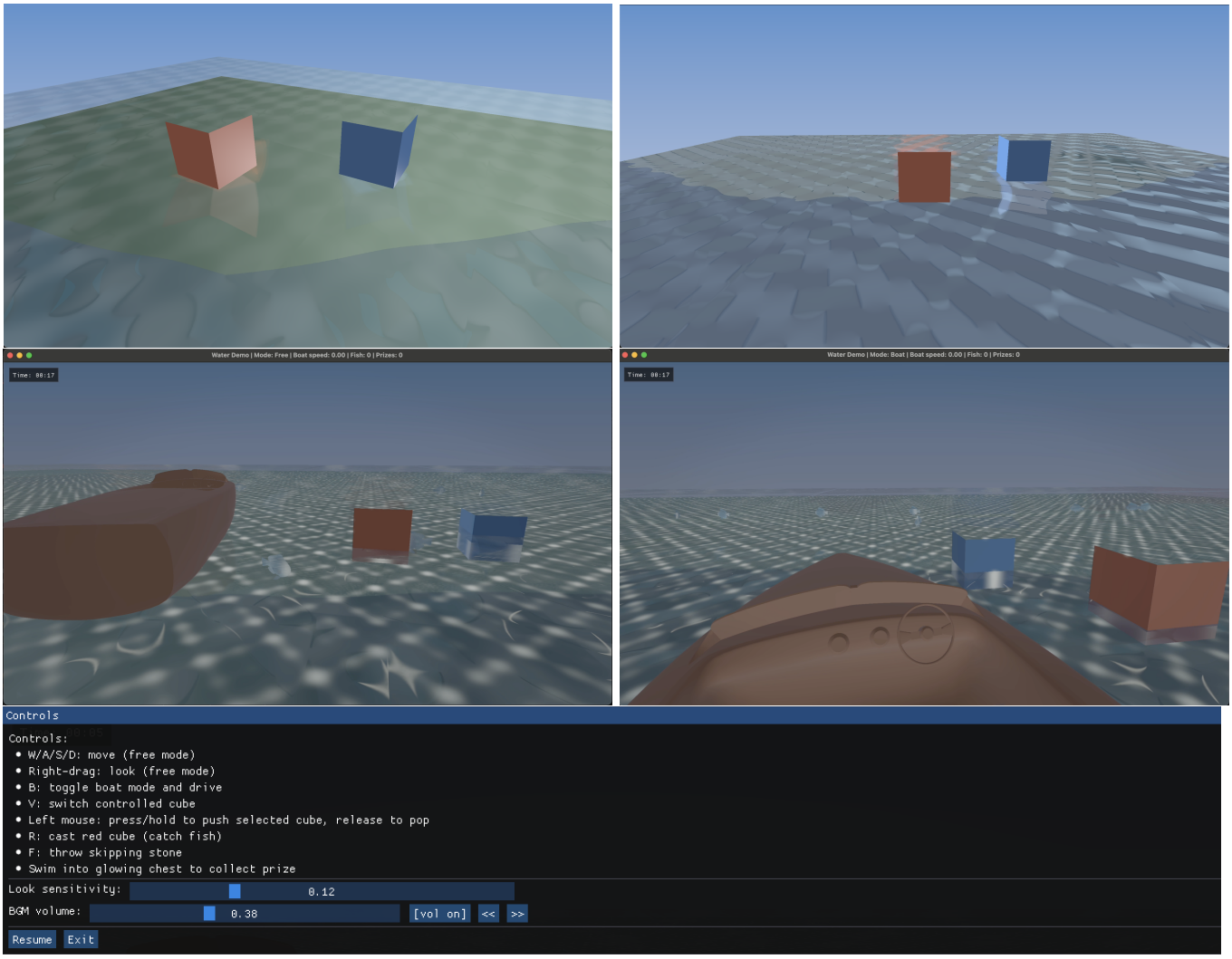
CS1750 Final Project Report

Iris Zhang, Audrey Zhang

December 2025

1 OpenGL Water World: Real-Time Waves, Interactions, and Visual Effects

1.1 Photos of Process



1.2 Overview (Video Demo: `https://youtu.be/X6Ih3aCoQeQ`)

This project is an interactive OpenGL/GLFW sandbox centered around a physically inspired water environment that supports stone skipping, fishing with a lure, a drivable boat, floating cubes, wandering fish, treasure chests, an ESC menu panel, and audio. The main loop and render pipeline are implemented in `main.cpp`, which coordinates input, physics updates, timing, rendering passes, and high-level counters such as fish caught and prizes collected.

