R-Tree and Point Location

Sorbonne Université

Thomas Thanh-Long Nguyen (28626005)

January 16, 2026

Introduction

This project was made within the scope of the **Algorithm and Data Structures (4MA316)** course at **Sorbonne Université**, taught by Didier Smets and Ani Miraci.

The project involves implementing an R-Tree data structure in C to efficiently handle spatial queries on a 2D mesh. The primary objective is to optimize the search for a triangle containing a specific query point (x, y).

With a basic approach, finding a containing triangle requires checking every single triangle in the mesh. We could call this the Naive search, that has a $O(N)$ time complexity. The R-Tree search improves this by using a hierarchy of bounding boxes (Minimum Bounding Rectangles, MBR) to narrow down the search space, achieving a logarithmic average search time complexity, $O(\log N)$.

This data structure has been introduced in 1984 by Antonin Guttman (`http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf`).

Acknowledgements: I would like to thank my teachers for their guidance and for providing the necessary resources and foundations to carry out this project.

Contact:
nguyen_thomas_t-l@hotmail.com
thomas.nguyen@etu.sorbonne-universite.fr

Contents

Chapter 1

What is an R-Tree?

1.1 Concept

An R-Tree is a tree data structure used for spatial access methods, i.e. for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. A common use case is storing geospatial coordinates.

1.2 Structure

The minimal bounding rectangle (MBR) is the key concept. It represents the smallest axis-aligned rectangle that completely contains a given object (such as a triangle) or a set of objects. In an R-Tree, these boxes are used to approximate the shape of data, allowing for rapid exclusion of irrelevant branches during a search.

- **Leaf Nodes**: Contain pointers to the actual data objects (in our case, Triangle IDs).
- **Internal Nodes**: Contain pointers to child nodes and a bounding box covering all rectangles in the child nodes.
- **Root**: The top-level node covering the entire dataset.

The key idea is that "nearby" objects are grouped together, allowing queries to quickly filter out large portions of the data that do not intersect the query region.

Chapter 2

Implementation and Results

2.1 Mesh Loading

We parse `.mesh` files (Medit format) to extract vertices and triangles. This data serves as the foundation for our spatial index.

For handling mesh operations, we integrated the mesh library from **TP3** from the course (provided by Didier Smets). This library provides the necessary data structures (`Mesh`, `Vertex`, `Triangle`) and functions to efficiently parse and load `.mesh` files.

2.2 R-Tree Construction

The construction algorithm proceeds as follows:

1. **Bounding Boxes**: For each triangle, we have to compute its minimal bounding rectangle. Instead we compute its minimal Axis-Aligned Bounding Box (AABB). We strictly use AABBs (defined by min/max X and Y) rather than arbitrary oriented rectangles because their intersection tests are significantly faster computationally ($O(1)$ comparisons). While they may fit the triangles less tightly than oriented boxes, the speed gain in the R-Tree logic outweighs the slight increase in overlap.

2. **Recursive Insertion**: We insert these rectangles into the R-Tree.

3. **Splitting**: When a node exceeds its capacity (also called max branching factor), it splits.

2.3 Library Source

We utilized the R-Tree implementation provided by **Superliminal** (`http://superliminal.com/sources/`), as suggested by our teachers.

One of the reasons this library is interesting is because it is made to be memory efficient. For instance, for the splitting process we mentionned the need to define the capacity of a node (max branching factor, `MAXCARD`), in that library it is determined such that each node fits exactly inside a single memory page of the machine (`PGSIZE`, typically `512` bytes in this implementation). The maximum number of branches is calculated by taking the page size, subtracting the node header (two integers: count and level), and dividing by the size of a `Branch` structure:

$$\texttt{MAXCARD} = \frac{\texttt{PGSIZE} - 2 \times \text{sizeof}(\texttt{int})}{\text{sizeof}(\texttt{Branch})}$$

This ensures that each node fits within a page, optimizing memory access.

$$\texttt{MAXCARD} = \frac{512 - 2 \times 4}{24} = \frac{504}{24} = 21$$

where $\texttt{PGSIZE} = 512$ bytes, $\text{sizeof}(\texttt{int}) = 4$ bytes, and $\text{sizeof}(\texttt{Branch}) = 24$ bytes (16 bytes for `Rect` + 8 bytes for `Node*`).

2.4 Modifications to the Library

We adapted the library to our needs for it to compile with our code:

- Added a lot of comments on top of the comments already there, to help ourselves understanding the code. Theses comments are indicated with "Ed." as for "Editor's Note".
- Used `intptr_t` instead of `int` in the code for casting pointers to integers (ID storage). We had problems with the fact that pointers are 8 bytes on 64bit architecture and 4 bytes on 32bit architecture, while `int` is 4 bytes on both.

2.5 Split Strategy: Linear vs. Quadratic

The library provides two node splitting algorithms:

- **Linear Split** ($O(N)$): Selects two seeds that are furthest apart along each dimension, then distributes the remaining entries. It is fast but may produce less optimal trees with more overlap.
- **Quadratic Split** ($O(N^2)$): Compares all pairs of entries to find the two that would waste the most area if put in the same group (the worst seeds). It then distributes the rest based on area expansion criteria.

Choice: We chose the **Quadratic Split** (default in our build) because it typically produces a tree with less overlap, leading to faster query performance ($O(\log N)$) at the cost of slightly slower build time. Since we build the tree once and query it many times, this is the optimal trade-off.

2.6 Search Algorithm

The search for a single point $P = (x, y)$ starts at the root and descends down to finding the exact triangle index. The mesh used for testing is `mesh2-tp2.mesh` from the **TP2** of the course.

