衝突処理と動的Treeの”伐採”

Atsuhito Fukuda

[概要]

My Game Programming in C++では自作のバイナリツリーを用いて衝突の処理を扱っています。

メモリ節約のために動的な八分木を用いて空間を分割します。

動的なツリーであるために定期的に不要になった枝を”伐採”する必要がありますが、幾つかの伐採方法についてその処理の速度を比較した内容となっています。

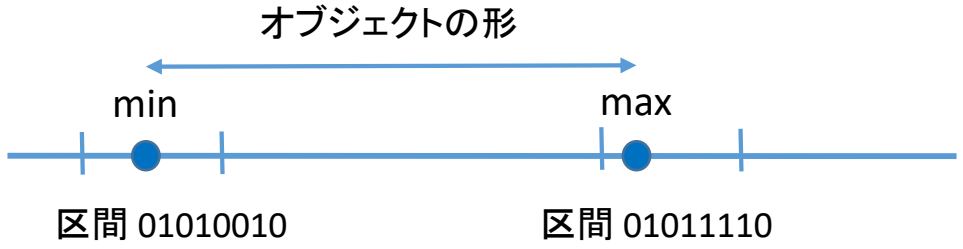
用いたコード等は”BinaryTree PerformanceCheck”フォルダにあります。

[Templateクラス BinaryTreeの説明]

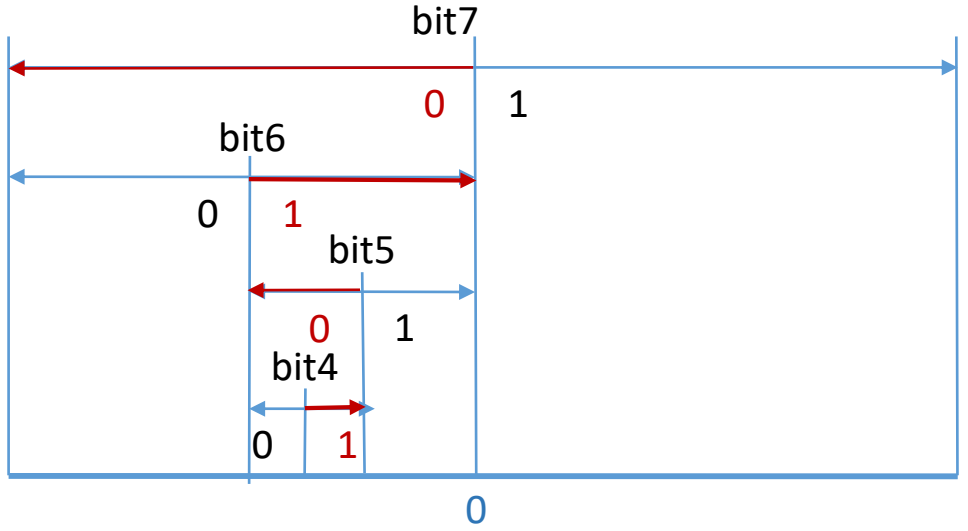
自作テンプレートクラスのBinaryTreeクラスは
BaseBinaryTreeクラスの派生クラスになり、2^n分木を構築します。以下で二分木を用いてクラスを説明していきます。

[BaseBinaryTree]

空間(二分木なので線分)は256の区間に分割されていて、各オブジェクトの端を表す2点min, maxが配置されている区間からオブジェクトのTree上での配置を割り出し登録します。



区間番号は1Byte整数で表されます



区間番号の先頭bitから順に、2等分した各区間の左右どちら側かを辿っていけば自身の配置される区間に到着します。この経路はTree上での分岐経路とまた同じになります。

