Ocean Simulator: Iteration 2

https://github.com/i12345/ocean-simulator/commit/c66a1a8c6b1573688c9a33a378faa0cb58b82449

Project Members: Shaina Ayer, Christian Blundell, Isaac Valdez

Vision:

The ocean simulator will be a video game that lets players virtually "swim" in an ocean with marine life using motion-control.

Target Users:

The target users will be an adolescent-focused group that enjoys things such as new technologies, games, science, animals, exercise, and exploration. A sample of this target audience is Jacob, a family member of group member Isaac Valdez, who falls into this target user category and is used for feedback. Jacob wants a simulator game that runs smoothly, has realistic graphics, and extensive gameplay.

Competitors:

Similar video games and interactive projects featuring marine life already exist, such as Ultimate Ocean Simulator, Ocean Mammals: Blue Whale Mari, Fish Abyss: Aquarium Simulator, and David Attenborough's Great Barrier Reef. They have many kinds of animals, and Ultimate Ocean Simulator lets the player "play" as one of the marine animals. David Attenborough's Great Barrier Reef has many educational videos and articles as well. Besides these, there are many non-interactive underwater experiences available on the Internet, such as regular and 3D videos. However, these games lack motion-controlled input.

Move! lets a person play a video game with motion-control, but it doesn't feature the ocean.

Requirements:

The major requirement for this project is to develop a motion control for the user to navigate through the simulator, with the user's webcam detecting movement and passing it out as movement in the simulation. The other major requirement is to design 3D models of marine life that can be discovered by the user while moving in the simulation. The marine life should have simple AI for generation and recognition of the type of marine life generated. Other requirements include generating AI patterns for certain types of marine life (schools of fish, shark behaviors, rarity of marine life) and dynamic generation of marine life. If possible, other requirements could include developing more in-depth patterns for certain types of marine life based on their natural behavior as well as fluidity of the water. Requirements will not include active interaction with marine life, or any simulation based outside of an ocean/sea environment.

Core Features:

- Navigation between main menu, "set up camera", and playing in the simulation
- 3D model / rendering for animals and diver
- Motion-control for the diver
- AI controls the animals' behavior

Project Plan:

The plan is to first develop motion control for the diver. This will be done with MediaPipe BlazePose for pose detection and PlayCanvas for 3D graphics and physics. The motion control foundation has been developed and tested during this iteration. We are developing this to work on a webcam, and we expect this simulator to work on any device that has access to a webcam. There may not be the option to develop a simulator that is connected via screencast, which is shown by the limited users in the target audience. This should take weeks to complete, finishing at iteration 3. The 3D rendering of the fish has also already begun, with 3D models of fish collected to be implemented into the simulator. We plan on getting more 3D models of various types of marine life to be generated (potentially dynamically) into the simulator. We anticipate that this will take a few weeks until iteration 3. The other requirements will be based on the completion of the previously mentioned steps and will take less time to implement. The simulator starting screen and user control setup will be developed during the third iteration. We anticipate 30 more hours of implementation of core features for each iteration, including testing.

Core Features I/O Table:

Input	**Output**
User: Enters Home Screen	**Program: Begins calibration for motion control**
Webcam: User moves using motion control	**Program: Defines movement to mirror movement of simulator**
User: Selects marine life found in simulator	**Simulator: Identifies fish based on value defined based on its 3D model**
Program: Generates marine life	**User: Sees marine life in simulated environment**

Domain Model:

The domain model shows the scope of each element of the Ocean Simulator. These elements include the motion control figure that guides through the simulator, the player pose that defines the motion control of the game, and the creatures that are generated using simple AI models into the simulator.

