Continuum: a Three-Dimensional Continuous Space Microbial Simulation Framework

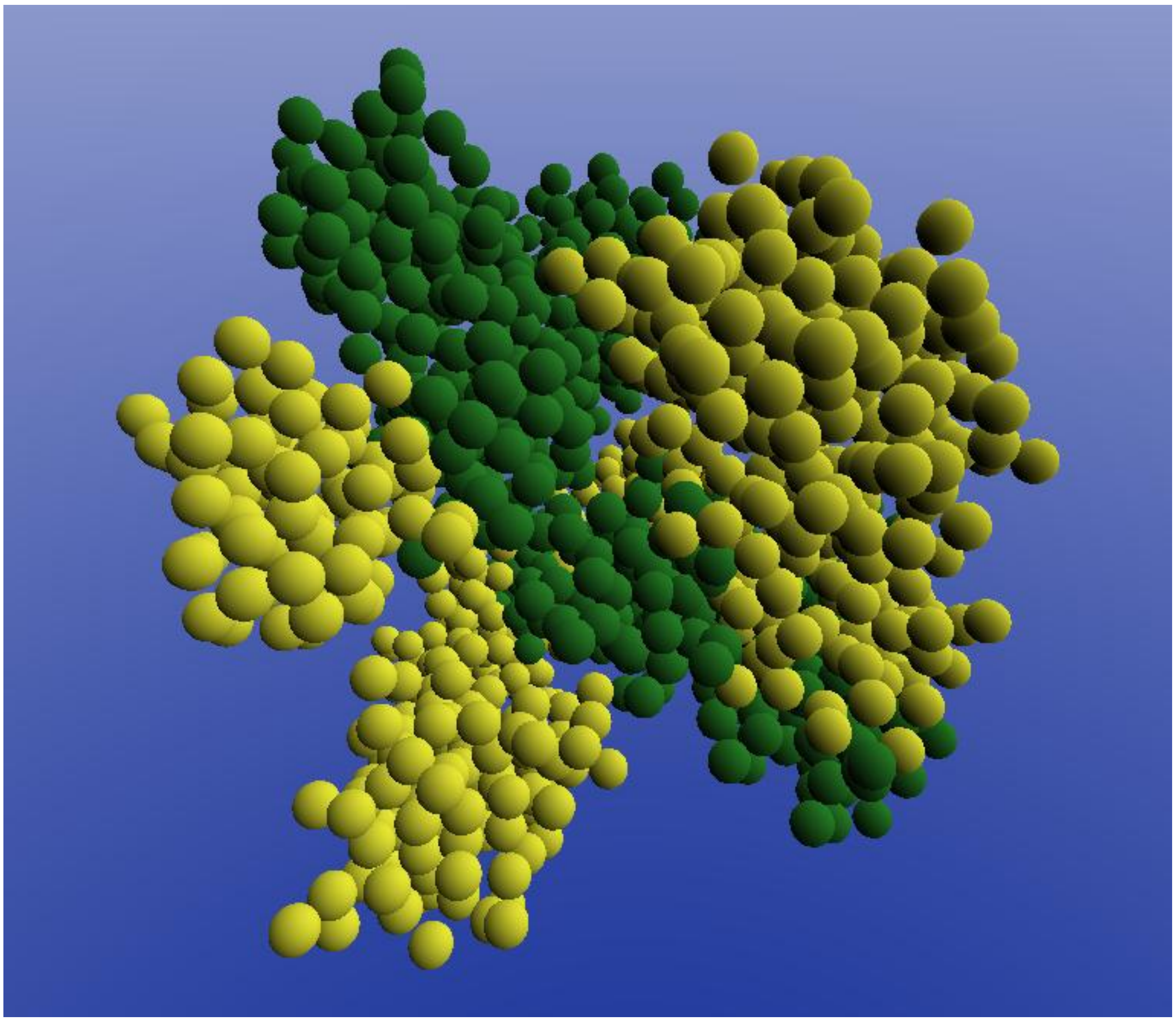

Figure 1 Screenshot of example simulation ran with 3D viewer

In this document we showcase our simulation framework for running microbial simulations in three-dimensional continuous space, we call it Continuum. It provides an optimised implementation of essential components most biological frameworks share. This allows users to quickly design efficient biological simulations without having to think about technical details and optimisation. Besides the simulation structure the project also contains solutions for rendering running simulations, writing results and intermediate states to files, and rendering from file. The framework is written in C# and will need C# code to be interacted with. We hope our solution enables biologists with less programming experience to create large scale simulations to aid in their research.

Table of contents

User guide

This chapter contains specific details about making use of Continuum. The chapter names correspond with the name of class and the file it is stored in. Note that for any property (int, float, string, ect) in the code there are comments that explain what it does, these are visible when they get recommended to the user by the auto-complete of an IDE, or when hovered over with the mouse. This guide is more about how these properties relate to each other and how to make your own version of some of the abstract classes in the code. If the provided explanation is unclear then you can look into the code to figure out how it is used, practically all IDEs allow for ctrl+clicking on a method or property to go to where it is defined, which saves a lot of time searching.

