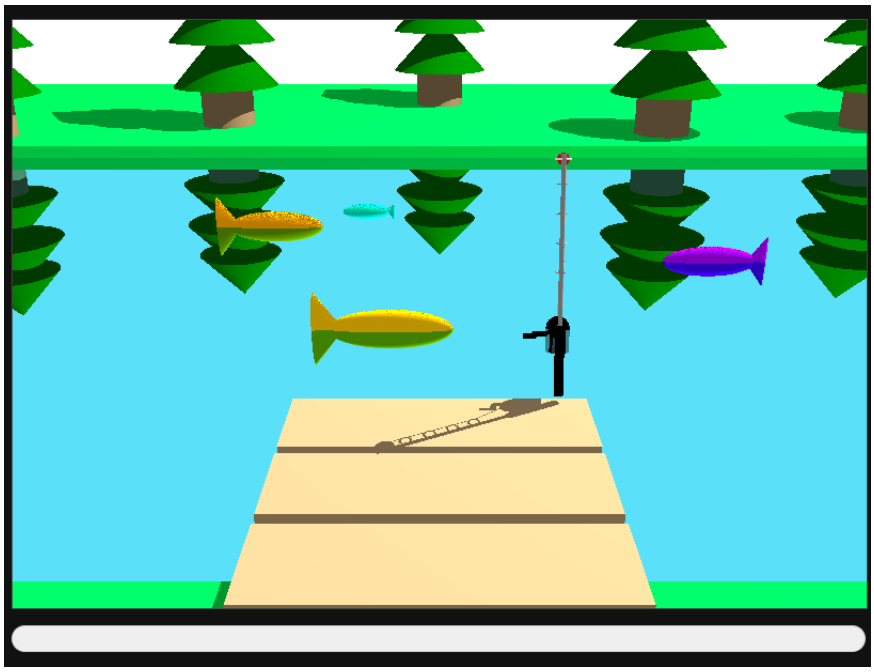
CS 412/512 Final Project Report - Fishing Minigame
Adrian Ruiz Vasquez
Christopher Jarek

Project Link: https://cjarek123.github.io/ray_tracing_version/



Game Instructions

The game starts as soon as the project webpage is loaded/reloaded, starting a timer countdown for one minute. In that one minute, the goal is to catch as many fishes and/or weight in fishes as you can.

To start fishing, you press and hold the spacebar. This will build up the charge meter beneath the canvas and allow you to cast your line farther into the water.

Once you've cast your line, you can begin reeling it back in by pressing and holding the spacebar again. By reeling in the bobber incrementally, you can aim for specific fish, and once you've caught one, reel it in all the way to secure it!

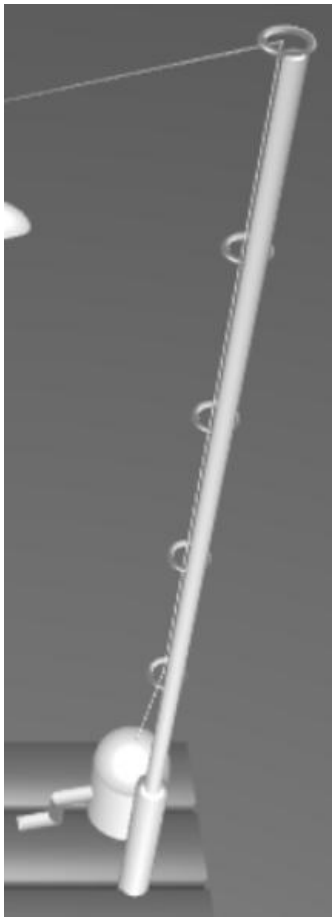
After catching a fish, its size will be recorded on the scoreboard, where you can see your total number of catches, and the size of your largest, and smallest catch! Keep playing to try to catch the largest fish you can!

The player can also look left and right to view the entire river scenery using the left and right arrow keys. The game also runs the smoothest when the river material is made diffuse!

Technical Methods Overview

The technical methods implemented in our project included *Hierarchical Modeling*, *Ray Tracing*, and *State Machines*. We worked on this project in two tangent parts, one focused on hierarchical modeling, animations, and game state, and the other focused on the ray tracing shader, and entity instantiation. Then both parts were integrated together for the final version.