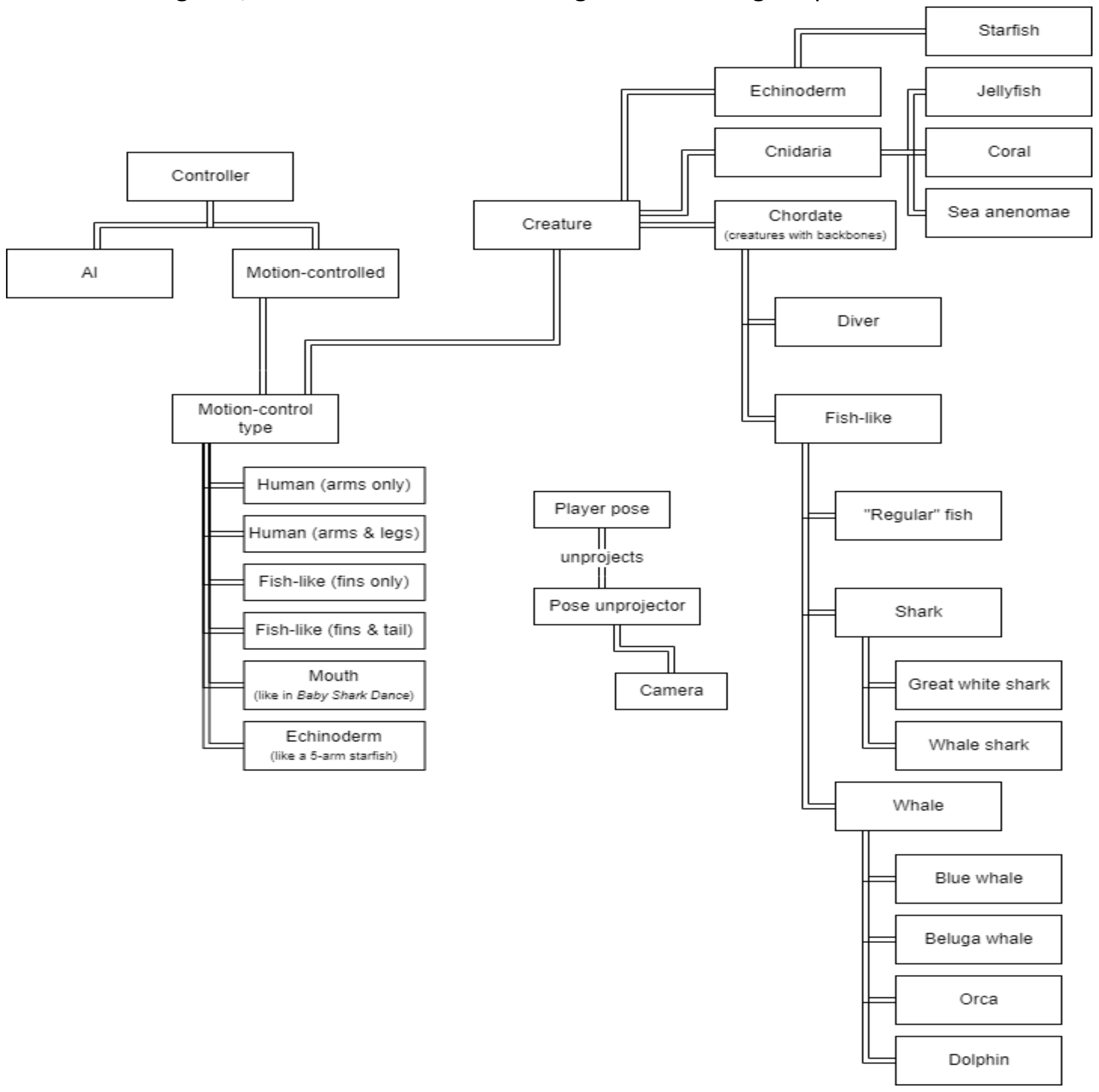

Figure 1: Domain Model for Ocean Simulator

Use Cases:

