Chromatic Arrow

William Li (wl14), Anna Qin (alqin), David Todd (dtodd)
COS 426 Spring 2020

Abstract

Chromatic Arrow is an archery-based web game developed using the ThreeJS library. Its core gameplay consists of shooting arrows at targets that randomly spawn in 360 degrees around the player. Arrows follow the laws of physics, so the player must account for gravity and wind while aiming. There are also invisible barriers throughout the scene that will block the player's shots; barriers are revealed by paint splatters whenever an arrow collides with them. The more the player shoots, the more of the level is uncovered. Points are scored by hitting targets and revealing barriers before time runs out.

1. Introduction

Goal

The goal of this project was to create a game that looked polished and that people would find fun to play. We all enjoyed the art contests throughout the assignments, so making something visually appealing was a high priority. We were also new to game design and thought this was a great opportunity to experiment and learn. We hope that our game will be fun and enjoyable for everyone stuck at home to admire and play over and over again.

Previous Work

The idea was inspired by some archery-based flash games that we used to play when we were younger, like Bowman 2 or Apple Shooter. These were both third person 2D games, but we thought that we could create a better, first person 3D version. The control scheme was inspired by common first-person shooter games, but we wanted to create a world where targets spawned around us instead of one where we had to walk around and search for targets on a map.

Our archery mechanics were inspired by the ones in *Legend of Zelda: Breath of the Wild*, in which the user holds down a button to draw back the arrow and can freely look around and aim. The player must account for gravity and wind while aiming, and can watch their arrow fly in a

visible arc through the air. We wanted firing an arrow in our game to give the player the same feeling of satisfaction.

The idea for using paint splatters to color and reveal the environment came about as we brainstormed ways to expand upon the archery concept. Inspiration came from the game *Splatoon*, where gameplay involves shooting paint to both attack opponents and color the stage. We were also inspired by past COS426 projects that used bloom post-processing very effectively, and we felt that stylizing our archery game with neon paint would make it very visually exciting.

Approach

We chose to use ThreeJS for our game because it is a strong, frequently-used, well-documented framework. It had all of the building blocks for the web game we wanted to create while still allowing us great flexibility in writing our own code. Furthermore, we had experience with it through previous assignments for this course, and we knew the course support and scaffolding code would be invaluable in the set-up stages. The bulk of our project was Javascript, but we used some CSS for styling the interface and some GLSL for shader code.

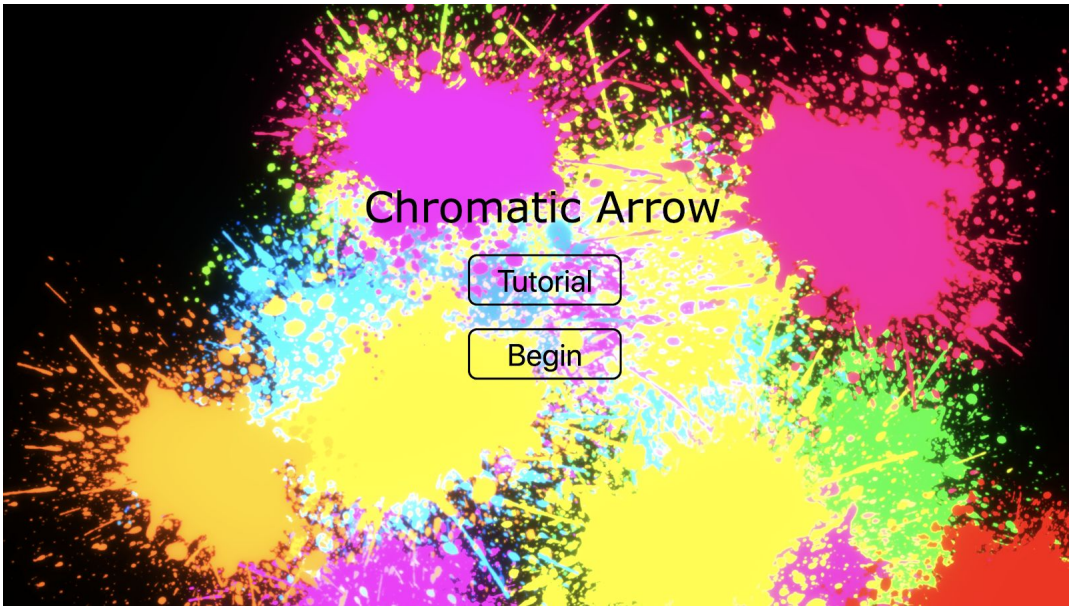
From a game experience point of view, we wanted to create a fast and simple game that would be easy to learn but hard to master. Our game was designed with a simple control scheme (it only uses the mouse) and a straightforward goal (to score points). We expect first-time-users to be able to quickly shoot targets and reveal barriers and have fun, but users trying to maximize their high score will spend time improving their strategy and understanding the game's mechanics, such as wind effects and barrier movement.

