A Concurrent Physics Engine

Nitish Gupta
College of Engineering and Computer Science
University of Central Florida
Orlando, FL 32816
Email: nitesh4146@gmail.com

Sumit Bhattacharya
College of Engineering and Computer Science
University of Central Florida
Orlando, FL 32816
Email: er.sumitbhattacharya@gmail.com

Tyler Truong
College of Engineering and Computer Science
University of Central Florida
Orlando, FL 32816
Email: graxrabble@gmail.com

***Abstract*—We will compare the performance of different spatial data structures for concurrent broadphase collision detection. This includes Lists, Hash tables, and Quad Trees.**

I. Importance of Work

The Broadphase Collision Detection is an important step within a physics engine which greatly impacts the performance of a physics engine. It is also the hardest part of the physics engine to parallelize. Kinematics and Collision Resolution is embarrassingly parallel: solved by dividing an array into sections and then giving each section to a thread. Brute Force Broadphase Collision detection has a runtime complexity of $O(n^2)$. Ignoring these data structures will result in a performance penalty even if all other parts of the physics engine are parallelized.

II. Technical Overview

We want to measure the performance of the different data structures in the context of a real time physics engine. This physics engine will move circles with random velocities around a screen. When a circle collides with the edge of the screen or another circle, it will bounce off and travel in another direction. The physics engine will divide time into equal slices called frames. The simulation will run at 60 FPS. Each frame will do the following in this order: Kinematics, Collision Detection, Collision Response. Kinematics requires add() ,remove() and update() operations as objects move around the world. Collision Detection requires a query() operation to find objects that are near each other.

Thread pooling is the basis for concurrency within the physics engine. All kinematics, collision detection and collision resolution will be divided into tasks and then submitted to the Thread pool. The threads will then take a task and execute it.

Spatial Data Structures are used by the physics engine to speed up the most intensive step, collision detection. Kinematics will update the objects position and will call data structures update() operation so objects which has moved can be reindexed. The collision detection will call the data structures query() operation to get a list of possible collisions. The collisions will be verified to produce a list of all collisions. These collisions will be passed to Dynamics which will update the position and velocities of the objects.

One of the data structures we propose is a concurrent grid. Objects will be indexed into a cell within the grid. Each cell uses a list to store references to objects that are contained within the cell. Objects that border multiple cells will be stored as a reference in multiple cells. Each object will have a corresponding reference object to keep track of its current references. The reference object is required by the update() and remove() method to help it delete its old references as it moves around within the world. References are lazily deleted because its preferable to have false positives over false negatives. False positives results in an object getting collision tested even if its not colliding which will slow down the simulation. False negatives results in missing collisions which can result in objects passing through each other. The query() operation should be wait free since it is read only and it returns objects that have not been logically deleted.

III. Technical Detail

Concurrent Grid is composed as an array of lists. Each list represents a cell on the grid. The lists are concurrent array based queues. The queue update(), add(), and remove() operations should only require quiescent consistency with operations taking place once the Kinematics stage of the simulation is complete. Below is pseudocode for the queue:

```
Class cell {
   Class Ref {
      Int object_id;
      // bitfield of the reference's status
      // 00 - free
      // 01 - locked
      // 10 - logical deleted
      // can be improved with bit stealing.
      int status;
   };


Ref array[];

Bool add(object_id) {
   For (auto &ref : array) {
     Test = ref.status.CAS(01)
       Switch (test) {
       Case 0:
          Ref.object_id = object_id;
          ref.status.set(00);
          Return true;
```

```
            Break;
        Case 1:
            Continue;
            break;
        Case 2:
            Continue;
            break;
        }
        Return false;
    };

Void remove(object_id) {
    For (auto &ref : array) {
        Test = ref.status.CAS()
        Switch (test) {
        Case 0:
            continue;
            Break;
        Case 1:
            ref.status.set(10);
            Return;
            break;
        Case 2:
            continue;
            break;
        }
    };
};
```

