, and a lock function that locks a block in place. Defining a block as an object helped us pursue our object-oriented goal and enabled the writing of all of these functions. Furthermore, we created this block object with the plan to make larger shapes made out of these blocks, such as an L-block. Thus, our blocks were constructed with several arguments: an (x,y,z) position that defines where in our grid it will appear, a number that indicates the color, and a pointer back to the parent to enable updating. This facilitates our creation of the different shapes that will appear above the box.

The image below demonstrates our attempt at keeping our code as modular as possible by having all the different kinds of blocks create their blocks from Block.js. These different kinds of blocks are subclasses of Shape.js, which contains all their characteristics.



2.3 Event Handlers

We added two event handlers in order to implement our user controls. The first event handler was in the window created in app.js which allowed us to switch between different levels

and the sandbox versions of our game. The second event handler was built into the scene that was loaded for the user. This allowed manipulation of the current tetris block for movement, rotations, and pausing the game as necessary. We chose the standard implementation of adding event handlers by adding a function handler for individual events to a specified parent object (the window or scene).

The event handler for the window utilized the number keys 0, 1, 2, and 3, where each number 1-3 was associated with that given level (i.e. pressing the 1 key would take you to level 1) and the 0 key was associated with the freeplay sandbox. The event handler for the scene utilized a range of different keys for movement and rotation. The left and right arrow keys would move the block left and right on the x-axis, the up and down arrow keys would move the block forward and backward on the z-axis, the space bar would move the block one unit downward on the y-axis to increment falling, and the Q, W, and E keys were associated with rotation where the Q key would rotate the block along the XY plane, the W key would rotate the block along the YZ plane, and the E key would rotate the block along the XZ plane. The space bar is additionally used to reset the game when the player reaches a game over or complete state.

The results of this approach were fairly simple and led to an event handler which was able to utilize a variety of user inputs for different controls that was not very difficult. One thing we could optimize in the future is creating a single event handler for the entire game instead of separating it into two pieces. This was done since the levels aspect of the game was added late in development and the events associated with changing the level needed to run outside of an individual scene.

2.4 Physics and Collisions

The physics of Tetris are fairly simple, so there was not much to implement here other than a constant falling rate that increases with the time of the game. When the game starts the fall rate is one unit every two seconds and this increases with the difficulty of the game which will be explained later. The collisions however were a much more difficult problem we had to solve. We needed to be able to handle collisions with both the walls surrounding the game as well as blocks that had already been placed. Collisions with the walls were easy since we could check out of bounds positions, but we quickly realized we needed to store information about the state of the game. We did this using a 6x6x6 grid of true or false values to tell whether a unit grid space was occupied. Using this grid we were then able to perform checks on individual blocks during different movements in order to tell if that movement was valid given the current arrangement of blocks. That is we could check the change in position from a given movement and find if all individual blocks of a given tetromino occupied a free space. This also allowed us to stack blocks on top of each other properly since a downward movement into a collision zone would result in the block becoming locked into place.

This approach ended up being more difficult to implement than we expected given potential out of bounds errors of blocks and properly checking collisions of all blocks before movements. We ran into several patterns along the way where blocks would morph into each other, stick to walls, rotate out of bounds, and more. We had to come up with a very careful checker system to make sure that all parameters were met before executing a movement and were able to do so using the described grid as well as checking for boundary collisions immediately. Overall, the results of this implementation were very successful, however there are a very large number of collision checks being done throughout the game loop which we may have been able to limit.

2.5 Shapes

At this point, it became clear that our code should be more modular. We had eight different classes for each block shape, and modifications to the code were becoming tedious. This motivated the implementation of a shape superclass. Within this code, block shapes were generalized such that there were no methods inside each subclass other than the constructor. That is, the superclass became responsible for all updates and event handling. The code was also simplified to be more coherent, efficient, and legible.

The constructor for the class takes in the parent, which is the scene. It also stores the grid of blocks as supplied by the scene. The class initializes an array that stores the relative positions of each block. This makes it easier to move the blocks, as they are relative to one block with relative position (0, 0, 0). The superclass also initializes an array that stores the blocks provided by Block.js.