`Math.*` defines vector/matrix operations and transforms, `Mesh.*` handles basic mesh creation and OBJ loading, `Waves.*` encapsulates Gerstner wave evaluation, and specialized systems for stones, rod/lure, boat, fish, chest, input, and audio are implemented in their respective `*.cpp` modules. Rendering helpers, including shader compilation and FBO creation, are in `GLHelpers.*`.

1.3 System Architecture

On startup, the application initializes GLFW/GLEW, sets up an OpenGL 3.3 core context, and configures Dear ImGui for in-game UI. It then builds a set of framebuffers: a reflection buffer, a scene buffer (color+depth for refraction and light shafts), an HDR buffer, ping-pong bloom buffers, an LDR buffer, and a shadow map depth framebuffer. SDL2_mixer is used to initialize an `Audio` instance, which loads background music and a library of sound effects. Meshes for the ground, cubes, and water plane are generated procedurally, while the boat, fish, and chest are loaded from OBJ files.

Each frame follows a consistent structure. Input is polled and processed into global camera state and mode flags (free vs. boat, menu visibility, cube selection, stone/rod charges). Gameplay systems are updated in a fixed order: cube buoyancy, stones, rod/lure, ripple pruning, fish, chest spawning/collection, and boat motion (including cube pushing). Time-of-day and sun direction are advanced based on elapsed real time. Rendering then executes a sequence of passes: directional shadow map, scene prepass for refraction, planar reflection, HDR scene (sky, geometry, water), bright-pass and separable bloom, light shafts, tone mapping, and FXAA. Finally, ImGui overlays (ESC menu, time HUD, chest screen-glow) are drawn and the back buffer is swapped.

2 Rendering and Water Simulation

2.1 Water and Wave Model

The water surface is defined by a sum of Gerstner waves configured in a global array. For each wave i, with unit direction $\mathbf{D}_i$, amplitude A_i, wavelength L_i, steepness Q_i, and phase speed c_i, the wavenumber is $k_i = 2\pi/L_i$ and the phase is

$$\phi_i(\mathbf{x}, t) = k_i\, \mathbf{D}_i \cdot \mathbf{x} + c_i t,$$

where $\mathbf{x} = (x, z)$ is the horizontal position. The displaced position of a water particle is

$$x'(t) = x + \sum_i Q_i A_i D_{i,x} \cos\phi_i,$$

$$z'(t) = z + \sum_i Q_i A_i D_{i,z} \cos\phi_i,$$

$$y(t) = \sum_i A_i \sin\phi_i.$$

The function `evalGerstnerXZ()` evaluates this at a given (x, z) and time and returns (x', y, z'). A global `kWaterHeight` offset (set to 0.0 in our implementation) is added so that the rest state water plane sits at $y = 0$, and this same function is used consistently for visual shading, cube buoyancy, and boat bobbing. Fish are generally kept near the surface using a simple height offset/oscillation rather than sampling the full Gerstner displacement in all states.

Geometrically, the water is rendered as a tiled ground mesh centered around the camera to give the illusion of an infinite lake. The water shader receives the main view-projection matrix, a reflection view-projection matrix, the scene color/depth textures, and procedurally generated DuDv and normal maps. Depth-aware refraction samples the scene depth buffer to vary refraction strength based on estimated water depth, while the normal and DuDv maps introduce high-frequency ripples on top of the Gerstner shape. Foam is applied where normals become steep or at grazing view angles, and Fresnel terms with tunable bias and scale blend between reflection and refraction.

2.2 Ripple System

On top of the base Gerstner surface, the simulation adds localized ripples triggered by gameplay events. A fixed-size array of `RippleEvent`s stores the center position, start time, and active flag of each ripple. When a stone skips or a lure hits the water, `addRipple()` records a new event and sets flags consumed later by the audio system to play splash sounds. In addition, stone collisions with cubes or the boat generate ripples at the impact point. `pruneRipples()` deactivates ripples whose age exceeds a short lifetime so the accumulation cost remains bounded.