Note that everything mentioned in this chapter can be done without altering any of the provided code by creating new classes that inherit from the abstract classes mentioned below and by overwriting existing methods. This is also the intended way of approaching it. Altering the original code should only be done if the provided implementation is not satisfactory and is done to update the release build of Continuum itself, regardless of use case.

General structure

Continuum exists in three sperate layers corresponding with the separate C# projects.

The first layer is the Simulation layer, which is the core of the simulation framework. It is where the Simulation.cs, Organism.cs, World.cs and DataStructure.cs are defined. This is the only required package to be able to make use of Continuum.

The second layer, which is built on top of the first, is the Implementation layer. This layer covers multiple different types of ways to run a simulation, alongside a SimulationRunner.cs which is a helper class to have all types of implementations run using the same interface.

The third and final layer, which is at least built on top of the first, and can make use of the second, is the User layer. This layer has no defined structure and is only used to help understand Continuum. This is your own C# project which imports the previous layers' projects to make use of them. Here you will define the rules that organisms should follow in your simulation, and a world that decides the conditions under which it is simulated. The code of Continuum also contains two example projects which belong to this layer. That being BasicImplementation.csproj and GrowthGridImplementation.csproj. You can use these projects as references to make your own.

Simulation layer

Simulation

This is the system that controls everything in the simulation. If you make use of the Implementation layer project, then you can skip this part as SimulationRunner.cs and all implementations have done all of this for you, if you decide to only use the Simulation layer, then you have to configure everything yourself.

Start by creating a new Simulation (where you can optionally enter a seed to have deterministic results) and a new World (more on the world later), and calling Simulation.CreateSimulation().

Then proceed by calling Simulation.SetDataStructure() and passing your chosen data structure as a parameter.

Finally, when you are ready you can call Simulation.Start() and the program will begin to run.

With this we have covered creating a simulation, the next step is to get your simulation to actively run. For this, Simulation contains a function Step() which should be called repeatedly by you. The Step() function returns a Task, if you are running the program single-threaded then the returned task can be ignored, if you are running the program multi-threaded, then make sure to change Simulation.Step() into await Simulation.Step(). This ensures the program only goes to the next line of code when the full step operation has been completed.

Once you have decided you are done with running the Simulation, call Simulation.Abort(). If you want to automatically quit the simulation under certain conditions, see the section World.

An optional feature that the Simulation class contains are events. These are the properties named: OnTick, OnFileWrite, OnEnd and are called once these actions happen. As a user you can *subscribe* to these events to run code as a reaction to such an event being invoked.

The simulation also has a few properties regarding file writing, these being: FileWritingEnabled, WriteToSameFile, and TicksPerFileWrite. These can be used, in combination with the SimulationExporter mentioned after this to have control over how the simulation contents are recorded.

Finally make sure you both support single-threaded and multi-threaded data structures, look at the current implementations for examples. Or leave out multi-threading if you don't want to support it by using 'throw new NotSupportedException()' on a multi-threaded data structure being assigned.

SimulationExporter

This class oversees how the simulation contents are written to a file.

The simulationExporter has two properties that need to be filled in to be able to write to files, those being FileName, SaveDirectory (affects where the file is stored), and a third optional boolean ShowExportFilePath (used to debug where your file was written to, because C# can be weird with local and absolute paths).

SimulationImporter

This class oversees how the simulation contents are loaded into a world. We are currently not really happy with how this works, it does work, but is a bit user unfriendly. A working example can be found in the 3DRenderer project.

The problem with inporting is that when a user creates their own types of organisms the system does not know about it before one is made, so each type of organism will have to be 'registered' before its type can be loaded in. To do so call SimulationImporter.RegisterOrganism(). Then after that you can call FromFileToOrganisms to populate a world. You can also call FromFileToObjectType to convert it to any type of object you want. These have a parameter 'timestamp index' which indicates which step in the simulation it should load in, this only works if all simulation timestamps are saved in the same file.

Organism

Now we are getting to the part that you must and want to do yourself. Organism is an *abstract* class, meaning it is a type of template class, you can make your own class that *inherits* from organism to make a complete version. Everywhere in the code that requires the class Organism will in practice requires a subclass (so something that inherits from) of Organism where all the known methods are worked out.

Organism already has a few systems made for you, some which you can overwrite if you want, those being Move() and Reproduce(). The default implementation of these should be enough, but if you want custom logic, then you can change the code here per individual simulation.

When creating your own organism, there are a few properties and methods you will have to define yourself, as they are unique per implementation. First of all, the property Key needs to be defined and is used by the Simulation to translate your type of organism to a string. Next you can optionally define Color, doing so will allow visual implementations to display your organism nicely.