区間 01010010
区間 01011110

Treeのルートノードから、
Left,Right,Left,Right
のノードに登録する

2点の配置されている区間番号の先頭からの共通部がそのオブジェクトのTree上での配置位置を示します。

登録・削除を終えた後にアップデート関数を呼ぶと、Treeのルートノードから順に幅優先探索を行い全ノードを配列に収納します。

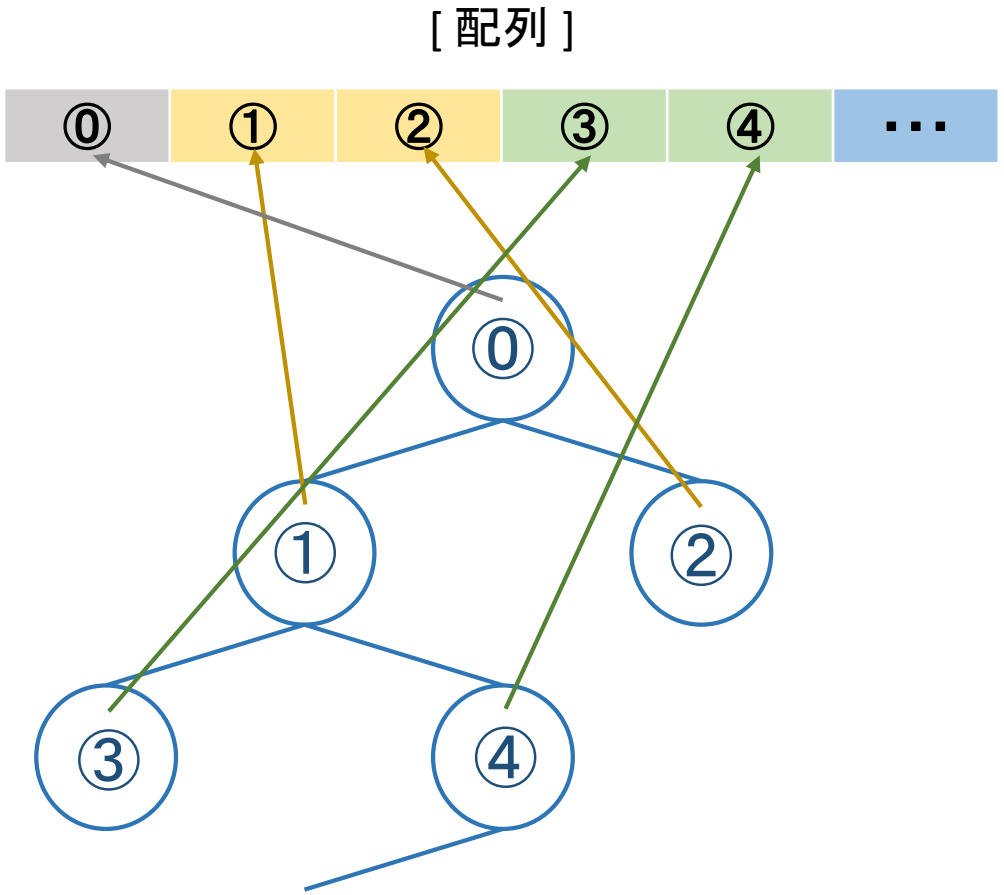
NextNode()関数は配列の要素を進め、NextParent()関数は現在のノードの親ノードにアクセスします。present()関数を用いれば現ノードの所持する要素(テンプレート)にアクセスできます。

[BinaryTree]

BinaryTreeクラスはBaseBinaryTreeクラスの派生クラスであり、

BinaryTreeクラスにはオブジェクトの配置変更に伴い不要となった枝を”伐採”する機構があります。

その他にも、登録ノード等を記録したクラスを受け渡すことで、オブジェクトの配置変更や削除時には前回の登録の修正を行います。



幅優先探索

[Tree上のオブジェクト同士の衝突処理]