Hierarchical Modeling



In order to render the scene, we utilized the hierarchical modeling from Christopher's hw4 assignment, building the fishing rod and fish objects from my previously implemented Node class. This Node allows us to create a tree structure wherein each node can have a parent node, and any number of children nodes. To propagate the matrix transformations down the tree, we use the updateWorldMatrix method, which recursively applies the local matrix of a parent to each of its children, who then apply their matrices to their respective children and so on. In this way we can connect primitives together to render a complex object.

In addition to the regular primitives, I also implemented a generateTorus function in order to create the rings on the fishing pole. Everything else was generated by applying transformations to spheres, cylinders, cones, and cubes.

One challenge I encountered while implementing the objects was when applying a scale transformation to a parent object. My intention for these was to adjust an individual component of the object to make it more accurate and visually appealing, however due to the way my Node class was designed, this scaling would be applied to each child node as well. Because of this I had to add an inverse scaling to each child node to cancel out the unwanted transformation.

State Machines

When creating the state machines for the Fishing Rod and Fish objects, I used the flow chart on the left to guide the process. For the fishing rod, I knew that we would need an idle position, a way to charge up the rod so that the player could cast the bobber varying distances out, and a way to reel in the line.

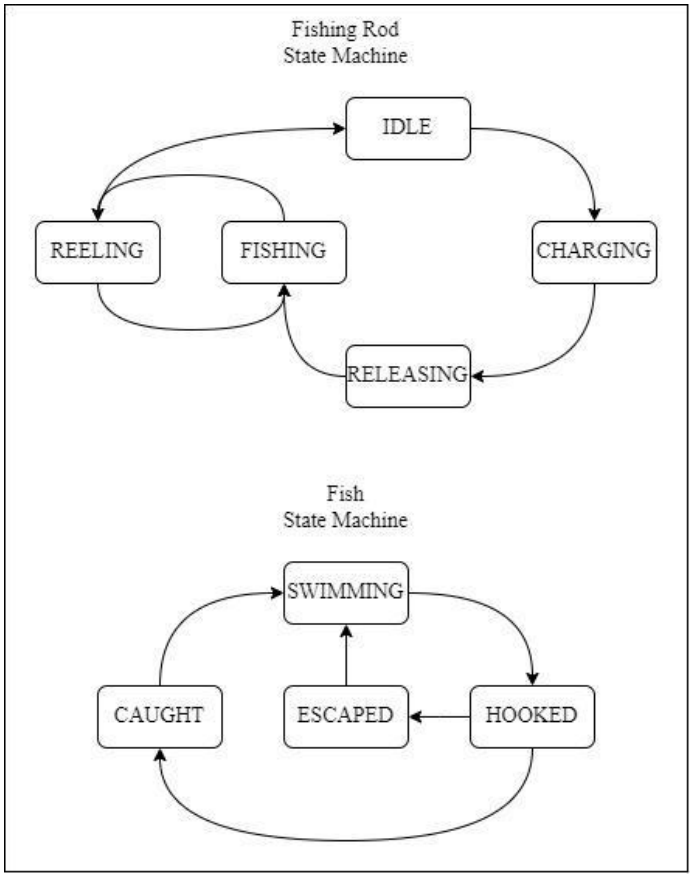
The IDLE state is the simplest, it is used to reset the bobber position back to (0, 0, 0), and the charge value back to 0.0. From idle, if the player presses and holds the spacebar, the fishing rod will enter its CHARGING state.

While charging, the fishing rod is pulled back, the charge meter ui element fills up, and the rod's charge increases. Also in this state, we update the rod's startTime value each frame to be used in the next state.

Once the player releases the spacebar the rod will enter its RELEASING state. Here I calculate the bobber's trajectory using the rod's charge and startTime values, both updated in the previous state. The player cannot interact with the game during this state, but when the bobber lands in the water the fishing rod will automatically enter its FISHING state.

While in its FISHING state, the bobber will slowly move up and down in the water, and logic within the fishes' SWIMMING state will check if it is near to the bobber, and if so, will hook the fish.

While in the FISHING state, the player may tap or hold down the spacebar to toggle into its REELING state. While reeling, the bobber is moved towards the player, in this way the player can navigate the scene in order to attempt to catch specific fish, or to reel in a hooked fish. If the bobber is reeled in enough, the rod will enter its IDLE state, reset the bobber, and catalogue the caught fish (if any) in the scoreboard. And the player may then charge up another cast.