For methods you probably first need to create a new constructor for your version of Organism, this contains code that is called when the class instance is made. Next a required method named CreateNewOrganism() it is used to pass on properties when a new organism of the same type is made after reproduction. After that you will have to define a Step() method, this will be called every tick by the simulation (on its Step() call) and can be used to tell the organism when and how to move and reproduce.

Finally, we have the two connected methods, ToString() and FromString(), these methods can be used to translate the organism to a file and retrieve it from a file. For these methods, the logic for saving the rest of the world is already done for you in SimulationExporter, you just need to explain how an individual organism is saved.

World

World is the other *abstract* class you will have to create yourself to be able to run your own simulation. The world is used to explain the experiment setup, handing starting conditions, boundaries and exit conditions.

When you create a world you can pass a bool for precise movement, doing so will let you choose between moving only if the destination is free of collisions, or moving as close to the destination as possible. The second being slower by about a factor of 2 but more accurate. The value will be applied to all organisms.

First, World contains two methods, Initialize() and Step(), both are optional to *overwrite* and add custom logic to, but by default they are empty and they can be left that way. The Step() function can be used to update any extra class that you made yourself, like for example a discrete grid used to store the number of resources in the environment.

The required methods to implement are StartingDistribution(), which you can use to fill the world with a few organisms to get started with (Example can be found in the example User layer projects), IsInBounds() can decide if a position is within the world, which is used in collision checks. And a final method StopCondition(), this can be used to abort the simulation once certain conditions have been met, but it can also be left empty with return false; to allow for a required manual halting of the simulation.

World also includes some pre-made methods, one being AddOrganism() this one is used for inserting organisms outside of the standard simulation loop (for things like user input). RemoveOrganism() if you want to forcefully remove something. And GetOrganisms() and GetOrganismCount() are used to get a list of all organisms and the length of that list respectively, these final two methods are rather slow, and it is recommended to only use them if strictly necessary, or if performance is less important.

Datastructure

Datastructure is the final class we will discuss regarding the Simulation layer. A few implementations of it have already been provided and it is recommended that you make use of these before trying to make your own. For the rest of this section we will refer to data structure as DS. Data structures used for speed ups are also sometimes called acceleration structures, so you can also find more information under that search term.

The simplest DS is NoDataStructure. As the name suggests it is not actually a DS, but it does follow the requirements set by the DataStructure abstract class. It is not used for performance, but it can be used to be absolutely sure that the data structure is not affecting the outcome of a simulation.

Next are the ChunkDataStructures. These have a 2D and 3D variant, where 2D is more optimal if the simulation grows very little in the Z axis (for example when simulating biofilms), its only difference to 3D is that it selects the chunk only based on an organism's X and Y position, without regards for its Z position. 3D makes use of all 3 axes as you would expect. The chunk DSs are significantly faster than not using any DS, with it being able to run at 4 tps for 100000 organisms depending on the conditions and the starting values. Using a chunk DS requires inputting a few parameters, the minimum- and maximum position the DS should account for (this should always be equal or larger that the bounds defined in your implementation of World. It has a parameter for chunk size, which, in relation to how large your organisms are, is very important for the eventual performance of the DS. The final parameter is largest organism size, which is the size of the largest organism that can appear in the simulation. This is used to make sure it always knows how far it needs to search to be certain of not creating a collision. The world size has a theoretical limit of 2 billion chunks, which can be 'easily' achieved by having at least 2000 chunks in each dimension. So, there is a limit to how large a world can be using this data structure. The best use-case for chunk DSs is a situation where are organisms are evenly distributed and do not to know their nearest neighbour when it is not in neighbouring chunk. The latter since the nearest neighbour implementation does not look further than an organisms neighbouring chunks.

For both chunk DS variants, there are two more variants for multithreading. These have roughly the same logic and can be created in the same manner as the DSs mentioned above. They include one extra parameter for the number of logical cores they can use. If this is left blank, then it will default to the maximum number of logical cores on your pc. Multithreading done through DSs is done internally and should not be combined with running multiple instances on the same computer simultaneously. The performance metrics are good, being able to reach about 200000 organisms at 4 tps (with 16 logical cores), but while using all your pc's CPU cores. It would be more efficient to run multiple instances of the single-threaded chunks or the RTree which will be mentioned after this. Note that multithreaded DSs are tested to work under many circumstances, but they are the most volatile of all DSs and might cause unpredictable bugs that are hard to resolve. Special care should be taken when using shared resources such as resource chunks, since concurrent access can cause memory inconsistencies.

