โครงสร้างข้อมูล Quadtree และการประยุกต์ใช้งาน

ธีรินท์ เพ็ชร์รัตน์ 6430185021
พิพรรธ จงพิพัฒน์ชัย 6431333721
วิวรรษธร ฐิตสิริวิทย์ 6432158421

สารบัญ

1. บทนำ (Introduction)

Quadtree ถูกสร้างขึ้นมาเพื่อใช้เก็บข้อมูลประเภทพิกัดในระบบพิกัดฉากบนระนาบ 2 มิติ เช่น พิกเซลภาพ พิกัดตำแหน่ง (geolocation) พิกัดกลุ่มข้อมูลบนกราฟ ฯลฯ โดยมีเป้าหมายเพื่อเพิ่มความเร็วการค้นหาจุดพิกัดใกล้เคียงที่อยู่ในระยะค้นหา โดยอาศัยความต้องการลดจำนวนข้อมูลที่ต้องตรวจสอบทั้งหมดบนกริด ($O(N^2)$) เหลือเพียงบางข้อมูลที่อยู่รอบ ๆ จุดค้นหาโดยเฉลี่ย ($O(N \log N)$) ทำให้ทรัพยากรที่ใช้ตรวจสอบลดลงมาก เมื่อจำนวนข้อมูลมากขึ้น หลักการเข้าถึงข้อมูลแต่ละตัวอาศัยการแบ่งพื้นที่เป็นพื้นที่ที่เล็กลงที่ข้อมูลควรจะอยู่ในนั้น และทำการหาด้วยการแบ่งซ้ำไปเรื่อย ๆ โดยใช้กระบวนขั้นตอนแบบเวียนเกิด (Recursion) ซึ่งการแบ่งเป็นพื้นที่ที่เล็กลงนั้น ลดปริมาณข้อมูลที่ต้องหาทีละครึ่งเท่า ทุก ๆ ครั้งที่แบ่ง จะทิ้งข้อมูลที่ไม่จำเป็นต้องค้นหาไปทีละครึ่งหนึ่งในทุก ๆ การแบ่ง ทำให้โครงสร้าง Quadtree สามารถใช้ประโยชน์จากประสิทธิภาพการค้นหาแบบนี้ได้

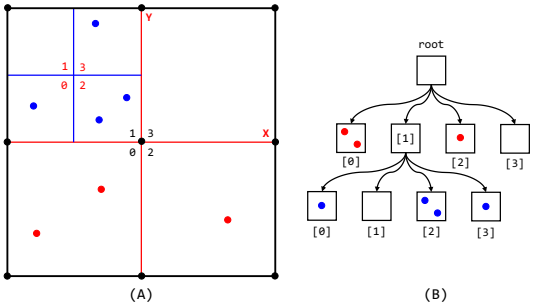
2. โครงสร้างข้อมูลต้นไม้ของพื้นที่แบ่งพาร์ทิชัน (Space Partitioning Tree)

โครงสร้าง พื้นที่ แบ่ง พาร์ ทิชั น เป็น รูป แบบ โครงสร้างข้อมูลที่อาศัยหลักการการแบ่งแยกพื้นที่ย่อยให้เล็กลงแบบเวียนเกิด (Recursive subdivision) โดยใช้โครงสร้างต้นไม้มาอธิบายความสัมพันธ์ของแต่ละพื้นที่ย่อย โดยตัวอย่างที่เห็นภาพได้อย่างง่ายคือการแบ่งพื้นที่แบบ Binary Space Partitioning (BSP) ซึ่งแบ่งพื้นที่ทีละครึ่ง เป็นจำนวนครั้งตามปริมาณข้อมูล ซึ่งคล้ายคลึงกับโครงสร้างข้อมูลแบบ Binary Search Tree (BST) โดยโครงสร้าง Partitioning เหมาะสมแก่การหาตำแหน่งที่เหมาะสมกับการเก็บข้อมูลนั้นด้วยการเปรียบเทียบค่าของข้อมูล และมีความรวดเร็วเป็น log ของจำนวนข้อมูลทั้งหมดในการค้นหา การแทรก และการลบ โดยการสร้าง Tree บนหน่วยความจำนั้น สามารถสร้างขึ้นมาด้วย Node (ปม) ได้ และเชื่อมความสัมพันธ์ (edge/relation) ด้วย pointer ที่ชี้ไปยังตำแหน่งของอีก Node หนึ่ง ดังนั้น การเก็บ tree ไม่มีความจำเป็นต้องใช้ช่องที่ติดกันบนหน่วยความจำ (contiguous memory block) ที่ใช้เวลาตามจำนวนข้อมูลเมื่อต้องย้ายตำแหน่งของข้อมูลเมื่อขยายพื้นที่ (reallocation) แต่จะใช้ dynamic allocation บน heap แทน โดยจองพื้นที่เพียงพอแค่ Node ที่สร้างขึ้นใหม่ โครงสร้างพื้นที่แบ่งพาร์ทิชันนั้นมีอยู่หลายวิธีในการแบ่ง โดยสามารถแบ่งเป็น 2 ประเภทได้ตามแกนการแบ่งดังนี้

