3d Spatial Partitioning and Collision Detection using Octrees

November 3, 2024

Nachiket Avachat (2023CSB1106) ,
Asodariya Harsh (2023CSB1103) ,
Som Nainwal (2023CSB1163)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Shubham Thawait

Summary: This project centers on creating an efficient collision detection system for 3D environments using an octree-based spatial partitioning method. By breaking down 3D space into a structured hierarchy, the octree optimizes memory and processing resources, allowing it to handle complex spatial queries and large point datasets with improved speed and accuracy. The system includes functions for inserting and deleting points, conducting nearest neighbor and range queries, and detecting collisions. Its efficacy is validated through a simulated environment where points move dynamically, with real-time collision handling ensuring accurate interactions within the space. This approach demonstrates the scalability and practical utility of octrees for real-time applications such as gaming and simulations, where efficient spatial partitioning is essential.

---

1. Introduction

This project explores efficient 3D spatial partitioning and collision detection using an octree data structure. An **octree** is a hierarchical tree structure commonly used in 3D computer graphics and spatial indexing to partition space into smaller cubic regions, or **octants**. Each node in an octree represents a cubic region and can subdivide into eight smaller octants, recursively partitioning the space to allow for efficient organization, storage, and retrieval of spatial data. This property makes octrees particularly useful in applications like **collision detection, spatial queries, and nearest-neighbor searches**.

Structure and Functionality

The octree used in this project supports various spatial operations, including:

- **Insertion** and **deletion** of points: Adds or removes points within the octree, with nodes subdividing as necessary to accommodate data density.
- **Search** and **range queries**: Locates specific points or retrieves all points within a defined spatial range.
- **Collision detection**: Checks if objects or points occupy overlapping space, facilitating quick collision detection by limiting checks to relevant octants.
- **Nearest neighbor search**: Efficiently finds the closest point to a given target, leveraging spatial partitioning for rapid querying.

Game Simulation

In the simulation component of this project, the octree's functionality is tested through an interactive 3D game. The user controls a point in 3D space, moving it by pressing "w, a, s, d, e, f" keys to traverse along different axes. Each movement updates the octree for collision detection, nearest neighbor, and boundary checks. This movement loop is represented in the code by a continuous `while` loop, where each iteration simulates one frame

of the game. The loop processes the user's input, updates the position of the point, and performs relevant spatial queries, providing a real-time response that mimics 3D navigation within a bounded space.

Project Applications

This project has practical applications in **3D graphics, game development, robotics, and physics simulations**, where efficient spatial partitioning and collision detection are essential for performance and interactivity. The octree's structure reduces the computational load, allowing complex spatial operations to be performed quickly and effectively, which is demonstrated through the game simulation's smooth traversal and response. Overall, this project showcases the versatility of octrees in managing spatial data and highlights their potential in various real-world applications.

2. Equations

This section details the mathematical foundations and formulas used in the octree structure, focusing on spatial relationships, distances, and boundaries.
1. **Octant Determination**: For a given point $P(x, y, z)$ relative to the center of an octree node $C(cx, cy, cz)$, we determine the octant by:

$$\text{octant} = (\text{if } x \geq cx \text{ then } 4 : 0) + (\text{if } y \geq cy \text{ then } 2 : 0) + (\text{if } z \geq cz \text{ then } 1 : 0)$$

2. **Distance to Cube Boundaries**: The squared distance from a point P to the nearest boundary of a cube defined by minimum min and maximum max coordinates:

$$\text{dist\_squared} = (\max(\min_x - x, 0, x - \max_x))^2 + (\max(\min_y - y, 0, y - \max_y))^2 + (\max(\min_z - z, 0, z - \max_z))^2$$

3. **Nearest Neighbor Distance Calculation**: For finding the nearest neighbor, the Euclidean distance squared is computed to compare distances:

$$\text{distance}^2 = (p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2$$

4. **Collision Detection within a Bounding Box**: To detect collisions, a range query checks if a moving point intersects with any other points within a bounding cube:

$$\text{collision\_box\_min} = (x - \text{box\_size}, y - \text{box\_size}, z - \text{box\_size})$$

$$\text{collision\_box\_max} = (x + \text{box\_size}, y + \text{box\_size}, z + \text{box\_size})$$