Tree上の要素同士の衝突はCollision関数で行われます。

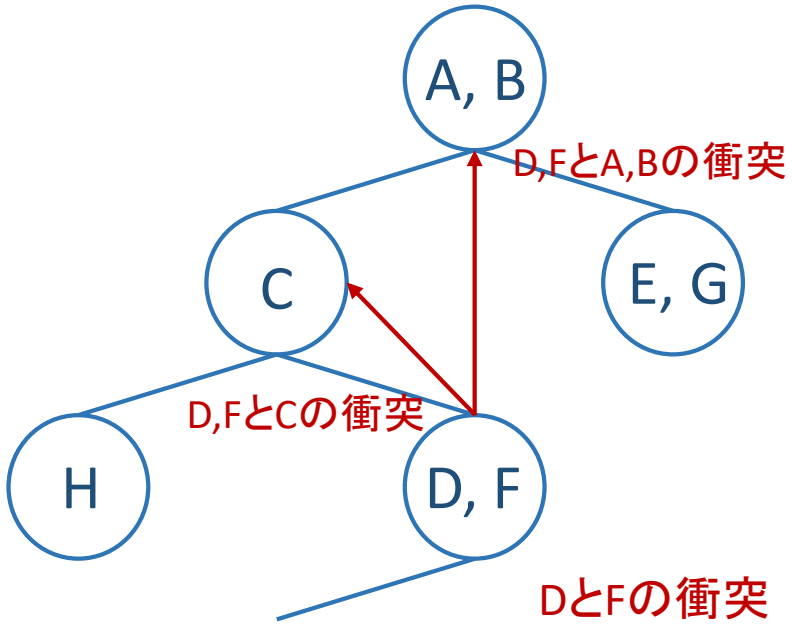
処理の概要は次のようになります。

① ノードをまとめた配列から現在のノードを取得

② 現在のノードの親ノードを辿っていき、ノードにオブジェクトが登録されている場合に衝突チェックリストに追加する

③ 現在のノードの登録オブジェクト同士、または親のノードのオブジェクトとの衝突判定を行う

④ ノードをまとめた配列の要素を1つ進め、①に戻る



[処理速度の検証方法]

[初期設定]

・3次元空間上での衝突処理を扱うためTreeは八分木を用います。

・1000の球体オブジェクト(半径2)を用意し、各々ランダムに初期位置、速度を与えます。

・空間は距離10×10×10の立方体を最小の区間として$8^8 (= 16{,}777{,}216)$分割します。

- ・区間番号は8進法8桁の数字となります
- ・球を包むAABBから配置されている区間を判断します

[検証部分]

・1000のオブジェクトのうち一部のみが動いているものとし、残りは位置の変わらない静的なものします。

・”全オブジェクトを初期化した後に200フレーム分の衝突反応を行う”工程を50回行い、1工程にかかる平均の処理時間を計測し、比較します。

・Treeの”伐採”は毎フレーム行います

[不要となった枝の”伐採”方法]

