

# 2048 Effect

Md Abid Sikder & Kanka Gupta

## Abstract

The goal of our project was to create a game “2048-Effect” which combines the online game 2048 with the aesthetics of Tetris Effect.

We began by writing a working game which we tested by playing in the console. Next, we designed the board border, tiles, title, score, and other text in Figma to get the color schemes down. We then translated that to 3D models using Three.js.

We then worked on improving the aesthetics of the game by using spotlights to create a gradient effect on the title, making all objects glow using the UnrealBloom post processing filter, using fragment and vertex shaders to add a time-based procedurally generated pattern to the board border, and using procedurally generated flow fields to add particle effects. We also added sound to further enhance the game experience.

Overall the outcome is similar to what we hoped to achieve. Due to time constraints, the particle effects we implemented are simple compared to those in Tetris Effect and we had to make some design changes (from the original we created in Figma as a template prior to coding) to ensure that we met the assignment specifications. Additionally, while our game is 3D it looks best if you look at the front of the board or slightly angle it.

## Introduction

Our goal for this final assignment was to create a new and improved version of 2048 which provides a better user experience by integrating sound, and interesting visual effects and simulations. This new game benefits the sizeable number of people who enjoy playing 2048. The original game has been well received: it has generally positive reviews from critics including a Wall Street Journal article which called it “almost like *Candy Crush* [a very popular game] for math geeks.” There already exist some spinoffs of 2048 including ones with elements inspired by the Doge meme, Doctor Who, and Flappy Bird. There is also a 3D version of 2048 (the actual game is in 2D but it has 3 grids representing the the top, bottom and middle layers of a cube; the objective of this game is to join numbers inside the cube until you create a 2048 tile). These related games are pretty different from the original as the doge meme and doctor who version don’t require you to combine numbers, instead you combine images that look the same to create a new image. These two implementation seem a little confusing because it unclear which image is the most desirable unlike in 2048 where the aim is very clearly to get tiles with larger numbers. In our opinion, these games do not provide the satisfaction of create a particularly large value tile in 2048. The flappy bird version is actually more similar to flappy bird than 2048 as the objective of this game is to ensure that your tile keeps “flying” (doesn’t hit the ground) and then you try to get it to fly into a tile of the same value to increase your tile’s value.

While, the 3D 2048 is very similar to the original game, especially in terms of visuals, it is much more complex. These related projects fail to provide the true 2048 game experience. Therefore, our aim was to create a game that resembles 2048 almost exactly in terms of game play (unlike the version mentioned above) that features a style we like: that of Tetris Effect. Our approach was to create a functioning game (by referring to 2048 source code), designing elements in our chosen style (tiles, board, texts, etc), and then combining them. 2048-Effect works best in certain browsers like Safari which do not block website sound autoplay. The game works in Google Chrome and Firefox too, however, you might have to tap a random key on your keyboard before the background sound starts playing. For the best and most consistent particle effects, the user should either be able to combine tiles relatively frequently or press the 's' key as these are the two circumstances under which new "sparkles" are generated. If the user is idle or unable to combine tiles for a while, the particles will drift away from the board resulting in a game which appears to have very sparse particle effects which do not look as good as the explosion of new particles that are created in the ideal game play conditions. The game also works well when played from the default front view or from an angle (this is a cool view which allows you to see and appreciate the 3 dimensionality of the board and tiles) and not well from the back (all the text appears to be backwards and the title is not lit as no lights were added to the back of it). Having a good sound system is also beneficial as it allows you to more clearly hear the faint "ding" sound that is made every time tiles combine.

## Methodology

To execute our approach we had to implement several pieces such as game execution, lighting, generating the board frame, tiles, and text, creating particle effects, adding sound, making everything glow, and fixing memory management issues.

## Game Execution

We created a data structure to hold the model of the game, which consisted of a 2D array data structure that contains either a null value or an actual object representing the tile at that location on the grid. The advantage of this approach was that the arrays stayed a constant size and it worked well with the Typescript type system to enforce us casing on every retrieval of one of those values.

We also had a controller class that was responsible for manipulating this grid when the user issued a move command.