The final pre-implemented DS is the RTree, which can reach about 100000 organisms at more than 1 tps. This is slightly less that the 3D chunk DS single-threaded, but with the advantage that it stays performant in a wider array of use-cases. In a case where multiple clumps of organisms form far away from each other, the R-Tree does a better job at indexing them since it can dynamically partition the search space. Furthermore, the nearest implementation of R-Tree can efficiently return the nearest neighbour regardless of its distance. So when this is essential pick the R-Tree over the chunk based data structures.

The following explanation can be ignored if you do not plan on making your own data structure, but if you do want to do so, then here is a short guide on how to use it.

Datastructure is again an *abstract* class meaning it defines an implementation. It has access to the World through a SetWorld() method which is handled for you. It contains a few abstract methods for you to implement: AddOrganism() and RemoveOrganism(), GetOrgansims() and GetOrganismCount(), a Step() function used to handle most logic in the data structure and CheckCollision() and NearestNeighbour() both of which can be used by Organism. All of these methods are rather self-explanatory in their goals, so they will not be restated here. There is also a required method named FindFirstCollision(), this is used by the precise movement option in World. Most of the logic is already implemented by FindMinimumIntersection() and RayIntersection(), but the rest has to be done by you. A non-required property (due to the open-endedness of its implementation) is somewhere to store all the active Organisms. You will have to come up with the rest yourself depending on what type of data structure you are trying to implement. If you do not know where to go from here, then it is probably not recommended that you try to add your own data structure. But in the case you do want to, look at the code of the provided data structures as examples. Chunk2DDataStructure and Chunk3DDataStructure are rather easy to understand and RTree is a lot more complex but the most effective currently provided. For an example on how to use a multithreaded data structure see MultithreadedChunk2DDataStructure or MultithreadedChunk3DDataStructure.

Note that some differences exist between DSs that first of all cause seeds to run differently for different data structures (as would be expected), but also methods like Nearest neighbour always returning a nearest neighbour for the RTree, while only returning a nearest neighbour within a range for the chunk DS. This is mostly due to implementation differences and some differences being extremely inefficient for some types of DSs. Multithreading does not support nearest neighbour range queries as it is not deterministically possible while running concurrently.

Implementation layer

SimulationRunner

This is a helper class that allows the user to connect their simulation to a specific type of program executable, which we call a program medium (more on that later). It is there to simplify the process to the user. Two example user simulations are coupled with the code and can be used as a good example of how to use this class, but a guide will also be provided here.

SimulationRunner already does the simulation creation for you once it is made, so there is no need to make your own Simulation. It does however not make organisms or a world for you, so both should be done by the user.

The user should set values for DataStructure, World, ProgramMedium and all values related to file writing (which have already been discussed in Simulation and SimulationExporter). After these steps have been done the user can call Start() for SimulationRunner to have the simulation run, at which point everything works as expected. Start() can be passed an integer to set a seed, doing so will result in the simulation ending exactly the same every time, which is useful for debugging. If start is left empty or passed the value 0, then it will behave pseudo-randomly.

NOTE: We encountered a bug where setting a seed will cause multithreaded data structures to perform incorrectly, the fix to this will be placed in Future work. But for now it is recommended to not combine multithreading a seeded simulations (so leaving the seed empty is still fine).

ProgramMedium

The above-mentioned terms in SimulationRunner should all be familiar to you with the exception of the ProgramMedium. We have chosen this name for an abstract class that allows for every type of program executable to have the same interface (meaning it can be controlled with the same methods being called).

Three types of ProgramMedium have already been provided, the first being a 3D viewer using OpenTK (a library to run OpenGL graphics card code in C#). This viewer is the slowest of the 3 and will make use of the graphics card, but it is useful for debugging and is easy to showcase with so it does have its purpose. A 2D viewer has been provided using Monogame, note that most of the draw logic here is on the CPU so it loses a bit of performance there, but it is still miles faster than the GPU implementation. It does make it very hard to debug so do not use it for that. Lastly, a console application has been provided to run the simulation without any drawn visuals, using this will only provide simple statistics about the state of the simulation and the performance of the program. It is by far the fastest and should be used in every situation when the actual simulation results is the only thing that matters. By default it prints a status update every 500 ticks or on file write, but this can be changed with the arguments “print=false” or by adjusting “tpp=[number]” for the amount of ticks before it should print. Disabling automatic printing can be better for performance as it prevents having to call GetOrganismCount() from the data structure, which is costly depending on how it is implemented in the data structure.

It is also possible to make your own program medium to allow for a different type of rendering or to make slight adjustments to an existing one. If you choose to do so, then you will have to define the functions StartProgram() StopProgram() and FileWriten() where the rest can be left up to the user depending on the framework you are using to make a program medium.

User layer

As mentioned earlier the user layer is not something we provide ourselves as a package, but we do have examples of what the user layer looks like, you can view the C# projects named ‘BasicImplementation’ and ‘GrowthGridImplementation’. Both are used as examples and contain less detailed documentation than the two previous layers, the contents of the simulations in these examples are also not grounded in reality, they are merely used as an example of how to make your own implementation.