1. พื้นที่แบ่งพาร์ทิชันแบบตามแนวแกน ซึ่งแบ่งโดยระนาบหรือเส้นที่ขนานและตั้งฉากกับแกนใดแกนหนึ่งเสมอ ซึ่งเป็นการแบ่งที่เหมาะสมกับการเก็บข้อมูลพิกัด และการแบ่งตามแนวฉาก ซึ่งเป็นรูปแบบการแบ่งที่ใช้ใน Quadtree และ Linear octree (หลักเกณฑ์เดียวกับ Quadtree ในระบบพิกัด 3 มิติ)

2. พื้นที่แบ่งพาร์ทิชันแบบไม่ตามแนวแกน ซึ่งใช้ ไฮเปอร์เพลน (Hyperplane) ซึ่งเป็นระนาบใด ๆ ในพิกัดฉาก n มิติ ซึ่งอธิบายด้วยสมการระนาบ $\mathbf{x} \cdot \mathbf{x_0} = d$ ที่ไม่จำเป็นต้องมีแกนใดแกนหนึ่งขนานหรือตั้งฉากกับพิกัด ซึ่งรูปแบบการแบ่งนี้เหมาะกับการใช้เก็บข้อมูลที่มีการกระจุกตัวกัน หรือสร้างระนาบด้วยกันระหว่างจุด (mesh) ใน Computer Graphics

3. แนวคิดการเก็บข้อมูลด้วยพื้นที่แบ่งพาร์ทิชัน



Class `QuadTreeNode` layout diagram

แนวคิดการเก็บข้อมูลด้วยพื้นที่แบ่งพาร์ทิชัน การเก็บข้อมูลที่สามารถเปรียบเทียบค่าได้ บนเส้นข้อมูลแบบแบบ 1 มิติ สามารถทำได้โดย Binary Partitioning แต่หากเราต้องการเก็บข้อมูลพิกัด 2 มิติ หรือ 3 มิติ ขึ้นไป จะใช้การทำ Binary Partitioning เป็นจำนวนครั้งเท่ากับมิติพร้อม ๆ กัน เช่น พิกัด (x, y) บนระนาบพิกัดฉาก ใช้การแบ่งตามแนวแกนราบ 1 ครั้ง และตามแกนตั้งฉากอีก 1 ครั้ง จะได้ Grid ที่แบ่งออกเป็น 4 ควอดรันต์ (Quadrants) ดังนี้

1. ควอดรันต์ซ้ายล่าง (SW) ให้ index พื้นที่นั้น เป็น 0
2. ควอดรันต์ซ้ายบน (NW) ให้ index พื้นที่นั้น เป็น 1
3. ควอดรันต์ขวาบน (NE) ให้ index พื้นที่นั้น เป็น 2
4. ควอดรันต์ขวาล่าง (SE) ให้ index พื้นที่นั้น เป็น 3

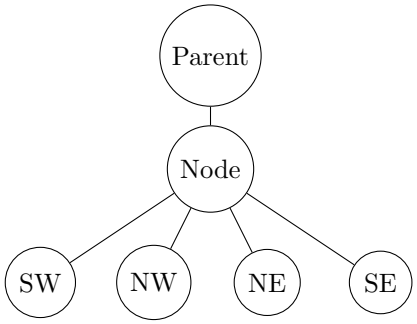
เมื่อปริมาณข้อมูลที่ถูกเก็บในพื้นที่ที่แบ่งนั้น มีมากกว่า 1 พิกัด/จุด เราจะทำการแบ่งพื้นที่ในควอดรันต์นั้นซ้ำแบบเวียนเกิด (recursively subdivide) จนในควอดรันต์ย่อยนั้นมีข้อมูลเหลือเพียง 1 ชุดเท่านั้น และให้ควอดรันต์เดิมที่ควอดรันต์ย่อยอาศัยอยู่ ทำตัวเสมือนไม่มีข้อมูลอยู่ในตัวมัน แต่อยู่ในควอดรันต์ย่อย (children nodes) ของควอดรันต์นั้น ๆ