The ripple height contribution at a position $\mathbf{x}$ is computed by

$$h(\mathbf{x}, t) = \sum_j h_j(r_j, t_j), \quad r_j = \|\mathbf{x} - \mathbf{x}_j\|, \quad t_j = t - t_{0,j},$$

where each active ripple contributes a damped radial sine wave with fixed global parameters:

$$h_j(r, t) = S\ \sin(\omega(r - vt))\, e^{-\alpha r}\, e^{-\beta t}.$$

Here ω, v, α, β, and scale factor S are hard-coded constants chosen empirically to produce visually plausible ripples rather than being configurable per ripple.

`rippleFieldHeight()` evaluates this sum on the CPU for physics, while `rippleFieldGrad()` computes a finite-difference gradient in x and z to estimate slope from this fixed ripple field. This gradient nudges floating cubes down the ripple slope, and the same ripple events are uploaded to the water shader so visual ripples match the physical perturbations.

2.3 Rendering Pipeline and Post-Processing

The rendering pipeline uses several offscreen framebuffers. A directional shadow map (depth-only texture) is rendered first from a sun-relative orthographic view. A scene framebuffer then captures solid geometry (ground, cubes, boat, stones, fish, chest, rod) with color and depth, using a shader that supports directional shadows, water-height-based fog above and below the surface, and simple caustic modulation driven by time and depth. This scene buffer is later used to compute refraction and light shafts.

Planar reflection is handled by rendering into a dedicated reflection framebuffer with the camera mirrored about the water plane and a clip plane enabled to discard fragments below the water. This reflection texture is mipmapped to reduce shimmering at glancing angles. The main HDR buffer then receives the sky (a full-screen gradient whose top/horizon colors interpolate between day and sunset based on sun height), followed by the lit geometry and finally the water surface, which samples both the reflection and scene textures along with the shadow map and ripple data.

Post-processing operates in screen space. A bright-pass shader thresholds the HDR buffer to isolate highlights, which are then blurred using a separable Gaussian filter ping-ponged between the two half-resolution bloom buffers. A light shaft pass reads the scene depth and a projected sun position and performs a radial blur along the vector from each pixel to the sun, with parameters controlling decay, density, weight, and exposure; this produces "god rays," especially visible when underwater. A tone-mapping shader combines the HDR color and bloom, applying an exposure curve (e.g., $C_{\text{out}} = 1 - e^{-EC_{\text{hdr}}}$) and a configurable gamma. Finally, an FXAA shader reads the tone-mapped LDR buffer, uses local contrast to detect edges, and filters them to reduce aliasing before presenting the final image to the default framebuffer.

3 Gameplay Objects and Interactions

3.1 Boat

The boat is represented by a `Boat` struct containing position, yaw angle in degrees, scalar speed, and an `active` flag. `updateBoat()` implements simple arcade-style physics. Given input booleans for forward/backward and left/right, it applies acceleration

$$v \leftarrow v + a_{\text{accel}}\, s_f\, \Delta t,$$

where s_f is $+1$ for forward, -1 for backward, and 0 otherwise, with clamping to a maximum forward speed and a reduced backward speed. Yaw is updated by $\pm\omega_{\text{turn}}\Delta t$ when turning. The forward direction is

$$\mathbf{f} = \texttt{boatForward}(b) = (\cos\theta, 0, \sin\theta), \quad \theta = (\text{yawDeg} + 90^\circ)\frac{\pi}{180},$$

and position advances in the XZ plane as $\mathbf{p} \leftarrow \mathbf{p} + \mathbf{f} v \Delta t$. A lateral component proportional to turn input is added using the right vector to simulate side slip, and speed is exponentially damped by $v \leftarrow v\, e^{-1.2\Delta t}$ for gradual coasting.