3. Figures, Tables and Algorithms

3.1. Figures

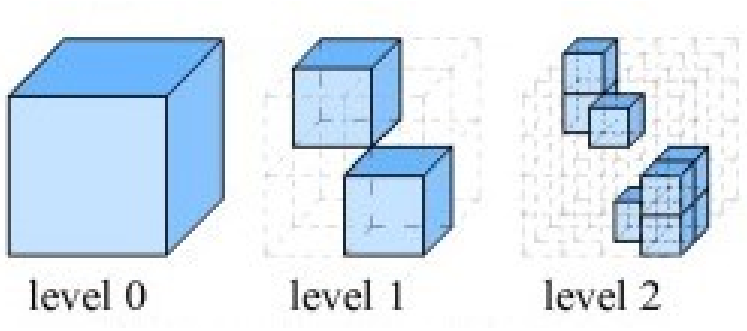

Figure 1: 3d Space Decomposition.

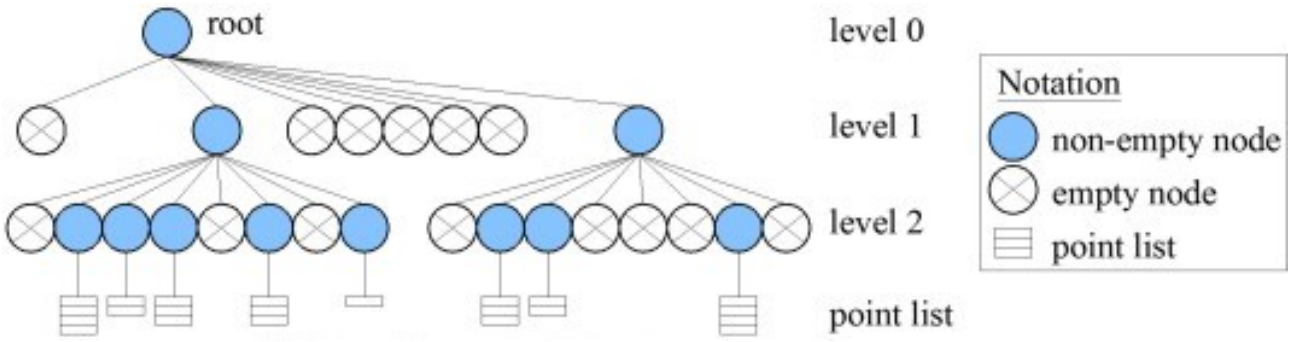

Figure 2: Octree Hierarchial Structure.

3.2. Algorithms used:

Algorithm 1: Insert

1.Introduction:

The function `insertPoint` is designed to insert a point into an octree node, either by adding it to a leaf node or by subdividing the node when it reaches capacity. This process continues recursively, creating finer spatial subdivisions until the point is successfully placed or the maximum depth constraint is reached.

Algorithm 1 Insert Point Operation

Require: Node *node*, Point *point*
Ensure: Initially *root* is kept as node.

```
if node is a leaf then
    if node's point count is less than the maximum allowed then
        Insert point into node
        Print success message
        return true
    else if node's depth is at the maximum then
        Print failure message (max depth reached)
        return false
    else
        Subdivide node into eight children
        Redistribute all points from node into appropriate children
        Clear point count for node
        Determine the octant for the new point
        Insert the point into the appropriate child node
    end if
else
    Determine the octant for the point
    Insert the point into the corresponding child node
end if
```

2.Mathematical Analysis:

Let D be the maximum depth of the octree (denoted as `MAX_DEPTH`), and each node hold up to M points (denoted as `MAX_POINTS`). The following analysis outlines the algorithm's complexity.
Space Complexity:
Each subdivision of a node creates 8 child nodes. The number of nodes $T(D)$ follows:

$$T(D) = 1 + 8 + 8^2 + \cdots + 8^D = \frac{8^{D+1} - 1}{7}$$

If D is constrained, the exponential growth is limited, reducing space requirements. For example, with $D = 6$, the total number of nodes is manageable. Thus, depth constraints mitigate exponential memory usage.
Time Complexity:
The worst-case time complexity for inserting a point occurs when it must be recursively inserted into child nodes down to the maximum depth D. With constant time $O(1)$ per depth level, the insertion time complexity is:

$$O(D) = O(\log N)$$

where N is the number of points, assuming balanced subdivisions.

Algorithm 2: Delete

1.Introduction:

The `deletePoint` function removes a specified point from an octree and optimizes memory by merging nodes when possible. Merging consolidates points from child nodes into the current node when conditions allow.

