1. Overview

1.1. Problem Description

The primary objective of this project was to simulate water flow and extraction in a dynamic environment, initially in a 2D grid and subsequently in a 3D space. The challenge was to accurately model the behavior of water under the influence of varying gravity directions and implement a rendering technique that visualizes this behavior effectively. Furthermore, the project aimed to develop an AI agent capable of optimizing the extraction process using advanced decision-making algorithms.

1.2. Solution Overview

I started by simulating water in a 2D grid using a modified cellular automata approach to support a general gravity direction. This involved calculating the free-fall paths of cells using the Bresenham algorithm and creating a helper class for traversing the 2D grid horizontally and vertically. This class indexed the cells of the grid to support the general gravity direction such that the lower the cell in the down direction (in relation to gravity), the lower the index. Additionally, I implemented water compression physics to simulate realistic water behavior.

Horizontal water balancing was achieved using the flood fill algorithm, which allowed for an even distribution of water across the grid. After establishing a functional 2D simulation, I expanded the simulation to 3D by adapting the helper traversing class, Bresenham algorithm, and flood fill algorithm for three-dimensional space.

To visualize the 3D grid, I employed the marching cubes algorithm, which is a widely used technique for rendering isosurfaces in volumetric data. This allowed for a clear and accurate representation of the water simulation. Additionally, I was tasked with creating a user interface for the simulator, where users could import their `'.obj'` models, these models were used to generate the grid, with the model's volume defining the barrier cells within the simulation. The scan line algorithm was implemented to approximate the imported model's geometry using the marching cubes algorithm, allowing the simulator to accurately represent the barriers within the 3D grid.

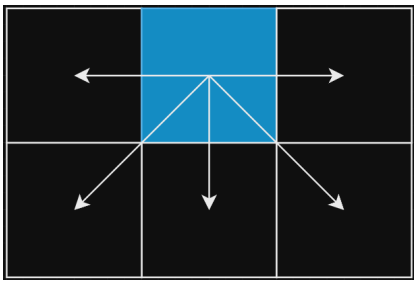
To optimize the simulation, I introduced multithreading by dividing the simulation into different stages and splitting the grid into chunks, each handled by a separate thread. To mitigate race conditions, I limited the maximum speed of water cells and implemented an odd-even simulation stage pattern. The 3D grid was rendered using the marching cubes algorithm, with each chunk rendered in a different thread to enhance performance.

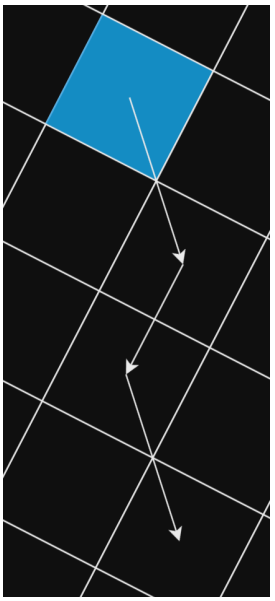
For the AI component, I used a modified Monte Carlo Tree Search (MCTS) algorithm tailored to support a single-agent game. The MCTS algorithm was chosen for its ability to return good results in a reasonable amount of time, given the high branching factor of the problem. The modified MCTS was designed to efficiently navigate the decision space and optimize the water extraction process by evaluating potential actions and their outcomes.
Overall, this solution integrates advanced simulation techniques with robust AI decision-making to address the complexities of simulating and optimizing water flow in a dynamic environment.