Vertical motion is coupled to the water surface. Each frame, the boat samples the Gerstner height at its XZ position and defines a target height y_{target} slightly above the surface. The boat's y coordinate is interpolated toward this target to create bobbing:

$$y \leftarrow y + (y_{\text{target}} - y)\,\lambda,$$

with λ chosen small so motion is smooth. Boat-cube interactions are handled via `pushCubesWithBoat()`, which treats both the boat and cubes as spheres in XZ and pushes cubes out along the collision normal if their centers lie closer than the sum of radii. In boat camera mode, the camera is locked to a "seat" point near the bow, and an additional yaw state smoothly eases the view direction back toward the hull heading while still allowing look-around via mouse and arrow keys.

Steering the boat required a lot of tuning. One issue was keeping the boat's bow correctly positioned in view during large camera turns. Because the visual model has an orientation offset relative to the physics forward direction, early versions caused the `W/A/S/D` inputs to move the boat in wrong directions. This was resolved by realigning the boat's yaw to the camera's view direction whenever steering input is given.

3.2 Floating Cubes

3.2.1 Buoyancy, Spring Movement

One interaction implemented was a "press down, spring up" behavior for the floating cubes in the water. The idea was that the user would press the ((`V`)) key to choose which cube to control, then hold down the left mouse button to push that cube down into the water, and upon release, it would pop back up with a distance/force proportional to how far down the cube was pushed. This was layered on top of the buoyancy system implemented earlier (where each cube has a vertical velocity and target height just above the water surface, computed from the Gerstner waves and ripples).

The target height is computed from the Gerstner waves and ripple field:

$$y_{\text{surface}} = kWaterHeight + y_{\text{Gerstner}}(x, z, t) + h_{\text{ripple}}(x, z, t), \qquad y_{\text{target}} = y_{\text{surface}} + h_{\text{offset}}.$$

The cube's vertical velocity v_y is then updated using a spring-damper rule

$$v_y \leftarrow v_y + k_{\text{stiff}}(y_{\text{target}} - y)\,\Delta t, \qquad v_y \leftarrow v_y\, e^{-k_{\text{damp}}\Delta t},$$

and the cube's height is integrated as $y \leftarrow y + v_y \Delta t$. Adding a spring-damper update made it so that the cube would bob naturally on the surface of the water.

The first attempt involved driving the cube's position directly toward a point below the water surface based on how long the button was held (forcing a lower y_{target}). This ended up fighting the existing buoyancy code, however, so the result just failed. Eventually, the interaction was reworked to solely manipulate the cube's vertical velocity: pressing down on the cube would apply a downward acceleration each frame:

$$v_y \leftarrow v_y - a_{\text{press}}\Delta t, \qquad \texttt{cubePressTime} \leftarrow \texttt{cubePressTime} + \Delta t,$$

and accumulate a `cubePressTime` variable; upon release, an upward impulse from `cubePressTime` was computed and added to the cube's velocity:

$$v_y \leftarrow v_y + \big(J_0 + J_1 \cdot \min(\texttt{cubePressTime},\, T_{\max})\big),$$

where J_0 provides a base upward kick and J_1 scales with how long the cube was held underwater, capped by $T_{\max}$ to prevent excessive launches. That spring-damper system then took over to create that pop-up motion.

3.3 Skipping Stones

Skipping stones are managed in a small global pool `g_stones`, each storing position, velocity, lifetime, and bounce count. Holding `F` charges a normalized value in $[0, 1]$ that maps to a downward throw angle α and speed s on release. The throw direction is

$$\mathbf{d} = \text{normalize}(\mathbf{f}_{\text{flat}} \cos\alpha + (0, -\sin\alpha, 0)),$$

so the initial velocity is $\mathbf{v}_0 = s\,\mathbf{d}$.

`updateStones()` integrates motion with gravity $\mathbf{g} = (0, -9.81, 0)$:

$$\mathbf{v} \leftarrow \mathbf{v} + \mathbf{g}\Delta t, \qquad \mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}\Delta t.$$

When the stone crosses the water surface from above, its speed and impact angle

$$\alpha = \arctan 2(-v_y, \sqrt{v_x^2 + v_z^2})$$

determine whether a skip is allowed (sufficient speed, shallow angle, bounce-limit not exceeded). For a valid skip, velocity is decomposed relative to the water normal $\mathbf{n} = (0, 1, 0)$:

$$\mathbf{v}_N = (\mathbf{v} \cdot \mathbf{n})\mathbf{n}, \qquad \mathbf{v}_T = \mathbf{v} - \mathbf{v}_N,$$

then reflected and damped:

$$\mathbf{v}' = -e_n\,\mathbf{v}_N + c_t\,\mathbf{v}_T.$$

The stone is snapped slightly above the surface, a ripple is spawned, and a "full" splash sound (`drop1/2/3`) is played. If the conditions fail, the stone produces only a tiny splash, spawns a weak ripple, and deactivates.