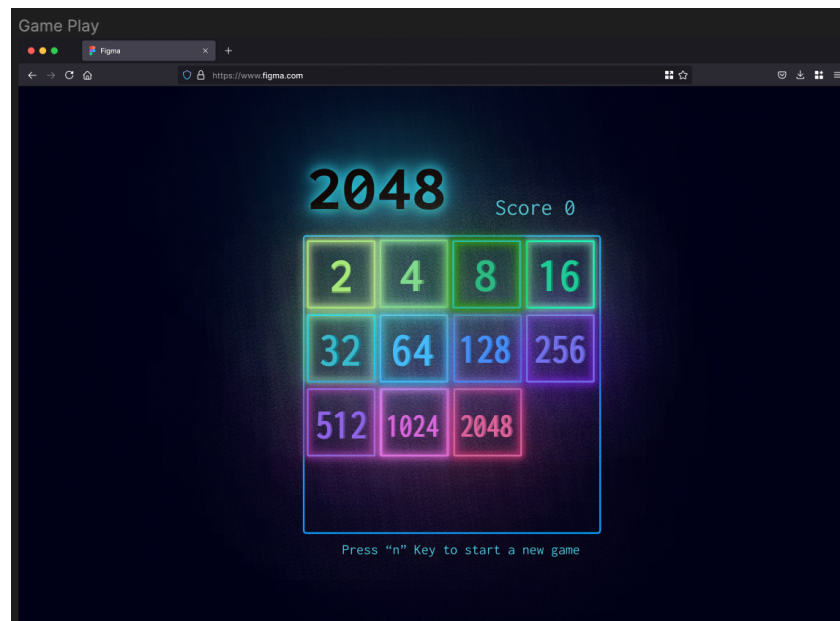
The disadvantage to this approach is that this model is just that: an abstract data type that made it harder for us to later on add effects on tile merges and whatnot.

## Lighting

To implement lighting, we created two point lights of different colors (red and blue) and added them to the scene. We played around with their positioning until we were satisfied with the way they lit the game title (we were trying to create a gradient-like effect). In the `generateTitle()` function in `view.ts` which creates the title text, we set the material of the text mesh to a Lambert material to ensure that it can interact with the point lights we added to the scene.

While the above describes how we implemented lighting in our game, there were several other possibilities. We could have included more lights (including other kinds of lights like spot lights and ambient lights) and a more complex lighting model. The advantage of doing this would have been that we would have been able to light the elements in our scene in more interesting ways. We might have even been able to light our title in a way that more closely resembles our initial figma design (see fig.1). However, the disadvantage would be that it would have taken more time and effort. We decided to have very simple lights in our game because we had already planned for the bulk of our “lighting” to come from making objects in our game glow using the UnrealBloom postprocessing effect that we found in 3js examples. We were also happy with the lighting of our title using the simple pointlights.