Update works by checking each block to see if there is a block underneath them. If so, the block and shape are locked. Otherwise, each block is updated, moving down. Action works by determining the keydown event. If an arrow key is pressed, the shape moves in that direction provided that there is no resulting collision. Rotation for each direction is also implemented.

Implementation for Shape was fairly tedious, as it was best to simplify the code as much as possible. This meant that shape movement would be generalized to the relative positions of blocks with generality. This was a big change from the hard-coded values from before. Once the shape class was implemented and began to work, it became easier to modify the code. This made it a worthwhile project.

2.6 Rotations

Rotation was initially developed to occur in the xy direction. This was hard-coded into each block shape according to different block compositions. However, with the introduction of the shape superclass, rotation could be generalized. This took form via the rotate method in Shape.js.

Initially, it was difficult to reason about rotations in a formulaic manner. It was hard to visualize the constraints on collisions as well as how each block in the shape should change. However, this was made easier when considering that rotation for blocks was similar to that of a rotation matrix. For the xy plane, it was reasoned that for every orientation, x and y would flip then y would be negated, relative to the source block of the shape. This was then applied for the other directions. For the yz plane, y and z were swapped and z was negated. For the xz plane, x and z were swapped and x was negated.

Once the relative rotation was applied, the position was translated by its previous absolute position. From there, collisions were tested. If no collisions occurred, the blocks were updated to their new positions. To maintain invariance across orientations, the relative positions were also updated.

2.7 Clearing and Scoring

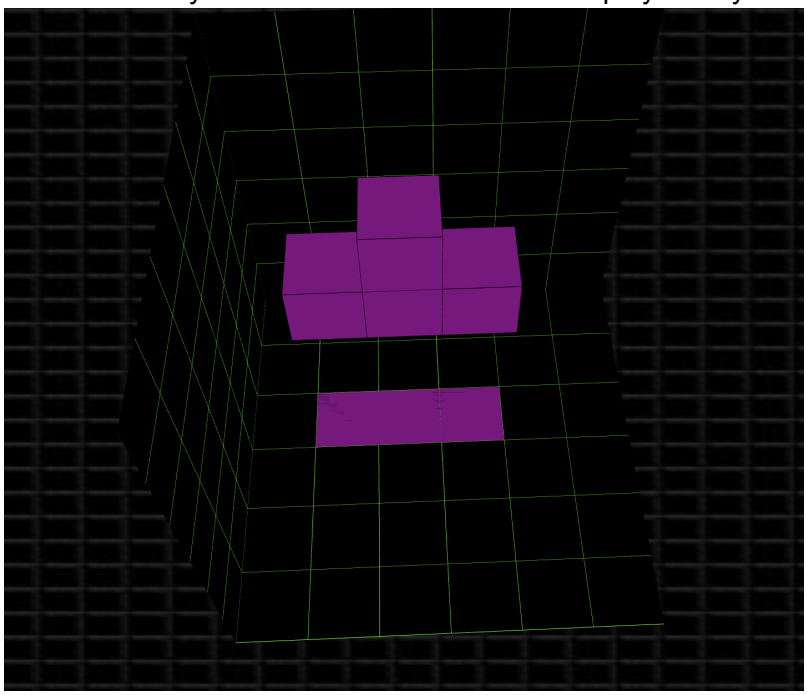
Given that we had already implemented a grid storing the locations of blocks that had been locked into place, clearing rows was fairly simple to implement. We just needed to check if any of the Y-planes were completely filled with blocks. If they were then that plane would be cleared (all block objects would be destroyed and removed from the scene), all blocks in the planes above would be moved down one position, and the user would add to their score based on how many rows they cleared after a single block was placed. This implementation of clearing was relatively straightforward given that we already had the grid, but we needed to create a second grid of block references to be able to destroy and remove blocks from the scene. We had to be careful not to remove the block's parent from the scene as this would potentially get rid of some blocks that still needed to be cleared.

