Feline Fishing Frenzy

Henry Wertz, Manya Zhu, Sarah Pedersen

Abstract.

What would a cat dream about? Fishing, of course! Feline Fishing Frenzy features the player as a dreaming cat character named Chub, who floats on the ocean in a boat and catches fish with a fishing rod. In this 3D fishing game, inspired by the nostalgic and discontinued 2D fishing game from Club Penguin, the goal is to catch as many fish as you can while avoiding obstacles that knock off your bait and your fish. The player is limited to a fixed number of bait, but is given the opportunity to collect more bait from the water as the game progresses and becomes more challenging with time. The captivating gameplay, combined with the cute artistic design and the soothing music and sound effects, make Feline Fishing Frenzy the game of the century!

1. Introduction

We took our initial inspiration from the classic Club Penguin ice fishing game, which was sadly discontinued in 2017 (RIP). Feeling nostalgic for the hours spent in our childhood, we sought to reimagine it with some twists/additions that we wished were in the original game. Feline Fishing Frenzy is a new fishing game built using Three.js from the perspective of Chub, Henry's housecat. Since the silly pet turtle (Bro Turtle) never wants to play, Chub is resigned to a life of sleeping all day. In Chub's dreams, however, he and Bro Turtle sail the high seas, catching endless fish.

Playing as Chub, the player controls the fishing line using the mouse, reeling them above water to catch them. But there's a catch: the ocean is filled with obstacles! Pufferfish and turtles knock caught fish off the player's line. Even worse, sharks eat line's bait, and jellyfish shock it. Players start with 3 bait, and if lucky, will find more floating in the water. The game gets progressively more difficult as time passes, as obstacles spawn more often and fish swim faster. Our goal is to create a fun, relaxing (yet difficult!) game that invites players to take a break from reality and beat their personal best.

Figure 1: Title Screen

1.1 Goal

After consulting our advisor and discussing with the group, our MVP was as follows:

To implement at 3D fishing game with the following features:

- Having a cat character in an ocean scene, including a boat and fishing pole
- Various sea creatures (fish, turtles, etc.) “swimming” across the screen, with weighted random generation of which creatures appear and how often
    - The longer you play the game, the more obstacles will be generated.
- The ability to “fish” for fish objects by moving the fishing hook/line using up and down using your mouse
- The ability to collect fish by bringing the fish to the surface of the ocean and clicking your mouse, while avoiding obstacles (hitting a non-fish creature drops your fish)
- A set number of “bait” per game, or how many times the player can attempt to fish
- Jellyfish that appear in the ocean and can shock the fishing line (hitting a jellyfish loses a bait)
- Dynamically keeping the score of the player during the game, showing the score of the player at the end.
- Correct logic for switching between a start page, pause page, game over page, and the game itself.

Additionally, these were our Stretch Goals:

- Waves in the ocean scene that move the boat/fishing hook up and down (adding movement to the game)
- Background music and sound effects (for most significant actions), with the ability to toggle the sound on and off during the game.
- Sharks that appear in the ocean that can eat your bait
- Big fish that are worth more points and harder to catch (heavier, so your line moves slower)
- Meow button (!!!)
- Dark mode/night mode: limited lighting in the sky, sparse underwater lighting
- Have objects swimming in across background – you know when they are coming
- Bonus bait: the ability to collect extra bait from the ocean once the game gets sufficiently challenging

As shown by the specific goals of our game above, our aim was to create a captivating and aesthetically pleasing fishing game that perfectly balances challenging gameplay, strategy, and relaxation for the player! Anyone who was a Club Penguin Ice-Fishing fan, who loves cats and everything cat related, or is simply looking for a fun new game to play would benefit from Feline Fishing Frenzy.

1.2 Previous Work

Club Penguin Ice Fishing Minigame

Club Penguin's Ice Fishing is a popular minigame within the virtual world of Club Penguin, an online multiplayer game that was developed by New Horizon Interactive. The game was later purchased by Disney and eventually shut down in 2017. Our game is very similar to this game, because one of our main goals was to improve upon this nostalgic, discontinued game. Ice Fishing specifically has mouse controlled movements, obstacle avoidance, and a similar frame of reference (character sitting above the water while the player gets a view of the underwater objects in motion).