Algorithm Steps:

1. Start at the R-Tree root.
2. Iterate through all entries (child nodes or leaves) in the current node.
3. If an entry's bounding box contains P, we recursively search that child branch.
4. If we reach a leaf node, we retrieve the candidate Triangle ID.
5. We verify if the point lies strictly within the triangle, using the fact that we indicate its coordinates with its vertices (barycentric coordinates).

2.7 Performance Results

To evaluate the efficiency of our R-Tree implementation, we compared it against a "Naive Search".

Naive Search Definition: This method iterates linearly through *every single triangle* in the mesh ($O(N)$) to check if it contains the point. It serves as a baseline for correctness and performance.

Accuracy Test: We generated 1000 random query points uniformly distributed within the bounding box of the entire mesh. For each point, we ran both the R-Tree Search and the Naive Search. The results were compared to ensure that both methods returned exactly the same triangle ID (or both returned "not found"), guaranteeing the correctness of our implementation.

Search Time Comparison (1000 queries on `mesh2-tp2.mesh`):

- **R-Tree Search Time**: ~ 0.000867 seconds.
- **Naive Search Time**: ~ 0.022523 seconds.
- **Speedup**: The R-Tree is approximately **25.98x faster** than the naive approach.
- **Absolute Difference**: The R-Tree saves ~ 0.021 seconds per 1000 queries.

Comparison results confirmed 100% accuracy, meaning the R-Tree always found the same triangle as the naive search, as it should.

Chapter 3

Visualizations

All visualizations in this project were generated using **Gnuplot**. The C program exports geometry data (triangles, boxes, points) into `.dat` text files, and generates corresponding `.gp` scripts to render them. The mesh used for these visualizations is `mesh2-tp2.mesh`, which was provided in the **TP2** of the course.

3.1 R-Tree Visualization

We first can see the figure 3.1 which shows the overall structure of the R-Tree and the search result for a single point, highlighting the found triangle. To better understand the tree structure, we next visualize the levels. This visualization helps confirm that the tree is balanced and that nodes properly partition the space. Inspection of several consecutive levels reveals how parent nodes include their children.

Why 3 rectangles at the root?

In our visualization of `mesh2-tp2.mesh`, at the Root (Level 0) we can see exactly 3 rectangles (branches) that contain level 1 nodes (see Figure 3.3). This observation is a direct mathematical result of the **max branching factor** (`MAXCARD`) defined earlier (21 branches per node) and the total number of triangles.

For the mesh `mesh2-tp2.mesh` with 780 triangles:

1. **Leaf Level (Level 2)**: The 780 triangles are stored in 50 leaf nodes.

$$\frac{780 \text{ triangles}}{50 \text{ nodes}} = 15.6 \text{ avg items/node} \quad (\approx 74\% \text{ fill})$$

2. **Parent Level (Level 1)**: These 50 Leaf Nodes are indexed by their parents.

$$\frac{\text{50 children}}{\text{21 max capacity}} \Rightarrow \lceil 2.38 \rceil = 3 \text{ Parent Nodes}$$

   The 50 children are distributed across 3 parent nodes (avg 16.6 children/node).

3. **Root Level (Level 0)**: The Root acts as the parent for these 3 Level 1 nodes. Since 3 fits easily into the Root (capacity 21), the Root has exactly **3 branches**.

So, the "3 rectangles" effectively divide the entire mesh into 3 large spatial partitions.

Remark on Insertion Strategy: The shape of these partitions is not random. The algorithm chooses to place each new triangle in the branch that would require the **smallest increase in MBR area**. This *Quadratic Split* strategy is crucial for keeping the bounding boxes as small and compact as possible, minimizing overlap and ensuring the efficient structure we see in the visualizations.