จากความสัมพันธ์แบบเวียนเกิดของควอดรันต์และควอดรันต์ย่อย เราสามารถใช้โครงสร้างข้อมูลแบบต้นไม้มาช่วยเก็บควอดรันต์บนหน่วยความจำได้ โดยให้แต่ละควอดรันต์เป็น Node และควอดรันต์ย่อยเป็น children ของ Node นั้น และให้ Node นั้นเป็น parent ของทุก ๆ children

4. โครงสร้างของ Node ใน Quadtree

ความเป็น Node ของ Quadtree นี้นั้น จะต้องเป็นไปตามกฎของความเป็นต้นไม้ ที่ว่าแต่ละ Node จะเป็นต้นไม้ และ children ของ Node จะต้องเป็นต้นไม้เช่นกัน

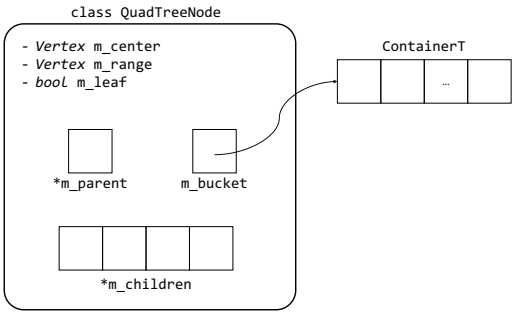
ในโครงสร้าง Quadtree การเก็บข้อมูลนั้นอยู่ในรูปแบบของ 4-ary tree ซึ่งภายในโครงสร้างนั้นอธิบายความสัมพันธ์ของ Node ก่อนหน้า (Parent) และ Node ถัด ๆ ไป (Children) โดย Node นั้นจะเก็บในรูปแบบของ Dynamic object บน heap และมีการจัดการหน่วยความจำตลอดเวลาเพื่อประสิทธิภาพเชิงพื้นที่ และป้องกัน memory leak ซึ่งรูปแบบความสัมพันธ์ของแต่ละ Node ดังภาพ



แผนผังปมราก-ปมลูก

โดยใน Node นั้นสามารถแบ่งประเภทด้วยตัวแปรประเภท boolean ว่าเป็นประเภทใบ (leaf node) หรือไม่ ซึ่ง Node จะทำการเก็บข้อมูลใน bucket เมื่อ Node นั้นเป็นประเภทใบ และ Node ที่ไม่ใช่ใบจะไม่เก็บข้อมูลใด ๆ และจะให้ children nodes ทั้ง 4 node เก็บข้อมูลทั้งหมด

โดยตามปกติ แต่ละ Node ในต้นไม้นั้นจะเก็บข้อมูลได้ 1 ชุด แต่หากจำนวนข้อมูลเพิ่มขึ้นมหาศาล จะทำให้ความลึกของต้นไม้สูงขึ้นตาม และหากต้นไม้นั้นมีความลึกมาก ๆ จะส่งผลให้ การค้นหาข้อมูลนานมาก และกินพื้นที่หน่วยความจำมาก เพราะต้องสร้าง Node เพิ่มในการจุข้อมูลนั้น ๆ จึงแก้ปัญหาด้วยการ chaining ข้อมูลใน Node เหมือนกับใน Hashtable แบบหนึ่ง ที่ให้ Node หนึ่งมีข้อมูลได้หลายชุด และทั้งหมดนี้จัดเก็บอยู่ใน bucket ซึ่งเป็น container รูปแบบหนึ่งที่สามารถ find, insert, และ erase ได้ โดยค่าปกติจะใช้ dynamic array หรือ `std::vector<T>` ในการเก็บข้อมูล แต่หากต้องการประสิทธิภาพที่มากขึ้น อาจใช้ Binary Search Tree หรือ hashtable ในการเก็บแทนได้ แต่มักไม่เป็นที่นิยม เนื่องจาก Quadtree สามารถเก็บข้อมูลใหม่ในแต่ละชั้นเพิ่มขึ้นเป็น 4^n ข้อมูล ซึ่งมหาศาลมาก จึงเลือกให้ขนาดของ bucket น้อย ๆ เพื่อให้เวลาการหาสูงสุดเสมือนคงที่ เทียบกับปริมาณข้อมูลทั้งหมด โดย layout ของ class `QuadTreeNode` เป็นดังภาพ