Stardew Valley: Fishing Minigame, SpiritFarer: Fishing Minigame

Many different games include fishing minigames. Two examples of fishing minigames within farming simulation/fantasy management games are the fishing games in Stardew Valley and SpiritFarer, which operate similarly. In these minigames, the player must cast a reel into a body of water, then must keep a “meter” or “rod tension” within a certain limit for a certain length of time. This is often controlled by either pressing and releasing a computer mouse or a controller button at the right time, while paying attention to a dynamic meter. Additionally, there are no obstacles to avoid in the water, only varying types of objects to reel in that are based on luck (ex.

normal fish, rare fish, garbage, seaweed. etc.). Where these games fall short is their simplicity of gameplay and their unintuitive movement control.

1.3 Approach

Feline Fishing Frenzy: Our Approach

Our game is unique because unlike Club Penguin Ice Fishing, we offer a fuller 3D experience and two modes: day mode and night mode. Additionally, what differentiates our game from the other fishing minigames is the full underwater visibility, the intuitive mouse movement controls (using raycasting), the infinite gameplay, and the progression of difficulty in stages. Feline Fishing Frenzy is the intersection between real time strategy games and cozy games – it offers the player a challenge with many moving elements, but also has a satisfying movement control and aesthetically pleasing design.

2. Methodology

2.1 User controls – mouse & raycasting, reeling in fish when click (MANYA)

One of the most important aspects of a game is its user controls. Since we wanted to develop a game that would be very satisfying, relaxing, and intuitive to play, it was paramount that our handling of controls was seamless. Mouse movement is often considered one of the best user controls. First and foremost, it provides precise and fine-grained control, allowing users to navigate and interact with graphical interfaces with high accuracy. The fluidity and versatility of mouse movement make it well-suited for a fishing game, and it aligns with the natural movements of the hand. Moreover, the mouse has become a standard input device for desktop computers, contributing to its widespread acceptance and familiarity among users. This consistency across systems enhances user experience and reduces the learning curve when transitioning between different devices.

In order to implement the mouse movements, we have an event listener that calls `handlePointerMove(event, pointer)` on every `pointermove` event, and `handleMouseDown(event, scene, sounds, playSound)` at every `mousedown` (click) event. Additionally, at every update step, we call the function `handleCharacterControls(scene, pointer, raycaster, camera)`.

First, `handlePointerMove(event, pointer)` simply sets the x and y values of the pointer variable to the normalized horizontal and vertical positions of the mouse pointer within the viewport, ranging from 0 to 1. Then, `handleCharacterControls(scene, pointer, raycaster, camera)` uses the pointer value and the camera from `app.ts` to set a raycaster,

using `raycaster.setFromCamera(pointer, camera)`. In order to get the correct y-world coordinate of where the pointer is raycasted onto the 3D screen, we use an object called "gamePlane," which simply is a transparent yz-plane placed at x = 0: this plane is where all the fish, obstacles, line, hook and bait spawn. We raycast the pointer to the gamePlane using `raycaster.intersectObject(gamePlane)` and extract the y-value from the intersection. This determines what the y position of the end of the reel, the hook, the bait, and the fish (if a fish is on the hook) should be at. Now, the reel, hook and bait follow your mouse movement up and down the screen

Finally, `handleMouseDown(event, scene, sounds, playSound)` handles whenever the mouse is clicked. If the hook is above the water level on the click and has a fish, we increment the player's score, play the appropriate sound effect, and "catch" the fish. Otherwise, if the hook is still below water level, the fish swims away from the hook and the player has dropped the fish.

2.2 Collisions

The key goal of the game is to bring fish to surface without losing them; therefore, the challenge must come in the form of obstacles. At every update step, we call the function `handleCollisions(scene, sound, playSound)`. We considered handling collisions in objects' "update" functions, but since the collisions affect other objects in the scene (nearly always involve removing/adding things to the scene), we chose to handle all collisions in a separate function. Many of them share logic elements, such as the result when the fish "swims away" off the hook, so this also allows us to modularize that element.

We use the Three.js bounding boxes to handle our object collisions. We use the `scene.getObjectByName` function to obtain the hook, reel, and bait since there is only 1 object of those names on the screen at any particular time. We then create bounding boxes using Three.js Box3().setFromObject for each component.