Algorithm 2 Delete Point Operation

Require: Node *node*, Point *point*
Ensure: Initialize function by keeping *root* as node.

if *node* is null **then**
  return
end if
if *node* is a leaf **then**
  Search for *point* in *node*'s points array
  if *point* is found **then**
    Remove *point* by shifting elements left
    Decrease the point count of *node*
    Print that the point has been deleted
  end if
else
  Determine the octant for the target *point* relative to the *node*'s center
  Recursively call **deletePoint** on the child node in that octant
  After deletion, check if all children are leaf nodes
  Calculate the total number of points across all child nodes
  if all children are leaves and total points $\leq$ MAX_POINTS **then**
    Merge all points from child nodes into the current node's points array
    Free all child nodes and set them to null
    Set the current node as a leaf
    Print that children were merged
  end if
end if

2.Mathematical Analysis:

The only time required to delete is for searching which is $O(\log N)$ other operations require $O(1)$ time.
Time Complexity:
$O(\log N)$, where N is the total points.

Algorithm 3: Search

1.Introduction:

The function `searchPoint` is a recursive algorithm that searches for a specific point in an octree. It traverses down the tree, selecting child nodes based on the point's spatial coordinates relative to each node's center. The function either finds the node containing the point or concludes that the point does not exist.

Algorithm 3 Search Point Operation

Require: Node *node*, Point *point*
Ensure: Initialize function by keeping *root* as node.

```
if node is null then
    return null
end if
if node is a leaf then
    for each point in node's points do
        if the point matches the target point's coordinates then
            return node {point found}
        end if
    end for
else
    Determine the octant for the target point relative to the node's center
    Recursively search in the child node corresponding to that octant
end if
return null {point not found}
```

2.Mathematical Analysis:

Let D represent the maximum depth of the octree (denoted as `MAX_DEPTH`) and M the maximum number of points per leaf node. The analysis below discusses the space and time complexity.
Space Complexity:
Each recursive call uses $O(1)$ space. Since the worst-case recursion depth is D, the overall space complexity is: $O(D)$
Time Complexity:
The time complexity depends on the depth D of the tree and the number of points M in each leaf node.

- **Internal Nodes:** Each internal node requires $O(1)$ time to determine the octant.
- **Leaf Nodes:** A linear search over M points is performed at the leaf node, resulting in $O(M)$.

Thus, the total time complexity for the search is:

$$O(D + M)$$

Bounding the Depth:
For a balanced octree, $D \approx \log_8 N$, where N is the total number of points. The complexity can therefore be expressed as:

$$O(\log_8 N + M)$$

Algorithm 4: Range Query

1.Introduction:

The `rangeQuery` function searches within a specified axis-aligned bounding box in an octree, efficiently identifying points in a 3D space using spatial partitioning.

2.Mathematical Analysis:

Let D be the tree depth and M the max points per node.
Depth Bound:

Algorithm 4 Range Query Operation

Require: Node *node*, Bounds *min*, *max*, Integer *count*, File fp for giving output

if *node* is null **then**
  return
end if
{Step 1: Check if the node's bounding box is outside the query range}
if *node*'s $max.x < min.x$ or *node*'s $min.x > max.x$ or
    node's $max.y < min.y$ or *node*'s $min.y > max.y$ or
    node's $max.z < min.z$ or *node*'s $min.z > max.z$ **then**
  return
end if
{Step 2: If the node is a leaf, check each point within it}
if *node* is a leaf **then**
  for each point in *node*'s points array **do**
    if point is within the bounding box defined by *min* and *max* **then**
      Write point to file fp
      Increment *count*
    end if
  end for
  {Step 3: If the node is not a leaf, recursively perform rangeQuery on each child}
else
  for each child node in *node*'s children array **do**
    rangeQuery(child, *min*, *max*, *count*, fp)
  end for
end if

For a balanced octree, $D \approx \log_8 N$, where N is the total number of points.
Time Complexity:
Bounding box checks reduce unnecessary calls. In the worst case, if the query range covers many nodes, complexity is:

$$O(M \times D) = O(M \times \log_8 N)$$

Algorithm 5: Collision Detection

1.Introduction:

We have used collision detection for the simulation of game done in `gameloop`. Thus, we will study what collision detection does in that case.
The `detect_collision` function detects nearby points within a defined region around a moving point using an octree. The function inserts the point, defines a collision box, performs a range query, and checks if multiple points are within the box, indicating a collision.

