Spatial Partitions

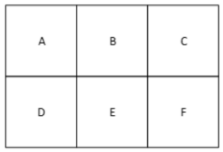
jodavis42@gmail.com

What is a Spatial Partition?

A data structure to quickly prune objects

Typically achieved through algorithmic complexity reduction

Grid Example:



Objects in a room can only collide with objects in nearby rooms

No optimization is better than not doing something!

*Spatial partitions return false positives

A spatial partition is a data structure that sub-divides space to quickly prune queries against objects. This is often achieved (although not always) through algorithmic complexity reduction. Normally, finding what objects intersect with each other is an $O(n^2)$ operation while most spatial partitions strive for $O\big(nlog(n)\big)$ or less. Spatial partitions are built upon the principle of: no optimization is better than not doing something. If we can do a little work (the less the better) to remove a lot of work then a spatial partition is “good”.

One of the simplest conceptual examples is a uniform grid. Think of each grid cell as a room in a building. An object in one room can only possibly intersect objects in that same room or neighboring rooms. This means objects in cell A and F don’t even have to check each other when they are far away.

One very important note about spatial partitions is that you should view their results as possible intersections. Spatial partitions are allowed to determine false positives, that is they can safely determine that two objects do not intersect but cannot determine if two objects intersect. This is typically left for a later part of the pipeline known as narrow-phase.

Spatial Partition Classifications

There's two main kinds of spatial partitions:

1. Object independent (Grids)
2. Object dependent (BVH)

Both have their pros and cons.

Can anyone list some?

Before delving into any specifics we'll first establish the two main kinds of spatial partitions. I haven't seen any official names for these so bear with my made-up names.

First we have object independent spatial partitions. These spatial partitions typically sub-divide space independently of where any objects are. Objects are then inserted into cell(s) based upon where they are in space. Some common examples are: Uniform grids, H-Grids, and Quad/Oct-trees.

Second we have object dependent spatial partitions. These spatial partitions sub-divide space depending on where objects are. As objects are inserted/removed/updated the space that is represented constantly changes. Most typically, the objects themselves represent a "cell" in this kind of spatial partition. Common examples include: SAP and Bounding Volume Hierarchies.

Both of these spatial partition types have their pros and cons so it's important to understand at a high level what issues can arise from them.
A con for each of these types is dealing with object boundaries. If an object straddles two grid cells in an object independent spatial partition how do we deal with it?

Similarly if two objects overlap with an object dependent spatial partition how do we handle multiple cells overlapping?

Spatial Partition Considerations

Unfortunately, there is no “best” spatial partition

Common evaluation metrics:

- Insertion/Removal/Update Cost
- Cost of queries: (Pair tests, ray casts, frustum casts, etc…)
- Tight Fitting
- Memory Requirements

Unfortunately, this is another part of the class where there is no right answer. No spatial partition is best and almost every single one can be “best” in certain situations. It’s important to understand what goes into making a spatial partition in order to properly evaluate for a given scenario how well one spatial partition performs against another. There are several common metrics used:

The first metric is the cost of inserting removing, and updating objects. This should go without saying that if it takes an extreme amount of time to build or update a spatial partition it might not be fit for real-time scenarios. And yes, it’s easy to make something on the order of $O(n^3)$ that will do very well at our other metrics (Static Trees). Since we want to focus on real-time applications, removal and update are also important to understand. Most commonly, a spatial partition’s update will just be a remove followed by an insert, but sometimes a more efficient update can be implemented.

Next we have the cost to perform any query. The 3 main kinds of queries done on spatial partitions are self-pair tests, ray casts, and other kinds of casts. Self-pair tests are most often performed by physics when needing to know what objects in a collection could possibly intersect. Ray casts are needed by all sorts of applications,

especially gameplay, and they need to know what objects are hit by a ray and in what order (time). Finally, other kinds of casts are sometimes needed such as a frustum cast. Frustum casts are most commonly used by graphics for frustum culling. Other kinds of casts are often useful for gameplay, such as casting a box in front of a player to determine if a switch is there.