Our scene's state has lists for fish and each potential obstacle on screen at that time. We considered putting all obstacles in one list, but as they have different properties for different purposes (such as the jellyfish/shark needing to be active or inactive), we opted to split the lists to have more specificity with the object types in TypeScript. As updates to the scene are run, we remove objects that have passed through the field of view by checking if their z-position is outside the edge of the screen in 3D coordinates. Therefore, we can loop through the current lists and create a bounding box for each object in view. We check for collisions in the following cases:

- Anytime an extra bait intersects the hook: add a new bait to the scene's state variable numBait. Remove the extra bait from the scene and updateList.
- If we have bait on the hook:

- Hook hits a fish: If no fish is on the hook, attach that fish to hook, and stop updating its motion.
- Hook **or** line hits a jellyfish: Lose a bait. If a fish is on hook, re-active it + swim away.
  - Make jellyfish's state "inactive" to prevent the remaining timesteps that it intersects from reducing bait.
  - Bait "falls off" in the water.
- Hook hits a shark: Lose a bait. If a fish is on hook, re-active it + swims away.
  - Make the shark's state "inactive" to prevent the remaining timesteps that it intersects from reducing bait.
- Hook hits a blowfish or turtle: If fish is on hook, re-active it + swim away.

2.3 Fish spawning algorithm for better gameplay (HENRY)

We originally implemented the fish spawning as entirely based on randomness throughout the game with a basic model in which there was a high likelihood of spawning fish and lower likelihood for obstacles. This entirely-random model, however, made the game feel chaotic, especially when multiple obstacles would spawn on top of one another. Therefore, we developed a more organized spawning algorithm.

This algorithm is rooted in four state variables for the scene: `spawnIntervals`, `spawnTimers`, `stage`, and `spawnSet`. `spawnIntervals` is a key (string) / value (number) dictionary that denotes how *often* (in real time) to spawn each sea creature when they are actively being spawned. This manages the rarity of sea creatures: for example, we want fish to be more common than sharks, and we want extra bait to be the most rare. We regulate the spawnIntervals with `spawnTimers`, which are initialized to 0. This is also a key (string) / value (number) dictionary that keeps track of how long it has been since the sea creature was last spawned until reaching the interval length. This way, we can reset them whenever the creature is put in/taken out of the spawnSet.

`stage` allows us to create a cohesive progression throughout the game and `spawnSet` keeps track of which entities are supposed to spawn during the current stage. The spawning is divided into 5 stages. At the beginning of the game (stage = 0), only fish are spawned to allow the user to get accustomed to gameplay and controls. This stage lasts for 10 seconds, whereas the later stages are longer. Each stage is progressively more difficult as more obstacles are added to the `spawnSet`; at stage 4, the fish speed is 8 (compared to 2 when the game starts), and shark, jellyfish, turtles, and pufferfish are all on screen.

2.4 Night mode

To provide the player with different experiences (or if they're playing it late at night like we are and want it dimmer), we decided to create a "daytime" scene and a "nighttime" scene. Players are able to toggle between the two scenes on the start page before beginning the game.

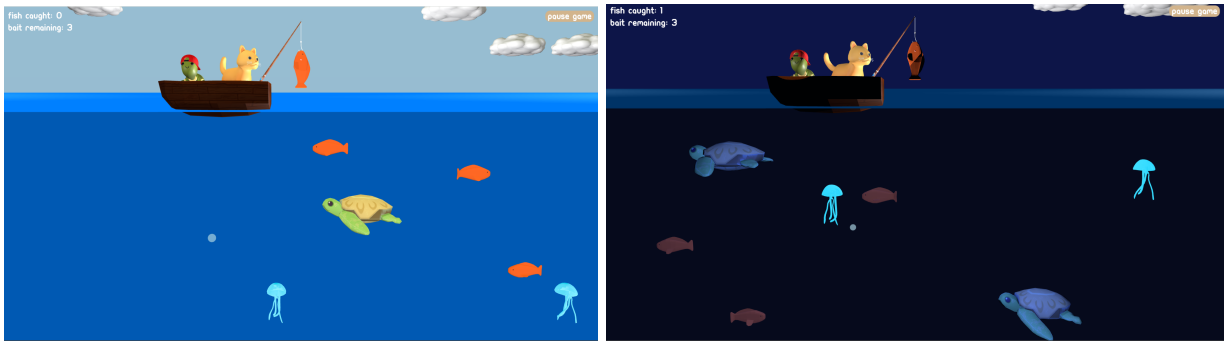

Figure 2/3: Day versus night mode scenes

To create these different effects, we altered the lights that go into the creation of the scene. We did this through the `BasicLightsNight.ts` class. Rather than using DirectionalLights or HemisphereLights, the night lighting has a series of SpotLight objects for each "bright" element of the scene. We set the position of these lights strategically to create an immersive effect. For example, we have one light set in the boat illuminating the cat and creating a shadow of the reel/boat on any fish brought up to the surface. We also have two lights focused on the water to darkly illuminate the fish/obstacles that pass through. This really emphasizes the "glowing" effect of the jellyfish.