For the scoring element we took inspiration from the original Tetris game which gave scores depending on how many rows were cleared after a single block was added. In our implementation the scoring was as follows:
1 plane cleared = 1000 x {current difficulty} points
2 plane cleared = 3000 x {current difficulty} points
3 plane cleared = 5000 x {current difficulty} points
4 plane cleared = 8000 x {current difficulty} points
This followed the original Tetris specifications for adding to the score except our point values were multiplied by 10 since our game is more difficult than the original version of Tetris and we wanted to give extra incentive.

To update scores and high scores, we stored the current score in an individual scene and stored high scores in the entire application such that high scores would remain “saved” when the user switches between levels and the sandbox. This approach allows them to continue playing and trying to beat different scores when going between the sandbox and other levels of the game. We did not run into any problems scoring but had to make sure we were updating the DOM properly each time scores were updated or levels were changed. We do not see any major ways to improve this implementation other than changing the fundamental scoring rules of Tetris or adding in modern Tetris scoring rules which had some rules that did not fit the 3D nature of our gameplay.

2.8 Shadows and Translucence

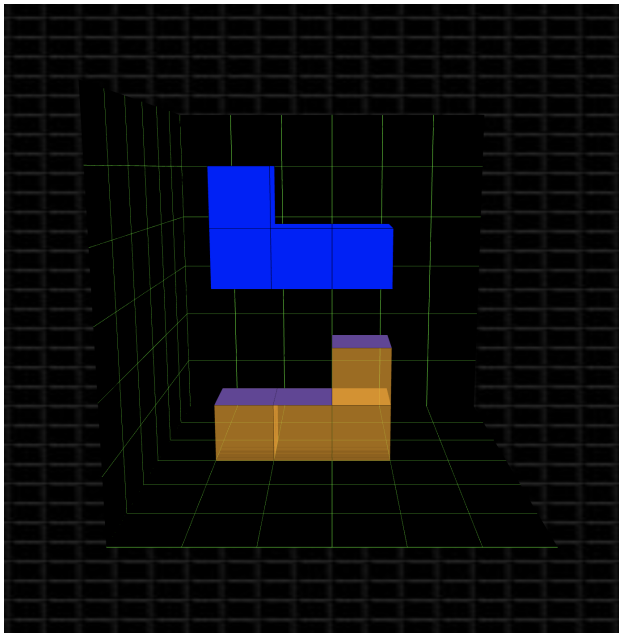
In the 3D environment, shadows are incredibly important. Since the depth can sometimes be difficult to interpret, the player may not know exactly where their shapes will land. Thus, a shadow is used to show the grid coordinates that the shape is over. Some 3D Tetris games include a shadow that is the block itself, just showing how it will land, but we aim to use shadows that highlight blocks that the falling shape is over. In the image below, the purple highlight is the block’s shadow. We made sure that the shadows are the same color as the block to minimize any confusion or uncertainties the player may have.



To implement shadows, we added the function in our javascript file that defined the individual blocks. This is because our thinking was the following: if a shape is falling, then we calculate the shadow, but do it for all of the blocks in the shape, because all of the blocks will ultimately have the same shadow as the blocks in the lowest layer of the shape. If we calculated the shadow for just the blocks in the bottom layer of the shape, this would require calling our shadow function in every javascript file that defines all of the shapes, rather than just calling it for every individual block. This requirement would require a lot of additional code for a small optimization.

In our shadow calculation, we maintained a 3D array of boolean values that indicated whether or not a stationary block existed at coordinate (x,y,z) in the grid (this is the same array mentioned in an earlier part). Since a shadow can only be present in the same x and z coordinate, we just checked the y coordinates for all the blocks, going top down. Whenever we found a stationary block, we added a new plane (highlight) at that (x,y,z) coordinate. If no stationary block exists underneath the falling block, then the plane is just added to the floor. Implementing this function involved one major difficulty: mapping the grid coordinates to the actual (x,y,z) space of the scene and then mapped again to coordinates relative to the falling block. This is because the x,y,z coordinates of the 3D boolean array had to be mapped to the actual coordinates of the scene, where we could then add the shadow. Since the shadows were calculated for each individual block, the shadow had to be added at a position relative to the falling block (position (0,0,0) equates to the position of the falling block). Thus, after calculating the position in the scene's x,y,z coordinates, these coordinates then had to be mapped again to coordinates relative to the falling block. This was solved by performing two mappings/functions on the (x,y,z) coordinates.