Things get tricky with the previous two metrics if we add in how tight fitting a spatial partition is. Ideally a spatial partition will return as few false positives possible by having the most tight-fitting space representation. Becoming more tight fitting often increases the cost of both queries and insertion. At the opposite end you can have such a loose fit spatial partition that everything collides, effectively pruning nothing. This is affected not only by how much time is spent on making a specific bounding volume as tight fit as possible, but on what bounding volume you use. From my personal experience, a cheap to compute bounding volume (such as an aabb) tends to perform best. This is because the cost to build a tight-fit aabb is cheap and the cost of intersection is low.

Finally, memory requirements are always important. Cache coherency is often affected by how big a spatial partition has to grow. Also, some systems have surprisingly low amounts of memory still, such as 64MB.

The cost of queries are obviously important as that is the whole reason we are using a spatial partition: to more efficiently reduce object tests than brute forcing them. If it took more time to find pairs with a spatial partition than with directly testing them then the spatial partition is of no use.

Collision Detection Phases

1. Broad-phase
2. Mid-phase
3. Narrow-phase

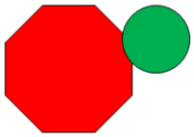
Before discussing some basics of spatial partitions it's important to understand the 3 phases of the collision detection pipeline. It's important to understand the roles of these phases, such as where spatial partitions are used and how to use them. These phases don't always explicitly exist, but should all be considered. Note: I'm using physics terminology for these phases.

Listed above are the update loop order of these phases. Instead of this order, I will go over them in detail in the order that I recommend implementing them.

Narrow-Phase

Determines if two primitives intersect

- Often creates contact information
- Don't return false positives



Most expensive phase, avoid where possible

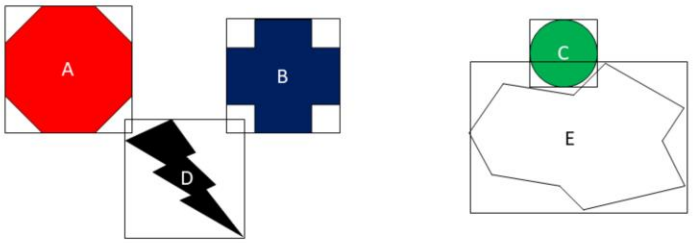
Narrow-phase is the final phase that determines if two geometry primitives intersect. This phase should never return false positives as it's the last phase in the pipeline. Sometimes these tests just return Boolean information, but some applications (such as physics) also need information about how the objects are intersecting. This information typically includes: contact normal, points of contact and penetration depth.

This phase typically takes in two primitives (or a list of primitive pairs) and returns intersection information for each pair that intersects. It's important to understand my definition of this phase as two primitives. If a primitive is a triangle then a mesh of triangles does not count as a primitive (we'll deal with that in another phase).

As this phase must return intersection information with certainty, it is the most expensive phase. We want to avoid sending as many pairs to this phase as possible, hence why the previous two phases exist. Note that this phase should be the first that you implement as this one is always required and the other two are purely for optimizations!