Class `QuadTreeNode` layout diagram

ดังนั้น ขนาดของ bucket ที่น้อย ๆ จึงเทียบไม่ได้กับความลึกของต้นไม้ Quadtree เราจึงสามารถปรับขนาดของ bucket ให้เหมาะสมที่สุดกับจำนวนข้อมูล และเวลาการค้นหาสูงสุดจะแปรผันตามความลึกข้องต้นไม้ รวมกับขนาดของ bucket หรืออาจประมาณได้ว่าแปรตามความลึกของต้นไม้ได้

โครงสร้างของ Node นั้นจะประกอบไปด้วย 2 สิ่งที่สามารถเข้าถึงได้โดยผู้ใช้งาน คือ พิกัดบนระนาบ 2 มิติ (x, y) ในรูปของ two-vector และ data ซึ่งเป็นข้อมูลที่ผู้ใช้ต้องการเก็บ เช่น ที่ พิกัด $(32, 57)$ ของโลกเสมือนในเกม มีวัตถุ entity เกมตัวหนึ่ง เป็น data ที่อยู่ ณ พิกัดนั้น โดย data ไม่มีผลต่อการค้นหา การเรียง การแทรก หรือการลบเลย แต่ใช้พิกัดระนาบสองมิติเป็นตัวบ่งชี้และตัวเปรียบเทียบ

5. โครงสร้างของ Quadtree

Quadtree นั้นจะประกอบด้วย Node ต่าง ๆ ตามกฎความเป็นต้นไม้ โดยเริ่มที่ปมราก (Root node) ซึ่งเป็น Node บนสุด หรือควอดรันต์นอกสุดของระนาบพิกัดฉากนี้ และ children nodes จะมีขนาดระนาบที่เล็กกว่า parent ของ Node นั้น ๆ เป็น 2 เท่า เหมือนกับการฝังระนาบย่อยใน 4 ควอดรันต์ ภายในระนาบใหญ่ และปมรากนั้นก็จะแบ่งลูก ๆ ออกไปตามความลึกของต้นไม้ ตามจำนวนข้อมูล และ ตามความถี่ของข้อมูลที่อยู่ใกล้ ๆ กัน

โครงสร้างของ Quadtree นั้น ในกรณีที่ข้อมูลมีการกระจายอย่างเท่ากัน ต้นไม้จะมีความใกล้เคียงกับต้นไม้สมดุล (balanced tree) แต่หากข้อมูลพิกัดมีการกระจุกตัวอยู่ใกล้กันมาก ๆ (close proximity) ต้นไม้จะมีกวามคล้ายกับ BST ที่มีลำดับการแทรกข้อมูลแบบเรียงติดกันจากน้อยไปมาก หรือมากไปน้อย ซึ่งเพิ่มความลึกของต้นไม้ฝั่งใดฝั่งหนึ่งมาก ซึ่งอาจเป็นข้อเสียของโครงสร้างข้อมูลประเภทนี้

6. การท่องต้นไม้ Quadtree

การท่องต้นไม้ ใช้หลักการเข้าถึงข้อมูลของ Node โดยอาจทำโดยฟังก์ชันเวียนเกิด หรือไม่ใช้ก็ได้ ซึ่งสามารถทำได้ 2 วิธี คือใช้ Function recursive call chain และอาศัยหลักการ Function call stack มาใช้กับ Node โดยไม่ต้องผ่านฟังก์ชันเลย โดยสามารถทำเป็น Preorder Traversal เพื่อแสดงลำดับชั้น (hierachy) ของต้นไม้ และข้อมูลในต้นไม้ได้ ด้วยการใช้โครงสร้างข้อมูลประเภท stack เก็บ pointer ที่ชี้ไปยัง Node นั้น ๆ จนสุดสาย แล้วจึงไล่ลำดับไปจนครบทุก Node

7. ขั้นตอนวิธีการเข้าถึง การแทรก และการลบข้อมูล

7.1. การเข้าถึงข้อมูล (Data access)