2. The Simulation Logic

2.1. 2D Simulation Logic

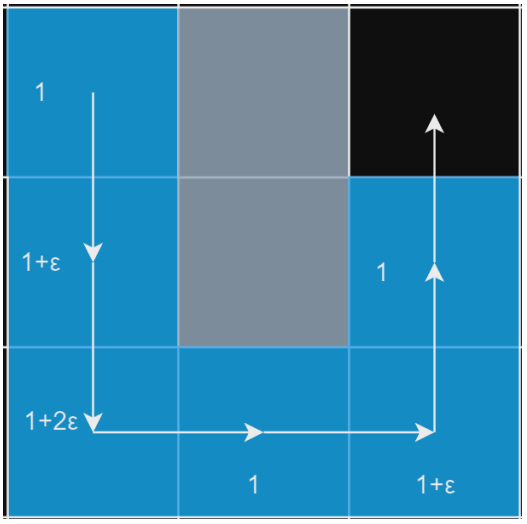
The foundation of the simulation began with defining the basic transition logic of cells within a cellular automata framework. Each cell follows a priority system where it first attempts to flow downwards, then diagonally, and finally spreads horizontally.





To accurately simulate free fall, I incorporated a momentum float and utilized the Bresenham algorithm to calculate the fall path, allowing each cell to move to the farthest possible cell along this path.The simulation order of the grids needs to be from the bottom up to avoid one water cell falling all the way down in one frame.

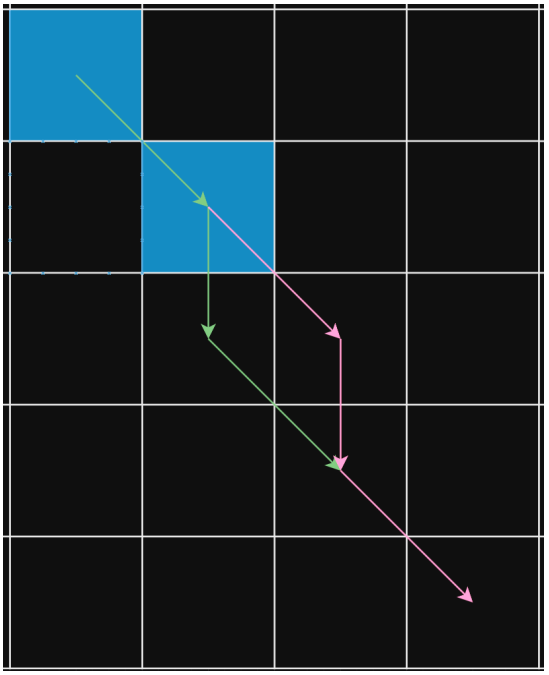
To implement water compression, each cell was given a float representing the volume of water it holds and the maximum volume it can support. At the beginning of each update loop, the simulation starts from the top cells. For each cell, the maximum supported volume is updated to be the maximum supported volume of the water cell directly above it on the Bresenham path, plus a global constant called epsilon. If there are no water cells above, the maximum supported volume is set to a default value.



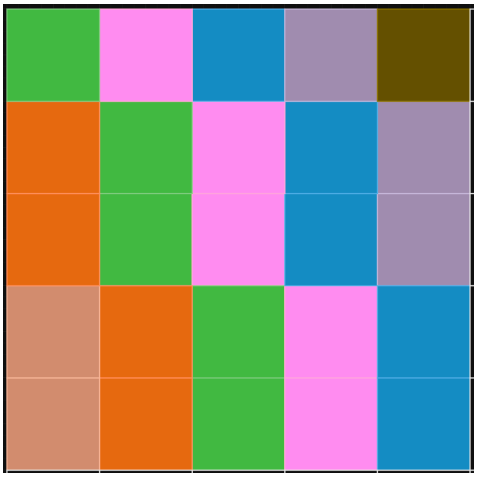
Next, the water cells are balanced horizontally using a flood fill scan, which identifies all horizontally

connected cells and balances the water volume among them. Following this, the grid is iterated over again, and for each cell with a volume exceeding its maximum supported volume, the excess volume flows upwards, thereby mimicking water compression physics.

A significant issue with this approach was that calculating the Bresenham path from different origins could result in inconsistencies. Cells on the path from one origin might not be included if the path were calculated from a different origin, causing failures in the calculation of the maximum supported volume and horizontal balancing.