2. Methodology

Scenes

In total, our game has four scenes: StartScene, GameScene, InterfaceScene, and EndScene.

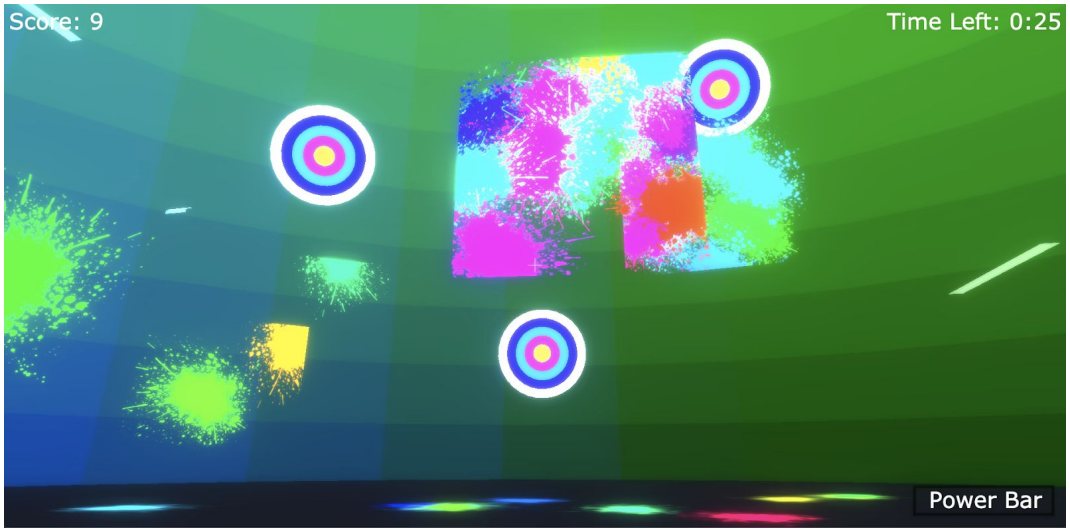
StartScene is the scene that appears when the game is first loaded, and it displays the title of the game, a button for the tutorial, and a button to start the game. The background starts out as black, but randomly generated splatters of bright paint appear in rapid succession. The first five splatter positions are fixed to ensure the title text and buttons are revealed. The splatters happen quickly enough that this still looks random.



The StartScene

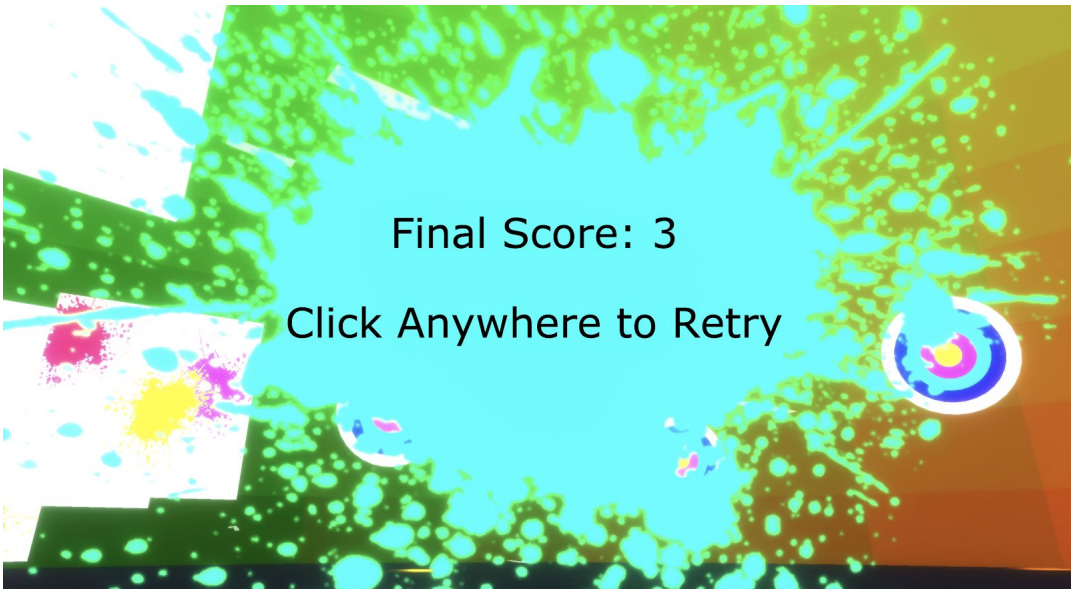
GameScene is the scene in which gameplay occurs. Immediately visible in this world are the dark colored ground and large rainbow dome. The dome limits the user's "game world" and gives them a stronger sense of perspective and distance while still adhering to the colorful aesthetic of the game. Present but invisible, barriers also exist in the world and continuously move around the player. Targets spawn randomly in the world and arrows are shot out from the center, where the player stands. The player cannot move around the scene, so they are able to focus entirely on aiming and firing arrows while using the camera, a ProjectiveCamera, to look around in all directions.