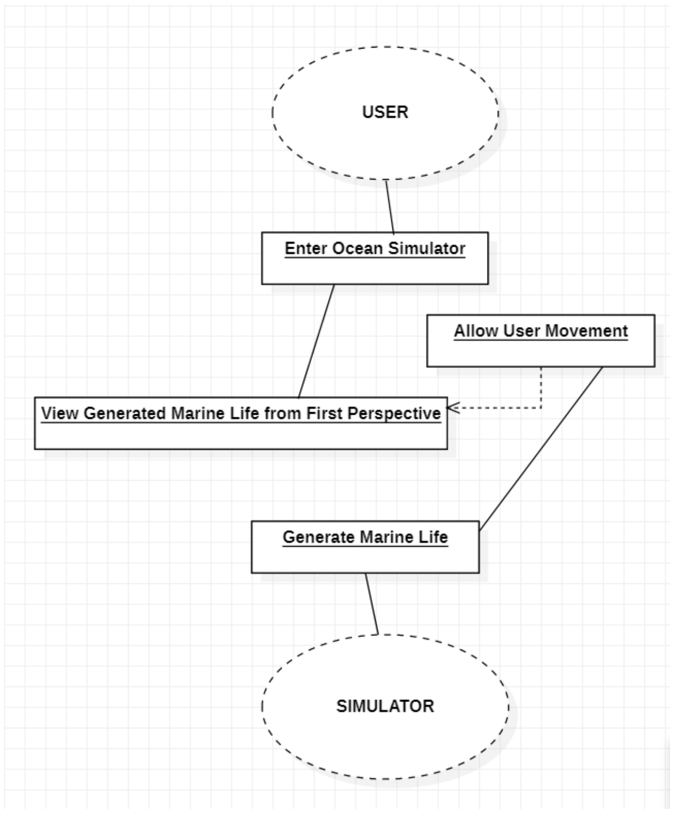

Figure 1: Use Case for General Ocean Simulator Project

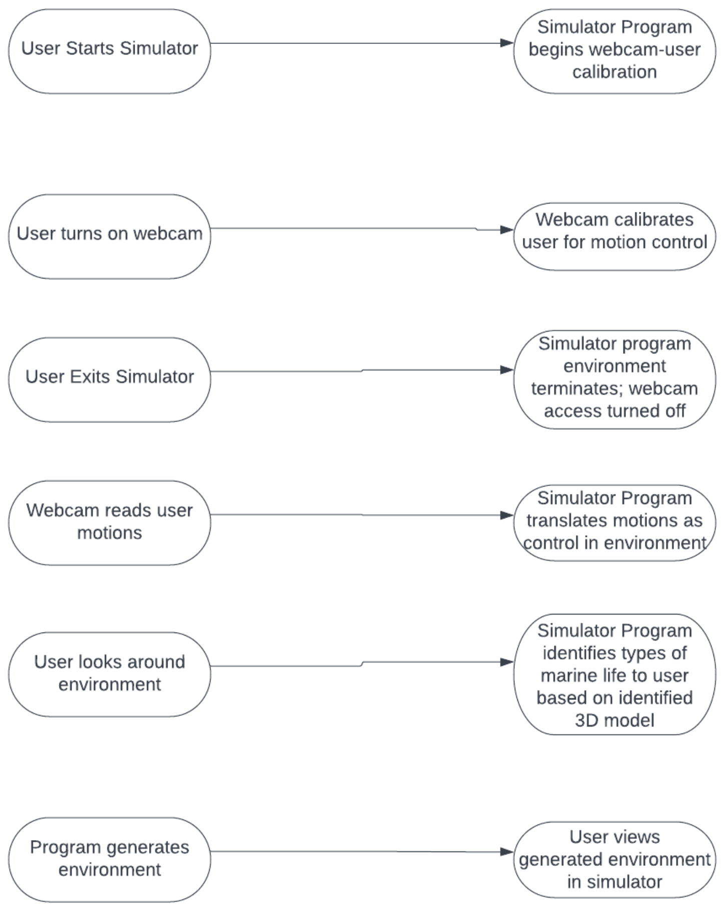
User Starts Simulator
Simulator Program begins webcam-user calibration
User turns on webcam
Webcam calibrates user for motion control
User Exits Simulator
Simulator program environment terminates; webcam access turned off
Webcam reads user motions
Simulator Program translates motions as control in environment
User looks around environment
Simulator Program identifies types of marine life to user based on identified 3D model
Program generates environment
User views generated environment in simulator

Figure 2: Use Cases for each aspect of simulator setup/gameplay

Risks and Risk Mitigation:

- **TV/Large monitor implementation: 100% x remaining project hours = Effect on total project (30 hr)**
  - TV/Monitor implementation difficult to achieve, only able to test and develop on single monitor setups.
  - Market Study: 20 people that fit target audience demographic took market study; results confirmed that they do not have a setup that justifies the implementation of large monitor device for motion control
  - Mitigation: We will implement and test this project using a single webcam setup on a computer as it is the most available for our target audience. We will potentially develop a simple prototype for multi-device setups that work on all types of large monitors/TV that support screen casting. (We will not make large monitors a must-have requirement for our project's final delivery)
- **Unrealistic Schedule: 60% x remaining project hours (30 hr) = 18 hours**
  - The time frame for each iteration may not be enough to complete each feature to the highest quality anticipated. This will result in a potentially incomplete product for the final delivery.