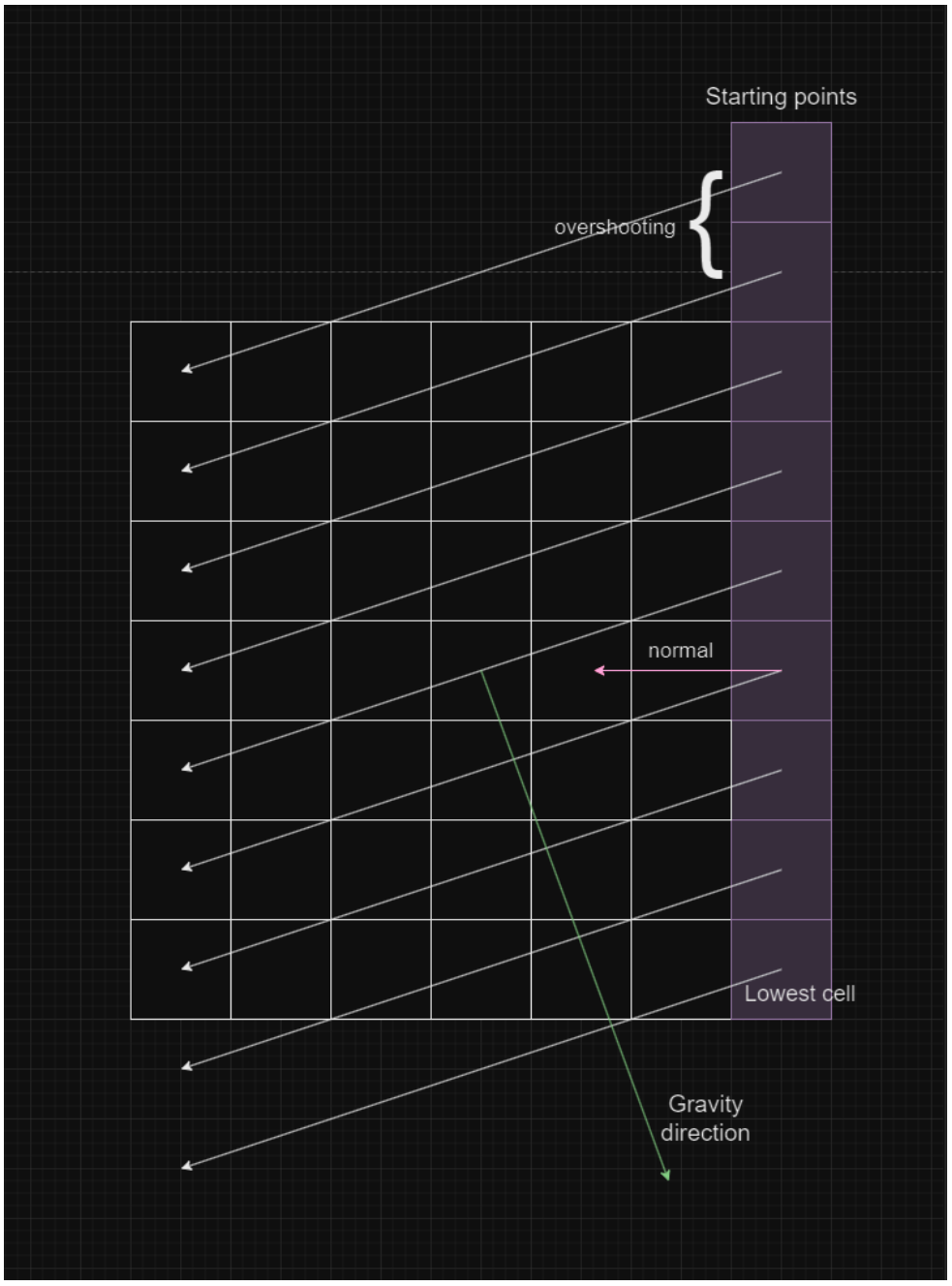


To resolve this, I implemented a helper class for grid traversal. This class calculates a single global Bresenham vertical path for each vertical layer in the grid, ensuring that all cells on a single Bresenham line use the same fall path. A similar approach was applied for horizontal traversal, ensuring consistent behavior.



