Physics Engine

Technical Document

by Oleg Loshkin

0. Foreword:

Starting this project I have set myself a few additional goals for this project following my failure of the procedural generation one.

With the approach of the deadline I realise that my first goal of bettering my time management for complex projects, while partially achieved, will not be sufficient. I have therefore set my focus on my second personal goal of managing the resulting situation in a more professionnal manner.

The official goals for this project are:

1. Implement a 2D physics engine with the following features into an existing game engine:
    - Believable simulation of forces interaction with rigidbodies of different types.
    - Believable collision resolution between simple shapes.
    - Collision checking optimisation in the form of a QuadTree.
    - Rotation of colliders and appropriate believable physical simulation of the phenomena that come with it.
    - Friction and bounciness of rigidbodies.

2. Acquire experience in working with an in-house game engine written in C++ with embedded Python scripting support.

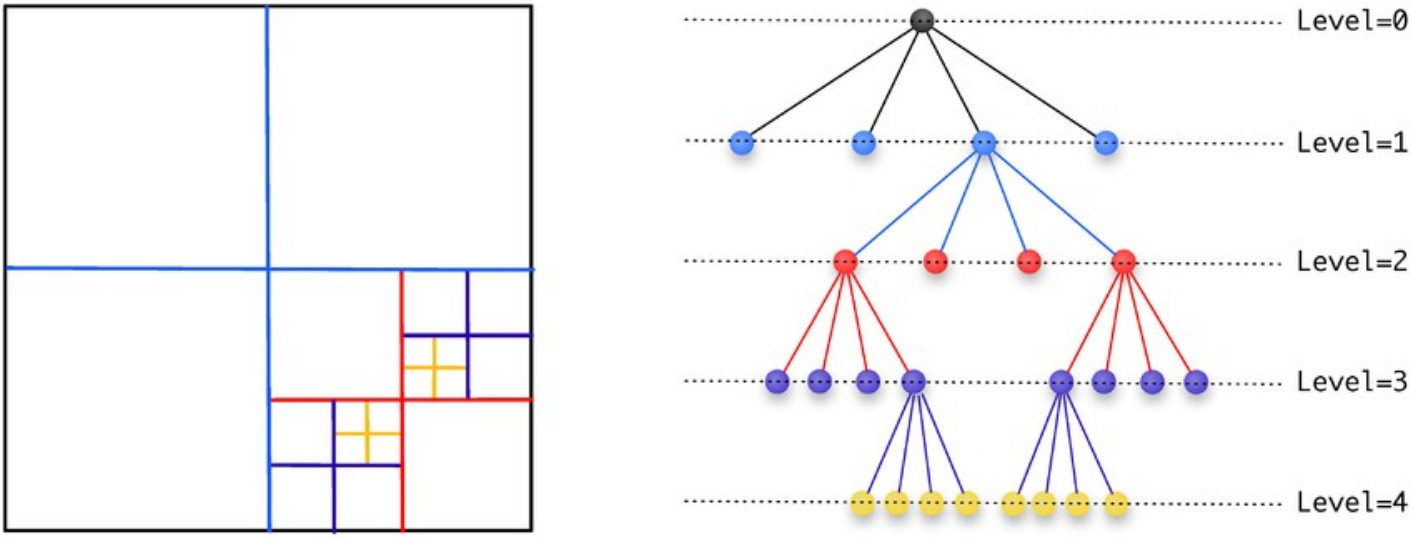
My personal goals for this project are:

1. Better my time management for complex projects.

2. Acquire a more professionnal approach to project management:
    - Failure anticipation and planning accordingly.
    - Appropriate communication with superiors in case of difficulties.

3.1: QuadTree behaviour overview:

The quad tree system is a dynamic spatial partitioning algorithm which organises a quad tree's contents by proximity. This is used in this engine to reduce the number of collision checks per frame.



taken from: https://www.researchgate.net/figure/Discretization-of-a-two-dimensional-domain-left-and-its-quadtree-representation_fig11_298796721

The root quad is an attribute of p2World which is instanciated upon game enging start. The root quad either has no or 4 quad children.

All instances of a quad can hold a limited amount of contents. When that limit is reached, the quad creates 4 children to hold it's contents instead. A limit of tree depth is also defined as it would be counter productive to have a tree that is too deep.

3.2: Abstract generalisation of the QuadTree algorithm:

- Every frame:
  - Clear the QuadTree.
  - For each object to store in QuadTree:
    - Insert object in root quad recursively.
  - Retrieve all potential collisions from the updated QuadTree.
  - Solve collisions.

