Quad tree-based collision detection engine

Everything is in the “col” namespace, except where otherwise noted.

UniqueId:

Note: This class currently doesn’t exist and is represented by the GenericPtr class from the “qao” namespace.

An object of this class is used to represent a collideable instance in the collision system. It is composed of two unsigned integers, index and uid. It will be up to the user program to assign unique IDs to collideable instances, with the following constraints.

The index must be a reusable number of 0 or higher. The collision system uses a vector to store instance information and uses the index part of the instance’s unique ID to quickly index that vector. What reusable means here is that if the instance with the index X is destroyed, X will be assigned to a new instance when it is constructed. It doesn’t matter in which order the indices are recycled, but if you only use a counter variable and increment it each time you need a new index, the information vector will keep growing until there is no more memory.

The uid is a value that needs to be unique over the course of the program’s lifetime and using a global counter variable to assign it is fine. Zero is considered to be null, so start from 1.

Simply put: No two instances can have the same index parts of their UniqueIDs at the same time, and no two instances can have the same uid parts of their UniqueIDs during the whole program’s lifetime (unless you reset the whole program state at once, of course).

(constructor)(size_t index_, size_t uid_)
Initializes the unique ID object with the provided index_ and uid_ arguments.

(constructor)()
Same as the above with zero as both arguments.

(constructor)(std::nullptr_t p)
Same as the above.

BoundingBox:

A bounding box represents the position and size of a minimal rectangle needed to envelop a collideable instance. The position is relative to the top-left corner of its domain.

Public data members:

Note: GroupMask is a typedef for unsigned integer (width depends on the platform).

double x, y, w, h;

GroupMask groups;

Public methods:

BoundingBox(double x, double y, double w, double h, GroupMask groups);

Constructs the BoundingBox object. The first 2 arguments represent the x and y coordinates of the bounding box relative to the top-left corner of its domain. The next 2 arguments are its width and height within the domain. The final argument, „groups“, is basically a bit mask. When checking for collisions, two instances can collide only if their groups overlap on at least one bit, that is:

collision = (inst1.group & inst2.group != 0);

BoundingBox();

Constructs the BoundingBox object with default values (all zeros).

bool overlaps(const BoundingBox & other) const;

Returns true if two BoundingBox objects overlap (taking their groups into consideration) and false otherwise.

bool enveloped_by(const BoundingBox & other) const;

Checks whether a BoundingBox object is completely enveloped by another BoundingBox object. Groups are not taken into consideration here.

void reset(double x, double y, double w, double h, GroupMask groups);

With this method you can set all the fields of the BundingBox object.

Domain:

A domain object represents an area in which all collideable objects reside and move. There are two different implementations, but both conform to the following interface.

Public methods:

(constructor)
(double x, double y, double w, double h, size_t maxdep, size_t maxobj);
Constructs the domain object. The first 4 arguments specify the domain’s x and y coordinates and its width and height. Note that the domain does not use the x and y variables in any way – when you insert collideable instances their coordinates must be relative to the top-left corner of the domain. You can, however, fetch the domain’s x and y coordinates, but if you don’t need that, you can just leave them as zero. The “maxobj“ argument specifies how many objects can be in a single node of the tree before it’s split into 4 sub-nodes. The “maxdep” argument specifies the depth limit after which the nodes won’t split under any circumstances.

(destructor)
Deconstructs the domain object. Is not virtual.

void clear();
Resets the state of the domain object. After calling clear(), it will be as if it were just initially constructed.

void inst_insert(UniqueId instance, const BoundingBox & bb);
Registers a new collideable instance with the domain object. Later, when you check for collisions, the domain will refer to this instance with the UniqueId provided here as the 1st argument. You'll also need to refer to this instance with the same UniqueId when you want to change its position within the domain or remove it. The second argument represents the instance's bounding box – its position, size and group within the domain.

Be careful as to not try to insert an instance whose UniqueId would overlap with another already in the domain.

void inst_update(UniqueId instance, const BoundingBox & bb);
Updates the bounding box of an instance already present in the domain.

bool inst_exists(UniqueId instance) const;
Checks whether an instance with the given UniqueId already exists within the domain.

void inst_remove(UniqueId instance);
Removes an instance with the given UniqueId from the domain.

void pairs_recalc_start();
If the domain in question has worker threads that check for collisions, this wakes them. Otherwise, it does nothing.

size_t pairs_recalc_join();
Waits for the worker threads to finish their work and return the total number of collisions found.

size_t pairs_recalc();
Equivalent to calling pairs_recalc_start(); return pairs_recalc_join();

bool pairs_next(UniqueId & inst1, UniqueId & inst2);

You can call this method after calling pairs_recalc_join() or pairs_recalc(), but before any subsequent inst_* method calls (otherwise you'll get outdated information). It is used to loop over pairs of instances whose bounding boxes collide in the domain, and UniqueIds of those instances will be written to the locations pointed at by the 1st and 2nd argument.

Note: If instances A and B collide, either A+B or B+A will be (eventually) returned by this method call, but not both.

Returns false if there were no more pairs to loop over (and thus nothing was written to locations pointed at by the arguments), and true otherwise.

Work-in-progress methods:

GenericPtr scan_point_one(GroupMask groups,
double x, double y) const;

void scan_point_vector(GroupMask groups,
double x, double y, std::vector< UniqueId > & vec) const;

GenericPtr scan_rect_one(GroupMask groups, bool must_envelop,

```
                    double x, double y, double w, double h) const;

void scan_point_vector( GroupMask groups, bool must_envelop,
                    double x, double y, double w, double h,
                    std::vector< UniqueId > & vec) const;

GenericPtr scan_circle_one( GroupMask groups, bool must_envelop,
                      double x, double y, double r) const;

void scan_circle_vector( GroupMask groups, bool must_envelop,
                    double x, double y, double r,
                    std::vector< UniqueId > & vec) const;
```

Public data members:

```
const double X, Y;
const double WIDTH, HEIGHT;

const size_t MAX_DEPTH;
const size_t MAX_OBJECTS;
```

All are set in the constructor and cannot be changed afterwards.

Setting maxdep and maxobj:

When all nodes are split to their limits, the tree will have 4^maxdep leaf nodes, and each will have a width and height of DOMAIN_WIDTH / (2^maxdep) and DOMAIN_HEIGHT / (2^maxdep). So you'll want to set the values in a way that each leaf node is about 3-4 times the size of the average size of a collideable instance (it's not good for performance when an instance needs to change its parent node often).

For maxobj, 5-10 is usually good, though you'll need to profile your program to be sure.

Implementing classes:

Classes "QuadTreeDomain" and "MTQuadTreeDomain" implement the domain interface. With the multithreaded version (MT prefix) there are 4 worker threads, each processing collisions while the calling thread is free to do other work in the meantime. Each thread will process a single quadrant of the domain, so this works best if collideable instances are distributed evenly.

With the singlethreaded implementation, the calling thread does all the work. Note that for many programs, this can be more than enough. You'll probably only need the multithreaded version when you get to thousands, or tens of thousands of collideable instances in the domain.