InterfaceScene consists of the elements overlaid atop the screen while the user is playing. This was done using an OrthographicCamera and rendering the scene simultaneously with the GameScene [4]. Its components include the crosshairs in the center, the points score in the upper left, the time remaining in the upper right, and the power bar in the lower right. The power bar changes color depending on the color of the current arrow.



The GameScene and InterfaceScene

EndScene is displayed after a game finishes and is rendered with an OrthographicCamera. It overlays a large paint splatter of a random color atop the game scene, with text showing the user's final score and the option to restart the game. When this scene loads, the underlying GameScene reveals all of the hidden barriers and disables shooting controls. While the retry option was initially displayed immediately, we realized users trying to land a last second shot would accidentally click through the screen. Thus, we added a short delay of one second.



The EndScene

Bloom effect post-processing was applied to all of our scenes using the UnrealBloomPass found in ThreeJS examples [2]. This causes our colorful paint splatters to have a glowing, neon aesthetic.

Random color generation

We initially generated colors by choosing a random hex value. Unfortunately, this often resulted in colors that looked dull or bland. The problem was exacerbated after applying the bloom post-processing filter, which caused bright colors to look washed out. We then decided to change our approach to one based around the HSL color scheme. We choose a random hue, set saturation to 1, and set lightness to 0.5. This more restricted range of colors gave a much improved appearance (especially under bloom) that significantly contributed to the cohesive aesthetics of our game.

Models

We use basic ThreeJS geometries to create all of the objects of our game scene. Although we initially tried to load in models from Google Poly, we decided that we preferred the simplicity and flexibility of composing built-in geometries.

Ground is represented by a large box with y-position = 0. It spans the bottom of the scene and allows arrows to collide and splatter onto it.

Dome is the hemisphere enclosing the area. While we initially set the game in an open field, we were disappointed that arrows could fly far away and leave no visible splatter. To solve this, we added a dome that the user could see, hit, and splatter with their arrows. We used flat shading to color the inside of the dome: using GLSL shaders, every face of the dome geometry was colored a different color by mapping the face normal to an RGB value. This coloring style made the dome's geometry distinct, adding a feeling of depth and perspective as the user looked around the scene, while also contributing to the vibrant aesthetic of the game.