When the app screen loads, we build a scene for each mode. They are from the same `SeedScene` class, but we pass through a parameter that creates a different light object in the scene depending on the mode. We then render the scene that corresponds to the toggle.

2.5 UX/look and feel

2.5.1 Loading Manager

To ensure a smoother user experience, we decided to use a loading manager for a few of our larger and more complex models. This way, while the initial app is rendering, the user is shown an aesthetically pleasing loading screen instead of seeing the page render piece by piece. This was done by initializing a loading manager before loading the day and night scenes, and then displaying the loading page on `manager.onStart` and removing it on `manager.onLoad`. The loading page is a CSS/HTML animation of blue waves within a circle, which matches the theme of the entire game. The loading page is adapted from an open-source implementation credited in the works cited.

Figure 4: Loading screen

2.5.2 World Building Details and Movement

In order to fully build out the 3D world and to immerse the player, we added subtle movements, fog, and other design choices. First, we have the boat, cat, and bro-turtle bobbing up and down in the water throughout the entire game. Next, we added 3D cloud models that float across the sky, at random y and x values (to show depth). We also have bubble objects that float up from the ocean floor in a sine wave motion, to the surface of the water. Finally, we have each shark swimming across the background hidden by the fog, before they eventually come around and swim across the foreground and become a real obstacle. This adds to the strategy of the gameplay, where users are given a warning before having to encounter the large, fast-moving, deadly shark obstacle. It also contributes to building out the world further, making the 3D ocean scene more believable.

2.5.3 Background Music and Sound Effects

To make the game more immersive, we wanted to add audio in the form of background music and sound effects. We implemented this by using the AudioLoader/Audio Three.js libraries and finding audio using open source website FreeSound.org. We initially encountered issues because Chrome/other browsers do not allow auto-play of sound until the user interacts with the site. We fixed this by setting up the audio context and loading the sounds after the user clicks the “Play Game” button. The background music Audio object is loaded into a global variable; depending on whether the user toggles background music on/off, the audio is either played or stopped. We have a global variable holding the sound effects in the form of a dictionary where the keys are strings and the values are Audio objects. We pass this “sound” global variable and another global

variable controlled by the sound effects enabled toggle into our event handlers, which play the respective sound effect for the action occurring on screen if sound is enabled. We ran into another issue in which the audio loader failed to decode the files when we kept them locally, so we uploaded them to an online storage site (JukeHost) that allows us to create links to the audio files.

2.5.4 Font and Design

While aspects like font and page design might seem trivial, they actually are a large part of making the game have a complete and polished look and feel. We chose a primary font called "Orange Juice," which we felt represented the whimsical, imaginative world we built. This font was used for all our titles and bigger texts. Then, we chose a secondary font called "Ubuntu-Title," which was more simple and readable, but still had the soft, rounded feel to complement the primary font. In addition, all our pages have cute, fantastical graphics and gifs to further enhance the cute and relaxing aquatic theme of our game.

2.5.5 Page Switching

In our HTML, we created different `div` elements for each page. We "switch" between pages by altering the visibility of these dividers. We encountered some issues importing HTML directly into our TypeScript, which resulted in us using a separate HTML "main" file and editing it in the TypeScript (rather than the other way around). We attach the rendering tools/scene to the `app` HTML element by getting it from the document in `app.ts`. Based on user input, such as clicking the play button, we have event handlers for those actions to toggle the visibility of the `start-screen` div to `none` and the app screen to `initial`. Similar transitions occur when the user clicks pause or the game ends.

3. Results and Discussion

To assess the success of our game, we made our friends play it incrementally throughout development, adjusting it based on their changes and expectations. For example, in an early version of the game, sharks played the same role as the blowfish and turtles. One person said they expected the shark to be more dangerous; this makes sense, so we changed the shark to "eat" the bait off the player's line to lose a life. When we were talking about how we wanted to implement the crabs, one person said they did not think it made sense because there would be no surface for the crabs to walk on to "cut the line" since the game was set in the open water. This made sense, so we added the jellyfish as an alternative that could swim.