การเข้าถึงหรือการหาข้อมูลใน Quadtree เป็นขั้นพื้นฐานของการแทรกและการลบ ซึ่งใช้วิธีการคล้าย ๆ กัน โดยการเข้าถึงข้อมูลใช้การค้นหาด้วยการท่องต้นไม้ด้วย stack เพื่อประสิทธิภาพในการทำงาน โดยการเข้าถึงในแบบโครงสร้างของผู้เขียนนั้นจะให้ค่าคืนมาเป็น pointer ที่ชี้ไปยังข้อมูลใน Node นั้น ดังนั้นผู้ใช้จึงสามารถแก้ไข หรือเปลี่ยนแปลงข้อมูลได้สะดวก โดยห้ามมิให้แก้ไขพิกัดที่ใช้เป็นโครงสร้างของต้นไม้โดยนัย

7.2. การแทรกข้อมูล (Insertion)

การแทรกข้อมูลใน Quadtree สามารถทำได้โดยการพยายามหา Node (ควอดรันต์) ที่ลึกที่สุด ณ ขณะนั้น แล้วทำการพยายามเพิ่มข้อมูลใน bucket ที่ควอดรันต์นั้นอยู่ หากขนาดของ bucket เกินกว่าที่กำหนด ก็จะทำการดึงข้อมูลทั้งหมดออกมาจาก bucket และทำการแบ่งพื้นที่ย่อยก่อนแล้วจึงใส่ข้อมูลแต่ละข้อมูลในควอดรันต์ที่เหมาะสมตามพิกัดของข้อมูล

7.3. การลบข้อมูล (Removal)

การลบข้อมูลสามารถทำได้โดยการพยายามหาข้อมูลนั้น เมื่อเจอพิกัดในควอดรันต์ที่ข้อมูลนั้นควรจะอยู่ ก็จะทำการลบข้อมูลออกไปจาก bucket จากนั้นทำการตรวจสอบว่าสามารถยุบรวมควอดรันต์ย่อยขึ้นมาอยู่ในควอดรันต์หลักได้หรือไม่ เพื่อคงความสมดุลมากที่สุดที่ทำได้ ให้ไม่เสียประสิทธิภาพด้านความเร็วในการค้นหา และพื้นที่จัดเก็บที่น้อยลง และจะยุบรวมควอดรันต์ก็ต่อเมื่อจำนวนข้อมูลในทุก ๆ ควอดรันต์ย่อยไม่เกินขนาดสูงสุดของ bucket ที่กำหนดไว้

8. แบบโครงสร้างข้อมูล Quadtree ใน C++ (Implementation in C++)

ทาง ผู้ เขียน ได้ จัด ทำ class `Quadtree` ใน ภาษา C+ + ซึ่ง ประกอบ ไป ด้วย ฟังก์ชัน ที่ จำเป็น สำหรับ การ find, insert, และ remove รวม ถึง โครงสร้าง ของ ข้อมูล พื้น ฐาน และ การ แสดง ตำแหน่ง ที่ อยู่ ของ Node ต่าง ๆ ใน ต้นไม้ และ ข้อมูล อีก ด้วย โดย class `Quadtree` นี้ไม่มี iterator (อยู่ ระหว่าง การ พัฒนา) โดย ราย ละเอียด เพิ่ม เติม เกี่ยว กับ class อยู่ ใน repository https://github.com/neil4884/quad_tree/

8.1. Quadtree Constructor

```
explicit QuadTree<T, PairT, ContainerT>
(Vertex center = Vertex{0, 0},
Vertex range = Vertex{1, 1},
unsigned bucket_size = 1,
unsigned depth = 16,
bool sort = false);
```

Quadtree เก็บข้อมูลในรูป `ContainerT` ซึ่งในตัวอย่าง นี้จะเป็น `vector` ของ `pair` ที่มี data และจุดพิกัด (x, y) ของ data นั้นๆ และมี data member ต่างๆดังนี้

- `m_root` เป็น node แรกสุดและบนสุด ซึ่ง `m_root` จะมี แค่ node เดียว
- `m_sort` ใช้ในการจัดเรียงข้อมูล
- `m_pair_comp` ใช้ในการเปรียบเทียบค่าของ pair และมี `m_size` ที่บอกจำนวนของข้อมูลที่ถูกเก็บอยู่ใน Quadtree
- ความลึก และขนาด bucket ที่ใช้เก็บข้อมูลสูงสุดมีค่าเริ่ม ต้นเป็น 16 และ 1 ตามลำดับ