    - Mitigation: Schedule each feature for each iteration properly and set realistic goals for each feature. In the project plan, we have anticipated that we will complete the most necessary features by iteration 3 so that the product will have the minimum functionalities we anticipate demonstrating.
- **Failure in Pose Detection Functionality: 40% x remaining project hours (30 hr) = 12 hours**
    - If pose detection technology is too technical and abstract to implement, core features of the product will not be available, leaving an unpolished product as the resulting effect.
    - Mitigation: Analyze pose detection and implementation in earlier iterations to ensure that pose detection remains a key part of the product. (Currently working on pose detection for this iteration)

Current Progress:

We have a foundation for the motion control diver, as we have set up the code and test cases for the math to decompose a set of key points (e.g., shoulder, elbow, etc.) into the corresponding rotations of limbs. The code uses MediaPipe BlazePose to detect poses. The test can be run by opening https://localhost:8080/ after running the commands "`npm i && npm run dev`" in the project directory and allowing access to the device's webcam.

3D models have been found and will be implemented in the next iteration. Using 3D objects extracted from zip files we can use these models and identify them before implementing them into the simulator. Each 3D model will have a form of identification that can be used to identify the type of marine species for the user. Various coral and marine plant 3D models will be accessed from the collection of coral 3D models developed by the Smithsonian Institution (Public domain access).

Different 3D environments have been found for the implementation of the simulator, which are found on open source sites such as CadNav.

Current Use Case for Webcam Calibration:

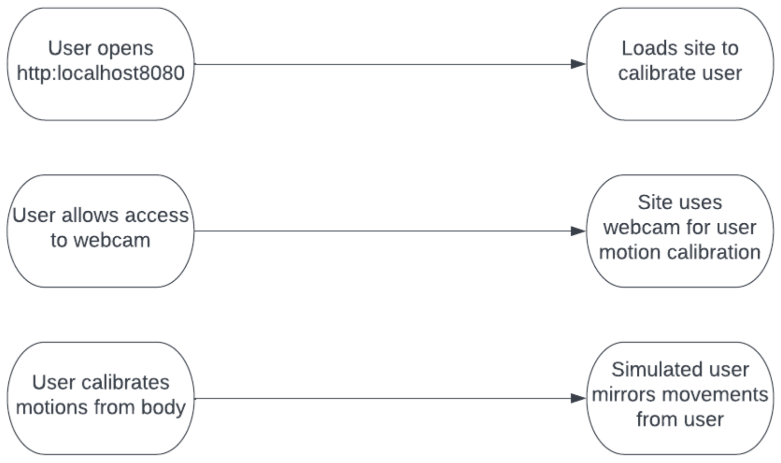
User opens http:localhost8080
Loads site to calibrate user
User allows access to webcam
Site uses webcam for user motion calibration
User calibrates motions from body
Simulated user mirrors movements from user