Broad-Phase

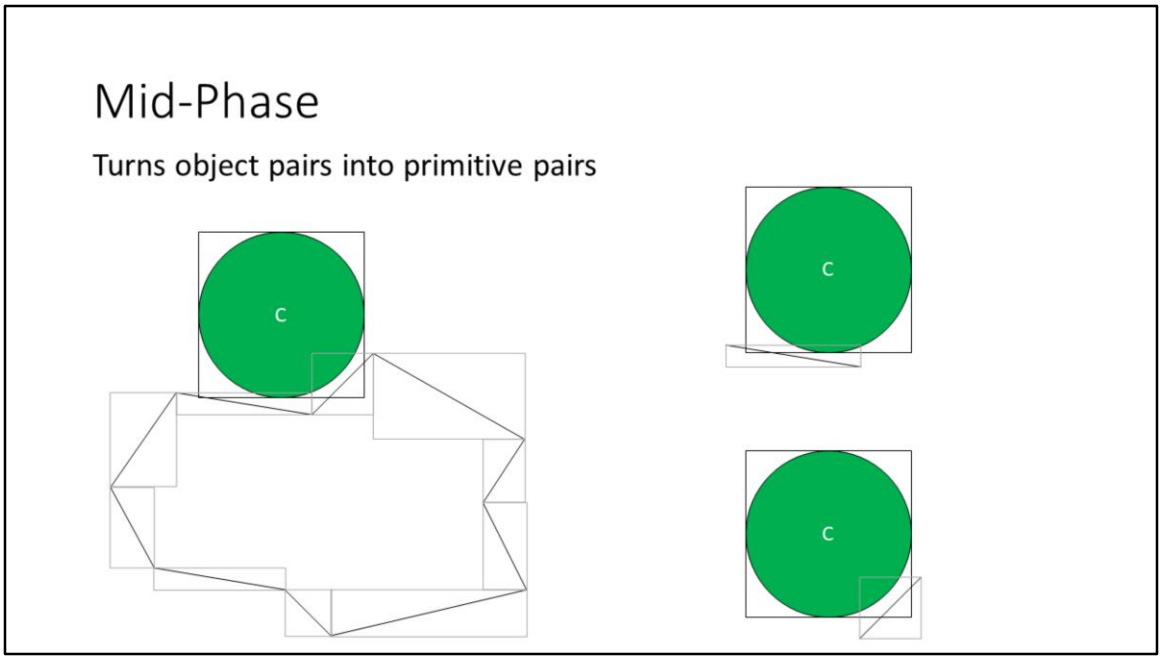
Results in pairs of objects that potentially intersect.



Above produces the pairs: [A-D], [B-D], [C-E]

Broad-phase takes in a collection of object and produces a reduced set of pairs of objects that could intersect. This is typically achieved through the use of a spatial partition. Broad-phase is also used to prune any other kind of query needed, such as ray-casts. Remember, broad-phase produces potential pairs that are not necessarily intersecting.

To further emphasize the distinction, broad-phase only takes objects in a scene and produces object pairs, not necessarily primitive pairs (which narrow-phase needs). A broad-phase is typically stored on a space. It's common practice to store more than one spatial partition in a broad-phase to deal with objects of different access patterns, such as dynamic and static objects in a scene.



Mid-phase is the least common explicit phase in a pipeline. Most people rarely conceptualize it even if they have one. At the end of broad-phase we have pairs of objects that we need to send to narrow-phase, but narrow-phase only accepts pairs of primitives. What if an object is composed of multiple primitives? This is the role of a mid-phase, to turn an object pair into zero or more primitive pairs.

In the above example we have a mesh against a sphere. We could test every triangle in the mesh against the sphere in narrow-phase, but it should be easy to see that most of these triangle tests shouldn't be needed. Using a simple mid-phase where each triangle is represented by an aabb, we can filter the possible primitive pairs down to two based upon the nearby triangles.

It's worth noting that we could avoid the mid-phase problem by changing our broadphase to have all primitives inserted instead of objects. This would avoid the need of a mid-phase, however this would also bloat a broad-phase and make all tests take longer.

The most common object complex enough to require a mid-phase is a mesh. Commonly mid-phases are achieved through the use of a spatial partition, but unlike

broad-phase, each “object” tends to have its own mid-phase. As a mid-phase can be expensive to compute and take up a bit of memory, it’s often stored in the local space of an object along with its mesh. All operations are then transformed into the object’s local space to test against the pre-built mid-phase.

When I talk about spatial partitions throughout the rest of the class I’ll mostly be focusing on an interface for broad-phases, however a properly written spatial partition can also be used for a mid-phase.