โดยมี parameter เป็น

- `m_center` คือจุดกำเนิดของ Quadtree
- `m_range` คือ จุด พิกัด ขอบเขต ของ Quadtree เช่น `m_range` = `Vertex{1,1}` นั่นคือระยะทางจากจุดกำเนิด ไปในแนวแกน Y ด้านละ 1 หน่วย ด้านบนจะเป็นพิกัด $(0, 1)$ ด้านล่างจะเป็นพิกัด $(0, -1)$ และระยะทางซ้ายขวาจากจุด กำเนิดไปในแนวแกน X ด้านละ 1 หน่วยเช่นกัน โดยด้านซ้าย จะเป็นจุดพิกัด $(-1, 0)$ และด้านขวาเป็นจุดพิกัด $(1, 0)$
- `bucket_size` คือค่าที่กำหนดไว้ให้ใน ควอดรันต์ใด ๆ ที่ กำหนดจำนวนข้อมูลที่สูงที่สุดที่สามารถเก็บได้ในควอดรันต์ นั้น
- `depth` คือ ค่า ความ สูง สูงสุด ของ ต้นไม้ โดย หาก มี แค่ `m_root` จะ มี ค่า ความ สูง เป็น 0 และ ถ้า มี node ลูก ของ `m_root` เพิ่มมา ความสูงของ Quadtree นี้ ก็จะเป็น 1 เป็นต้น

8.2. Insertion

```
std::pair<viterator, bool> insert(
const Vertex &point,
const T &data);
```

Quadtree สามารถ insert ข้อมูลได้โดยการใส่ parameter ไป 2 ตัว นั่นคือ จุดพิกัดและข้อมูลที่ต้องการจะ insert

8.3. Updating

```
bool update(const Vertex &point,
const T &data);
```

หากต้องการแก้ไขข้อมูลที่พิกัดใดๆ เราสามารถเรียกใช้ ฟังก์ชัน update โดยระบุ parameter `Vertex point` (พิกัด (x, y)) และ T data (ข้อมูล) ซึ่งจะคืนค่าผลลัพธ์การ update ว่าสำเร็จหรือไม่ หาก point ไม่อยู่ในระยะ `m_root` จะคืนค่า `false` หากอยู่ในระยะก็จะทำแก้ไขข้อมูลที่พิกัดนั้น หากที่ พิกัดนั้นไม่มีข้อมูลมาก่อน ก็จะทำการ insert ข้อมูลเข้าไปที่ พิกัดนั้น ๆ จากนั้นจะคืนค่า `true`

8.4. Find

```
bool contains(const Vertex &point);
```

ฟังก์ชัน นี้ เปรียบ เสมือน find function ของ data structure ตัวอื่น ๆ หากต้องการทราบว่าจุดพิกัดนั้นมีอยู่ใน Quadtree นั้นหรือไม่ function นี้ จะคืนค่า `true` เมื่อพบว่า มีจุดพิกัดนั้นอยู่จริง แต่จะคืน `false` หากไม่พบว่ามีจุดพิกัด นั้นอยู่ใน Quad Tree

8.5. Removal

```
bool remove(const Vertex &point);
```

หาก ต้องการ ลบ พิกัด ใดๆ ออก ให้ใช้ ฟังก์ชัน `remove` ที่มี parameter คือ จุด พิกัด ของ ตัว ที่ เรา ต้องการ จะ ลบ ออก จาก Quadtree และจะคืนค่า `true` เมื่อพบว่ามีจุดนั้นและทำการ ลบออกไปแล้ว แต่จะคืน `false` เมื่อไม่พบพิกัดนั้นในรูปของ

leaf node ในชั้นของ children ที่ลึกที่สุดใน Quadtree หรือไม่พบพิกัดนั้น ๆ

การ remove มีประเด็นเพิ่มเติมนั่นคือ เมื่อทำการลบพิกัดใดๆแล้วพบว่าจำนวนข้อมูลทั้งหมดใน sibling nodes ทั้งหมด ไม่เกิน `max_bucket_size` แล้ว จะทำการยุบ children node ขึ้นไปหา parent node ด้วย

8.6. ฟังก์ชันอื่น ๆ

```
std::vector<std::pair<Vertex, T>>
extract_all();
```

คืนข้อมูลทั้งหมดที่อยู่ใน Quadtree โดยคืนค่าเป็น `vector` ของ `pair` ของพิกัดและข้อมูล

```
data_in_region(const Vertex &bottom_left,
const Vertex &top_right);
```