IV. Related Work

There are few works based on concurrent spatial data structures. Most of the literature on spatial data structures was written before lock free programming. The exception is the QuadBoost QuadTree paper which was published in June of 2016, 4 months ago. The paper starts by mentioning a basic CAS QuadTree implementation. Because a quadtree has four children, all of their pointers cannot be changed atomically. This is solved with a flag which locks the node. Another problem mentioned in the paper is lazy deletion. Lazy deletion causes the tree to grow too large; objects will move around the world, forcing the quadtree to add and delete many nodes. When a node has no more objects, it becomes empty. Those empty nodes which are lazy deleted consumes lots of memory. One solution the authors found was to use state transitions to keep track of how many empty nodes are on a branch and then force a compress operation on that branch when its full of empty nodes. The compress operation will pull objects up to parent instead of leaving leaving them in the chain of empty nodes. This will allow the tree to free up old memory.

V. Appendix

A. Physics Engine

Physics Engine is a class or library which handles motions and collisions of a set of objects. Motion and collision are usually handled with Kinematics and Dynamics equations. Videogames are an example of programs that use physics engines. The video game will tell the engine to keep track of all players and projectiles. The engine will report to the video game when a projectile hits a player or when a player walks into a wall. This allows the programmer to focus on other tasks like Graphics or AI.

1) Collision Detection: Collision Detection is the process of detecting collisions between objects. Imagine two circles traveling towards each other in a head on collision. The physics engine must detect the exact time t when the two circles collide and then their new velocities after the collision. Physics engine literature commonly splits collision detection into two steps: narrowphase and broadphase.

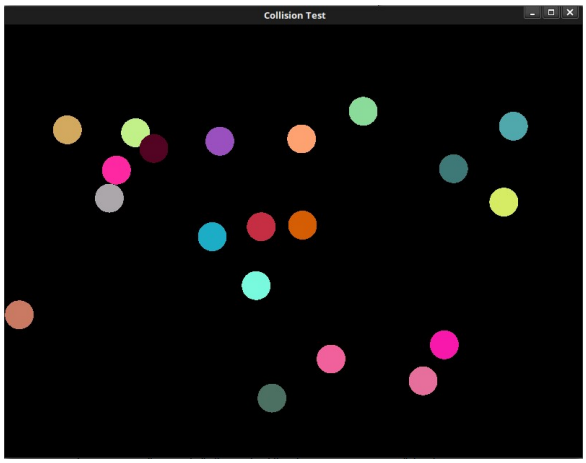

Fig. 1. Box2D implementation

2) Narrowphase: Narrowphase focuses on detecting the TOI of a collision between two objects. The two objects are then backtracked to a previous position so they stop intersecting with each other. A new velocity or trajectory is then computed using Collision Resolution.

3) Brute Force: Brute Force Collision Detection has a runtime complexity of $O(n^2)$. This is because the set of all collisions is a permutation of every pair of N objects.

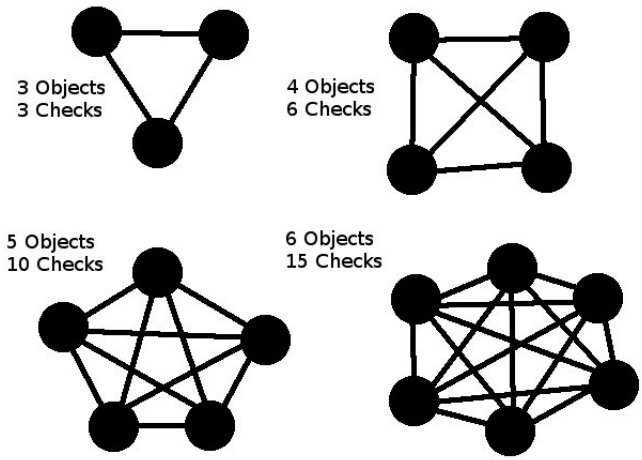

Fig. 2. Brute Force Collision Detection

This runtime is not noticeable with 100 objects but is very noticeable with 1000 objects. Even though the number of jects

increased by 10 folds, the runtime complexity increased by 100 folds.

4) Broadphase: Broadphase Collision Detection is used to accelerate Collision Detection by sorting objects into spatial data structures: Lists, Hash tables, and Quad Trees. Objects are sorted by their position within the world. All of the above data structures take advantage of the fact that only objects that are nearest to each other need to be checked for collision.