Stones also collide with cubes (AABB test) and the boat (sphere test), creating small impulses, generating ripples, and triggering a "thud" sound.

Stone splash thresholds were iteratively tuned to distinguish between full splashes (that should trigger ripple sequences and loud drops) and small impacts that should only produce a tiny splash.

3.4 Fishing Rod and Lure

The fishing system uses a `Rod` struct tracking position, velocity, charge, and state flags. Holding `R` increases the charge linearly in $[0, 1]$. Releasing the key maps this charge to a throw angle and speed,

$$\alpha = \alpha_{\min} + (1 - \text{charge})(\alpha_{\max} - \alpha_{\min}), \qquad s = s_{\min} + \text{charge}(s_{\max} - s_{\min}),$$

yielding flatter, faster casts for longer holds. The lure is spawned near the camera with a small forward/right/up offset.

During flight, `updateRod()` applies scaled gravity and exponential drag,

$$\mathbf{v} \leftarrow (\mathbf{v} + \mathbf{g}'\Delta t)\, e^{-k_{\text{drag}}\Delta t}, \qquad \mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}\Delta t.$$

When the lure drops below the sampled water height, it's snapped to float just above the surface, its vertical velocity is zeroed, a ripple is spawned, and `flying` becomes false. The lure stays active briefly and can catch at most one fish during this window. A looping reel sound plays while charging and stops on release, and a `hasCaught` flag ensures only one attachment per cast.

3.5 Fish Catching

Fish are stored in a `Fish` struct containing position, velocity, yaw, yaw velocity (for banking), timers, and state flags. `initFish()` randomly distributes fish beneath the surface with initial headings, ensuring separation. In `updateFish()`, inactive fish count down a respawn timer and reappear with new positions and velocities.

Active fish behave in three modes. In the wander state, yaw is perturbed by small random turns, with occasional larger rotations. A steering helper `steerAway()` adjusts yaw away from the boat, cubes, and the underwater player, strongest at close distances. Yaw is wrapped to $[-180°, 180°]$ and the yaw rate determines a banking angle during rendering. Forward motion is based on the heading,

$$\mathbf{v}_{\text{desired}} = v_{\text{swim}}(\cos\theta, 0, -\sin\theta),$$

and velocity is blended toward this target using a first-order filter,

$$\mathbf{v} \leftarrow \mathbf{v} + (\mathbf{v}_{\text{desired}} - \mathbf{v})(1 - e^{-a\Delta t}),$$

while vertical position is kept near the surface using a small oscillation/offset for a natural bobbing effect.

If the rod is active and a fish enters a catch radius, the fish enters the fighting state. During fighting, the fish's height is clamped to stay just below the water surface (a fixed offset from the sampled water height), rather than continuing free vertical swimming. Its yaw is set from the lure's horizontal direction plus a sinusoidal wiggle (about 20°), and its own velocity is zeroed. A timer runs until a randomly chosen duration is exceeded, after which the fish is marked `caught`, the global `fishCaught` counter increments, and the fish deactivates before later respawning. A `hasCaught` flag on the rod prevents multiple fish from attaching during a single cast.

We experimented with the amplitude and frequency of the tail wiggle, as well as fight duration ranges to make the fighting look energetic without being erratic. The mouth anchoring was also tuned so the fish stayed attached to the lure.

3.6 Treasure Chest and Glow Marker

The treasure chest feature was inspired by a Roblox game I played as a kid called SharkBite. Treasure chests would spawn and users would have a short period of time to go underwater and find the chest before it disappeared.