⓪ オブジェクトを登録している本Treeとは別に、削除するべきノードを集めたTreeを作成します。

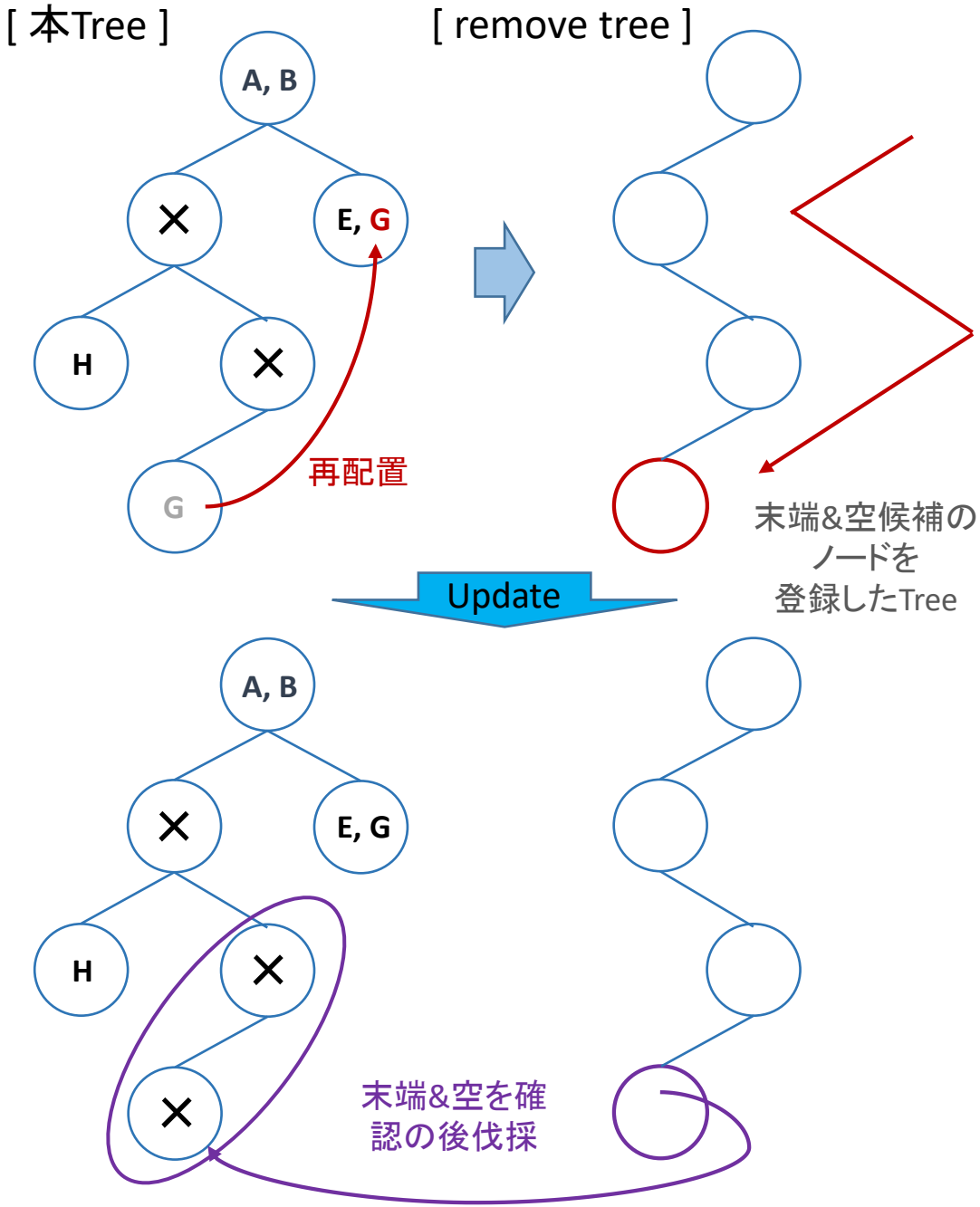
オブジェクトを動かす際に元々配置されていたノードが末端&空となった場合にそのノードをこの”remove tree”に登録します。　その後別のオブジェクトが同ノードに配置されることもあります。

一通りオブジェクトの再登録を終えた後にBinaryTreeクラスのアップデートを行いますが、この際に、

”remove tree”の末端に登録されているノードを辿っていき、本Treeで該当ノードが末端&空だった場合に本Treeからそのノードを含めた”余分な枝”を伐採します。

(余分な枝を伐採し終えた本Treeの各ノードを配列に収納していきます。)

各オブジェクトの移動時に加えて衝突処理時にもオブジェクトの配置が変わるため、重複登録を避けるために”remove tree”はTree構造を採用しています。



① “remove tree”を用いることをせず、BinaryTreeクラスのアップデート時の全てのノードを配列に収納する際に各ノードが末端&空ノードでないかを確認していき、そうだった場合にそのノードを別途用意した配列“delete table”に収納していきます。