In the 2D case, to implement the traversing lines class, a starting points vector was chosen for traversing the grid with horizontal layers, in the implantation of the Bresenham algorithm the axis thats more aligned with the direction vector gets incremented along the path, therefore, since the direction of the horizontal traversing is perpendicular to the gravity direction, the starting point vector needs to be the most aligned edge with the gravity direction, otherwise some points will appear multiple times in the traversing.
For simplicity, the starting points vector is chosen along an edge that includes the lowest cell in the grid (the normal of an edge that includes the lowest cell will always have a non positive dot product with the gravity direction), therefore, to find the starting point vector we take the edge that has the biggest non positive dot product with the gravity direction.

To include the cells that cant be reached by any horizontal traversing line originating from a cell on the grid, we define an over shot direction for the starting point vector.



Another problem arose from horizontal traversing from right to left, causing cells to prioritize flowing left due to the traversal order. To address this, I introduced a random permutation at the beginning of each frame, making the horizontal traversal order random.

Additionally, when a water cell is surrounded by barriers on the sides and above, receiving volume from below due to compression would cause the water to become infinitely trapped. To fix this, I set an upper bound on the volume each cell can hold, preventing infinite trapping.

2.2. Expanding to 3D

Expanding the simulation to 3D involved adapting the traversal class to support a 3D grid, we find the three faces of the cubic grid that include the lowest cell and defined the starting points vector to be one of the intersection lines between any pair of those faces that's most aligned with the gravity direction, the horizontal layers are 2d planes instead of lines, while vertical traversing became a 2D list of vertical columns. The Bresenham algorithm was expanded to accommodate 3D space, and the flood fill algorithm was modified to work on 2D planes instead of lines.

By extending these fundamental algorithms and traversal methods to 3D, the simulation could accurately represent water flow and compression in a three-dimensional environment, setting the stage for realistic rendering and AI-driven optimization.