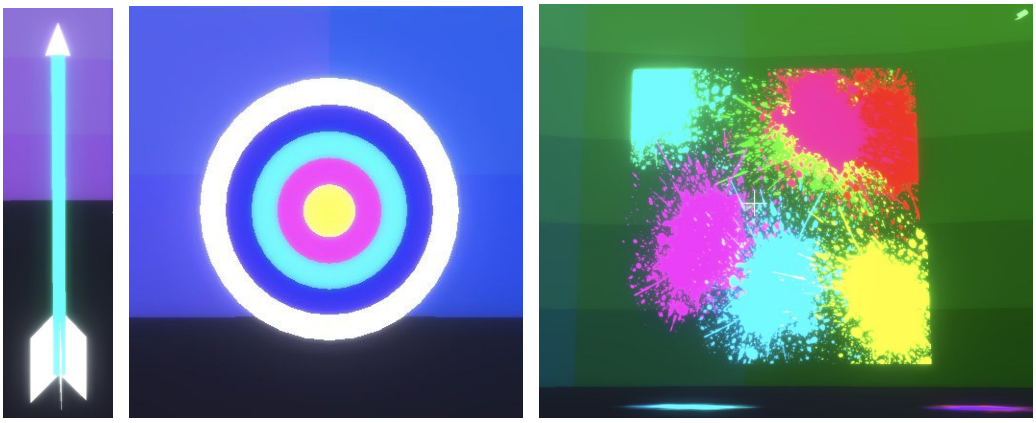
Arrows are built from a long, thin cylinder, with a white cone at the tip and three white parallelogram planes for the tail feathers. Using the TrailRenderer library [1], a trail is attached to the end of the arrow which shows a colorful streak as the arrow flies through the air for greater visibility. Every arrow is assigned a random color as described above, which determines its shaft color, trail color, and splatter color. An arrow's initial velocity is determined by user input. Arrows are affected by gravitational force and wind force. Multiple arrows can exist in the scene at once, and there is no lag or cooldown when firing arrows. An arrow is removed after it collides with any solid object.

Targets are created by combining five flat concentric cylinders, all rotated to face center toward the camera position. They were modeled after targets in an archery range, but colors were altered to make bloom post-processing look more pleasing. They are all the same fixed size and they spawn randomly in a zone between the barrier-spawning zone and the dome. This was achieved by using spherical coordinates to control the distance away from the camera but randomize the relative location around the camera. Also, we made sure that targets did not spawn too close to any existing targets to prevent any overlap. They spawn into the scene in four-second intervals, with a maximum of 10 onscreen at a time.

Barriers are created from flat, square-shaped boxes, which are positioned upright to face the central y-axis. All barriers initially spawn in a random location between the user and the target-spawning zone and move around the user in either a clockwise or counterclockwise path. On their path, they will move up and down with sinusoidal vertical motion. Each barrier has the same base speed and movement but a little noise applied to each one allows them to have unique movement patterns. To avoid barriers colliding, they are spawned at increasing distances away from the camera position, starting at the beginning of the barrier-spawning zone to the beginning of the target-spawning zone. During normal gameplay, the barriers are invisible, and the user is only able to track their presence by splatter meshes from arrows that have hit them. At the end of the game, all of the barriers are revealed to show their positions.

Paint splatters are meshes created using ThreeJS's DecalGeometry [3] and then attached to a given mesh at the location the arrow hit. The splatter also needed to be rotated to align with the impacted surface. Although this approach uses the same fixed image for each splatter, the image is rotated by a random amount each time, which gives the illusion of a new splatter. While we experimented with trying to draw custom shapes on a canvas and convert these into ThreeJS texture objects, none of these attempts yielded convincing results.

Wind is a force applied to the arrow (described below), but it is also visually represented in the scene by transparent, curved surfaces that appear, trace a path, then disappear. This path is determined by the direction and speed of the wind at that moment. The approach was modeled after a similar application to generate a road [5]: a Catmull-Rom spline was used to interpolate a set of points, and then a set of triangular faces was generated from the created spline. The animation then proceeded by only rendering a fixed number of vertices and tracing the length of the geometry as time progressed. Here, the control points began as a straight line in the direction of the wind whose spacing was proportional to the wind speed so that faster gusts led to longer paths. A loop was then added at the end, and the points were more densely packed here so that the object did not clip through itself while tracing.



From left to right: an arrow, a target, an invisible barrier revealed by paint splatters

Wisps of wind

Physics

All of the physics and collisions calculations for this game were coded ourselves, without leveraging physics libraries.

Only fired arrows are affected by physical forces in the game scene. In order to simulate their movement correctly, at every timestep we accumulate the forces on the arrow and then calculate the arrow's next position using Verlet integration using the same process as Assignment 5. We also ensure that the arrow is always pointed in the direction of its motion to give more realistic results.

The two physical forces in the game are gravity and wind. Gravity is applied as a constant downward force, in the negative y-direction. Wind force is applied in the same manner as gravity, but it is not constant. Every 12 seconds, the direction and magnitude changes are chosen randomly (for any direction and between a fixed range of magnitudes).