衝突処理を行う前に”伐採”を行うわけにはいかないので、余分な枝も含むTreeで衝突処理を一通り終えます。

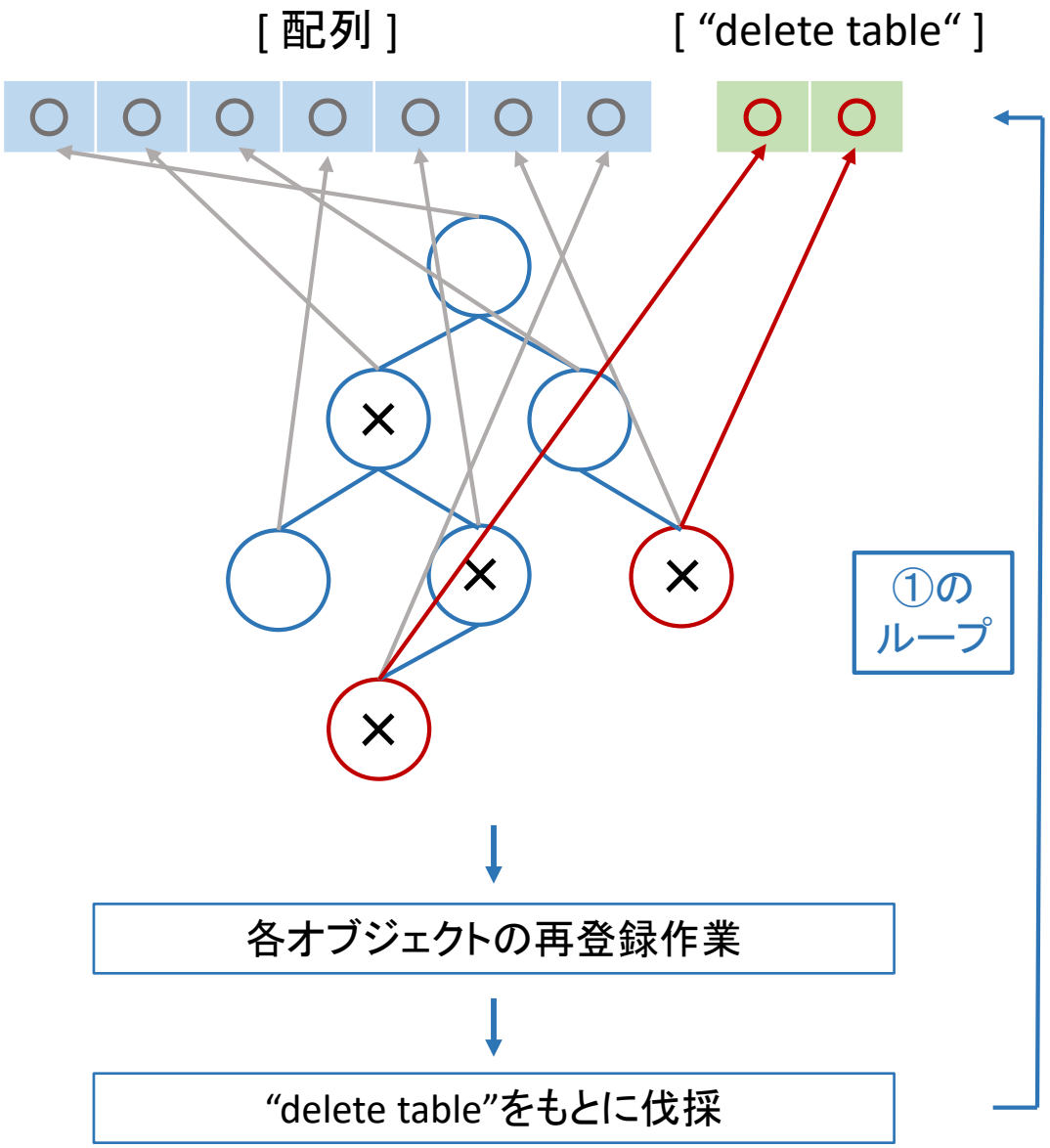
その後一通りオブジェクトの再登録を終え、遅れて前回のアップデート時に”delete table”に収納したノードを確認していき、末端&空だった場合にここで”伐採”します。