Figure 1: Use Cases for Webcam Calibration

Code for Pose Angle:

```
/**
* Computes the angles needed to describe an arm's rotation.
* @param angles where to store the resulting angles in
* @param invert_local_y whether or not the local y axis should be inverted;
* `true` if on the left side, `false` on the right.
* @param keypoints the keypoints to work with
*/
const arm = (
        angles: Partial<LimbAngles>,
        invert_local_y: boolean,
        keypoints: {
            shoulder?: Vec3
            shoulder_other?: Vec3
            hip?: Vec3
            elbow?: Vec3
            wrist?: Vec3
            thumb?: Vec3
            index?: Vec3
            pinky?: Vec3
        }
    ) => {
    /**
     * Pose angles for an arm are computed by progressing through arm segments,
     * starting with a basis, then decomposing the rotation for that segment,
     * then computing the next basis and repeating.
     *
     * For the starting (upper arm) basis:
     * x+ = direction upper limb segment is pointing,
     *      (focal shoulder)<-(other shoulder)
     * y+ = forward, straight ahead of player (if they were looking forward),
     *      found by crossing z and x.
     * z+ = up, hip->shoulder
     *
     * The elbow only rotates along the y-axis (pitch), so that means the upper
     * arm has to roll. To compute this roll, the z_elbow vector is computed by
     * double-crossing the elbow->wrist vector with the shoulder->elbow vector.
     * The z_elbow vector is then used along with the shoulder->elbow vector to
     * compute the upper arm rotation.
     *
     * Then, within the lower arm basis, the elbow->wrist vector will only need
     * a pitch component to describe it. The wrist basis can then be computed.
     *
     * The wrist can pitch, yaw, and roll. The hand is assumed to have the palm
     * facing up (Z+) by default, and this up vector is computed by crossing
     * the wrist->thumb and wrist->pinky vectors. The forward vector for the
     * wrist, the vector to compute the rotation for, is the wrist->index
     * vector.
     *
     * That's all that's computed for now. Later, the finger rotations may be
     * computed here also.
     */

    if (!keypoints.shoulder ||
        !keypoints.shoulder_other ||
        !keypoints.hip ||
        !keypoints.elbow)
        return

    const x = new Vec3().sub2(keypoints.shoulder,
keypoints.shoulder_other).normalize()
    const z = new Vec3().sub2(keypoints.shoulder, keypoints.hip).normalize()
    const y = new Vec3().cross(z, x)

    if (invert_local_y)
        y.mulScalar(-1)

    const basis_upper = { x, y, z }
    console.log(`basis_upper:\nx: ${prettyPrint(basis_upper.x)}\ny:
${prettyPrint(basis_upper.y)}\nz: ${prettyPrint(basis_upper.z)}`)

    const x_shoulder_elbow = projectToBasis(new Vec3().sub2(keypoints.elbow,
keypoints.shoulder), basis_upper).normalize()
    let x_elbow_wrist = keypoints.wrist ? projectToBasis(new
Vec3().sub2(keypoints.wrist, keypoints.elbow), basis_upper).normalize() : undefined

    console.log(`x_shoulder_elbow: ${prettyPrint(x_shoulder_elbow)}`)
    console.log(`x_elbow_wrist: ${prettyPrint(x_elbow_wrist)}`)

    let z_elbow = x_elbow_wrist ? new Vec3().sub2(x_elbow_wrist, x_shoulder_elbow) :
undefined
    if(z_elbow && z_elbow.length() < 0.1) z_elbow = undefined
    z_elbow?.normalize()
    if(z_elbow) z_elbow = new Vec3().cross(new Vec3().cross(x_shoulder_elbow,
z_elbow), x_shoulder_elbow).normalize()
    console.log(`z_elbow: ${prettyPrint(z_elbow)}`)
    console.log(`z_elbow dot x_shoulder_elbow: ${z_elbow?.dot(x_shoulder_elbow)}
(should = 0 | undefined)`)
    angles.upper = decomposeEuler(x_shoulder_elbow, z_elbow)
    const angles_upper_inverse = angles.upper.clone().mulScalar(-1)

    if (!keypoints.elbow ||
        !keypoints.wrist)
        return

    console.log(`angles.upper: ${prettyPrint(angles.upper, 0)}`)
    console.log(`angles_upper_inverse: ${prettyPrint(angles_upper_inverse, 0)}`)
    const basis_lower = rotateBasis(basis_upper, angles_upper_inverse)
    console.log(`basis_lower:\nx: ${prettyPrint(basis_lower.x)}\ny:
${prettyPrint(basis_lower.y)}\nz: ${prettyPrint(basis_lower.z)}`)
```

```
    x_elbow_wrist = rotate(x_elbow_wrist, angles_upper_inverse)

    console.log(`x_elbow_wrist: ${prettyPrint(x_elbow_wrist)}`)
    angles.lower = decomposeEuler(x_elbow_wrist)
    console.log(`angles.lower: ${prettyPrint(angles.lower, 0)}`)
    const angles_lower_inverse = angles.lower.clone().mulScalar(-1)

    if (!keypoints.thumb ||
        !keypoints.pinky)
        return

    const basis_wrist = rotateBasis(basis_lower, angles_lower_inverse)
    console.log(`basis_end:\nx: ${prettyPrint(basis_wrist.x)}\ny:
${prettyPrint(basis_wrist.y)}\nz: ${prettyPrint(basis_wrist.z)}`)

    const x_wrist_thumb = projectToBasis(new Vec3().sub2(keypoints.thumb,
keypoints.wrist), basis_wrist)
    const x_wrist_index = projectToBasis(new Vec3().sub2(keypoints.index,
keypoints.wrist), basis_wrist)
    const x_wrist_pinky = projectToBasis(new Vec3().sub2(keypoints.pinky,
keypoints.wrist), basis_wrist)
    const z_hand = new Vec3().cross(x_wrist_thumb, x_wrist_pinky).normalize()
    console.log(`x_wrist_thumb: ${prettyPrint(x_wrist_thumb)}`)
    console.log(`x_wrist_pinky: ${prettyPrint(x_wrist_pinky)}`)
    console.log(`z_hand: ${prettyPrint(z_hand)}`)

    angles.end = decomposeEuler(x_wrist_index, z_hand)
    console.log(`angles.end: ${prettyPrint(angles.end, 0)}`)
}

/**
* Calculates the angles needed to rotate a default rig to match the given
* keypoint 3D positions.
* @param pose the pose keypoint 3D positions to calculate angles for
* @returns the angles needed to rotate a default rig to match the given
* keypoint 3D positions
*/
export function calcPoseAngles(pose: Partial<Keypoints3D>): DeeplyPartial<PoseAngles>
{
    /**
     * Right now, just the arms are calculated for, using the arm() function.
     */

    let angles: DeeplyPartial<PoseAngles> = {
        arms: {
            left: {},
            right: {}
        }
    }

    const blazePose = pose as Partial<KeypointMap<Vec3, BlazePoseKeypointIDs>>
    arm(angles.arms.left as Partial<LimbAngles>, true, {
        shoulder: pose.left_shoulder,
        shoulder_other: pose.right_shoulder,
        hip: pose.left_hip,
        elbow: pose.left_elbow,
        wrist: pose.left_wrist,
        thumb: blazePose.left_thumb,
        index: blazePose.left_index,
        pinky: blazePose.left_pinky,
    })

    arm(angles.arms.right as Partial<LimbAngles>, false, {
        shoulder: pose.right_shoulder,
        shoulder_other: pose.left_shoulder,
        hip: pose.right_hip,
        elbow: pose.right_elbow,
        wrist: pose.right_wrist,
        thumb: blazePose.right_thumb,
        index: blazePose.right_index,
        pinky: blazePose.right_pinky,
    })

    return angles
}
```