3. Optimization and Multithreading

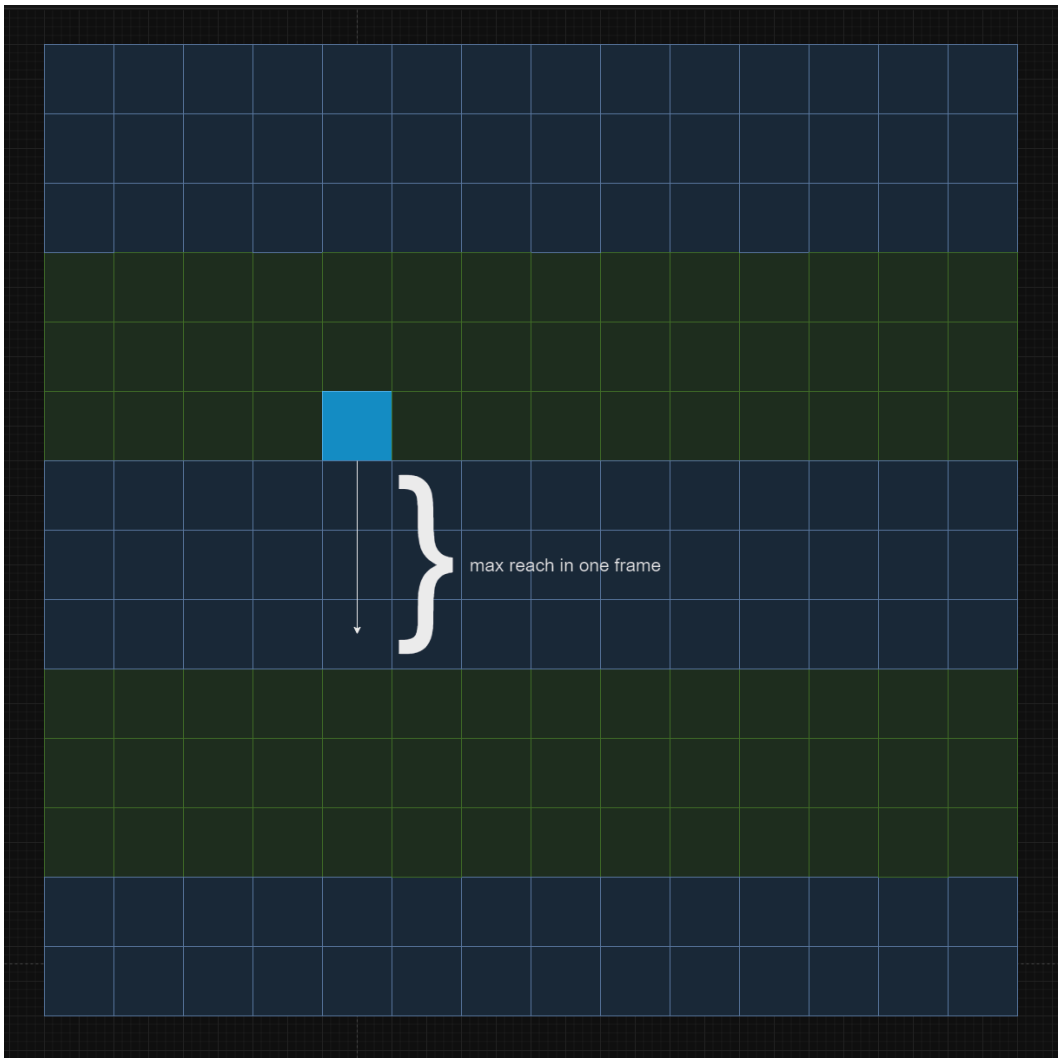
To enhance the efficiency and performance of the water simulation, the process was divided into distinct stages and parallelized using multithreading. This approach reduced the computational load on individual threads and minimized race conditions.

3.1 Updating Compression

In this stage, the compression of each water cell by the cells above it is simulated. The 3D grid is divided into vertical columns, with each column handled by a separate thread. This division allows for concurrent processing of the compression effects across the entire grid, improving the overall simulation speed.

3.2 Simulation

The grid is split into horizontal layers, with a constant value c set as the maximum speed for water cells. The height of each horizontal layer is also set to this constant. The layers are indexed, and during each simulation step, water cells can only remain in the same layer or move to the direct layer below. By simulating all even layers concurrently, followed by all odd layers, race conditions between threads are avoided. Each layer is processed independently by different threads, ensuring efficient parallel computation.



3.3 Horizontal Water Balancing

In this stage, the grid is divided into horizontal layers, each of which is processed by a separate thread. This step balances the water horizontally across the grid, ensuring that water spreads out evenly and preventing bottlenecks in the simulation.

3.4 Water Upwards Flow

After balancing the water horizontally, the simulation handles the upwards flow of water due to compression. Similar to the compression stage, the grid is split into vertical columns, with each column managed by a different thread. This parallel processing allows the simulation to accurately model the effects of water being forced upward due to the pressure from compressed water cells below.

3.5 Rendering with Marching Cubes

For the rendering phase, the 3D grid is divided into cube-shaped chunks, with each chunk rendered by a separate thread. This approach allows the rendering process to be parallelized, significantly reducing the time required to visualize the simulation results. By leveraging multithreading for both the simulation and rendering stages, the entire process is optimized for performance, enabling real-time interaction with complex water simulation scenarios.

4. Marching Cubes Rendering and User Interface

The user interface for the simulator is designed to provide a clear and interactive way to manage and visualize the water simulation. It is divided into three distinct screens, each catering to different aspects of model import, environment configuration, and simulation control.

4.1 Screen 1: Model Import

The first screen allows users to import their desired model into the simulator. Users can upload a 3D model file, typically in the .obj format. This screen provides a straightforward interface for selecting and importing the model, which will serve as the basis for generating the grid used in the simulation.

