Project Outline

Title: The Void Rescuer

Project Overview:

Time is wrapping, your fuel is screaming, and the massive shadow of a singularity is devouring everything in sight—can you pull your team back from the brink of the void?

The Void Rescuer is a 3D space adventure game where the player acts as a pilot on a high-stakes rescue mission. The goal here is to save stranded astronauts floating near a massive Black Hole. The game is built around "Gravity Physics"—the closer a player gets to the center, the stronger the pull becomes, making it harder to fly away. Players must use a "Tether Beam" (a tractor beam) to grab astronauts and pull them to safety while dodging moving asteroids and managing their ship's energy.

Core Features:

1. Realistic Gravity & The Danger Zone

The center of the game map is a Black Hole that constantly pulls the player and all objects toward it. We use a math formula called the "Inverse Square Law" so that the gravity feels weak at a distance but becomes incredibly powerful (and dangerous) as you get closer to the center. To make this look exciting, the Black Hole is surrounded by a "swirling ring" of particles, and if you touch the center, it is an instant Game Over.

2. The Tether Beam (Rescue Mechanic)

The player fires a "Tether Beam" to catch astronauts. This uses Raycasting to detect a hit. Once caught, the astronaut is linked to your ship by a visible energy line. Because of physics, the astronaut doesn't move perfectly with you; they have weight and momentum, meaning if you turn too fast or fly too wildly, the tether can snap, sending the astronaut drifting back toward the Black Hole.

3. Dual-Screen HUD and Speed Effects

To help the player navigate in 3D space, the screen is split into two parts. The main screen shows the pilot's view (what is in front of the ship), while a smaller "Radar" window shows a top-down map of the whole area. When the player uses their thrusters to go fast, the camera's Field of View (FOV) stretches out. This creates a "warp speed" visual effect, making the rescue feel more intense and fast-paced.

4. Moving Asteroid Field & Energy Survival

The game world is not static; it is filled with asteroids that orbit the Black Hole. These act as moving obstacles that the player must weave through. Additionally, the ship has a limited energy supply. Players must find "Solar Crystals" floating in space and use their beam to "suck" energy from them to stay powered up. This forces the player to choose between playing it safe or diving deep into dangerous territory for more fuel.

Work Distribution Among members:

Member 1: Physics & Movement:

- **Gravity Logic:** Create the code that pulls the player and astronauts toward (0,0,0).
- **Tether Connection:** Write the logic so that a rescued astronaut follows the ship smoothly.
- **Ship Controls:** Build the W-A-S-D movement so the ship accelerates and slows down naturally.