その後①の初めに戻ります。

② ⓪ではオブジェクトが動く際に、Tree上での配置に変更がない場合でもオブジェクトを一度Treeから取り除き再登録していました。

今回は、Tree上の配置に変更があった場合にTreeから取り除き、”remove tree”に登録するようにします。

③ ②と比較のために、①の場合でもTree上の配置に変更があった場合にTreeから取り除き、再登録するようにします。

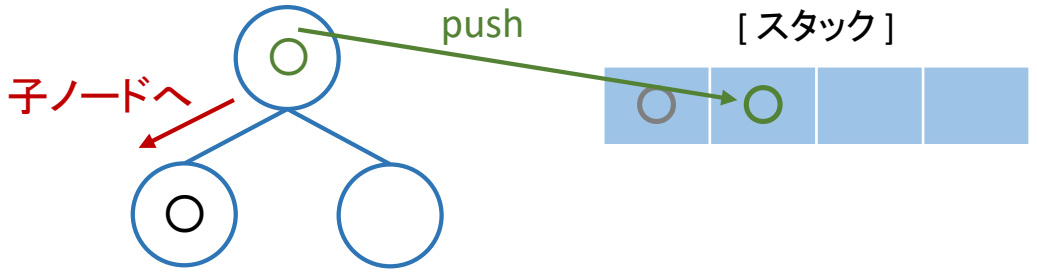


[ノードを配列に収納する過程は必要か？]

BinaryTreeクラスは汎用利用を想定して作成したクラスでしたが、衝突を扱う処理には無駄があると感じます。

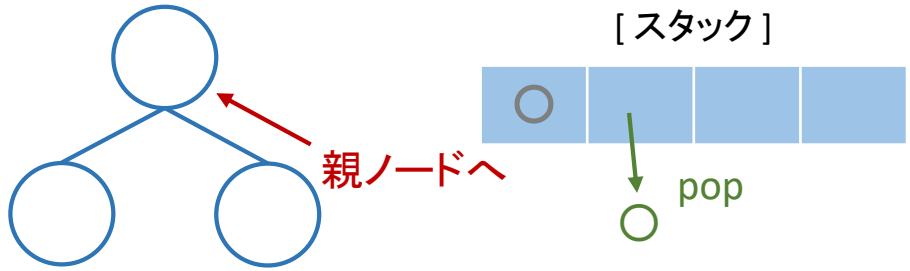
BinaryTreeクラスはTreeの全ノードを配列に一度収納しますが、この過程を行わず、Treeをルートノードから走査しつつ衝突処理を行えば無駄がないのではないかと思えます。

そこで、スタックに親ノードのオブジェクトを登録しつつTreeを走査していき、各ノードでは登録されているオブジェクト同士、またはスタックに登録されているオブジェクトとの衝突処理を行うこととします。



子ノードへ移る際にオブジェクトをスタックへpush

子ノードの”〇”同士でのオブジェクトの衝突処理
&
“〇と”スタックの”〇”, “〇”のオブジェクトの衝突処理を行う



全ての子ノードで処理を終えると
親ノードへ戻る際にオブジェクトをスタックからpop

この場合でも”伐採”は必要となりますが、

④ ②と同様にオブジェクトの再登録を行う際、Tree上の配置に変更があった場合にTreeから取り除き、”remove tree”に登録するようにします。

“remove tree”もまたルートノードから走査しつつ末端&空ノードを探していきます。

余分な枝を伐採した後に前述のスタックを用いた衝突処理を行います。

⑤ ④で”remove tree”を用いない替わりに、前述のスタックを用いた衝突処理の走査と同時に各ノードが末端&空でないかを調べ、そうだった場合に伐採します。

(⑥ ④,⑤で伐採を一切行わない場合を検証します。)