Code for Euler Angle Decomposition:

```
// Currently the xyz coordinates are used to mean (roll, pitch, yaw)
export type EulerAngle = Vec3

/**
* Rotates a vector by given euler angles.
* @param v the vector to rotate
* @param eulerAngle the euler angles to rotate by
* @returns that vector, rotated by the given euler angles
*/
export function rotate(v: Vec3, eulerAngle: EulerAngle): Vec3 {
    return new Mat4().setFromEulerAngles(eulerAngle.x, eulerAngle.y,
eulerAngle.z).transformVector(v)
}

/**
* Rotates a basis by given euler angles around its own axes.
* @param basis the basis to rotate. The components for this basis also serve
* as the axes of rotation.
* @param rotation the rotation to rotate to this basis by. The rotations for
* these angles are done around the basis component vectors themselves.
* @returns a new basis computed from the original basis but rotated around its
* own axes.
```

```
*/
export function rotateBasis({ x, y, z }: Basis, rotation: EulerAngle): Basis {
    const mx = new Mat4().setFromAxisAngle(x, rotation.x)
    const my = new Mat4().setFromAxisAngle(y, rotation.y)
    const mz = new Mat4().setFromAxisAngle(z, rotation.z)
    const m = new Mat4().setIdentity()
    m.mulAffine2(mx, m)
    m.mulAffine2(my, m)
    m.mulAffine2(mz, m)

    return {
        x: m.transformVector(x),
        y: m.transformVector(y),
        z: m.transformVector(z),
    }
}

/**
* Computes the euler XYZ angles needed to rotate [1, 0, 0] to get to {@link v}
* optionally pointing with a local Z+ vector of {@link up}.
* @param v a vector to find the euler XYZ angles to get to from [1, 0, 0]
* @param up the final local Z+ vector this rotation should orient to
* (defaults to [0, 0, 1]). This vector should be perpendicular to {@link v}.
*/
export function decomposeEuler(v: Vec3, up?: Vec3): EulerAngle {
    /**
     * First we find the pitch and yaw needed to get [1, 0, 0] to point to `v`,
     * then we compute the roll component that would align with `up`.
     *
     * To compute the pitch, we imagine a triangle with its adjacent leg on the
     * xy plane and its opposite leg extending upward, only in the z-dimension.
     * The pitch is the angle opposite the opposite leg of this triangle.
     *
     * The yaw is just the angle of rotation on the xy plane:
     * atan(y/x) adjusted for a complete 360 degrees.
     *
     * The roll can only be computed if an up vector was given. Using the pitch
     * and yaw, an ideal up vector is computed as if there was a roll of zero,
     * and then the roll component can be computed using
     * acos(dot(real up, ideal up)). To discern if the roll is positive or
     * negative, this angle is also computed between the up and ideal right
     * vectors (found by crossing the given vector and its ideal up).
     * Whether or not that angle (between real up and ideal right) is more than
     * 90 degrees determines whether or not the roll is positive or negative.
     */

    v = new Vec3().copy(v).normalize()

    const xy_plane = xy(v)

    const pitch = Math.atan2(-v.z, xy_plane.length()) * 180 / Math.PI
    const yaw = Math.atan2(xy_plane.y, xy_plane.x) * 180 / Math.PI

    let roll: number

    if (up) {
        const ideal_up = rotate(Vec3.BACK, new Vec3(0, pitch, yaw))
        const ideal_right = new Vec3().cross(v, ideal_up)
        const roll_up = Math.acos(ideal_up.dot(up)) * 180 / Math.PI
        const roll_right = Math.acos(ideal_right.dot(up))

        roll = roll_up * ((roll_right > (Math.PI / 2)) ? -1 : 1)
    }
    else {
        roll = 0
    }

    return new Vec3(roll, pitch, yaw)
}
```