When making these and any other changes, we also played the game ourselves. In response to the shark change, we felt the game was more interesting and difficult. Our primary guiding

principle in creation was finding things to make the gameplay more fun and relaxing. This included details like the clouds floating in the background or bubbles floating up on the screen. We also wanted to create “easter eggs” for players – this is not documented anywhere in the instructions, but pressing the spacebar actually makes Chub “meow” (by playing audio). These little details were super important to use in creating a game to take a basic concept and make it extremely satisfying. We also stress tested the game by playing it in weird ways, intentionally hitting obstacles or making things collide unexpectedly.

Ultimately, we were very happy with our approach, and the final game turned out to be a lot more extensive than we expected. We accomplished multiple of our stretch goals, including the audio, sharks, day/night mode, objects in the background (sharks), and bonus bait. Follow-up work for the game could investigate fine-tuning the spawning algorithm to balance out things that are currently random (such as the number of fish that swim onto the screen from the right side versus the left side). Additionally, we would be interested in adding some kind of visual for the obstacles like a bloom effect on the line when hit by a jellyfish. We also would create a more visual instruction page - such as a picture of a shark next to its role in the game (“knocks off your bait”).

We learned a lot through the experience, and it tested our ability to try out new tools by exploring documentation and examples. None of us had ever used TypeScript before, which was a fun challenge. The typing, while sometimes annoying, forced us to think more about our design and implementation *before coding*, which was a good experience in the long run. The Three.js library was also very helpful. For things that we initially thought of implementing on our own (such as the collisions), we found we were able to take advantage of Three.js.

4. Conclusion

Overall, we feel that we effectively obtained our goal in making a fun, challenging, and visually-appealing game that fulfills what is missing from our childhood. We completed our MVP and accomplished multiple of our stretch goals, even adding elements that we came up with throughout the development process (like the “extra lives"). We encountered issues throughout the process – fish moving in weird directions (up into the air??!), spawning in the middle of the screen, collisions bringing the bait count to -30, and audio playing unexpectedly. For each issue, we found a way to debug it, usually re-factoring and making better code along the way. Some larger issues we encountered were the development of the algorithm for timed fish spawning and creating different timed “stages” of the game to make the gameplay more cohesive.

If we were to build on the game, we would add more levels with different obstacles or even new scenes (since the concept is a dream sequence, one we discussed was a “sky” scene where Chub

sits on the moon and fishes amongst the clouds). Additionally, we would vary the types of gameplay, such as doing a timed "challenge" mode in which players compete to see how many fish they can catch in one minute. There are so many different fun and creative ways to extend the game, which we would love to do in the future.

5. Contributions

Henry

- Initial setup of the scene / layout of the objects + camera
    - Figuring out initial loading in of objects and how to position them
- Class design for entities
- Raycasting the mouse movement and implementing the reel
- Collisions
- Fish spawning algorithm
    - Timed spawning
    - Stages of gameplay with different obstacles
- Lighting for day mode
- Night mode
    - Underwater lighting

Manya

- Raycasting the mouse movement
    - Having the reel, hook, and bait follow the y-position of the mouse pointer
    - Handling mouse clicks for catching the fish/dropping the fish
- Collisions
    - Basic preliminary collision logic, making bounding boxes
- Ocean scene
    - Creating the ocean surface and sky in the background, creating the depth, dimensions, and patterns
    - Boat/cat/turtle bobbing up and down in the ocean continuously
    - Clouds randomly spawning and moving across the sky
    - Bubbles randomly spawning and moving through the ocean in sine waves
- Bait
    - Loaded in the model and attached it to the hook
    - Falls off the hook when colliding with pufferfish or shark in sine wave motion
- General design / start page
    - Loading page: adapted from open source

Sarah

- HTML + CSS logic/setup

- Audio loading and integration
- Reeling fish in when user clicks & fish is above water
- Framework between app / SeaScene / handlers
    - Maintenance of state variables during interactions between the app, the scene, and the HTML (i.e., toggling between modes, displaying score on screen, toggling background audio)
- Bait: Restocking bait button, extra bait floating in ocean

6. Works Cited

6.1 Code:

- Loading screen animation: https://codepen.io/FauxyCoder/pen/LYYmKJG
- Computing the size of the window in 3D coordinates: https://discourse.threejs.org/t/functions-to-calculate-the-visible-width-height-at-a-given-z-depth-from-a-perspective-camera/269
- HTML inspired by Lightsaber Dojo project from last year: https://github.com/arcturus3/lightsaber-dojo/blob/master/src/main.ts#L47