When creating the fishes' state machine, I wanted the fish to swim across the scene and only get caught if the player cast their bobber into the fishes' path. To accomplish this, the fish are initialized in their SWIMMING state. In this state they travel across the scene from left to right, resetting when they exit the scene. Also in this state, the fish will check if the fishing rod is in its FISHING state, if so they will then check to see if the bobber is close to themselves and if the fishing rod doesn't already have another fish hooked. If so, they will enter their HOOKED state.

While hooked, a fish is added to the fishing rod's hierarchy, connected to the bobber. This allows us to pull the fish in with the rod's REELING state.

Once a hooked fish is reeled in all the way it will enter its CAUGHT state. It remains in this state for only one frame, while the game updates the scoreboard and resets the fish with a new randomized size before releasing it back into its SWIMMING state.

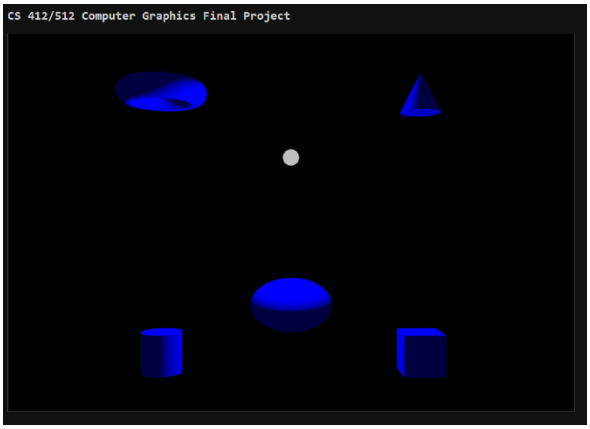
I had initially intended to add a secondary minigame after the player hooked a fish. This minigame would have the fish fight against the line, and the player would have to wait until the fish became exhausted to reel it in without snapping the line. Unfortunately, I didn't have enough time to implement this as well, so the ESCAPED state remains unused.

Ray Tracing

The ray tracing pipeline used in the final version of our game was expanded upon the version Adrian submitted for homework 6, which was based on the skeleton code provided by Dr. Wang.

The first issue with the initial pipeline was that it did not include intersection functions for cylinders, cones, or torus, and the sphere and cube functions were not generalized for ellipsoids and rectangular prisms. The second issue was that the camera, light source, and primitives were instantiated inside of the shader.

Therefore, the work done on the ray tracer, included adding the functionality to buffer camera origin, and forward, up, and right vectors to *dynamically change the camera view* in the game. Also, the functionality to buffer the light source origin, enabled our *day/night cycle* feature into the game to show moving shadows. You can also change the material of the river water; when refractive, you can see the fish' bodies through the water, and when reflective, you can see the reflection of the trees and fish' upper halves on the water surface.



More advanced intersection functions for the cone, cylinder, ellipsoid, rectangular prisms, and torus had to be added. The main challenge with these were that the transformations such as rotation made the primitives no longer axis aligned or oriented, which led to issues related to intersection distance, because of the math involved with calculating the intersection and normal in the primitive's local space, instead of world space. This challenge was overcome with help from AI for debugging, by describing to AI the visual effect that was occurring due to the errors, and it recommended the issue was with the way we were transforming in and out of world and local space, specifically for the *t* variable, which was being normalized and not denormalized at some point.

```
// Struct for general primitive
struct Primitive {
    mat4 worldMatrix;
    vec3 color;
    float reflectivity;
    float refractiveIndex;
    int material;
    int shapeType;
    int pad0;//padding
    float field1;
    float field2;
    float field3;
    int pad1;//padding
};
// Uniform Block Index
layout(std140) uniform PrimitiveBlock {
    Primitive primitives[70];
    int primitiveCount;
};
```