A final mention for the ‘3DRenderer’ project coupled with the code. This is a version of the 3D renderer that does not run a simulation, but plays back a previously saved one. It is not that extensive, nor is the code quality up to the standard of the provided simulation and implementations layers, but you can use it to get an idea of how to make a renderer to view past simulations.

Theory

Introduction

This section aims to provide some background information on the data structures we have implemented, how they work and why we have chosen them. To understand the answer to these questions it is important to first understand the problem they try to solve.

Problem definition

During a simulation organisms will need information about other organisms in their neighbourhood. An example is checking for collision with other organisms or a Nearest Neighbour Search so an organism can interact with or react to other organisms in its environment. If all organisms of a simulation are naively stored in a list, answering these queries becomes very computationally expensive. Since the structure of the list does not provide information on location, the entire list must be searched to answer these queries. In each step of the simulation all organisms will perform such queries. This means that in each iteration all organisms need to inspect all other organisms. This leads to quadratic scaling which is catastrophic for performance. Increasing the simulation size tenfold would mean computation time is multiplied by 100.

Solution

Clearly, a way to access organisms based on their location is essential to ever achieve large scale simulations. This is where spatial data structures come into play. They provide a way to access entries based on their location. In our search for appropriate data structures, we came across two main types of spatial data structures: Bucket/chunk -based structures, and tree structures. We chose to implement one of each and in the following sections we discuss how they work, their strengths and weaknesses, and when to use which.

Challenges

One challenge of using spatial data structures in this context is the dynamic nature of the simulations. Organisms are stored based on their location, yet that location is constantly changing. This means that frequent updates to the structure of the data structures are needed. Depending on how often and how much the locations change this can become quite computationally expensive. Some data structures are affected more by this than others so the best data structure to use heavily depends on the context.

R-Tree

Our implementation largely follows the original definition of the R-Tree in the paper by Guttman et al. [1] So for a more in depth and technical explanation we refer to their paper.

Main Idea

In the r-tree, entries are characterised by their minimum bounding box (MBB). This is the smallest n-dimensional axis aligned rectangle the entry fits in. In our case an axis aligned box. Entries are stored in leaf nodes. Each leaf node then also has an MBB, in which all its entries fit. Similarly, leaf nodes are contained in branch nodes. These branch nodes can also contain other branch nodes and have an MBB in which their entries fit. Together this forms a tree structure which can be traversed based on the MBBs.

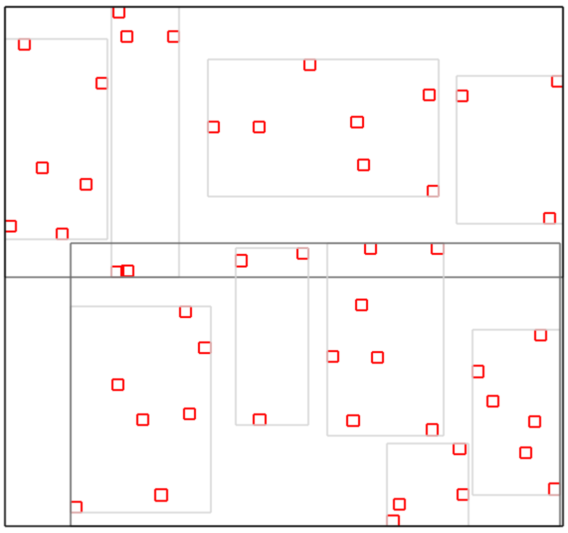

Figure 3: Example of R-Tree spatial partitioning, red squares correspond to leaf ***entries****. The black square is the root, branch nodes become greyer the higher their level is.*

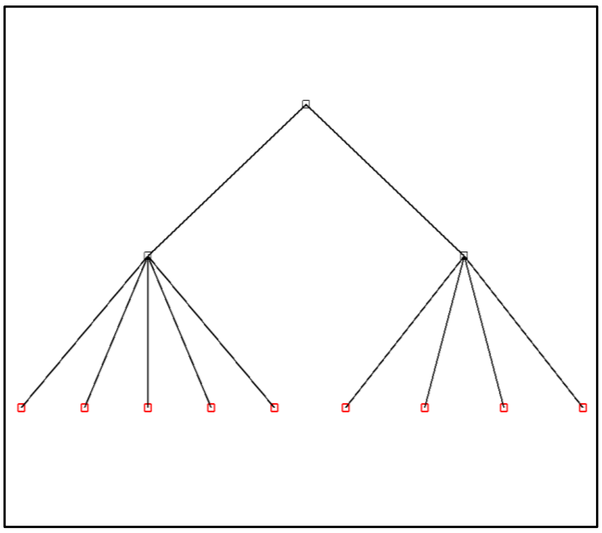

Figure 2: Corresponding graph representation of the same R-Tree, red squares correspond to leaf ***nodes***

Properties

- Each node contains at least m entries and at most M entries. M and m are configurable, and optimal values depend on the use case.
- The MBB of a node is the smallest rectangle that contains the rectangles of its entries.
- The root node has at least 2 entries unless it is a leaf node.
- All leaves appear on the same level, i.e. the tree is height balanced.