To make the game more visually appealing and friendly, we also added translucence to the blocks. That is, when a block falls into place, it becomes translucent. This lets the player see through blocks that may be in their way, it lets the shadows stand out, and it differentiates the falling blacks from the stationary ones. This simply required changing the opacity value when a block becomes "locked". The image below shows this opacity change in the orange block.



In some cases, we notice that the rendered shadow is incomplete or inaccurate. This is likely due to aliasing and issues with Three JS's buffering. To improve, we could potentially make some modifications to our code (such as adding EPS when defining where the plane is added) and methodology for adding the planes. We could also attempt to use other libraries.

2.9 Game Loop

Tetris ends when one places a block above a certain height. In 3-D Tetris, it works the same way. This established the game loop, whereby the game over screen allows a player to reset the game. The logic for the game loop was implemented by the scene classes. In the scene classes, a field called game determines whether the player is in a game or looking at the game over screen. If the game field is 0, then the game is over. If this is the case, the game over text displays on the screen. If the game value is 1, the player is in a game and plays as usual.

When the game is going on, determining if the game is over goes through the Scene -> Shape -> Block pipeline. At the lowest level - the block - a try-catch statement determines if the block is within the grid. This occurs when the block is locked, meaning that the next shape would be called. The only case when the block is outside of the grid is when it is above it, meaning that failing to update the grid values entails that the game is over.

When the game is over, the scene's current item refers to the game over text geometry and none of the game logic is reached. Instead, the only way to change the game state is a spacebar keydown event. If this is the case, the game over object calls the scene to reset, refreshing the game to its initial state, more or less.

Implementing the game loop was not too difficult. However, it was tricky to consider invariants about the game state. Nevertheless, the game loop allowed for replayability.

2.10 Difficulty and Levels

To implement a difficult component we simply had a difficulty integer that would interact with both the fall rate and the scoring elements. After a designated number of seconds the difficulty of the game increments causing the blocks to fall every two/difficulty seconds with the score being influenced by difficulty as described earlier. We found this to be a simple way to increment the difficulty of the game using time elements to incentivize players to play faster in order to achieve higher scores.

Additionally, we used this difficulty in a variety of ways in order to create levels for the players to attempt to beat. We decided to add these different level scenes in order to make our game a bit more dynamic and allow the player to feel the satisfaction of completing a stage similar to Tetris Classic. The levels had difficulties as follows:

<u>Sandbox</u>
Infinite blocks (chosen by player)
Starting difficulty: 1
Difficulty Increment: 0.1/10 seconds

<u>Level 1</u>
20 blocks excluding S and Z blocks
Starting difficulty: 1
Difficulty Increment: 0.1/10 seconds

<u>Level 2</u>
40 blocks
Starting difficulty: 1.5

Difficulty Increment: 0.1/7.5 seconds

Level 3
60 blocks
Starting difficulty: 2
Difficulty Increment: 0.2/10 seconds

2.11 Sandbox GUI
The sandbox GUI allows players in the sandbox to choose which types of blocks are able to fall during the next block creation. This was made to allow players to tinker with different strategies of the game and try out different formations, types of rotation, and positioning for different blocks. We found that the sandbox began to get very difficult for novice players when S and Z blocks were falling, so this allows a player to get acclimated to the game slowly in the fashion that they choose.

This GUI was fairly simple to implement as we just used the GUI object from the DAT library. This was standard for our other Three.js projects. This allowed us to store true or false values for the different blocks and made it possible for the random block creation to ignore certain types of blocks it was trying to create. The implementation choices for this were straightforward, we did not run into any hiccups, and do not see how this could be improved in terms of what we wanted to achieve with the GUI other than using a different type of GUI.

3. Results
Our game is fully-functioning with a number of game states and settings, interesting gameplay, and graphics that stay true to the simplicity of Tetris. From a qualitative point of view, the gameplay was very compelling. A number of people saw us working and wanted to play our game, and it was generally met with even more enthusiasm after playing. It seems fair to say that the addictiveness of Tetris rubbed off on this game.