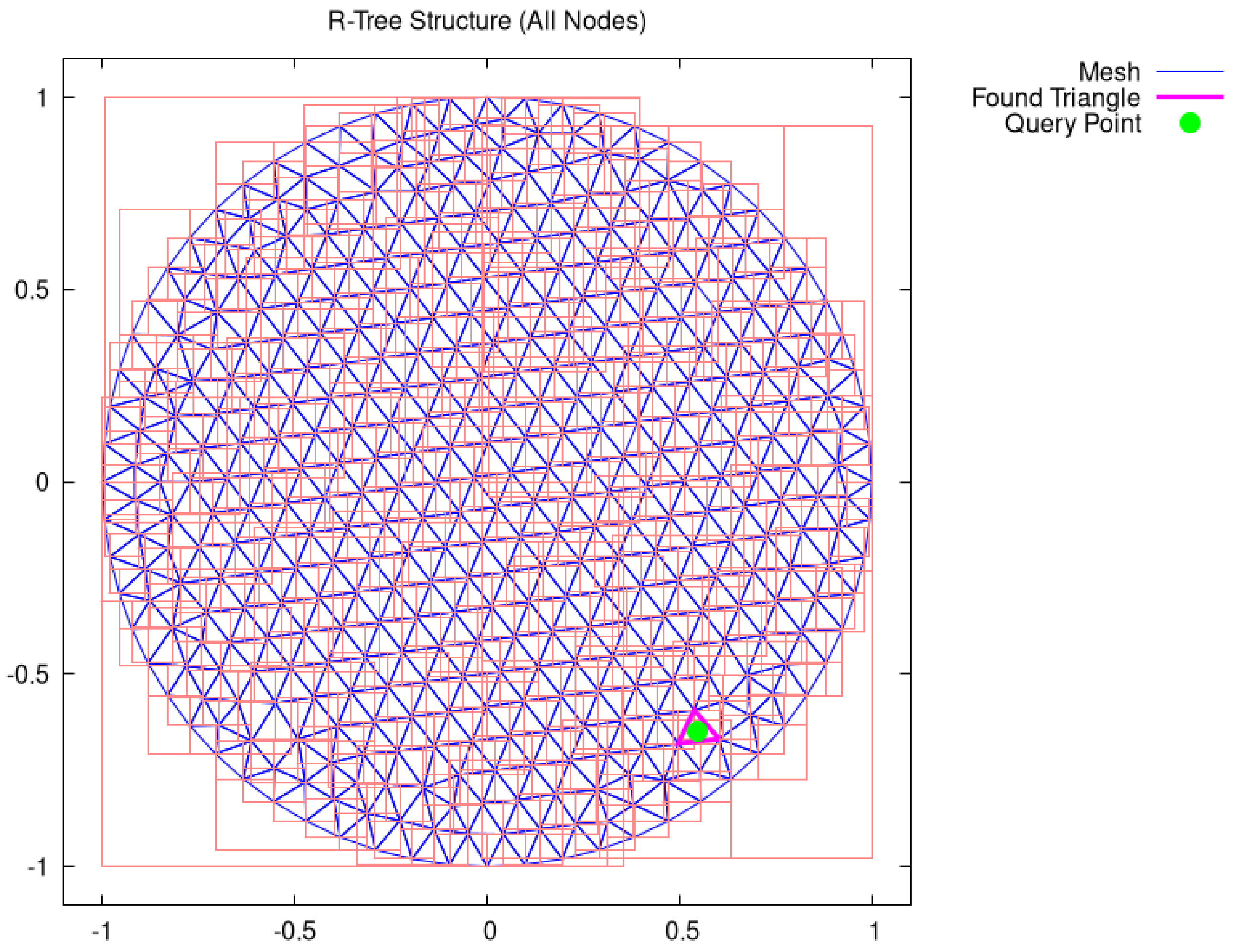

Figure 3.1: Overview: R-Tree Nodes (Light Red), Found Triangle (Magenta), Query Point (Green). The mesh is shown in Blue.

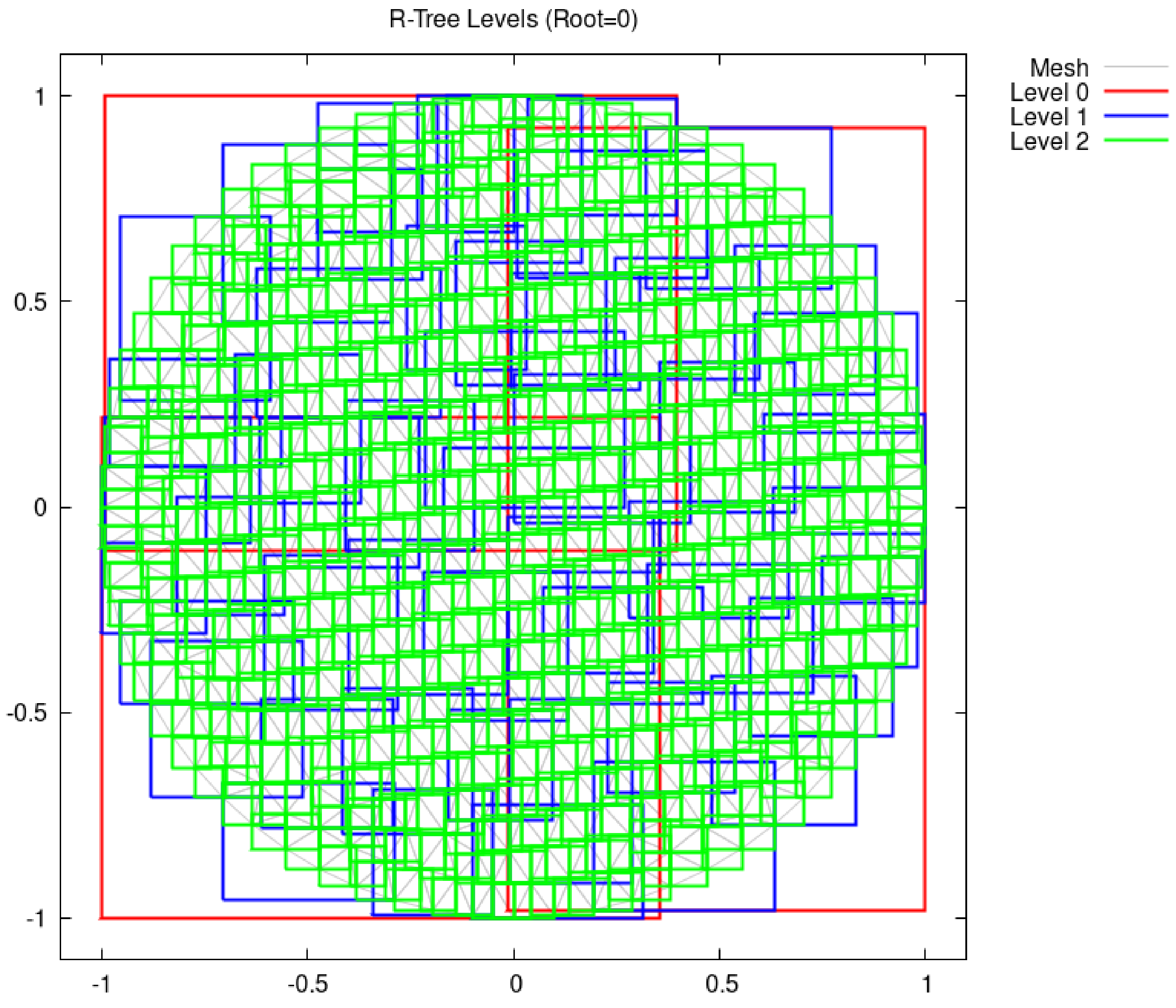

Figure 3.2: All R-Tree Levels overlaid. Different colors represent different depths.