A `Chest` struct stores whether it is active, its spawn time, and world position. A global timer `nextChestTime` is initialized to about five minutes in the future; in the main loop, if the chest is inactive and the current time exceeds this value, `spawnChestNear()` is called with either the boat or camera position as a center. This function picks random XZ offsets within a given radius and places the chest slightly above the seabed so the mesh remains fully visible. `chestExpired()` marks the chest inactive once its lifetime exceeds a fixed TTL (e.g., 15 seconds), and the respawn time is reset to the next five-minute interval.

Player collection is handled by `tryCollectChest()`, which computes the squared distance between the player (camera) position and chest position. If this distance is less than a collection radius squared, the chest is deactivated, the `prizes` counter increments, and `nextChestTime` is scheduled a fixed interval in the future (with support for randomized respawn intervals via a uniform sampler if desired). Visually, the chest is rendered using a dedicated mesh (or a fallback cube) and accompanied by a tall, additive-blended glow column drawn as a scaled cube standing above the chest, both in the main scene and in the reflection pass. A screen-space glow bar is drawn using ImGui's background draw list at the projected chest position, making it easy to locate the chest on the horizon even when it is far away or below the camera. The chest spawn and pickup both play distinctive "magic" sound effects.

4 Audio and Ambience

For audio, we created an `Audio` class that wraps SDL2_mixer. Initialization sets up the audio subsystem, opens the device at 44.1 kHz stereo with a modest buffer size, and allocates a fixed number of mixer channels. The class has methods to load short clips into memory (one-shot or looping), play them on specific or any free channel with optional volume overrides, query whether a channel is playing, and stop individual channels. Background music is handled separately with functions to load a streaming track, start playback with volume, stop it, adjust its volume, and seek to timestamps, which are used for cue-skipping.

In the main loop, audio responds closely to gameplay state. A relaxing Zelda BGM track plays in a loop, with its effective volume controlled by a normalized `bgmVolume` and a `bgmMuted` flag. When the camera goes underwater, the BGM volume is reduced (ducked) and a looping underwater ambience clip is played on a dedicated channel; resurfacing restores the music level and stops the underwater loop. The boat engine clip plays while the boat's speed exceeds a small threshold, with channel volume scaled as a function of normalized speed and a minimum floor to stay audible. If the boat stops, this channel is halted. Stone splashes set pending flags that are consumed once per frame: tiny splashes play a single `tiny_splash` clip, while full water splashes advance through a small sequence of `drop1/2/3` to give variation across bounces. Cube or boat impacts trigger a short `hit_thud`. The rod system starts a looping `reel` sound when charging and stops it upon release, while fish catches and chest pickups play success and magic stinger sounds respectively. Menu open and button presses generate click/menu effects.

5 UI, Controls, and Time-of-Day

Controls are divided into a free-fly mode and a boat mode. In free-fly mode, the camera moves with `W/A/S/D` in the horizontal plane and `Q/E` vertically. Yaw and pitch are updated from mouse deltas scaled by `g_mouseSensitivity`, with pitch clamped to avoid gimbal lock, arrow keys for rotation. Pressing `B` enters boat mode, attaching the camera to a "driver seat" on the bow. Steering input updates the boat's yaw, while a separate camera yaw state and view-offset allow looking around without immediately rotating the hull. The view gradually interpolates back toward the boat heading. Other interactions include `V` to switch cubes, left mouse to press/release a cube, `F` to throw stones, and `R` to cast the lure.

UI is handled through ImGui. ESC opens a "Controls" panel listing keybinds and containing sliders for look sensitivity and BGM volume, plus mute and cue-skip buttons using `setMusicPosition()`. "Resume" and "Exit" buttons close or terminate the program. While the menu is open, input is disabled and mouse capture is released. A time-of-day HUD shows a 24-hour clock based on a phase in $[0, 1)$ that also drives sun direction and sky color. Additionally, the window title is updated each frame with mode (Boat/Free), boat speed, fish caught, and prizes collected.

6 Optimizations

Uniform locations for each shader program are cached in small structs so that per-frame updates don't repeatedly query OpenGL. Ground tiles, chest, fish, stones, cubes, and the boat are grouped by VAO and shader to minimize state changes and draw calls. The tiled ground transforms are rebuilt once per frame around the camera and reused across passes. Framebuffers and textures are created once and reused across frames, with mipmap generation limited to key textures (scene color and reflection) where needed for filtering.