We also examined our game for bugs. Testing the game extensively, we found that the game was pretty reliable. There were off occasions where collisions didn't behave expectedly, but for the most part, the game played as it was intended. This was observed especially with the toggling of different game settings and stages. Transitions were seamless and there were no serious bugs in the game.

Considering player satisfaction and the extensive testing of edge cases, 3-D Tetris is a successful project. The game is enjoyable and reliable, and the graphics complement the game well. The gameplay is complex, but learnable, and the levels are challenging. Overall, 3-D Tetris is a fun game.

4. Discussion & Conclusion
Our goal for this project was to create the best version of 3D Tetris. This involved rotation for all 3 axes, configurable difficulties, a friendly and appealing UI, and a large enough environment to play the game.

First, we achieved our MVP for this game in that we were able to implement the core functionality of Tetris. Namely, blocks moved down gradually and were able to be moved and rotated. Players were able to clear and the game was able to end and restart when a player lost.

We also managed to implement a number of stretch goals, taking our game to the next level. In terms of gameplay, this involved adding shadows for falling blocks and translucence for locked

blocks. It also meant rotation in all three planes. We also implemented features outside of the scope of gameplay. This included different level designs and a configurable UI.

With these features, the game is successful in terms of playability as well as reliability. However, there is always room for future improvement. For example, we could add music and sound effects to further improve our gameplay immersion. We could also consider how we could make our controls more intuitive, as well as make it easier to perceive the blocks in the environment. Overall, though, the game is entertaining and can be considered a success.

5. Contributions

Gavin Leising
My focus of the project was mainly on maintaining the flow of the game. I worked on the general physics of the block object in time, the collisions with other blocks and walls, and the clearing of planes in the game that increment the score. I created our grids which gave information about which blocks were stored within our game to make it easy to remove different block objects when we needed to clear different planes from the game.

I added in the event handlers for the game to recognize user inputs for a variety of different commands including movement, rotation, level changing, and speeding up the block's fall movement and worked on creating 3 levels of varying difficulty outside of our sandblock level in order to give our project more of a game-like feel where the user has a completion objective. I had to tinker with the blocks that were falling to make the levels easy, medium/hard, and hard/impossible to provide varying levels of difficulty. I additionally added the GUI to the infinite sandbox allowing users to choose which kinds of blocks they want to fall in order to tinker with strategies of the 3D Tetris game for different kinds of block objects.

Aidan Walsh
A lot of my work was focused on visuals, ease-of-use, and the environment. Firstly, I focused on the scene. This involved obtaining the background, designing the meshes and shapes for the box that the shapes fall into. This involved a lot of planning with how the scene will develop, and a lot of 3D vectors to finalize the location of everything in the 3D space. I also designed the "starting page" which also involved some thinking of visuals. Later on, I created the block object and constructed it to make it more usable for Drew and Gavin when they created all of the different shapes and updating functions.
To pursue our goals defined in the Introduction, I also developed the shadows and translucence features. Developing the shadows took a lot of work to figure out the mapping of coordinates and coming up with a scheme to add the planes, but ultimately it worked out. Adding the translucence feature was very simple, but I feel it makes our game more user and visual-friendly.

Drew Curran
My focus on the project was on the simplification of code. The starting code was implemented by the time I was able to start to help. Blocks were already moving, but the game logic was not all the way there. I noticed there were a number of ways to make the code more efficient, such as constructing superclasses and making existing code more concise. I also took on the responsibility of polishing up the code near the end so there were not so many nested conditions.
As a result of this, I was able to implement the more complex logic when it came to gameplay. Specifically, rotation in all three axes was an important and pretty complicated undertaking. I implemented a number of new variables to accomplish this task, as well as removed those that

were redundant. As I was responsible for converting hard-coded information into concise abstractions, I was familiar with the capabilities that came with the new code. I also worked on the game loop, working through ThreeJS addons to add the game over text geometry.

6. References

Background and Motivation:
https://en.wikipedia.org/wiki/3D_Tetris
https://tetris.fandom.com/wiki/Scoring
https://www.britannica.com/topic/Tetris
https://3dtetris.com/
https://www.ebhasin.com/games/Tetris3D/index.html

Code:
All meshes and models are from https://threejs.org/