We also must ensure that the arrow is properly colliding with the solid objects of the scene: the ground, the dome, the barriers, and the targets. The position of the arrow's tip is used when checking for collisions. When a collision with a target is detected, the target is removed from the scene along with the arrow. For collision with the other three solids, the specific position of the collision must be calculated so that a splatter mesh can be displayed there.

Collisions

Since all of our objects were constructed from simple primitives, we checked intersections manually rather than heavily relying on library functions or bounding boxes. We think this gives both good precision and good performance since we can tailor our approach directly to our specific application. We make sure to stop checking for collisions with other objects if a collision is found. For all cases below, the arrow is treated as a single point at the tip.

Ground collision was as simple as checking if the y-position of the arrow tip was within epsilon of zero, the ground's y-position.

Dome collision checked if the arrow position was within epsilon of the dome radius from the origin.

Target collision begins by checking if the arrow tip is in the target-spawning zone. If it isn't, then we assume no targets were hit. For each target, we check if the arrow is as far away from the camera as the target is and if so, we calculate the score that it would get for hitting the target (explained in the next section). If the score is positive, then there is a collision.

Note: calculating the score requires finding the distance to the center of the target, so we used the score instead of calculating the distance twice.

Barrier collision similarly begins by checking if the arrow tip is in the barrier-spawning zone. If it isn't, then we assume no barriers were hit. Because each barrier is always facing the y-axis, the normal vector and distance are known, which means we can calculate its plane. Since we know each barrier is vertical, we can also find its orthogonal basis vectors: the first is parallel to the y-axis while the second is in the xz-plane and perpendicular to the normal. Finally, given these basis vectors and the dimensions of the barrier, we can obtain 2 boundary points for each axis and then check if the plane intersection point is between both pairs of points.

Scoring

There are two ways to score points: hitting a target or hitting a barrier with an arrow. For the target, we take the distance from the arrow tip to the line with endpoints at the camera and the center of the target - this works because targets were rotated to directly face the camera. We divide this distance by the width of each ring of the target to determine how many rings away the arrow tip was and award points starting with 1 for the outermost ring and 5 for the innermost ring (which is a circle). We award zero or negative points if the distance is greater than the target's radius. Barriers keep track of how many times it has been hit, and when a barrier collision happens with a barrier with no prior hits, 1 point is awarded.

Points are scored in the GameScene, but displayed in the InterfaceScene, so we needed a method to share the current score. To do this, we fired a custom event "addScore" from GameScene with the points to be added in the event details. In InterfaceScene, we added an event listener to the "addScore" event that would update the text on screen.

3. Results

During the development process, we play-tested each new component and evaluated the effect on both the aesthetics and gameplay. After obtaining a viable MVP, we used our own thoughts along with the TA feedback to make some substantial improvements. For instance, it was initially hard to track the path of the arrow, so we later added a trail and the bloom post-processing to make colors stand out more. The gameplay mechanics were not immediately obvious, so we also added a tutorial. We knew that the game was successful when we were genuinely interested in playing our own game and competed amongst ourselves to get the highest score. When we were closer to a finished product, we solicited feedback from friends. Some of the more notable responses are listed below:

> "Those barriers can f*** themselves"
>
> "This is art"
>
> "LOL this is nice. I like the wind"
>
> "It's really cool though! I like the bright colors and the splats"
>
> "Thanks y'all for choosing to [make] such a fun project"

These encompass both the playable and aesthetic aspects of our game. The barriers were an ever-present obstacle and the wind affected aiming as desired. Simultaneously, the random color

generation, splatter effects, and dome texture created appealing visuals. This experiment indicated to us that we had accomplished our main design objectives.

4. Discussion

High-level approach

Tutorial

The tutorial leads the user through a scripted interaction with the core aspects of the game. This includes firing an arrow at a target; looking around the scene; dealing with gravity and wind; and hitting invisible barriers. While a simple screen of text may have been sufficient, we thought that approach was very unsatisfying. It is much easier to learn by playing when the player can see and interact with the elements described. Completing the tutorial returns the user to the start screen.

Main game

The user controls the game entirely with their mouse, which simplifies the controls greatly and cuts down on what the user has to learn and remember. The camera direction follows the position of the user fires arrows by holding and releasing click. The arrow's initial speed is then determined by how long the user held click, and the arrow's initial direction is the current direction of the camera.