Operations and efficiency

In this subsection we will provide a short explanation on some of the different operations on the R-Tree, and their asymptotic complexity. These performance figures are not guaranteed, since there are situations where the tree might form a lot of overlap. However, in most real-world scenarios and especially in most simulations you can expect these levels of performance. For those not familiar with asymptotic complexity, in a nutshell, it provides a measure of how well an algorithm scales. For example: O(n) indicates computations time scales linearly with n and O(log n) indicates computations time scales with the logarithm of n. In this case n is the number of organisms in the data structure.

Search

Given a search area defined by an MBB, search returns all leaf entries whose MBB intersects the search area. The algorithm works by recursively traversing the tree structure. At a branch node it calls search on all child nodes whose MBB intersect the search area. At the leaf level it adds all leaf entries whose MBB intersect the search area to the results list. Calling search on the root node gives the desired result. The performance of this operation is O(log n) with relatively low overhead, so calling this for each organism every tick is fine.

Insert

Given a (leaf) entry, insert places it in the right leaf of the data structure. This is another recursive procedure. It traverses the tree and at each node it chooses to call insert on of its child nodes. The child whose MBB needs to be expanded the least to contain the new entry is chosen. If two nodes have equal expansion the node with the least children is chosen. Insert is relatively cheap most of the time. However, if a nodes maximum size is exceeded the node needs to be split. Since splitting the node introduces a new child in the parent node changes need to be propagated up the tree. More splits may occur, when the root is split a new root should be installed. For deciding how to split, the Linear Split algorithm [1] is used. So, insert is often quite cheap but can get more expensive. Exact figures are hard to provide, but on average this probably converges to O(log n). It is still considerably slower than search however since there is much more overhead.

Delete

Deletion is one of the most expensive operations on the R-Tree. Given a leaf entry, the right leaf entry needs to be found first. This works like a search operation. Then, since the leaf node might become smaller the bounding box needs to be recalculated. This is more expensive than a simple enlargement. Since the MBB could have gotten smaller changes need to be propagated upwards. Now the real problem lies in when the leaf node has less entries than the minimum amount after the deletion. The node should then be removed, and remaining entries need to be reinserted. Since the parent branch node could now also have too little entries, changes need to be propagated upwards. When ‘orphaned’ child nodes of branch nodes need to be reinserted they should be reinserted at the right height, because all leaf nodes should be at the same level. Solving these problems causes quite a lot of overhead. The delete often does not lead to problems and can be performed quite fast, but sometimes it can cause great amounts of reorganisation leading to longer computation times. On average, depending on the situation, it is often again O(log n) but with much overhead. This is probably the slowest of the basic operations on the R-Tree.

Update

According to the paper[1], updating a node requires first deleting the leaf entry and then inserting it again. This is quite expensive, so we have added some code to detect whether it is necessary. When a leaf entry moves outside the leaf nodes MBB this method can not be avoided. When the leaf entry moves inwards from the edge of the MBB, the leaf nodes MBB could require shrinking. The MBB must then be recalculated, and the change should be propagated upwards similar to delete, but without having to restructure the tree. When the leaf entry moves from any other place within the MBB to another place in the MBB no recalculations or propagations are required. So apart from having to find the corresponding leaf node (O(log n) with little overhead) in the final case no extra computations are required. Thus, performance heavily depends on how far and in what direction leaf entries are moved. Little movement only requires O(log n) with little overhead, and an occasional O(log n) with extreme overhead (Insertion and deletion). In situations with great amounts of movement, especially when directed away from other leaf entries, Update becomes the bottleneck and can get very slow. It must then often call insert and delete which are already quite slow by themselves. Regardless scaling is still O(log n), so despite having high overhead it still scales relatively well.