Collision Detection Phases Recap

1. Broad-Phase: All Objects -> Pairs of Objects
2. Mid-Phase: Pairs of Objects -> Pairs of Primitives
3. Narrow-Phase: Pairs of Primitives -> Intersecting Primitive Pairs

*Mid and Narrow-Phase typically happen at the same time

To recap the 3 phases:
Broad-phase takes in all objects in a scene which are then reduced to potentially intersecting pairs of objects.
Mid-phase takes these pairs of objects and turns them into pairs of primitives as necessary. Note that some pairs will entirely skip mid-phase as they are already primitive pairs.
Finally, narrow-phase takes the pairs of primitives and determines what ones are actually intersecting, often producing intersection data for each pair.

In typical implementations, broad-phase happens as a standalone phase first where all pairs are found. The mid and narrow-phase tend to happen at the same time though. Each object pair is tested and the mid-phase either directly calls the associated geometry test for the primitives or it calls the mid-phase and then immediately calls the narrow phase for those primitives.

Spatial Partition Interface Basics

How should our main functions (insert/update/remove) work?

A typical student interface look like:

```
class SpatialPartition
{
  void Insert(Collider* collider);
  void Update(Collider* collider);
  void Remove(Collider* collider);

  void SelfQuery(ColliderPairArray& results);
  void RayCast(Ray& ray, ColliderRayCastArray& results);
};
```

First look at insert, any problems?

Before going into two simple spatial partitions we first need to look at the basics of our interface to help establish good practices, also we'll get to an interface similar to what you'll have for assignments. Pictured above is a simplification of a typical student interface. We insert/update/remove colliders (or whatever your collision object is) into the spatial partition and queries return arrays of them (or arrays of pairs).

To start let's look at insert/update/remove as they're the first thing we have to do. For simplicity sake let's first look at the insert function. Currently we just take a collider to insert, nice and simple. Are there any problems with this implementation? (This one should be easy!)

Spatial Partition Interface Basics

Why does our spatial partition only work for physics?

Spatial partitions should not care about who's using them!

What data do we insert?

The bounding volume! (Assume aabb for examples)

```
void Insert(const Aabb& aabb);
```

Problems?

The most obvious problem is that we've hardcoded the spatial partition to only work for physics. Spatial partitions tend to be application agnostic, so a spatial partition for physics can just as easily be a spatial partition for graphics. The problem is if we don't pass in a collider as our information what do we pass in? Well most spatial partitions work on a bounding volume so how about we just pass that in? For simplicity sake I'll just assume this is an aabb and we'll deal with this issue later.

So now we have a new interface that takes in the bounding volume we're inserting. We're nice and application agnostic now, but what problems are still in this interface?

Spatial Partition Interface Basics

What do intersection tests return now? Aabbs?

Physics wanted colliders…

We need to take some identifier for what the user is inserting!

```
void Insert(const Aabb& aabb, void* clientData);
```

Now physics, graphics, etc… work as long as we're given an aabb

Any problems now?

So the question now is, what do our intersection tests return. Previously they would return colliders but we can't do that anymore because we were never given any. We could return aabb pointers but that's not very helpful. For example, physics would have to store a hashmap of aabb pointers to colliders or something. We need to give the client a way to add some data along with what they're inserting. This is achieved simply by adding a void* of client data that the spatial partition will store when inserting. Now the spatial partition can be used for all sorts of different applications!

Now that we can be have data generically inserted into us that we can properly return, what problems could be left if any?

Spatial Partition Interface Basics

Insertion doesn't have any problems by itself, what about removal?

```
void Remove(const Aabb& aabb, void* clientData);
```

Well, the aabb isn't needed which is easy to fix:

```
void Remove(void* clientData);
```

Any more problems?

If we look at insertion by itself there's not really any problems so let's continue and look at removal. If we give removal the same interface as insertion the obvious problem is that we're passing in an aabb that's probably not helpful for removing. This is easy to fix though by just realizing that the bounding volume isn't necessary for removal and just not passing it in.

Now are there any problems?

Spatial Partition Interface Basics

How does a spatial partition find the data to remove?

For example, how does a tree find the node?

- Searching the entire tree?
- HashMap of client data to nodes?

Up to each spatial partition to decide

Any problems left?