Fig. 1

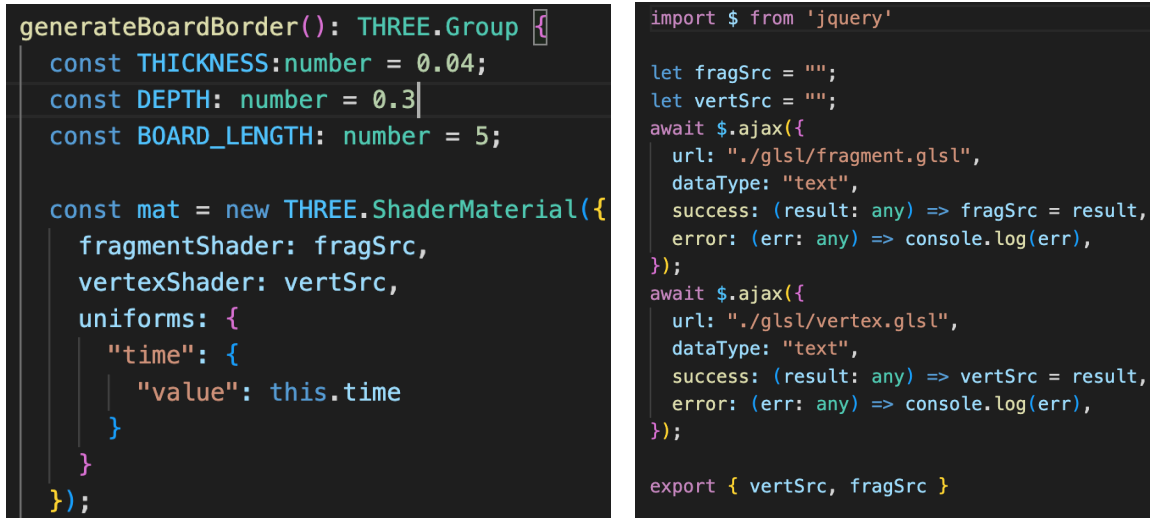


## Generating Game Elements (Board Border, Tiles and Text)

To generate the board border we used fragment and vertex shaders (see fig. 2 which displays the code for this). This allowed us to create a multicolored frame that changes color over time. There were certainly other possible ways to create the board border that did not include the use of shaders. In fact, our original design (see fig. 1) had a plain blue border. We ultimately decided

to use shaders because that was one of the extra features we wanted to implement. The advantage of this approach was that we were able to create a board border which was more visually interesting than a plain blue border but a disadvantage was that it took some more time to implement than a basic border.

Fig. 2



```
generateBoardBorder(): THREE.Group {  
  const THICKNESS: number = 0.04;  
  const DEPTH: number = 0.3;  
  const BOARD_LENGTH: number = 5;  
  
  const mat = new THREE.ShaderMaterial({  
    fragmentShader: fragSrc,  
    vertexShader: vertSrc,  
    uniforms: {  
      "time": {  
        "value": this.time  
      }  
    }  
  });  
};
```

```
import $ from 'jquery'  
  
let fragSrc = "";  
let vertSrc = "";  
await $.ajax({  
  url: "./glsl/fragment.glsl",  
  dataType: "text",  
  success: (result: any) => fragSrc = result,  
  error: (err: any) => console.log(err),  
});  
await $.ajax({  
  url: "./glsl/vertex.glsl",  
  dataType: "text",  
  success: (result: any) => vertSrc = result,  
  error: (err: any) => console.log(err),  
});  
  
export { vertSrc, fragSrc }
```

In order to generate the number tiles, we choose to create separate functions to generate each number tile (i.e a different function for the 2 tile, 4 tile, etc). We did this because to make the tiles look good we had to create a separate mesh for each digit (so for instance, to create the 2048 tiles four meshes were created: a “2” mesh, a “0” mesh, a “4” mesh and an “8” mesh and then the function returned a group of these four meshes). Doing this was more time intensive but allowed greater control over the spacing between digits in a multi-digit number. We choose to use this approach instead of creating a single mesh for each number because the default spacing between digits did not look very good. We played around with the x, y, and z positions of each mesh until we were happy with the way each tile looked. Additionally, we initially gave our numbers depth (currently we have 3D looking tiles with a depth but the number inside the tile is flat looking) but then decided against it. While the advantage of 3D numbers was that it made the 3D effect look better (i.e if you panned the view was better because there were more 3D elements), once we added the bloom effect the 3D numbers looked blurry. So we choose 2D-looking numbers (numbers with a very small depth) to make the text look sharp.