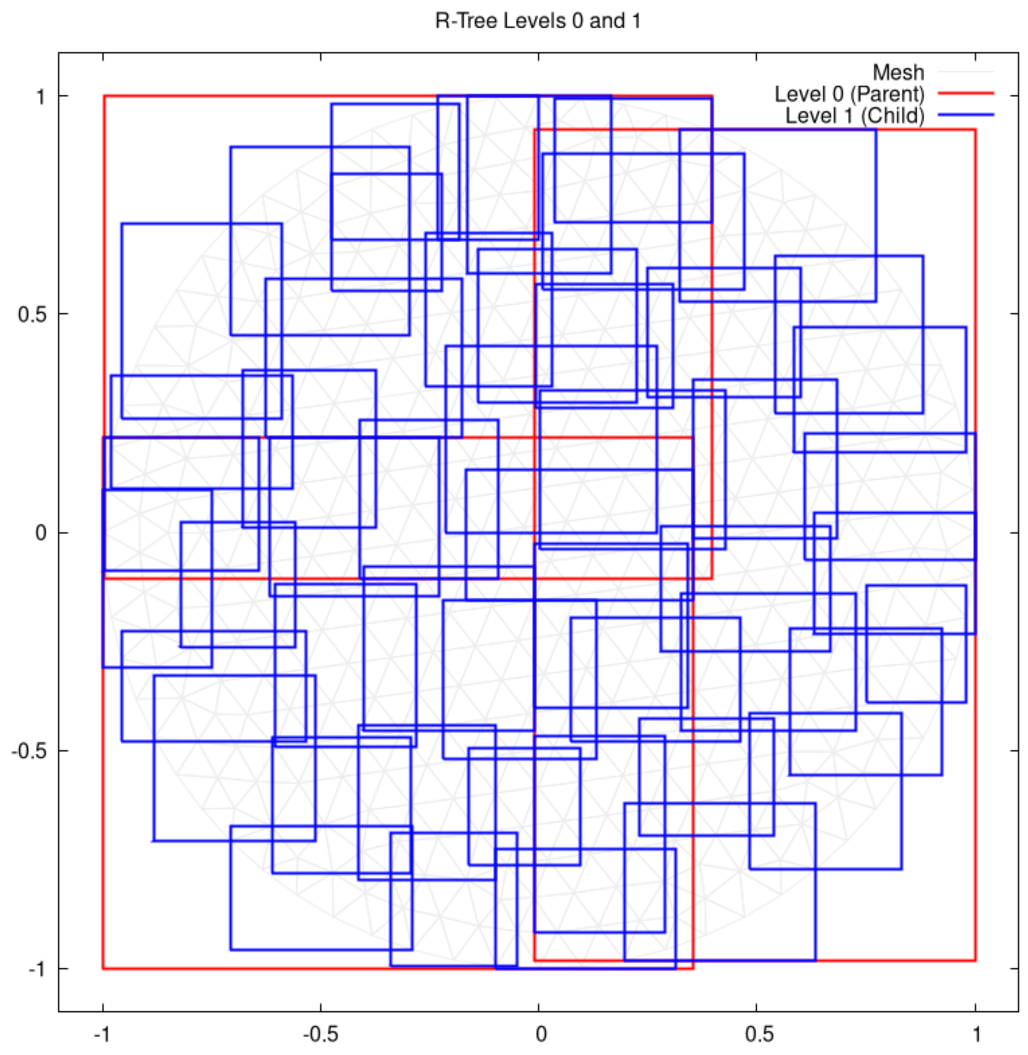

Figure 3.3: Level 0 (Root, Red) containing Level 1 (Children, Blue).