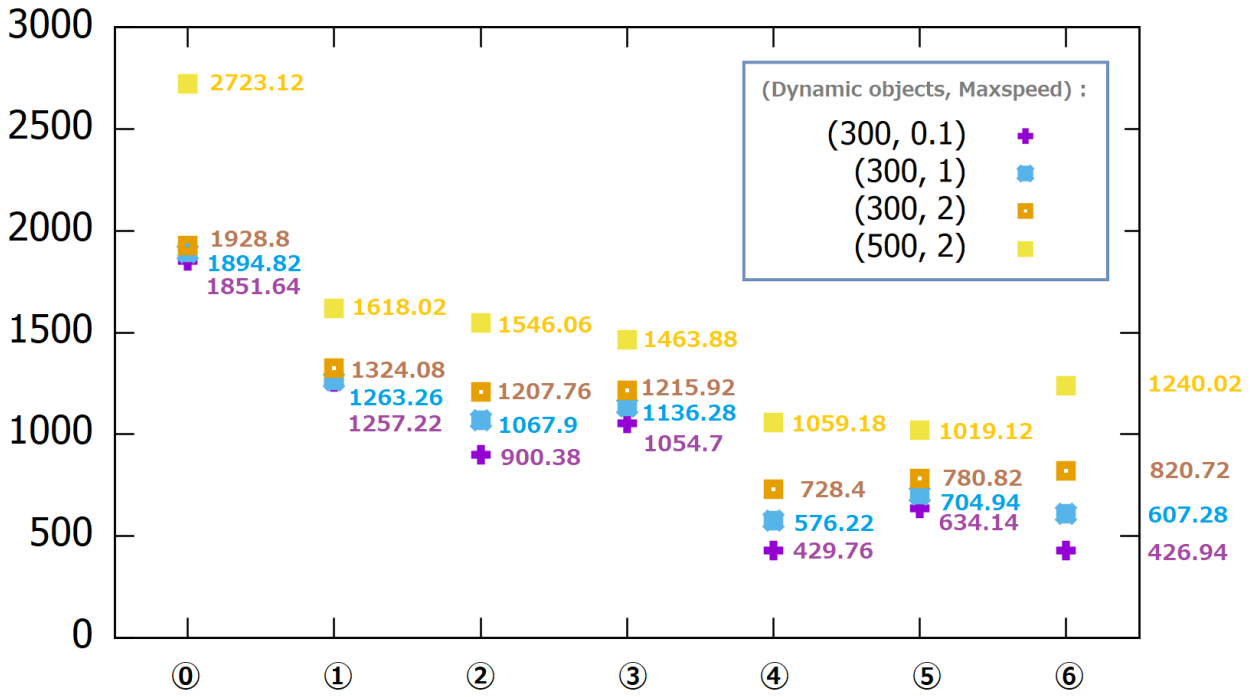
[結果]

各数値(ms)はオブジェクト総数1000のうち動いているものを300, 500とした場合、またランダムに設定された速度(距離毎フレーム)の最大値を0.1, 1, 2とした場合の値となります。

各オブジェクトは半径2の球であり、空間は最小10×10×10の区間で分割されています。

⓪, ①, ②, ③, ④, ⑤, ⑥は順にmain.cppでの(enum class) ”compare0”, ”compare2”, ”compare3”, ”compare4”, ”compare5”, ”compare6”, “compare7”にあたります。

Windows10、Intel Core i3-8100の環境下での計測値となります。(目安に)



例えば、速度最大値0.1、動くオブジェクト300の場合での④(“remove tree”方式)と⑤(走査中方式)では1フレーム中に1msほど差があります。

各場合を比較するに、

・Treeでの配置変更の有無に関わらず”remove tree”を呼び出す⓪は最も遅いが、配置変更時の場合に絞れば②と③を比べるに”remove tree”を用いる方が速いことがある

・全ノードを一度配列に収納しない④～⑥のスタックを用いた衝突処理方式の方が速くなる

・④～⑥の方式では、

・動くオブジェクト数が少なく速度が遅いと全く”伐採”しない(ただし200フレーム間)方式がわずかに速く
・動くオブジェクト数が増え、速度が上昇するにつれて”remove tree”を用いた方式が速くなり
・最終的には走査と同時に”伐採”する方式が最も速くなり、一切伐採しない方式は最も遅くなる

すなわち、

・配置変更されるオブジェクトが少ないのであれば動的Treeに削除ノード(候補)を登録するのが速く、
・配置変更が多くなると静的なものを含め全ノードを伐採対象として調べる方が速くなり、
・また伐採を放置することで速くなることがある、

と言えるのだと思います。

ありがとうございました
Thank You！

Atsuhito Fukuda