6.2 Models:

- Bait: "Low Poly Earthworm" (https://skfb.ly/6WCTs) by singingstranger is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).
- Boat: "Boat lowpoly" (https://skfb.ly/6EFnt) by Hawsky is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).
- Cat: "Toon Cat FREE" (https://skfb.ly/6TKrP) by Omabuarts Studio is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).
- Cloud: https://sketchfab.com/3d-models/clouds-116f49c23c4347eba340d0f59b0601f7 by farhad.Guli (https://sketchfab.com/farhad.Guli)
- Cool Turtle: "Cool Turtle" (https://skfb.ly/oANOo) by Sebastian Zayas is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).
- Fish: This work is based on "Fish - Low Poly - Rigged - Animated" (https://sketchfab.com/3d-models/fish-low-poly-rigged-animated-77b7c7c352c94ab08081ba221858c9c2) by kharalos (https://sketchfab.com/kharalos) licensed under CC-BY-4.0 (http://creativecommons.org/licenses/by/4.0/)
- Fishing Rod: "Old Fishing Rod plus Substance Config" (https://skfb.ly/o6TVu) by Dherian.Chacon.Chinchilla is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).
- Hook: This work is based on "Fish hook from Poly by Google" (https://sketchfab.com/3d-models/fish-hook-from-poly-by-google-927dea47c26a48fb8d2c99f8b12bfce2) by IronEqual (https://sketchfab.com/ie-niels) licensed under CC-BY-4.0 (http://creativecommons.org/licenses/by/4.0/)

- Jellyfish: This work is based on "Simple Jellyfish" (https://sketchfab.com/3d-models/simple-jellyfish-f77876d8297846eeb23c4ad82dbebb9) by ricksticky (https://sketchfab.com/ricksticky) licensed under CC-BY-4.0 (http://creativecommons.org/licenses/by/4.0/)
- Blowfish: Blowfish from Poly by Google (https://sketchfab.com/3d-models/blowfish-from-poly-by-google-1d9badce84914d58ab91825db85942a3) by IronEqual (https://sketchfab.com/ie-niels)
- Shark: This work is based on "Low Poly Shark" (https://sketchfab.com/3d-models/low-poly-shark-58eddd6fbc2448c38efd1e3df3d0f342) by AlienDev (https://sketchfab.com/AlienDev) licensed under CC-BY-4.0 (http://creativecommons.org/licenses/by/4.0/)
- Swimming turtle: This work is based on "Torti - Stylized Turtle" (https://sketchfab.com/3d-models/torti-stylized-turtle-29b083db392a468c8726e5b42d644c29) by Bato Balvanera (https://sketchfab.com/batobalvanera) licensed under CC-BY-4.0 (http://creativecommons.org/licenses/by/4.0/)

6.3 Images

- Sleeping cat gif: https://giphy.com/stickers/l0MYtNH6ZJPrlcOic
- Waking up cat gif: https://tenor.com/view/cat-kitty-waking-up-wake-up-hungry-gif-19249075
- Bait.png (restock bait button): https://www.freepik.com/icon/fishing-baits_9263736

6.4 Audio:

- Background music: https://www.bensound.com/royalty-free-music/track/the-elevator-bossa-nova
- Harpsichord dream effect (game start): https://freesound.org/people/BrianKatz/sounds/221275/
- Success jingle (turtle spin): https://freesound.org/people/JustInvoke/sounds/446111/
- Eating sound effect (shark eating bait): https://freesound.org/people/maugusto_sfx/sounds/521253/
- Soothing waterdrop (reel in fish): https://freesound.org/people/MATRIXXX_/sounds/702806/
- Typewriter ding (catch bonus bait): https://freesound.org/people/LeonelAle71/sounds/679016/
- Door bump (obstacle collision): https://freesound.org/people/Superex1110/sounds/77525/
- Tin can (refill bait): https://freesound.org/people/PhreaKsAccount/sounds/46276/
- Electric shock (jellyfish sting): https://freesound.org/people/inferno/sounds/18381/
- Cat meowing (space bar): https://freesound.org/people/lextrack/sounds/333916/

6.5 Fonts:

- orange juice (c) Brittney Murphy 2013, www.brittneymurphydesign.com, info@brittneymurphydesign.com
- ubuntu title font (https://www.1001freefonts.com/ubuntu-title.font), Andrew Fitzsimon