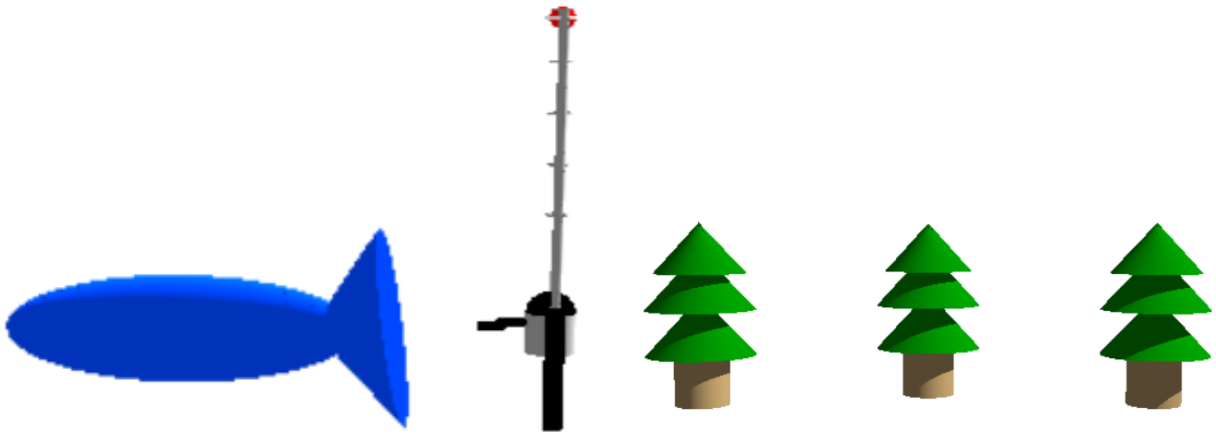
Then work on dynamically buffering primitives into the scene was done, utilizing a *Uniform Buffer Object*, represented by a list of structs inside of the fragment shader. A single struct was used for all primitives, but each primitive had an int representing its shape type, as well as world matrix, color, reflectivity, refractive index, material, and three fields for radius, height, dimensions, which differed per shape type.

Entity & Primitive Instantiation

We revamped the vertex-based primitive constructors into new constructors which were based on the geometric properties of the shapes, including height, radius, and dimensions. These

primitives were easier to integrate alongside the ray tracing pipeline, which uses the geometric properties to check ray-shape intersections.

Then an *Entity* base class was made which included a constructor function to instantiate the entity with Node(s) from the hierarchical model pipeline, an animate function used for updating every frame based on states if needed. Each node was assigned a primitive from the revamped library of primitives. The game rendering loop called each entity's animate function each loop.



Six classes extending the Entity base class were made: *Fish*, *FishingRod*, *PineTree*, *Dock*, *GrassBlock*, and *River*. The FishingRod and Fish class were essentially conversions of the FishingRod and Fish classes used in the Hierarchical Model pipeline, with added colors. The PineTree, Dock, and GrassBlock built most of the scenery, and the River class material can be customized between diffuse, reflective, and diffuse to show off the different materials in ray tracing. The reflectivity and refractive index of the river can also be customized using sliders. The difference in reflectivity and refractiveness is best seen when directly changing the river material. When the material is reflective, the river color becomes white because it reflects the sky color, while when refractive the river is blue. The fishing rod core cylinder is also reflective.

Team Contributions

Christopher Jarek - I began this project by using my hw4 assignment as a base. From my hw4 base, I removed the old objects and built the fishing rod and fish objects from transformed primitives and created the generateTorus() method in addition to my existing ones. I then created the fishingRod and fish classes and built the framework for their state machines, which I implemented step by step in order to create the game mechanics.

While implementing the CHARGING state of the fishing rod, I added a charge meter to the UI underneath the canvas that would give the player a visual indication of how far they would cast the bobber.

I also instantiated the fish so we could have more than one at a time in the scene, and added the scoreboard UI to the right of the canvas to track the player's score and largest/smallest catches.

Finally, I wrote the Game Instructions, Hierarchical Modeling, and State Machines sections of this report.

Adrian Ruiz Vasquez – I worked on the ray tracing pipeline, adding more advanced intersection functions for cones, cylinders, ellipsoids, rectangular prisms, and torus. This was the most complicated task I did, because of the heaviness of math.

I made the camera view movable using the left and right keys.

I added the day and night cycle and making the light sphere orbit in semi circles, adjusting light intensity and ambient based on day or night.

I revamped the primitives to use geometric properties (radius, height, dimensions) instead of generating vertices, indices, and normals.

I designed the Entity base class, and the classes which extended it: Fish, FishingRod, Dock, GrassBlock, PineTree, and River. I redesigned the buffering from the cpu to the gpu shaders around the Entity class. This made it easier to instantiate, render, and animate any number of different entities for all entity types.

I also added the timer counting down, which ends/halts the game when over. Made the river material changeable, and the river refractive index and reflectivity customizable, and made the fishing rod core cylinder reflective.

AI Disclosure

We used AI as an alternative for google in searching for the proper html objects and syntax to implement the charge meter. We used AI to research methods to pipeline entire structs (for the primitives) into the shaders, which we ended up discovering Uniform Buffer Objects and then used. We used AI in debugging the math involved in the intersection algorithms for Cone and Cylinder, because we were getting issues related to the t field returned.