Nearest Neighbour (Search)

The paper by Guttman et al. does not provide a NNS algorithm so this implementation is mostly based on the paper by Nick Roussopoulos et al. [2]. Given a leaf entry E, Nearest Neighbour returns the nearest other leaf entry. Since MBBs in the R-Tree can overlap multiple paths of the tree need to be searched to ensure that the nearest neighbour is found. For a given node, the smallest possible distance to E, any of its children can have can be calculated, let's call it MINDIST. The algorithm uses this to prune possible paths. Nearest neighbour is called on the root and initiated with a nearest neighbour distance of infinity. At a given node, MINDIST is calculated for all its children, they are then sorted by their MINDIST and put in an active branch list. The active branch list is then iterated in order. If the current child nodes MINDIST is larger than the current nearest neighbour distance iteration can be stopped early so that branches are pruned. If this is not the case Nearest Neighbour is called on the child node. At the leaf level, the actual distance to all leaf entries is checked and if, for any of them, the distance is smaller than the (up to then found smallest) nearest neighbour distance it is updated and that leaf entry becomes the nearest Neighbour. When the recursion returns to the root the nearest neighbour has been found since al relevant branches have been checked. The performance of this algorithm is worse then Search since it is more complex so when one can get away with just using Search that is preferred. Nearest Neighbour has much overhead since it must sort many values and search multiple paths. Performance is largely dependent on the amount of overlap in the Tree, for well balanced cases it is again O(log n), with great overhead.

Strengths and weaknesses

Strengths

- World size is not limited, the R-Tree works with any world size and can efficiently store (groups of) entries, even when there is a lot of dead space between them.
- Nearest neighbour search range is not limited, when the nearest neighbour is very far away it will still (efficiently) be found.
- When entries move little, the R-Tree provides O(n log(n)) scaling with relatively little overhead.

Weaknesses

- In very dynamic contexts the R-Tree will have a lot of overhead, despite scaling with n log(n) it will probably perform quite poor.
- Scaling cannot get better than n log(n). When there is a high branching factor the base of the log is also high so in non-extreme situations it might still outperform O(n) solutions. However, there will always be a number n for which the O(n) implementation becomes faster than the O(n log n) implementation, no matter the amount of overhead. Where this tipping point lies is heavily dependent on the context, so it is best to test which implementation works best for a desired number of organisms.

Chunks

Our implementations of chunk data structures work with a fixed grid of square/cube chunks (depending on 2D or 3D implementation). Chunks are a very typical data structure technique and our work almost exactly as one would expect, with one exception. The chunks have two sizes, the actual size as you would expect and a second larger size which overlaps with other chunks. This second size is used to keep track of all organisms in other chunks which could possibly collide with an organism in its own chunk. With this, we do not have to check neighbouring chunks on collision, which is often done for chunk implementations.

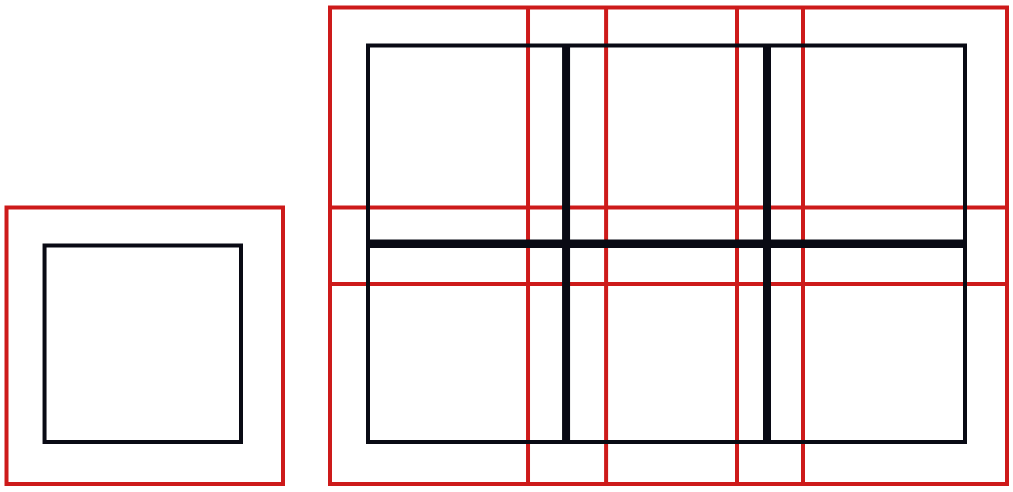

Figure 4: A group of extended chunks to visualize overlapping

Multithreading

Our chunk implementations have a multithreaded variant. This means that each logical core on the CPU can control a set of chunks and runs simultaneously with all other logical cores. How these chunks are divided over a logical is very important. Say we have an organism in chunk A and another organism in chunk B which neighbours chunk A. If both chunks run simultaneously, then it is possible for chunk A to check for collisions for its organism, when it does not find any collisions within its own chunk, it will check neighbouring chunks, but when it looks for chunk B, it is gone because the organism in chunk B could have moved towards chunk A after chunk A did its original check, but before it got to finish its check of chunk B. To solve this problem, we must make sure no neighbouring chunks ever run simultaneously. Our solution to this is to group the chunks into a few sets, where each set is denoted with a letter. For 2D the pattern is visualized below, for 3D we use the same pattern but extended to 3D dimensions (so with 8 total sets).