Test Cases:

There are currently 22 test cases for motion diver calibration. 4 of the new test cases are based on tests regarding the decompose identity rotation, and 19 test cases are based on the Euler angle rotation. For the "rotate by Euler angle" test: the test case presents a starting 3D vector and a Euler rotation (rotation around X, then Y, then Z axis), and an expected final 3D vector. For the "Decompose Identity Rotation" test, a given Euler angle is used to rotate the 3D vector [0, 0, 1], and then the decomposeEulerXY() method is tested to see if it can decompose this vector into a functionally equivalent Euler angle. Manual testing with the motion control scripts to make sure the pose of my arms would reasonably realistically rotate the diver's arms was done as well.

Instructions to Run Project:

1. Install nodejs
2. Clone ocean simulator repository
3. Run "npm install" in there
4. Run "npm run dev" in there
5. Navigate to https://localhost:8080 from the development computer and wait for it to load
6. Let web page access camera
7. Back up so the camera can see your arms
8. Test movement

Feedback:

Jacob (Isaac's twin brother) is a member of the target user and is our group's representative of the general user. He did not test iteration 2, but he gave these reviews after testing our progress in iteration 1: Positives: It's more responsive to all joints compared to previous iteration and results in more interactivity. Negatives: After considering our lack of requirements for motion detection of the lower body, there is nothing to report on the previous feedback given regarding the detection of lower body movement. Jason is eager to see a more immersive environment of the simulator and cannot produce anymore feedback until the 3D environment as well as multiple 3D models are implemented into the simulator.