Well if we think about the internals a bit more we should realize that on removal, a spatial partition will need to find whatever it allocated internally from the insert where it was handed the client data and remove it. The problem is, how does the spatial partition find this data?

One of the better examples to look at is a tree based spatial partition. How do we find the node allocated for this client data? Do we search through all nodes? Removal shouldn't take $O\big(nlog(n)\big)$ just to find the node! We could always have a map of client data to nodes or some other direct mapping scheme.
What about a grid? Sweep and Prune? Quad-Tree? Etc… Well, it's up to each spatial partition to determine how to efficiently map this data over.

Once again, are there any problems? This is the trickiest one to really come up with on your own so I'm not expecting much.

Spatial Partition Interface Basics

Why did we switch to the void*?

- To be application agnostic!
- Is it valid to assume any knowledge about this data?

We're assuming each insertion has a unique client data!

Why would a user ever do this?

- What if the user doesn't care about the client data (just raycasting)?
- What if the same object is inserted more than once (concave objects)?
- Etc…

How do we fix removal then?

There's a fundamental problem with the previous approach which becomes apparent if we look at why we even switched to the client data pointer. We did this so we could be application agnostic, that is we only care about the bounding volume being inserted and we simply added the client data so the user can get back their information.

So this begs the question, is it valid for us to assume anything about the client data? What if there's more than one insertion that has the same client data? Our spatial partition would break horribly because we assume this isn't the case. Shouldn't we actually assume that each call to Insert is unique? You might say, "why would anyone ever insert the same client data twice?" Well to give two examples I've run across personally:

1. What if the user doesn't actually care about the user data? What if they're building a spatial partition so they can efficiently perform a raycast to just get a t-value (for a mid-phase). In this case they have to assign unique ids to each insertion even though they will never use the data they're inserting…
2. What if the same object is being inserted more than once? What if for a specific application it's more efficient to break a larger object into multiple pieces, each with their own bounding volumes, to be inserted into the same spatial partition

more than once.

These are just the simplest reasons for why our spatial partition should never look at the client data. It should do nothing more than store and return it. This begs the question then, if we can't use the client data to find what object to remove, how do we remove anything?

Spatial Partition Interface Basics

Add more data: an intrusive key

```
void Insert(const Aabb& aabb, void* clientData, Key& key);
void Update(const Aabb& aabb, void* clientData, Key& key);
void Remove(Key& key);
```

What is this key?
Data that the spatial partition needs to efficiently perform update/removal
The client should simply hold onto it and pass it back for update/remove

Any problems left?

The only real place to go from here is to add more data that we can use to uniquely identify an object. Instead of making the user pass in a unique key for each insertion though, we'll assume that each insertion is unique and give the user a key to use in subsequent operations. That is, we are forcing the user to intrusively store data from an insert that they need to pass in when they remove/update the object. If the user doesn't care to remove/update then they don't need to store it.

So what is this key? Well, the client shouldn't care what's in this data, they should simply store it and pass it back in to update/remove objects. What does the spatial partition put in the key? Well it depends on the spatial partition. It should be whatever that spatial partition needs to efficiently find an object for removal/update. For example, a tree might store the node pointer.

With this we now have a clean divide for a spatial partition to be generic, give the client back what it needs from queries, and efficiently perform updates and removals. So are there any problems left?

Spatial Partition Interface Basics

Only real problem left is dealing with other bounding volumes

Easy solution, take a structure:

```
class Data
{
  Aabb mAabb;
  Sphere mSphere;
  void* mClientData;
};
```

```
class SpatialPartition
{
  void Insert(const Data& data, Key& key);
  void Update(const Data& data, Key& key);
  void Remove(Key& key);
};
```

Simplifies some issues with inheritance

What about the query functions?

When it comes to insert/update/remove there's no real problems left. The one thing you could nitpick is what if we need a spatial partition other than an aabb? This becomes tricky if you want to adhere to a virtual interface. One solution (the one in your framework) is to just take a structure that has all common bounding volumes used for a spatial partition. I don't have a great solution to this one as C++ is a bit limited in how virtual works. The best "generic" solution would be to take a "shape" that aabb derives from, but this has performance and other implications. For now it's easiest to just take a struct.