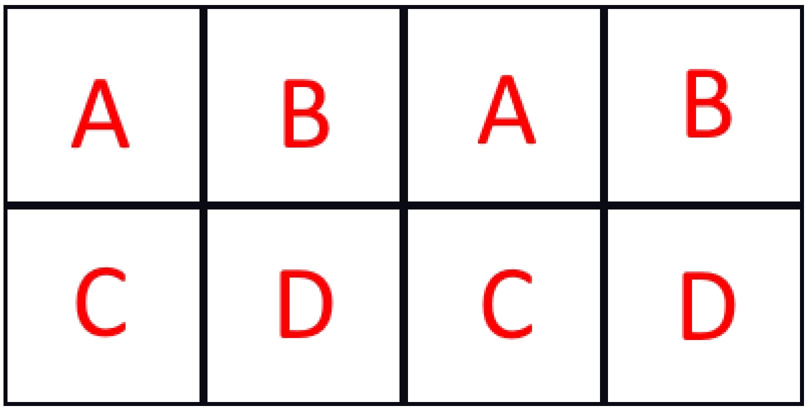

Figure 5 Example of sets of chunks for multithreading in 2D, chunks with the same letter get called at the same simultaneously.

Strengths and weaknesses

Strengths

- The biggest strength of the chunk-based data structure is its asymptotic complexity, which is O(n). This means that is scales very well with larger simulations. Within a chunk computations are O(m^2), but since m is limited to the maximum number of organisms that can be contained in a chunk, this does not affect the overall asymptotic complexity, since it can be interpreted as a large constant when the data structure becomes full.
- When the organisms are evenly distributed, chunk based implementations perform especially well, since the earlier mentioned 'm' stays as low as it can be.

Weaknesses

- The Nearest neighbour search implementation will not search further than its neighbouring chunks. So, when it is necessary to know an organism's nearest neighbour even when it is far away, this data structure should not be used.

- When organisms are clumped together in groups further away from each other, this data structure works less efficiently, as the earlier mentioned ‘m’ will be very large. A data structure that uses dynamic spatial partitioning like the R-Tree can index the organisms more efficiently in situations like this.

Future work

This framework handles large scale microbial simulations in a fast and efficient way. It allows users to easily set initial conditions and add custom behaviour to different organisms. Continuum also includes a way to visually display a simulation in 3D space. The framework has delivered on its original promise. But that doesn’t mean there aren’t any improvements that can be made.

Semi-Adjusting BSP-Tree

In Continuum we have implemented two data structures, the R-Tree and the hash chunks. As future work, we propose a semi-adjusting BSP-Tree as described by Luque et al. [3] to improve spatial partitioning for the microbial simulations. Unlike fully dynamic trees that continuously rebalance, this structure delays rebalancing, reducing computational overhead while maintaining efficient space partitioning. The tree partitions space using a set of pre-defined directions, selecting partition planes based on a criterion that balances organism distribution and minimizes redundancy. Leaves contain ordered lists of organisms relative to their parent's partition plane, allowing efficient access patterns. The data structure supports several operations—such as split, shift-split, merge, swap, and balance—to adapt the tree as organisms move or interact. This approach could offer a higher performance as mentioned in the paper. This data structure could require a lot off testing as it has many parameters that need tuning. This needs to be taken into account when developing the data structure. The tree’s responsiveness to the simulation's dynamics makes it a strong candidate for future integration into Continuum’s core.

Fix random for multithreading

As mentioned earlier in the user guide, setting a seed for simulation while using a multithreaded data structure will lead to errors and incorrect results. This is due to Microsoft’s System.Random not being usable in a concurrent environment. We have a fix that would work but did not have the time to implement. A custom Random class should be made that has a function that requires the organisms position as an input and combines it with the current tick to get a singular value (an uint or ulong) which then should be used in a LFSR (Linear-feedback shift register) to pseudo-randomize the result, which can then be used to get a random value in a range requested by a method.

Runtime-variable size

A currently unsupported feature is allowing the size of organisms to grow while the simulation is running. This would not only break data structures, but it would also have to check if it would not collide with other organisms due to the size growth. It is not too hard to do, but it does require extra time that we did not have.

Extra precision

The simulation framework currently uses floating points for all of its calculations. Doubles have more precision while only lowering performance by a minimal amount. We forgot to use doubles as floating points are in almost all situations enough, but specifically for simulations it is an added bonus to have double point precision. In all current example use-cases doubles are not needed for their precision, but it would be a nice small addition to have that final small boost in accuracy. To do this, simply convert every float into a double and replace all Vector2s and Vector3s with a new struct Double2 and Double3.

References

[1] Guttman, A. (1984, June). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (pp. 47-57).

[2] Roussopoulos, N., Kelley, S., & Vincent, F. (1995, May). Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data* (pp. 71-79).

[3] Luque, R. G., Comba, J. L., & Freitas, C. M. (2005, April). Broad-phase collision detection using semi-adjusting BSP-trees. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games* (pp. 179-186).