3.3: Clear() function:

Called every fixed update in p2World's Step() function. Clears all objects lists and deallocates all heap memory allocated to the quad tree except the root quad.

Heap memory is used here to handle a quad's recursive nature.

Algorithm:

- If quad has children:
  - For each child:
    - Call child's **Clear()** function.
  - Deallocate children's heap memory.
- Reset this quad's variables.

3.4: Insert() function:

Called once on root quad for every object that needs sorting. Called subsequently by the **Insert()** function itself to insert object in a child quad.

Algorithm:

- If quad has no children:
  - If there's still room in this quad:
    - Insert object here.
  - Else:
    - If QuadTree can be split further:
      - Split this quad.
      - For each old object stored in this quad:
        - Count how many children the old object overlaps.
        - If old object overlaps only one child:
          - **Insert()** object there recursively.
        - Else:
          - Insert old body back into current quad.
      - Count how many children the new object overlaps.
      - If the new object overlaps only one child:
        - **Insert()** object there recursively.
      - Else:
        - Insert new object here.
    - Else:
      - Insert object here.
- Else:
  - Count how many children the object overlaps.
  - If the new object overlaps only one child:
    - **Insert()** object there recursively.
  - Else:
    - Insert new object here.

3.5: Split() function:

Called from **Insert()** when a quad does not yet have children and aleready holds the maximal amount of objects. Instanciates 4 quad children on the heap and initializes them.

Algorithm:

- Do 4 times:
  - Instanciate a child quad on heap.
  - Set it's variables and it's AABB.
  - Store a pointer to child.

3.6: Retrieve() and RetrieveRecursively() functions:

Called once per fixed update on root quad recursively once the QuadTree has been updated. The function returns a list of PotentialCollisions that are to be filtered during the narrow phase and solved if needed.

In this implementation the function is split between RetrieveRecursively() and Retieve(). The first is called in Retrieve() and in itself whereas Retrieve() is called in p2World's Step() on root quad only. RetrieveRecursively() is a simplified variation of Retrieve().

Retrieve()'s Algorithm:

- Declare a list of PotentialCollisions to fill.
- If root quad is storing more than 1 object or if it has children:
    - Add an uninitialized instance of a PotentialCollision to list.
- If root quad has children:
    - Declare a current index counter.
    - If root quad stores more than one object:
        - Initialize PotentialCollision at current index with root quad's objects as siblings.
        - Add an uninitialized instance of a PotentialCollision to list.
        - Increment current index.
    - For each object stored in root quad:
        - Add root quad's objects to current PotentialCollision's potentialCollideesAbove list.
    - For each child:
        - Store a copy of current index and of current PotentialCollision's potentialCollideesAbove list.
        - Call child's **RetrieveRecursively()** with references to list to fill and current index as arguments.
        - If current index has changed (meaning the function call has resulted in new PotentialCollisions being added to list to fill.):
            - Set current PotentialCollision's potentialCollideesAbove list to be equal to the copy made above.
    - Remove last uninitialized PotentialCollision from list.
- Else:
    - Add root quad's objects to PotentialCollision at index 0's siblings list.
- Return filled list to fill.

RetrieveRecursively()'s Algorithm:

- If quad has children:
    - If quad stores any objects:
        - Initialize PotentialCollision at current index with quad's objects as siblings.
        - Add an uninitialized instance of a PotentialCollision to list.
        - Increment current index.
    - For each object stored in root quad:
        - Add root quad's objects to current PotentialCollision's potentialCollideesAbove list.
    - For each child:
        - Store a copy of current index and of current PotentialCollision's potentialCollideesAbove list.
        - Call child's **RetrieveRecursively()** with references to list to fill and current index as arguments.
        - If current index has changed (meaning the function call has resulted in new PotentialCollisions being added to list to fill.):
            - Set current PotentialCollision's potentialCollideesAbove list to be equal to the copy made above.
- Else:
    - Add quad's objects to current PotentialCollision's siblings list.
    - Add an uninitialized instance of a PotentialCollision to list to fill and increment current index.

3.6: Debugging functions:

GetQuadTreesAabbs():

Fills a referenced list with AABB's of every quad recursively. Used to retrieve AABB's in order to draw the quads for debugging.

LogQuadsBodyCount():

Instructs every quad to cout it's node level and it's the number of objects it's storing. Function called recursively.