7 Future Improvements

7.1 Floating Cube Spring Movement

Future improvements would involve making this interaction feel even more physical and realistic. On the physics side, a more realistic buoyancy model could introduce some tilting and rolling and even water drag so that pushing the cube beyond a certain point would encounter resistance from the water itself. In terms of visual effects, perhaps the spring pop-up movement of the cube could cause special effects like bubbles or foam.

7.2 Fishing

An earlier version of the fishing mechanic attempted to use a full 3D fishing rod model rather than the current red-cube lure. The goal was to render a visually convincing rod-and-line system: the line would originate from the rod tip, pass through guide rings along the rod, and bend dynamically under tension. Achieving this required a custom shader to render the line as a camera-facing curve and additional logic to deform the rod mesh according to the lure's position. However, keeping the rod model correctly aligned with the camera, ensuring the line connected precisely to the moving lure, and synchronizing the rod's bending were difficult and time-consuming. Because of these alignment and implementation challenges, the system was simplified to use a compact red cube as the lure.

7.3 Better Landscape

With more time, we would make the environment have better landscape features. We would add islands, rocks, colored patterns for the 3d objects, as well as underwater elements like seaweed or coral.

7.4 Player Avatar + Skins

We could also introduce a player avatar instead of just using a camera-only viewpoint. This, along with customizable skins and boats, would give the player a stronger in-world presence and allow for personalization. This would allow for more animations or emotes like swimming, dancing, which are popular features in Minecraft and Roblox that enhance the player experience.

References

Water Simulation and Physics

- Shallow Water Equations: `https://en.wikipedia.org/wiki/Shallow_water_equations`

- J. Tessendorf, *Effective Water Simulation from Physical Models*, GPU Gems, NVIDIA: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-1-effective-water-simulation-physical-models
- Dotcrossdot, *Water Ocean Shader*: https://medium.com/dotcrossdot/water-ocean-shader-9173e0977f98
- I. A. Strumberger et al., *Real-Time Water Simulation*, NAUN: https://www.naun.org/main/NAUN/computers/ijcomputers-41.pdf
- Derivation of skipping stones over water: https://physics.stackexchange.com/questions/176108/deriving-equation-for-skipping-stones-over-water
- Physics of Floating Cube Spring Movement: https://www.myphysicslab.com/springs/single-spring-en.html and https://gafferongames.com/post/spring_physics/

3D Models

- Fish model: https://free3d.com/3d-model/fish-v1--996288.html
- Speedboat model: https://free3d.com/3d-model/speedboat-v01--840133.html
- Treasure chest model: https://free3d.com/3d-model/treasure-chest-91359.html

Code Libraries and Tools

- jrafix post-processing effects: https://gitlab.com/kernitus/jrafix
- Dear ImGui: https://github.com/ocornut/imgui

Audio Assets

- water_drop1 / water_drop2: https://www.youtube.com/watch?v=6L0HBSu_ZaY
- water_drop3: https://pixabay.com/sound-effects/water-drop-stereo-257180/
- relaxing zelda music + ocean waves: https://www.youtube.com/watch?v=LfdCMBCt2r4
- Speed Boat Ride Ambience: https://www.youtube.com/watch?v=1frh0wqXb1M
- Magic WHOOSH: https://www.youtube.com/watch?v=pyzUhkKf2Qo
- Magical Twinkle Sound Effect (HD): https://www.youtube.com/watch?v=UZcSVrHi3-I
- fish_reel: https://www.youtube.com/watch?v=HStwgFswd18
- hit_thud: https://pixabay.com/sound-effects/cinematic-thud-fx-379991/
- mouse_click: https://pixabay.com/sound-effects/mouse-click-290204/
- game_menu: https://www.youtube.com/watch?v=sW8TKZtoND8
- tiny_splash: https://pixabay.com/sound-effects/tiny-splash-83778/
- underwater_sounds: https://www.youtube.com/watch?v=Pf4ZVHyA7Bg
- success: https://www.youtube.com/watch?v=n4qnDaSidJs