2.Mathematical Analysis:

The algorithm's efficiency is influenced by the pruning effect provided by the octree structure and the properties of range queries.
Time Complexity:
The primary steps contributing to the time complexity are:

- **Insertion and Deletion:** Inserting and deleting a point in an octree has an average complexity of $O(\log_8 N)$, where N is the number of points in the octree.
- **Range Query for Collision Box:** Using an octree, the range query within a bounding box of side length $2 \times$ box_size can be highly efficient. Pruning nodes outside the query region minimizes checks, leading to sublinear behavior on average. In the worst case, however, querying the entire octree could

Algorithm 5 Collision Detection Operation

Require: Octree *octree*, Moving Point *moving_point*, Box Size *box_size*

Insert the *moving_point* into the octree using **insertPoint**
{Define the collision box boundaries around the moving point}
$min \leftarrow (moving\_point.x - box\_size,\ moving\_point.y - box\_size,\ moving\_point.z - box\_size)$
$max \leftarrow (moving\_point.x + box\_size,\ moving\_point.y + box\_size,\ moving\_point.z + box\_size)$
{Initialize count to keep track of nearby points}
$count \leftarrow 0$
{Query the octree for points within the collision box}
rangeQuery(*octree*, *min*, *max*, *count*)
{Check for collisions}
if $count > 1$ **then**
    Print "Collision detected"
    deletePoint(*octree*, *moving_point*) {Remove the moving point after checking}
    return true {Collision detected}
end if
{Remove the moving point from the octree if no collision is found}
deletePoint(*octree*, *moving_point*)
return false {No collision}

reach $O(M \times \log_8 N)$, where M is the maximum number of points per leaf.

Space Complexity:
The algorithm only requires a constant amount of additional space for storing the bounding box boundaries and a counter variable. Therefore, the space complexity is $O(1)$, assuming the octree itself is pre-built.
[6]

Algorithm 6: Nearest Neighbor Search

1.Introduction:

The `findNearestNeighbor` function identifies the closest point to a target in an octree by limiting the search to relevant regions. Starting with a large initial minimum distance (`minDist`), it updates this distance as closer points are found.

Algorithm 6 Find Nearest Neighbor

Require: Octree *octree*, Target Point *target*
Ensure: Initialize the function by keeping *root* as node.
Initialize *minDist* to a large value (e.g., infinity)
Initialize *nearest* to null or an empty point structure
$found \leftarrow$ **findNearestNeighborHelper**(*octree.root*, *target*, *nearest*, *minDist*)
if *found* **then**
    $minDist \leftarrow \sqrt{minDist}$ {Convert squared distance to actual distance}
end if
return *found*, *nearest*, *minDist*

2.Mathematical Analysis:

Bounding Box Distance Calculation:
Given a point $P = (x, y, z)$ and a bounding box defined by minimum and maximum coordinates min =

Algorithm 7 Helper Function for Finding Nearest Neighbor in Octree

Require: Node *node*, Target Point *target*, Reference *nearest*, Reference *minDist*
Ensure: Update *nearest* and *minDist* with the closest point found

if *node* is NULL **then**
  return false
end if
{Calculate the squared distance from the target point to the node's bounding box}
$distToCube \leftarrow$ **distanceToCubeSquared**(*target*, *node.min*, *node.max*)
if $distToCube > minDist$ **then**
  return false {Current node's region is farther than the best distance found}
end if
$found \leftarrow$ false
if *node* is a leaf **then**
  for each point in *node* **do**
    {Check each point in the leaf node} *dist* $\leftarrow$ **squaredDistance**(*target*, *point*) **if** $dist < minDist$ and $dist \neq 0$ **then** {Ignore the target point itself} $minDist \leftarrow dist$ $nearest \leftarrow point$ $found \leftarrow$ true
    end if
  end for
else
  {Determine the order to traverse child nodes based on proximity to target}
  childOrder $\leftarrow [0, 1, 2, 3, 4, 5, 6, 7]$
  Sort *childOrder* based on distance from *target* to each child's bounding box
  {Traverse each child node in the sorted order}
  for each child index in *childOrder* **do**
    if *node.children*[*child index*] is not NULL **then**
      $childFound \leftarrow$ **findNearestNeighborHelper**(*node.children*[*child index*], *target*, *nearest*, *minDist*)
      if *childFound* **then**
        $found \leftarrow$ true
      end if
    end if
  end for
end if
return $found$

$(x_{\min}, y_{\min}, z_{\min})$ and $\max = (x_{\max}, y_{\max}, z_{\max})$, the squared distance d^2 between P and the box is:

$$d^2 = \sum_{i \in \{x,y,z\}} \max(0, \min_i - p_i, p_i - \max_i)^2$$

This calculation is efficient and enables early pruning of nodes that cannot contain closer points than the nearest found.