One game lasts 60 seconds, and the player's goal is to earn as many points as possible within that time frame by hitting targets and barriers. We felt 60 seconds was a good middle ground that gave users enough time to discover barriers and explore the environment while also not allowing gameplay to become repetitive and dull. This makes our game both fun initially and highly replayable. For targets, the point scheme creates a speed versus accuracy tradeoff that is central to gameplay. The user needs to shoot rapidly to hit all targets that spawn but also take the time to aim precisely. The barriers create an obstacle for the player, but also provide points and are easy to avoid when revealed. This incentivizes the user to explore and discover the barriers, which are key components for both the gameplay and visuals.

Low-level approach

Transitioning between multiple scenes and even rendering two scenes on top of each other worked well for displaying the game. While a simple menu screen could be created by overlaying an HTML element, having a separate scene gave much more flexibility and allowed use to reuse existing code. Splattering in the start screen and end screen worked exactly the same as splattering in the game. Being able to create an interface as a separate scene gave even more flexibility and allowed us to easily tie together the aesthetics of the game. For instance, the

powerbar fill and end screen splatter both get the bloom post-processing effect, which means their colors match the rest of the game. Having multiple scenes did make disposing of objects and preventing memory leaks much more challenging, though. We also valued the visuals more than the performance (provided the game ran smoothly), so the tradeoff was worth it.

Follow-up work

Improving current elements

The main element that could be improved is the splatter effect. After drawing our own textures failed, we discovered that just transforming a fixed image was sufficient. Other avenues might have yielded interesting results, though. One idea is to simulate the effect of gravity on the paint splatter. If an arrow hit a vertical barrier, the initial splatter texture would be applied as before, but then we could simulate a "drip" that descended over time. Another idea was to add small particle effects flying off the collision site to give the impression of a real impact. Ultimately, with the feedback we received above and the rapidly approaching Dean's Date, we decided to stick with our naïve implementation that still gave good results.

Future directions

One of the advantages of this game is that there are many ways to extend the core archery concept. Adding background music and sound effects for hit targets and splatter collisions would help enhance the game experience. For the competitively-minded, we could include a leaderboard for high scores by adding a backend. We also would have liked to add power-ups to the game: the user could hit a special, small object with an arrow and gain a special effect, such as firing multiple arrows at once, increasing the amount of time remaining, or temporarily revealing barriers. We could even add additional game modes that alter difficulty or allow the player to simply fire their arrows onto a blank canvas. Sadly, the limited time frame of this project did not allow us to implement these extra features.

What we learned

Through this project, we gained a greater understanding of Javascript and the ThreeJS library, and how ThreeJS's tools can be leveraged to create a simple yet fun game. We learned about game design: what makes a good game, how a game's components all interact with each other, and how to break the game creation process down into separate, actionable steps. We also gained experience in understanding and adapting code from example sources to fit our own needs.

4. Conclusion

We were successful in creating a fun and unique archery game that many of our friends enjoyed playing. We were very happy with how realistic the game physics were and how the neon theme turned out. Specifically, the game was easy to learn, and people had a fun time competing to reach the highest score.

5. Contributions

William started the project by setting up the camera and controls and adding a ground and dome to the world. Then he worked on building targets, barriers, and the scoring system. Anna created the arrow model and added Verlet integration for physical effects and an arrow trail for visual appeal. Also, she set up the bloom post processing and designed the dome and ground aesthetics. David started by making a rudimentary design of the interface, then worked on adding the splatter effect. He continued with creating the gameplay loop (start screen and end screen) that included a tutorial. Finally, he added the wind force and animation.

Links

Repository: https://github.com/WillPower264/Chromatic-Arrow

Live Demo: https://willpower264.github.io/Chromatic-Arrow/

References

A special thanks to ThreeJS, which was used throughout our project: https://threejs.org/

[1] TrailRenderer: https://github.com/mkkellogg/TrailRendererJS/tree/master/js

[2] Bloom post-processing example:
https://threejs.org/examples/webgl_postprocessing_unreal_bloom.html

[3] Decal splatter example and splatter image source:
https://threejs.org/examples/webgl_decals.html

[4] Overlay example: https://threejs.org/examples/webgl_sprites.html

[5] Wind computation inspiration:
https://discourse.threejs.org/t/create-a-curved-plane-surface-which-dynamically-changes-its-size/6161/8