4.2 Screen 2: Environment Configuration

In the second screen, users configure the simulation environment by setting several key parameters:
1. Size of the Grid: Users specify the dimensions of the simulation grid, determining the scale and resolution of the simulation space.
2. Simulation Steps Per Second: This parameter controls the frequency of simulation updates, defining how many steps are processed each second to simulate the water dynamics.
3. Simulation Steps Until a Render Step: Users set the interval at which the simulation results are rendered, balancing between simulation detail and rendering frequency.
4. Terminal Velocity: This defines the maximum speed c of water cells, a crucial parameter for managing multithreading efficiency during the simulation.

Additionally, users have the ability to move, rotate, and scale the imported model to ensure it fits within the grid boundaries. Once the environment parameters are configured and the model is properly positioned, users confirm their settings to generate the simulation environment. At this point, the simulation begins by filling all empty cells with water, establishing the initial state for the simulation.

4.3 Screen 3: Simulation Control

The third screen provides control over the simulation process. Users have two primary options:
- Manual Rotation: Users can manually rotate the grid to explore different angles and perspectives, allowing for interactive experimentation with the simulation environment.
- AI Decision: Alternatively, users can opt for the AI to determine the best next rotation for the grid. Leveraging the AI's decision-making capabilities, the simulation can be adjusted based on the calculated optimal rotation to enhance the results or achieve specific goals.

This multi-screen interface ensures that users have a comprehensive set of tools to both configure and interact with the simulation, facilitating an intuitive and effective user experience.

5. AI Agent Logic

The AI agent logic in this simulation utilises a modified Monte Carlo Tree Search (MCTS) algorithm, which is designed to make effective decisions by simulating and evaluating potential future states of the environment. MCTS was chosen for it's ability to find good solutions in a reasonable amount of time in a high branching factor problem such as this water extraction problem (do the infinite amount of rotations possible), mcts is also an anytime algorithm meaning it will still provide a good enough solution if stoped early. The core components of the MCTS algorithm implemented in this project include the tree structure, selection, expansion, rollout, and backpropagation phases. Here's a detailed overview of how the AI agent logic operates:

5.1 MCTS Tree Structure

The MCTS algorithm is built around a tree structure, where each node represents a snapshot of the simulation state. The `MCTSNode` class captures this structure, holding the following attributes:
- CellularAutomataSnapshot: Represents the state of the simulation at a given node, including the grid of cells, total water volume, and environmental rotation.
- Depth: Indicates the depth of the node within the tree.
- Parent and Children: Establishes the hierarchical relationships between nodes.
-Visits and Evaluation: Tracks the number of times a node has been visited and its evaluation score.
- Best Rollout: Stores the best result obtained from the rollouts.

5.2 Selection Phase

During the selection phase, the algorithm traverses the tree to select a leaf node for expansion. The node is chosen based on a balance between exploration and exploitation. During selection we navigate through the tree, at each node we determine whether to expand a new child or continue the selection process through an existing child node based on this formula:

depthFactor = **hyperparameter weight** * (maxDepth - parent.depth) / maxDepth
ShouldExpand = (averageUCTValueOfChildren - depthFactor) < **hyperparameter threshold**

It calculates whether the average UCT value of existing children, adjusted for depth, is below a threshold that favors expansion.

5.3 Expansion Phase

Once a leaf node is selected, the expansion phase creates a new child node representing a new state of the simulation. During expansion we restore the simulation state to that of the selected node, perform a simulation step, and then create a new child node for this resulting state. The depth of the child node is set to one more than the parent node.

5.4 Rollout Phase

The rollout phase involves simulating the environment from the newly expanded node to evaluate its potential. During the rollout phase we simulate the environment for a certain number of steps or until a predefined condition is met (e.g., water volume drops below 5%). The result of this simulation, including the percentage of extracted water and the depth of the simulation, is recorded in a `RolloutResult` class.