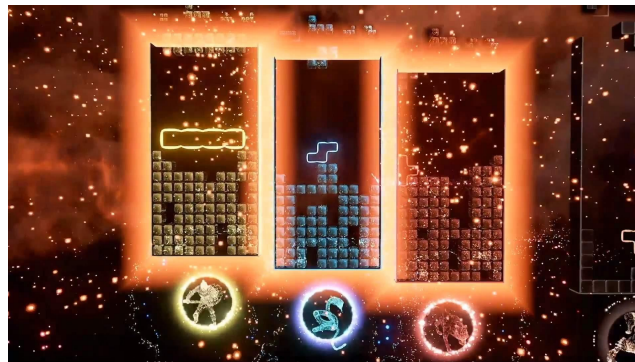
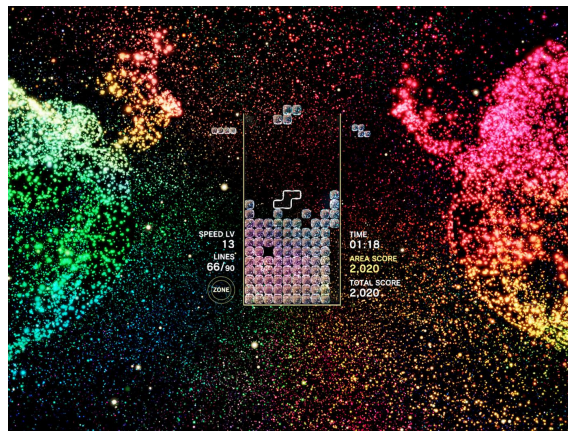
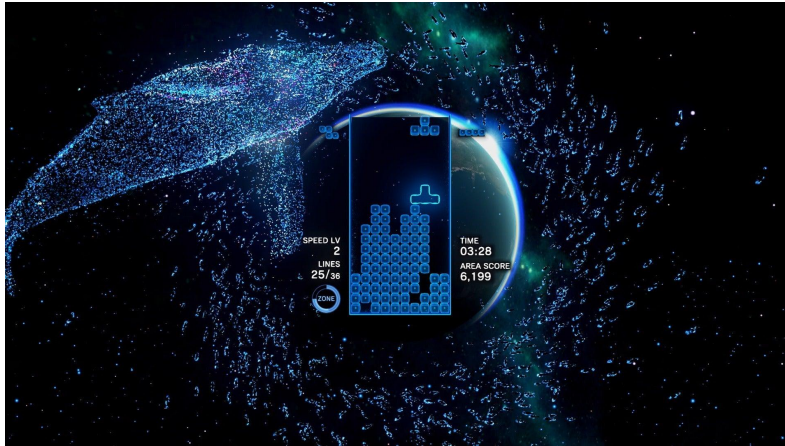
Text generation like the generation of the “press ‘s’ key to sparkle” message, the title and the score followed a similar procedure to the generation of tiles except this text did not have a 3D border around it. We made all text flat to preserve sharpness and ensure readability (as described in the above paragraph there were advantages and disadvantages to using both 2D and 3D text but we chose 2D as this looked better). To make all the text, we essentially added text geometry and then used this to create meshes which we then positioned appropriately. Some text like the title and score used multiple meshes. For the title this was done so that “2048” and “Effect” could be placed one on top of the other (again more meshes allowed for

greater flexibility with placement even though this took some effort in terms of having to correctly place more meshes) and for the score it was done because the score number varies which the word “score” remains the same.

## Particle Effects

To implement our particle effect we created a hundred `THREE.Points()`, and used three different 4D simplex noise functions to set each particle's x, y and z accelerations as a function of position x, position y, position z and time. This acceleration is used to update the velocity and the position of the particles in the next animation frame. A 2D simplex noise function which takes in number of particles and time in as parameters is used to get the color of all the particles. We first calculate color in hsl not rgb. This was done to make sure that the colors had a certain luminance so that the colors generated were not too dark (these would not stand out against the dark background). While this is how we implemented our particle effect, there were several other possibilities. For instance, 3js examples had some projects that used shaders to achieve particle effects: see [ex1](#), [ex2](#) and [ex3](#). We chose not to do this because we wanted to use procedural generation to do the particle effects since that was one of the extra features we implemented. We also could have used more complex geometries (instead of points) to do the effect. This would have allowed greater freedom to make more interesting patterns (see fig.3 for more complex effects from Tetris effect), however this would be more difficult and memory intensive to implement. Given the time constraints, we chose to use points and implement a basic but still nice looking particle effect.

Fig. 3



## Sound

To implement the background sound, we used the code shown in fig. 4. We created a listener which was added to the camera and then we created an audio loader which was used to load the required mp3 and adjust volume and other parameters. This is the standard method we found online to add background sound; we are not aware of any alternative methods for implementing this feature.