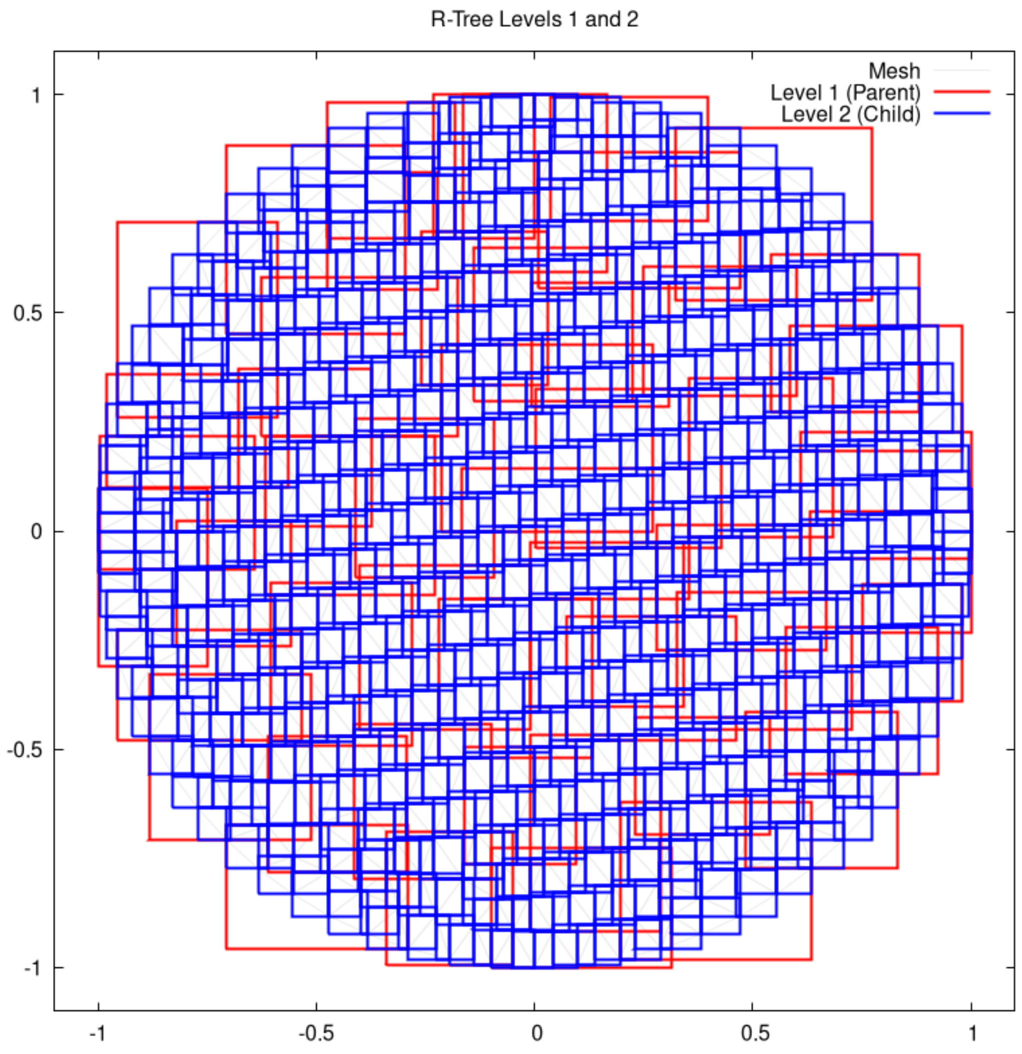

Figure 3.4: Level 1 (Parent, Red) containing Level 2 (Children, Blue).

Appendix A

How to Build and Run

This section details the steps to compile and execute the project from scratch.

1. Point Location Part

To build the project and run the point location search:

1. **Configuration (Optional)**: Before running the program, you can specify the coordinates of the query point in a file named `what_query_point.dat` in the project root directory. The file should contain two floating point numbers separated by a space (e.g., `0.1 0.1`). If the file is missing, the default coordinates $(0, 0)$ will be used.

2. **Build the project**:

   ```
   cmake -S . -B build
   cmake --build build
   ```

   This will compile the source code and link it with the R-Tree library.

3. **Run the executable**:

   ```
   ./build/RTreeRUN meshes/<mesh_name> <number_of_queries>
   ```

   Example:

   ```
   ./build/RTreeRUN meshes/mesh2-tp2.mesh 1000
   ```

This command loads the mesh from the meshes/ folder and runs the specified number of random queries.

2. Visualization Part

During the execution of RTreeRUN, the program automatically generates several .dat data files and .gp Gnuplot scripts in the plots/ directory. To generate the visualization images (PNG):

1. **Run Gnuplot**:

   ```
   cd plots
   gnuplot *.gp
   ```

   This will execute all generated scripts inside the plots/ folder.

2. **Output**: The images will be created in the plots/ directory.

   - visualization_overview.png
   - visualization_levels_legend.png
   - viz_level_X_Y.png (for each pair of start/end levels)

Appendix B

What about Greenland?

We also tested our implementation on a much larger dataset: the `greenland.mesh` file. This mesh contains 66,425 vertices and 131,189 triangles. It can be found in the medit mesh format at: `https://people.sc.fsu.edu/~jburkardt/data/medit/medit.html`.

Performance

Running 1000 random queries on this large mesh produced the following results:

- **R-Tree Search Time**: ~ 0.0013 seconds.
- **Naive Search Time**: ~ 0.5058 seconds.
- **Speedup**: **382.04x**.

The R-Tree demonstrates massive scalability, maintaining sub-millisecond query times even as the dataset has grown significantly.

Visualizations

We generated visualizations and locate *Nuuk*, the capital of Greenland. Approximate coordinates: $(280, 600)$.