ฟังก์ชันนี้จะคืนข้อมูลทั้งหมดที่อยู่ในช่วงพื้นที่ที่เราสนใจ ระบุพิกัดที่สนใจด้วยพิกัดที่อยู่ล่างซ้ายสุดของช่วงพื้นที่ (`bottom_left`) และพิกัดที่อยู่ขวาบนสุดของช่วงพื้นที่ (`top_right`) โดยจะคืนข้อมูลมาในรูปแบบ `vector` ของ `pair` ของพิกัดและข้อมูล

```
void print_preorder();
```

ฟังก์ชันนี้จะทำการพิมพ์ข้อมูลและ address บนหน่วยความจำของ node, children nodes, parent node และข้อมูลออกมาในลำดับแบบ preorder

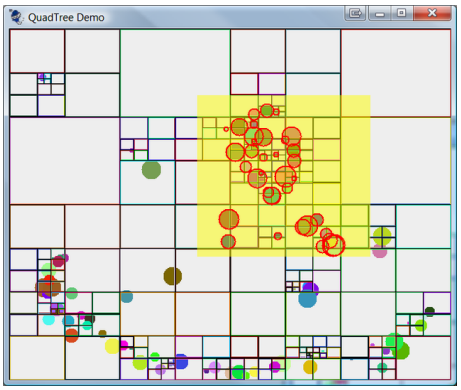
```
void print_data();
```

ฟังก์ชันนี้จะทำการสั่งพิมพ์ข้อมูลทั้งหมดออกมาในรูปแบบ “`<*> Point [(x, y)] has data = [data]`” ตามลำดับ preorder

9. การประยุกต์ใช้งาน Quadtree

9.1. การตรวจสอบการชนบนระนาบ (Collision Detection)

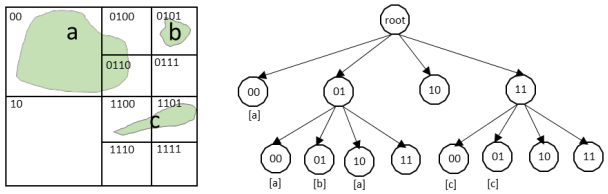
ในการตรวจสอบการชนบนระนาบสองมิตินั้น โดยปกติแล้วเราจะต้องทำการเช็คระยะห่างระหว่างคู่วัตถุทุกคู่เพื่อตรวจสอบว่าวัตถุสองชิ้นนั้นชนกันหรือยัง ซึ่งหากมีวัตถุไม่มาก การตรวจสอบระยะห่างของวัตถุทุกคู่นั้น ก็ยังไม่ต้องการการคำนวนที่เยอะ แต่หากวัตถุมีมากขึ้น การที่เราต้องจับตาดูวัตถุทุกคู่ที่มีนั้นเป็นวิธีที่ค่อนข้างเกินความจำเป็นไปมาก Quadtree สามารถช่วยให้เราโฟกัสกับคู่วัตถุได้เป็นกลุ่มเล็กมากขึ้น กล่าวคือ หากวัตถุใด ๆ อยู่ในลูกเดียวกันของ Quadtree ก็จะทำการตรวจสอบวัตถุทั้งหมดในลูกนั้น ๆ แทนดังภาพ



Collision Detection Illustration [1]

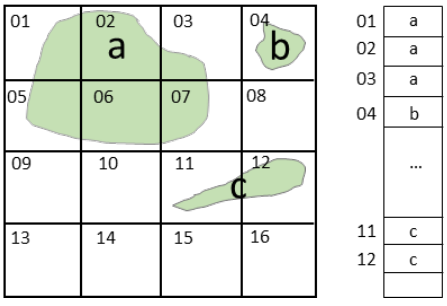
9.2. การทดสอบข้อมูลเชิงพื้นที่ (Spatial Index & Query)

เนื่องจาก Quadtree นั้นมีการเก็บข้อมูลในรูปแบบพิกัดและข้อมูลควบคู่กัน อีกทั้งยังมีการแบ่งรากหรือลูกแต่ละลูกเป็นควอดรันต์



Collision Detection Illustration [5]

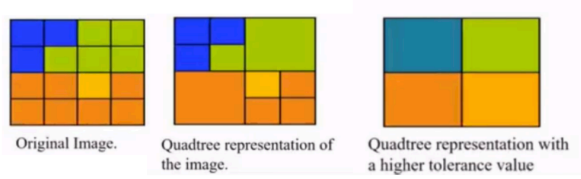
เพื่อการค้นหาข้อมูลที่ง่าย เราจึงสามารถใช้ประโยชน์จากวิธีการเก็บ และเข้าถึงข้อมูลเหล่านี้ในการวิเคราะห์เชิงสองมิติได้โดยตรง เช่น การหาพิกัดที่ใกล้กับอีกพิกัดมากที่สุด หรือจะค้นหาข้อมูลในขอบเขตที่สนใจก็ทำได้ง่ายเช่นกัน