Time Complexity:

The overall time complexity can be analyzed in the following parts:

- **Bounding Box Pruning:** The algorithm efficiently skips nodes whose bounding boxes are further away than the current `minDist`, significantly reducing unnecessary computations.
- **Leaf Node Check:** When a leaf node is reached, each point in the node must be compared to the target. If a leaf contains M points, this leads to $O(M)$ comparisons.
- **Recursive Traversal and Sorting:** Sorting child nodes based on their proximity to the target takes constant time $O(1)$ since there are at most 8 children. The recursive calls result in a depth of $O(\log_8 N)$ in balanced trees.

Thus, the expected average time complexity of the algorithm is:

$$O(M + \log_8 N)$$

where N is the number of points in the queried leaf node and M is the total number of points in the octree.

Space Complexity

The space complexity primarily depends on the structure of the octree. Each recursive call adds to the call stack, leading to a maximum depth of $O(\log_8 N)$. Hence, the space complexity is:

$$O(\log_8 N)$$

due to the depth of recursion.

3.3. Tables:

Summary of Algorithms

Operation	Average Complexity	Description
Insert	$O(\log N)$	Inserts a point in the correct octant
Delete	$O(\log N)$	Deletes a point and merges children if possible
Search	$O(M + \log N)$	Finds a specific point within the octree
Range Query	$O(M \log N)$	Queries points within a specified bounding box
Collision Detection	$O(M \log N)$	Detects collision by checking surrounding points
Nearest Neighbor Search	$O(M + \log N)$	Finds the nearest point to a target

4. Further Useful Suggestions

- **Data Visualization and Testing Framework**: Set up a visualization framework to test the spatial partitioning visually. This can help in fine-tuning the distribution of points and verify the correctness of nearest neighbor and collision detection.

- **Adaptive Thresholds for Subdivision**: Based on empirical data, adjust the threshold for node subdivision adaptively, which can help balance between depth and width of the octree, improving search times.
- **Batch Processing for Insertions/Deletions**: When inserting or deleting large batches of points, consider strategies to process them in bulk rather than individually. This could help minimize the restructuring of nodes and save computation.
- **Cross-Validation with Different Data Distributions**: Test the octree's performance using different data distributions (e.g., clustered, uniform, Gaussian). This helps ensure the structure is robust and performs well across various real-world scenarios.

5. Conclusions

In this project, the octree data structure has proven to be an efficient and scalable solution for 3D spatial partitioning, collision detection, and other spatial queries. The octree enables faster operations compared to naive approaches due to its recursive partitioning and selective traversal of nodes. Additionally, this octree system was successfully applied to a simulated game environment where a point moves through space, with real-time collision detection and spatial queries enhancing the interactivity and accuracy of the simulation. Further optimizations can be implemented to handle even larger datasets and more complex interactions, demonstrating the robustness and flexibility of the octree structure in handling 3D spatial data.

6. Bibliography and citations

The foundational work by Samet [5] provides comprehensive insight into multidimensional and metric data structures, outlining the principles behind efficient spatial indexing that facilitate octree-based approaches in 3D space. Meagher [4] first explored geometric modeling using octree encoding, demonstrating its capacity to represent complex spatial data hierarchically in a compressed form, significantly improving memory efficiency and query speed.
Additionally, Laine and Karras [3] introduced sparse voxel octrees, a highly optimized form that reduces memory usage for sparse datasets while maintaining high computational performance, which is particularly useful in real-time applications, such as gaming and simulations.
Practical implementations of octrees are studied through [1, 2].

References

[1] gamedev. Octree data structure. `https://www.gamedev.net/tutorials/programming/general-and-gameplay-programming/introduction-to-octrees-r3529/`.

[2] GeeksforGeeks. Octree data structure. `https://www.geeksforgeeks.org/octree-insertion-and-searching/`.

[3] S. Laine and T. Karras. Efficient sparse voxel octrees. *ACM Transactions on Graphics (TOG)*, 29(4):1–12, 2010.

[4] D. J. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[5] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2006.

[6] Z. Yao and R. Blaheta. Collision detection and distance calculation based on octree representation. *International Journal of Computer Applications*, 32(10):17–24, 2011.