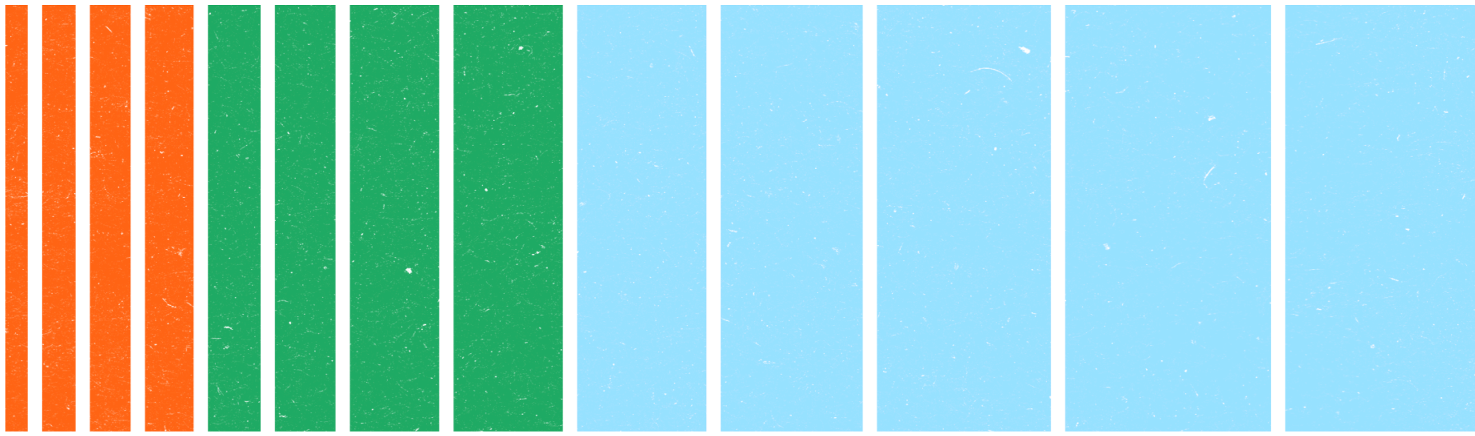
Member 2: Visuals & Camera

- **Split-Screen:** Set up the code to show the main game and the mini-map at the same time.
- **Black Hole Design:** Use spheres and spinning particles to make a cool-looking central singularity.
- **Speed Visuals:** Make the camera "stretch" (FOV change) when the ship goes fast.

Member 3: Gameplay & Rules

- **Spawning System:** Make asteroids and astronauts appear at random spots in orbit.
- **Rescue Goal:** Create the "Rescue Gate" where players must bring astronauts to win points.
- **Information Display:** Use text on the screen to show the player their speed, energy, and distance.

Work Distribution



The Void Rescuer

Cosmic Environment & Physics Set-Up

- ☐ Modify the `grid` and `floor` logic to create a dark "Space" background with star particles.
- ☐ Implement the **Singularity** (Black Hole) at the center (0,0,0) using a pulsing gluSphere.
- ☐ Create the **Inward Gravity Logic** by making a `moveBlackHole()` function so it pulls the *Player* toward the center at all times, and uses **Inverse Square** Law for determining the distance between the two.
- ☐ Define the `xbound` and `ybound` as the "Event Horizon"—if the player crosses these, `gameOver = True`.

Camera Set-up

- ☐ For setting up the 3D camera, we need 3 sets of coordinates:
    - ☐ **Eye (x, y, z):** Where the camera is standing.
    - ☐ **Center (x, y, z):** What the camera is looking at (The Ship).
    - ☐ **Up (x, y, z):** Which way is "up" (usually (0, 1, 0)).
- ☐ **The Math:** If the ship is at `(shipX, shipY, shipZ)`, the camera should be at `(shipX, shipY + 50, shipZ + 150)`.
- ☐ **Instruction:** In the `display()` function, call `gluLookAt` using the ship's current coordinates as the base. This ensures the camera moves whenever the ship moves.
- ☐ **The Math:** Use the shipAngle to position the camera *behind* the ship.
    - camX = shipX - 200.cos(math.radians(shipAngle))
    - camY = shipY - 200.sin(math.radians(shipAngle))

Ship Construction and Navigation in Space

- ☐ Build the SpaceShip Model using `gluCylinder` for the body and `glutSolidCube` for the wings.
- ☐ Set up the `player()` function to rotate using `turnAngle` (manual) and `cheatAngle` (automatic radar).

Ship Construction and Navigation in Space

- ☐ Implement **Thruster Physics**: Use the bullet movement math (50 times cos(theta)) to move the player forward when "W" is pressed (Similar to Assignment 03)

Astranaut and Asteroids

- ☐ **The Helmet:** Use `gluSphere`. We can make the visor a different color by calling `glColor3f(0, 1, 1)` (Cyan) just before drawing the sphere.
- ☐ **The Life Support Pack:** Use `glutSolidCube`. We will scale it using `glScale(1.2, 1.5, 0.5)` to make it a flat backpack on the astronaut's back.
- ☐ **Floating Animation:** To make them look like they are "drifting," we will apply the `wave` variable to their Z-axis.
    - ☐ *Code Logic:* `glTranslatef(x, y, z + (wave * 5))`
    - ☐ This makes the astronaut bob up and down slowly in the void.
- ☐ For Asteroids, **the Base:** Start with a `glutSolidSphere`.
- ☐ **The Jagged Look:** Before drawing the sphere, we use `glScalef(1.0, 1.3, 0.8)`. This stretches the sphere into an irregular "potato" shape.

Astranaut and Asteroids

- ☐ To make every asteroid look different, we can give each one a random `rotationAngle` in the `generate()` function.
    - *Code Logic:* `glRotatef(randomAngle, 1, 1, 0)`
- ☐ **Coloring:** Use `glColor3f(0.5, 0.4, 0.3)` to give them a dusty, brown rock appearance.
- ☐ **`astronautList`**: Stores `[x, y, z]` for people to save.
- ☐ **`asteroidList`**: Stores `[x, y, z, size]` for hazards to avoid.
- ☐ In the `display()` function, we will simply loop through these lists and call the `drawAstronaut()` or `drawAsteroid()` functions at those coordinates in space.