Collision Detection Illustration [5]

9.3. คอมพิวเตอร์กราฟฟิก (Computer Graphics)

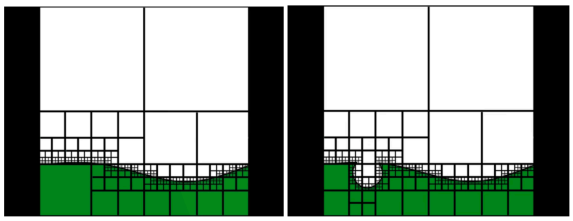
Quadtree มีประโยชน์ในการนำเสนอข้อมูลในรูปแบบ 2 มิติ เช่น รูปภาพ Quadtree สามารถใช้กับข้อมูลได้หลายแบบ ยกตัวอย่างเช่น ข้อมูลที่มีสี เนื่องจากข้อมูลแบบ Quadtree มีหน้าตาเป็น Grid หลาย ๆ ช่องมารวมกัน เนื่องจากว่าในภาพ ๆ หนึ่ง ช่อง pixel ที่อยู่ติดกัน มักจะมีสีรอบ ๆ ใกล้เคียงกัน เช่นว่าเป็นสีเขียว pixels รอบ ๆ มีความเป็นไปได้ที่จะมีสีเขียวมากกว่าสีอื่น ดังนั้น เราจึงไม่จำเป็นต้องแบ่งย่อยทุก grid ให้มีขนาดเล็กเท่ากันหมด แต่สามารถรวม grid ที่มีสีใกล้เคียงกันเป็น grid ที่ใหญ่ขึ้น grid หนึ่งได้ เพื่อประหยัด memory ดังรูป



Computer Graphics: Image compression by neighboring pixels

ซึ่งคุณภาพของภาพที่เราทำการย่อขนาดที่ได้นั้นก็ขึ้นอยู่กับ algorithm การรวม grid เช่นหากสั่งให้ย่อขนาดภาพโดยการรวมทุก grid เข้าด้วยกันครั้งเดียว คุณภาพภาพที่ได้ก็อาจจะแย่ลงอย่างเห็นได้ชัดดังตัวอย่าง เนื่องจากเกิดการรวม grid ที่มีสีที่ต่างกันมากเกินไป ดังนั้น หากต้องการคงคุณภาพของภาพ ควรเขียน algorithm ที่แยกว่า grid ใดมีสีต่างกันมากเกินไป จะได้คัดแยกและไม่รวม grid นั้น ๆ

อีก ตัวอย่าง หนึ่ง ของ การ ประยุกต์ ใช้ Quadtree กับ Computer Graphics คือ การ ใช้ Quadtree ช่วย ใน การประมวลของการเปลี่ยนแปลงภาพต่าง ๆ เนื่องจากลักษณะของช่องแต่ละช่องของ Quadtree เป็นสี่เหลี่ยมจตุรัส ซึ่งง่ายต่อการประมวลผลรูปภาพ เช่น ในกรณีที่อยากให้รูปเนินด้านล่างหายไปบางส่วนเนื่องจากการระเบิด



Computer Graphics: Element differences

Quadtree ที่มีความถี่มากจะเปรียบเหมือนเส้นต่าง ๆ ของรูปภาพ เราสามารถใช้ Quadtree คำนวนระยะระเบิดและเปลี่ยนแปลงลูกแต่ละลูกให้กลายเป็นภาพภาพใหม่ได้อย่างรวดเร็ว

10. บรรณานุกรม

References

[1] M. Coyle. *A Simple Quadtree Implementation in C*. 2008. URL: https://www.codeproject.com/Articles/30535/A-Simple-QuadTree-Implementation-in-C.

[2] S. Har-Peled. “Quadtrees - Hierachical Grids.” In: *Geometric Approximation Algorithms*. 2016.

[3] Pierre. *Quadtree and Collision Detection*. 2019. URL: https://pvigier.github.io/2019/08/04/quadtree-collision-detection.html.

[4] J. Savant. *Quadtree*. URL: https://www.jordansavant.com/book/algorithms/quadtree.md.

[5] UCGIS. *DM-66 - Spatial Indexing*. URL: https://gistbok.ucgis.org/bok-topics/spatial-indexing.