B. Lists

Lists are the simplest spatial data structures. Imagine two objects on a number line. The numbers 2 and 4 are closer to each other than 2 and 24, so the numbers 2 and 4 are more likely to collide with each other compared to 2 and 24. Lists are used by the SAP or Sweep And Prune algorithm which is based on the converse of the SAT or Separating Axis Theorem. SAT states that objects can only collide if they intersect on all axises. The converse is that objects cannot collide if they do not intersect on one of many axis. This number line will be referred to as the chosen axis. SAP works by checking for collisions between pairs of objects that are near each other on the chosen axis; this is the Sweep step. SAP will check the proximity of objects near a query point on the chosen axis. If an object does not intersect with the query point then the object will not collide; this is the Prune step. The remaining pairs that were not pruned could collide and will be passed to either another layer of SAP on a different axis or to the narrowphase collision detection.

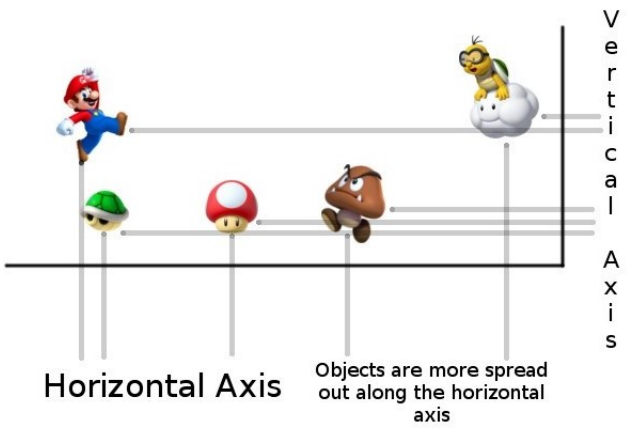

Fig. 3. Sweep and Prune: Axis Comparison

- Advantages of Lists:
  Lists are the simplest data structures to implement. This also means they are the easiest data structure to parallelize. The data structure will focus on implementing the swap() operation over the add() and delete() operations. Imagine a 2d world with two balls named A and B. A is located on the left while B is located on the right. These two balls are moving towards each other along the horizontal axis which is also the chosen axis for SAP. Note that it is possible for A and B to not collide if they have different heights within the 2d world. By not colliding they will pass through each other and swap positions along the chosen axis. Now, it is B that is located on the left while A is located on the right. Instead of deleting nodes and adding nodes to the List, they can be swapped. The swap operation() avoids garbage collection of the nodes.

- Disadvantages of Lists:
  They check for collisions on a single axis. The algorithm will break down if the objects do not spread out along the chosen axis. Imagine Super Mario World which has Mario and all of the enemies on running along platforms at different heights. The SAP will fail if the vertical axis is chosen because most of the objects will cluster along the same height. A better axis would be the horizontal axis because all of the objects on the screen will be spread out more. One solution is to calculate the standard deviation along multiple axis and choose the one with the largest deviation.

C. Grids

Grid data structure: imagine one airplane in New York and another airplane in California. The two airplanes are so far apart that checking for collision is not necessary. The 50 states within America act as the grid. Airplanes can only collide with each other in two cases: both planes are within the same state and both planes are near the borders between neighboring states. Grids are normally composed of square cells for ease of implementation. The Grid data structure works by mapping all objects into cells. This speeds up collision detection because each cell divides Broadphase Collision Detection into series of small Brute Force Collision Detections.

Fig. 4. Grid Analogy

- Advantages of Grids:
  Grids are the most efficient data structure for sparse,

uniformly distributed, and homogeneous objects. Imagine 100 circles of the same size spaced in a lattice pattern across the world where each circle has its own cell. The Query() operation works by returning all other objects within the same and neighboring cells. The add() and remove() operation can be implemented as a lookup table of lists where each cell is represented by a list. The list data structure can easily be parallelized by lock free techniques. And the lookup table can distribute threads across the data structure, reducing contention.