Fig. 4



```

//adding background sound
const listener = new THREE.AudioListener();
effect2048.camera.add(listener);
const audioLoader = new THREE.AudioLoader();
const backgroundSound = new THREE.Audio(listener);

audioLoader.load("./sound/background.mp3", function(buffer) {
    backgroundSound.setBuffer(buffer);
    backgroundSound.setLoop(true);
    backgroundSound.setVolume(0.5);
    backgroundSound.play();
});

```

The code and method to add a “ding” sound effect when tiles combine is fairly similar. We add the sound effect every time the score updates (this is equivalent to when tiles combines). Alternative methods to implement this feature would be to detect when tiles collide (they need to collide before combining) then creating a sound. This method is more complicated as it requires collision detection. There are no real advantages of this method by itself. It may, however, be useful if we decided to create a positional audio object instead of a non-positional one for the “ding.” We could have then created a ding sound that originated from the point of tile collision. This was not done because it requires a lot more work and was not feasible given the time we had to work on this project. Additionally, we are not sure if the positional “ding” would sound much different from the current sound effect that we have since the “ding” sound effect is fairly faint.

## Glow

To implement glow we used the UnrealBloom filter that we found in 3js examples. We are not aware of other methods of making objects glow but presumably we could have achieved a similar effect using a complicated lighting model. We choose to use the filter as the effect it produced looks aesthetically pleasing. We could have implemented selective bloom which allows use to choose which objects glow but this was hard to implement we did not have sufficient time to make it work. Additionally, we did not really need the ability to make only some objects glow; we were pretty happy with having everything glow. To use the Bloom filter, we had to create an effect composer (this allows us to add post-processing effects). We then added the render pass (current frame to render with no effects added to it) and the bloom pass (imported from a 3js example) to it (see fig. 5 for this code). We also had to effect composer to render the scene in the animate step.

Fig. 5

```

/* Unreal Bloom Pass */
const renderScene = new RenderPass(this.scene, this.camera);
const effectComposer = new EffectComposer(this.renderer);

const bloomPass = new UnrealBloomPass(
  // vec2 representing resolution of the scene
  new THREE.Vector2(width, height),
  // intensity of effect
  0.4,
  // radius of bloom
  0.04,
  // pixels that exhibit bloom (found through trial and error)
  0.01
);

effectComposer.addPass(renderScene);
effectComposer.addPass(bloomPass);

this.effectComposer = effectComposer;

```

## Memory Management

An issue we encountered when creating our game was that it made our browser crash. When we used the activity monitor we realized that the game was using up a lot of memory. After looking up possible causes of this memory leak, we realized that 3js does not automatically dispose of objects that have been deleted. This was a problem because we delete and regenerate all the objects at every animate step. We created a `disposeOfGroup()` function (see fig. 6) to deal with this problem. We do not know of any alternative ways to solve this issue.

Fig. 6

```

function disposeOfGroup(g: THREE.Group) {
  g.children.map(child => {
    if (child.type === "Group") {
      // @ts-ignore
      disposeOfGroup(child);
    }
    else if (child.type === "Mesh") {
      // @ts-ignore
      child.geometry.dispose();
    }
  })
}

```

## Results

We measured success based on how well the game ran (even if we played very quickly) and how similar it was to our original idea. Based on these metrics, we think that we were pretty



successful as the user experience of the game is great and we were able to create a product similar to what we hoped to create (although our particle effects are simpler than what we envisioned). An experiment we executed was having some of our friends test out the game and tell us what they thought. When we watched them playing the game things seemed to be running pretty smoothly for the most part and they enjoyed the game. One issue, however, was that the “ding” sound effect did not play on everyone’s device for some reason. We still need to investigate why this is.