Rescuing the Astronauts (The Tether System)

- ☐ Build the SpaceShip Model using `gluCylinder` for the body and `glutSolidCube` for the wings.
- ☐ Set up the `player()` function to rotate using `turnAngle` (manual) and `cheatAngle` (automatic radar).
- ☐ Implement **Thruster Physics**: Use the bullet movement math (50 times cos(theta)) to move the player forward when "W" is pressed (Similar to Assignment 03)

Rescuing the Astronauts (The Tether System)

- ☐ Create a generate() function that places astronauts randomly in space without putting them in danger immediately.
  - ☐ This function picks a random (x, y) coordinate and checks it against a "Danger Zone."
  - ☐ Use `random.uniform(min, max)` to pick a spot on the map.
  - ☐ **The Safety Check:** Use the Distance Formula: sqrt{dxˆ2 + dyˆ2}. If the distance from the **Center (Black Hole)** is less than 300*(estimated)*, discard that spot and pick again. We don't want astronauts spawning inside the hazard.
  - ☐ Save these coordinates in `astronautList.`
- ☐ Make a raycast() function that detects if the ship is pointing directly at an astronaut so we can "lock on." It is similar to a laser beam shooting out from the front of your ship. We check points along that beam to see if any point is touching an astronaut.
  - ☐ **The Beam:** Create a loop that "walks" forward from the ship's position in steps of 10 units (up to 2000 units).
  - ☐ At each step, calculate the point using: x_coordinate = shipX + (i . cos(shipAngle)), y_coordinate = shipY + i . dot sin(shipAngle))

Rescuing the Astronauts (The Tether System)

- ☐ **The Detection:** For every step, check the distance between that point and every astronaut. If the distance is less than the astronaut's size (radius), return the `position` of that astronaut. This is your "Target Lock."

☐ Make a RenderTether() function, which acts as a visible "rope" of light between the ship and the rescued astronaut.

- ☐ **The Primitive:** Use the GL_LINES mode.
- ☐ **The Points:** The line starts at the ship's (x, y, z) and ends at the tethered astronaut's (x, y, z).
- ☐ **The Animation:** To make it look like a "tractor beam," use a sine wave (math.sin(time)) to change the line's brightness or thickness (glLineWidth) over time. This makes the beam "pulse."

☐ Make a moveTethered() function that makes the astronaut follow the ship like they are being pulled by a tractor beam. We calculate the direction from the astronaut to the ship and move the astronaut a small "step" in that direction every frame.

- ☐ **The Direction:** Calculate the vector: direction = ShipPosition - AstronautPosition.

- ☐ **Normalization:** Divide the direction by the total distance. This gives you a "Unit Vector" (a direction with a length of 1).

Rescuing the Astronauts (The Tether System)

- ☐ **The Pull:** Multiply that Unit Vector by the ship's speed. Add this to the astronaut's position.
- ☐ **The Snap:** If the distance becomes too large (e.g., > 500 units), the tether "breaks." Set isTethered = False.

Ship Navigation using Arrow Keys

- **Left/Right Arrows:** These change the `shipAngle`. They don't move the ship; they just spin it in place.
- **Up/Down Arrows:** These move the ship forward or backward based on whatever way it is currently facing.

☐ Create a keyboard(key, x, y) function.

- ☐ **Rotation:** shipAngle += 5 (Left) or -= 5 (Right).
- ☐ **Thrust:** Use trigonometry to find the new position.
  newX = shipX + (speed.cos(math.radians(shipAngle))
  newY = shipY + (speed.sin(math.radians(shipAngle)))

Tether Shoot and Visuals using Special Keys

- **Left Mouse Click:** Tether Shot
- W/S/A/D/ right mouse click: for visuals, coming later.

Tether Shoot and Visuals using Special Keys

- ☐ Create a `mouse(button, state, x, y)` function.
    - ☐ **The Trigger:** `if button == GLUT_LEFT_BUTTON and state == GLUT_DOWN:`
    - ☐ **The Action:** Call the `raycast()` function.
    - ☐ **The Connection:** If `raycast()` returns an astronaut ID, set `isTethered = True` and store that astronaut's index.