- Disadvantages of Grids:
  Grids degrade when objects are not sparse, uniform and homogenous. The worse case is where all objects are clustered into same cell. This Grid will degrade into Brute Force collision detection. When an object is larger than a single cell of the Grid, the Grid must store that object into multiple cells. This means that the grid must perform multiple add() and remove() operations for every cell that contains the large object. The Grid must also handle objects that are on the border between multiple cells. There are two solutions: store the object into multiple cells similar to the large object problem or store the object in one cell and the query() operation will check neighboring cells.

D. Quadtrees

Quadtrees are similar to the grid based data structures with one added feature. They recursively subdivide the space of all objects into quadrants. This means that every node in the tree will have four children, one for each quadrant. The tree gives this spatial data structure a runtime complexity of $O(\log n)$ because each traversal in the tree will prune $3/4^{th}$ of the possible collisions.

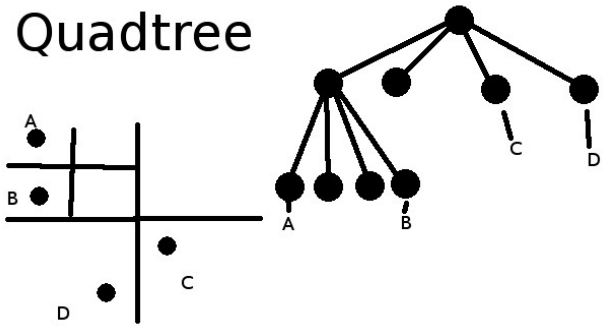

Fig. 5. A Quadtree

- Advantages of Quadtrees:
  Quadtrees can recursively divide space into smaller chunks. They can efficiently handle objects of different sizes. Grids store large objects in multiple cells while Quadtrees can subdivide cells until they fit around the object. Quadtrees are less vulnerable than Grids to objects clustering near each other. Where the Grid degenerated into Brute Force Collision Detection, the Quadtree can subdivide cells.

- Disadvantages of Quadtrees:
  Quadtrees are more complicated to implement because they involve creating and deleting nodes. This means that new problems such as memory management, garbage collection and ABA must be considered. The Quadtree will constantly change shape as objects move around in the world. Imagine a racing game where cars move around a track. The track will be subdivided into quadrants which are recursively subdivided until fitting the nodes fit the cars. When the cars move from the top right corner to the top left corner, all of the tree structure must be deleted from the top right quadrant and recreated on the top right quadrant.

E. Collision Resolution

Collision Resolution is the process of calculating an objects new velocity after a collision. The programmer has full artistic freedom to decide how objects should behave. An example of a parameter is elasticity or the bounciness of an object: a rubber ball should bounce of the ground while a sandbag will stick to the ground. Another parameter is mass: lighter objects should experience a greater change in velocity when bouncing off a heavier object. Resolution is not a focus of this paper, so a simple reflection is used to compute the new velocities of objects.

F. Thread Pooling

Thread Pooling is a technique which creates a set number of threads then delegating tasks to the threads. Initializing and Destroying threads is expensive which is why a pool of threads are kept alive. It works by dividing computation into a unit called a task. A task can be checking if a pair of objects are colliding. A FIFO concurrent queue is used to store the tasks. Available threads will fetch the tasks from the queue then execute each task. This provides load balancing between the threads.

REFERENCES

[1] K. Zhou, G. Tan and W. Zhou, *Quadboost: A Scalable Concurrent Quadtree*.
[2] M. Kornacker and D. Banks, *High-Concurrency Locking in R-Trees*.
[3] E. Hastings, J. Mesit and R. Guha, *Optimization of Large-Scale, Real-Time Simulations by Spatial Hashing*.
[4] Y. Serpa and M. Rodrigues, *Parallelizing Broad Phase Collision Detection for Animation in Games: A Performance Comparison of CPU and GPU Algorithms*.
[5] D. Tracy, S. Buss and B. Woods, *Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal*.
[6] Joshua Shagam, *Dynamic Spatial Partitioning for Real-Time Visibility Determination*.
[7] D. Coming and O. Staadt, *Kinetic Sweep and Prune for Collision Detection*.