## Discussion

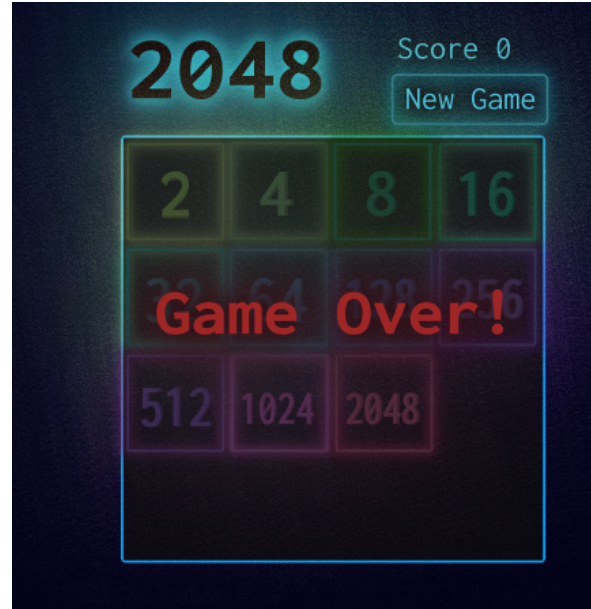
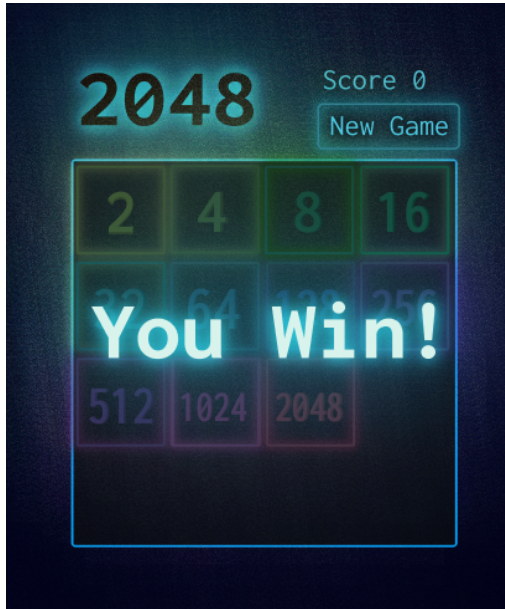
Overall, the approach we took was promising as it produced a result we are quite proud of. A variant of this approach that might have been better was to make our tiles into cubes to enhance the 3D effect. We could have also worked on adding different text to the front and back of these cubes so that the game looks good from both the front and back. Due to time constraints, we were unable to do this but this is a good follow-up idea to improve our game.

One major learning point, at least about Three.js, is that efficiency of code matters, since the underlying library itself is quite slow. This means strategies of animation should utilize moving existing objects around rather than just generating an entirely new object, in which case memory management and freeing memory after an object is removed from a scene is very important.

## Conclusion

Overall, we were able to effectively achieve our goal as we managed to create a pretty good game fairly quickly. Some next steps would be to add more complex particle effects (like those in fig. 3), to make the back side of the game look good, and to make the “you won” and “game over” pop up look better (we could make it look similar to our original figma design which is shown in fig. 7). We could also add additional sound effects to further improve the user experience. Another possible addition, could be adding multiplayer capabilities so that you can compete against a friend to see who can achieve a higher score in 2048-Effect. Some issues we need to revisit are the aforementioned problem with the “ding” sound effect (to be clear this feature does work but not in all browsers for some reason).

Fig. 7



## Contributions

### Abid

- Getting the game to work in the console
- Fragment and vertex shaders
- Particle effect using procedural generation
- Three.js memory leak fix

### Kanak

- Overall design and color scheme
- Game element generation (tiles, text, board)
- Sound
- Glow (Bloom effect)
- Lighting

## Works Cited

### Three.js

<https://threejs.org/>

Glow effect example:

[https://threejs.org/examples/?q=glow#webgl\\_postprocessing\\_unreal\\_bloom](https://threejs.org/examples/?q=glow#webgl_postprocessing_unreal_bloom)

## Vite

<https://vitejs.dev/>

## Typescript

<https://www.typescriptlang.org/>

## Simplex Noise GLSL

Simplex 3D Noise (permute+taylorInvSqrt+snoise)

Credit to Ian McEwan, Ashima Arts

<https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83>

also check out: <https://github.com/ashima/webgl-noise>

## Warping with Noise

<https://iquilezles.org/articles/warp/>

## Simplex Noise Javascript

<https://www.npmjs.com/package/simplex-noise>

## Flow Field Creation

<https://thecodingtrain.com/challenges/24-perlin-noise-flow-field>