Spatial Partition Interface Basics

What about queries?

For now a super simple interface.

```
void SelfQuery(QueryResultsArray& results);
void RayCast(const Ray& ray, SortedCastResultsArray& results);
```

This will be (roughly) the interface in your framework

More advanced interfaces will be described later in the semester

I won't go into much detail here about the query functions as the basic interface is good enough to not warrant further discussion for a while. I'll go into more details when I talk about spatial partition extensions towards the end of the semester. For now, each one simply fills out an array of the relevant information.
SelfQueries fill out an array of client data pairs.
RayCasts fill out a (sorted) array of a struct with the client data and a t-value.

You'll see some extra functions in class, such as frustum casting and debug drawing, but they're also obvious enough to not warrant any further discussion for now.

Basic Spatial Partitions

Two most basic:

1. N-Squared
2. Bounding Volume N-Squared

Now I'm going to go into the two most basic (implementation wise) spatial partitions that there are. These are mostly to get you comfortable with the idea of implementing a spatial partition and.

N-Squared

Simplest (and worst) spatial partition
- Does no pruning, returns all objects

Why use this?
- Optimizations should come last
- Use as a placeholder stub when first starting
- Also some fancy extensions later in the semester

*If anyone uses this in their final game I will be sad!

The n-squared bounding volume is the most basic spatial partition one could conceive of, however most people will probably not really consider it to be a spatial partition. The main reason is that it doesn't actually improve anything.
What does this look like? Well insertion simply inserts into an array, casts return every object, and self query just returns all n^2 pairs of objects, performing no checks whatsoever.
So this begs the question, why would I ever talk about this?

The first reason is that a spatial partition is an optimization. Optimizations aren't important until you have things working, so creating a complex spatial partition before you even have the rest of the engine working is somewhat backwards. However, a spatial partition can play a key role in the structure of a system. Creating an n-squared spatial partition allows you to play around structurally with a spatial partition before you're actually at the stages to optimize. It also lets you consolidate all code ahead of time. For example, if you have to check for collisions you'll write a double for loop somewhere, it might as well be behind a spatial partition interface so you can easily replace it later.

The second reason is mostlyfor logical understanding. We'll see quite a few places

come up later where you have an n-squared spatial partition implicitly defined even though you don't realize it.

The final reason is for comparisons. We'll see that reason towards the end of the semester when we cover spatial partition extensions.

All this to say that the n-squared spatial partition is a great starting point to wait until optimizations are needed, however I will be very sad if any of you ship a game that uses this for collision detection as it's trivial to do a little more work for a lot of gain.

Bounding Volume N-Squared

Simplest reasonable spatial partition:

- N-Squared, but first checks a bounding volume
- Relies on no algorithmic complexity, just intersection test complexity
- Works fine until around 100 or so objects
- Any bounding volume works

*I will not be sad if you have this in your game

To improve the n-squared spatial partition significantly with just a little more work we can just add a bounding volume. While the n-squared spatial partition would return every object during a ray-cast, this will first check the stored bounding volume (e.g. a sphere) against the ray and only add the object if this test passes.

This spatial partition doesn't reduce the algorithmic complexity of any tests, but it instead reduces the cost of each test. For example, a scene with a lot of meshes would be really expensive to test in an n^2 fashion. If we can first reduce a lot of those tests by checking a bounding sphere we can save a lot of time. This does start to break down at a certain point though as n^2 of a small number can still grow very fast. Typical numbers I've seen (I haven't tested recently) is that this works fine for around 100 objects. Also any bounding volume can be used: spheres, aabbs, k-dops, etc…

This spatial partition is such a large improvement on the pure n-squared that it's silly to not do. If you ship a game with this because it's all your game needed then I will not be sad, it's a reasonable one to use.

Questions?