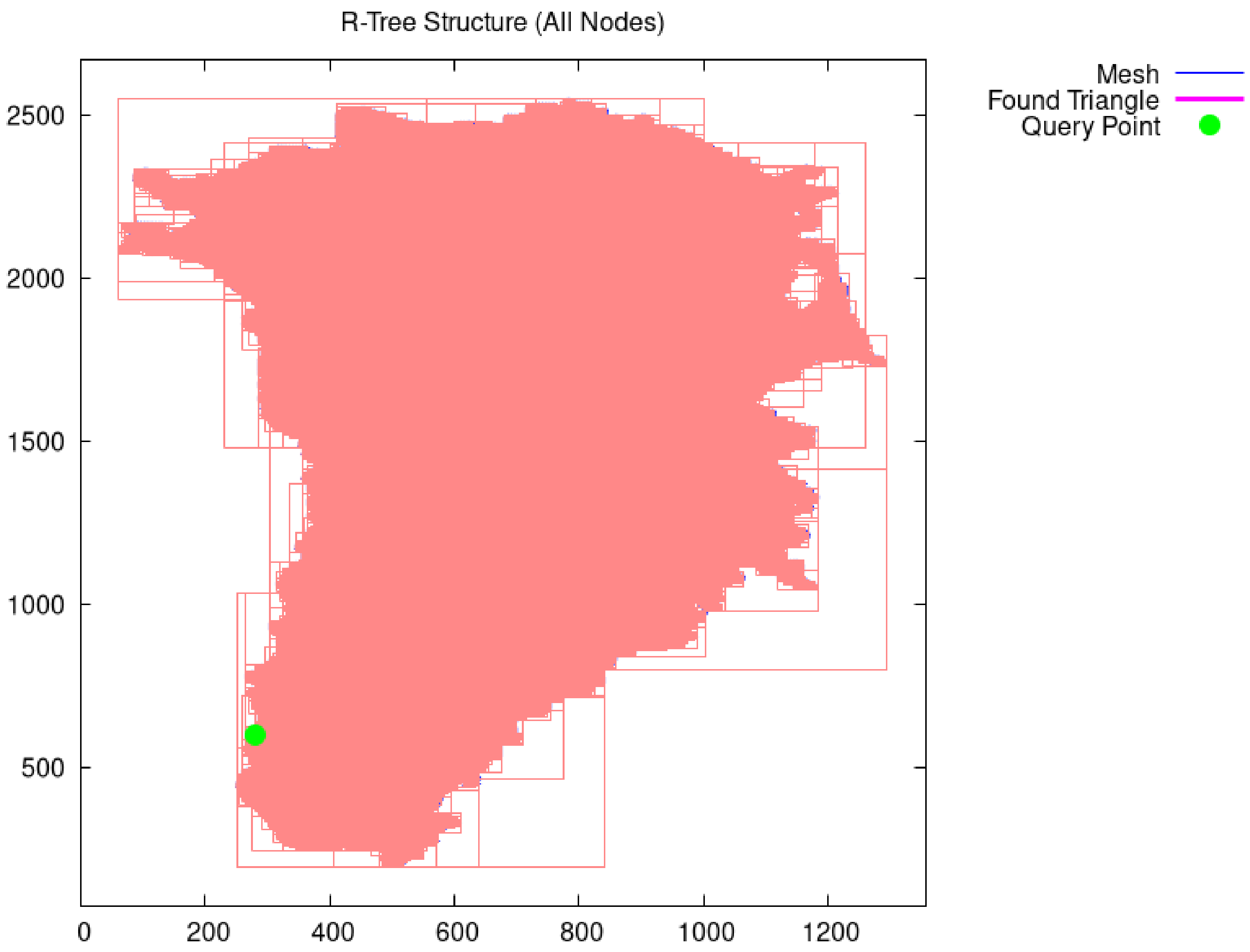

Figure B.1: Greenland Mesh with R-Tree Structure and Query Point at Nuuk (280, 600).

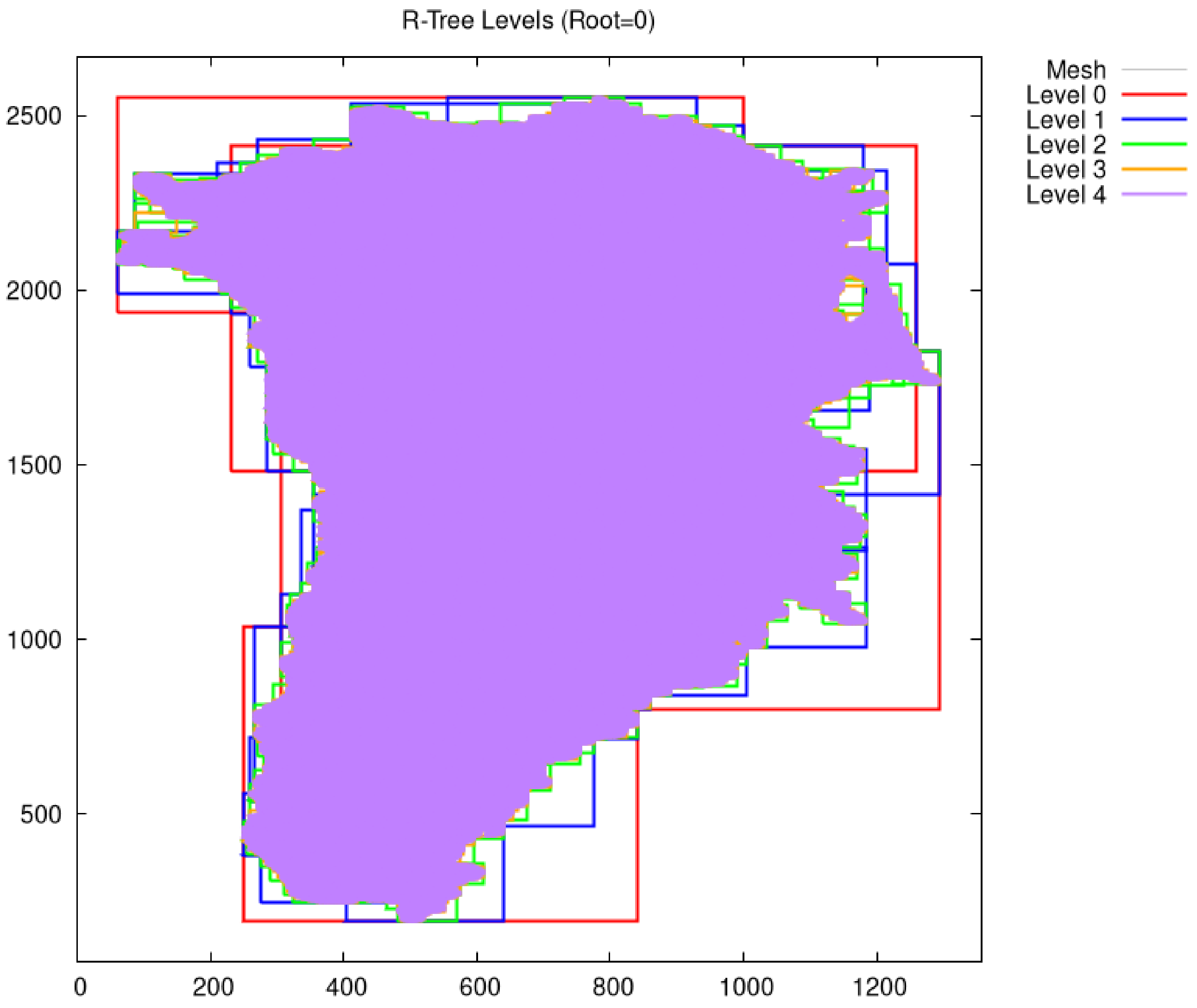

Figure B.2: All R-Tree Levels for Greenland (Depth 5).