5.5 Backpropagation Phase

The backpropagation phase updates the evaluation of nodes based on the results of the rollout. During the backpropagation phase we traverse from the leaf node up to the root, updating the visits count and evaluation score of each node. The evaluation score is computed using a weighted combination of various factors, including depth, extracted water percentage, best rollout results, and evaluations of child nodes using the following formula:

1. Depth Score:
  - Nodes that are closer to the root (i.e., shallower) are preferred over nodes that are deeper in the tree. This is done to favor solutions that require fewer steps to achieve.
  - depthScore = (1.0 − node.depth) / maxDepth

2. Extracted Water Percentage:
  - This measures how much water has been extracted compared to the initial total water volume. A higher percentage of extracted water indicates a better result.

3. Average and Maximum Child Values:
  - The evaluation of child nodes provides insights into the potential outcomes if the current node were to be expanded. The average and maximum evaluations of children help in assessing the quality of the node.

4. Best Rollout Result:
  - This factor reflects the best result obtained during rollouts from the node. It combines the extracted water percentage and penalizes nodes deeper in the tree.
  - rolloutFactor = rollout.extractedWaterPercentage * (1.0 - rollout.depth / maxDepth)

Combining all these factors, the overall evaluation score for a node is calculated as follows

node.eval = **hyperparameterWeightDepth** * depthScore +

hyperparameterWeightExtractedWater * extractedWaterPercentage +

hyperparameterWeightRollout * rolloutFactor +

hyperparameterWeightAverageChildren * averageChildrenValue +

hyperparameterWeightMaxChild * maxChildValue;

By combining these factors using weighted contributions, the algorithm calculates an evaluation score that helps determine the most promising nodes for further exploration.

5.6 Decision Phase

In the decision phase, the AI selects the best action based on the evaluation of children nodes. In this phase we restore the simulation state to its original form and then determine the optimal action by identifying the action corresponding to the node with the highest evaluation score.

This approach allows the AI agent to make informed decisions by exploring and evaluating potential future states of the simulation, ensuring effective management of the water simulation environment.

5.7 Hyperparameter Tuning

To optimize the performance of the MCTS model, a random search method was employed for hyperparameter tuning. This approach involved testing numerous random combinations of hyperparameter values across two different 3D models. The goal was to identify the best-performing configuration while reserving a third 3D model for final validation. After extensive testing, the optimal set of hyperparameters was identified as follows:

- **shouldExpandNewChildUCTThreshold**: 0.7152
- **selectionExplorationWeight**: 0.2
- **expansionExplorationWeight**: 100.0
- **expansionDepthWeight**: 0.1
- **evalWeightDepth**: 0.0494
- **evalWeightExtractedWater**: 0.2990
- **evalWeightRollout**: 0.1948
- **evalWeightAverageChildren**: 0.2777
- **evalWeightMaxChild**: 0.1791

This set of parameters was chosen for its ability to balance exploration and exploitation, depth management, and effective evaluation of both individual nodes and their children.

5.8 Performance vs random agent

The MCTS and the random agent were tested on a third 3D model, each agent was evaluated 10 times with 1000 iteration each, the average was taken:

- A solution is accepted if the extracted water percentage reaches 95% or above.
- The max depth is 6, if a solution is not found within a depth of 6, the agent will return the solution with the highest extracted water percentage possible with a depth of 6.

# iterations	25	50	100	500	1000
MCTS	Depth - 5.8 Extracred - 88%	Depth - 5.8 Extracred - 91%	Depth - 5.6 Extracred - 92%	Depth - 5.1 Extracred - 94%	Depth - 4.8 Extracred - 95%
Random	Depth - 6 Extracred - 65%	Depth - 6 Extracred - 69%	Depth - 6 Extracred - 71%	Depth - 6 Extracred - 76%	Depth - 6 Extracred - 76%

as can be seen from the testing results, the mcts model significantly outperformed the random agent