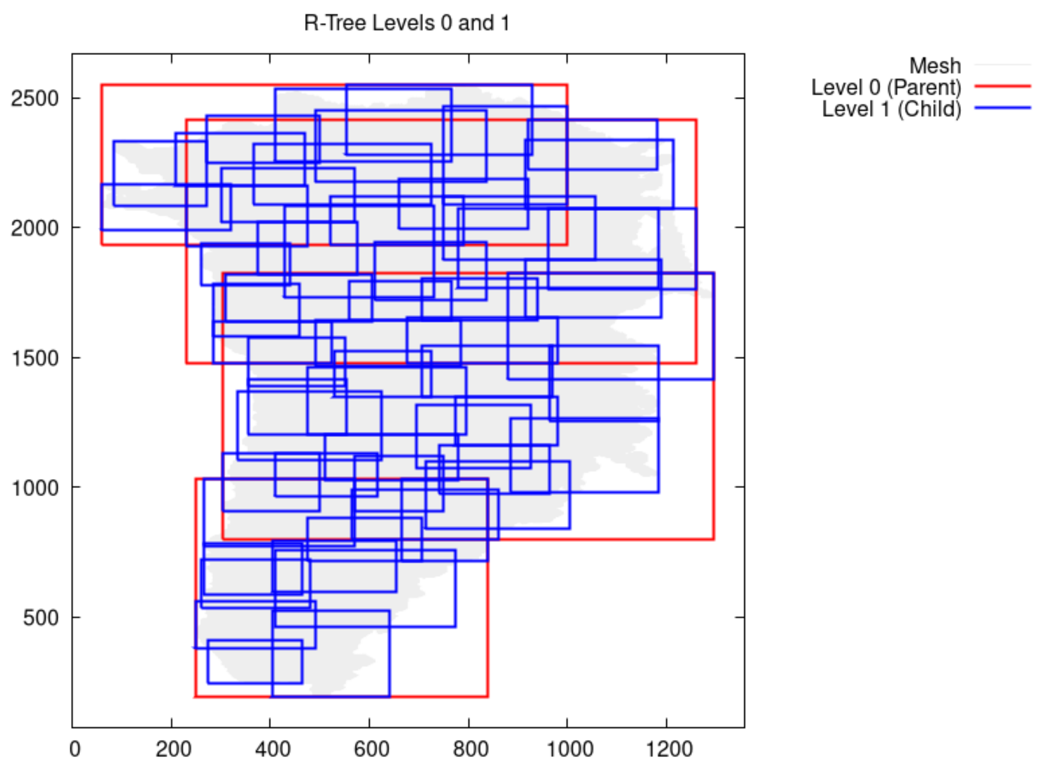

Figure B.3: Greenland: Level 0 (Root, Red) containing Level 1 (Child, Blue).

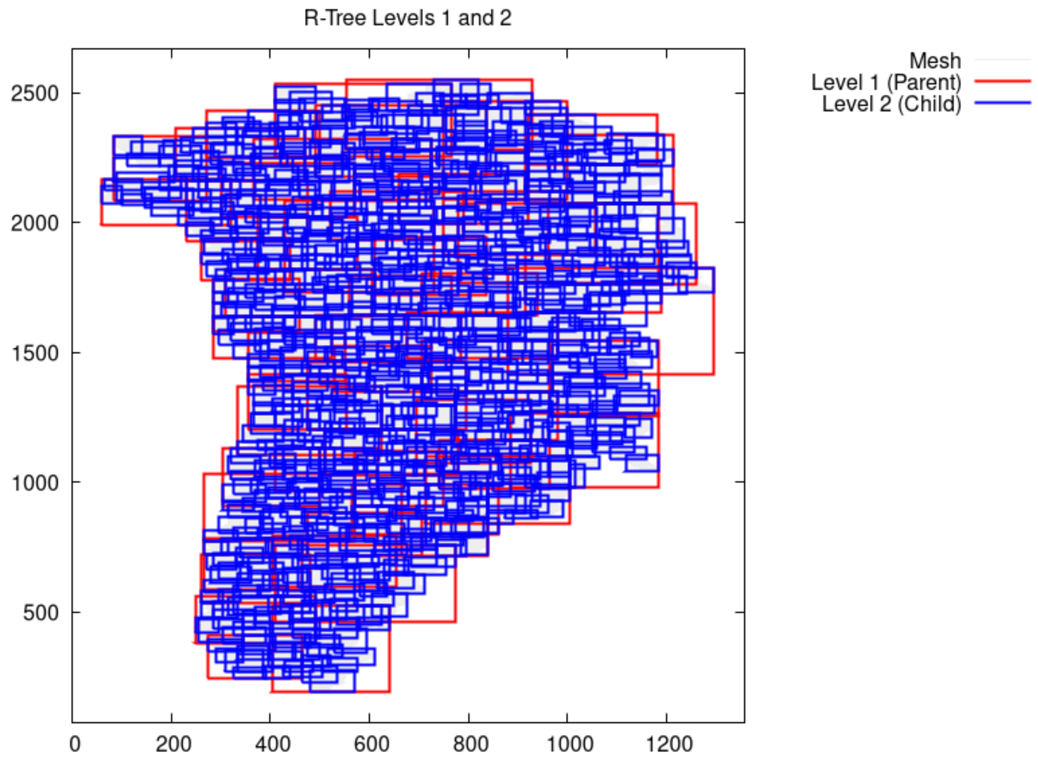

Figure B.4: Greenland: Level 1 (Parent, Red) containing Level 2 (Child, Blue).

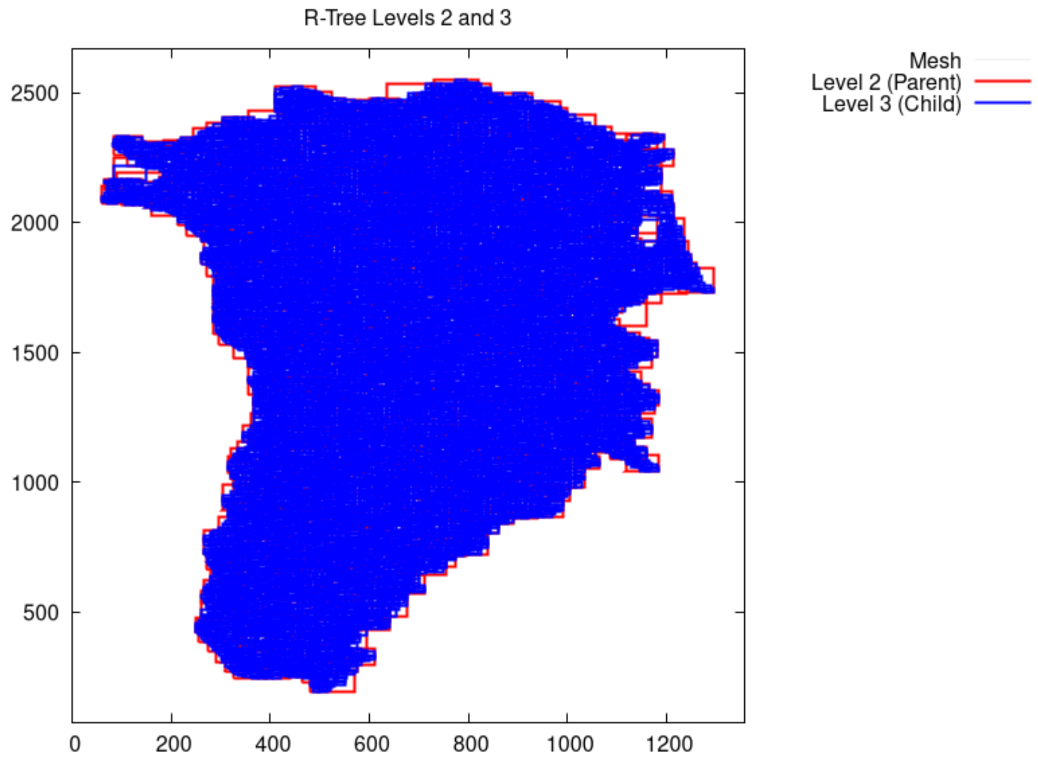

Figure B.5: Greenland: Level 2 (Parent, Red) containing Level 3 (Child, Blue).

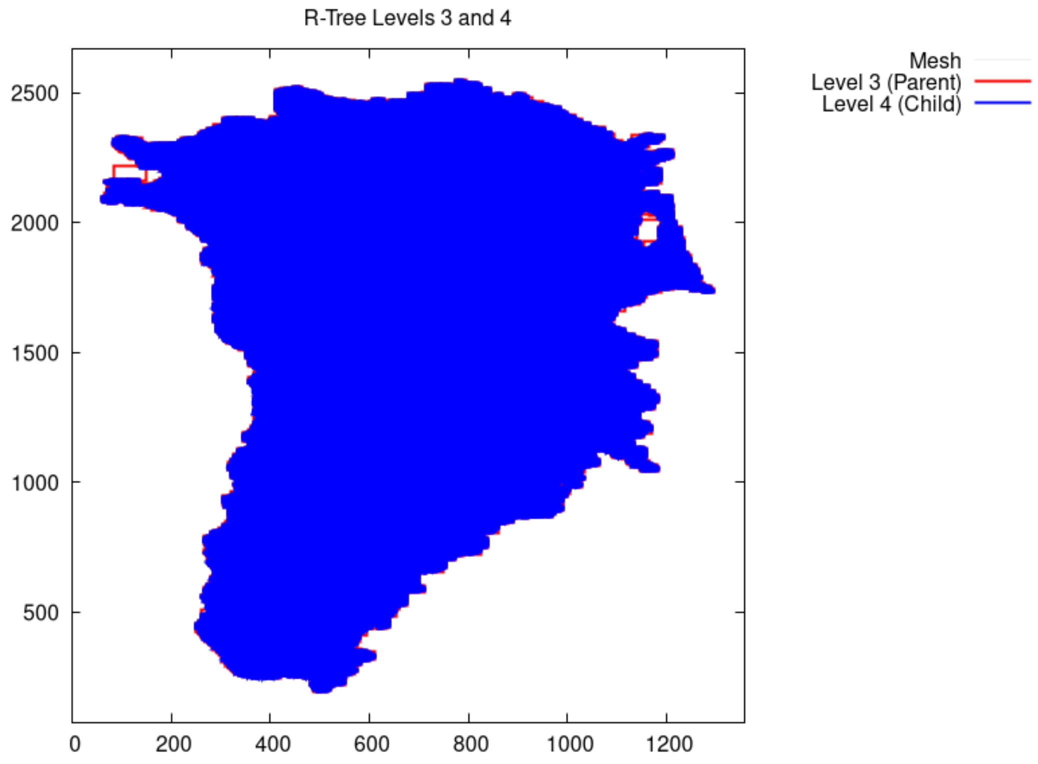

Figure B.6: Greenland: Level 3 (Parent, Red) containing Level 4 (Child, Blue).