Welcome, and thanks everyone for coming!
When you multi thread your game code, there are global systems that game objects need to talk to. These systems are often protected by a mutex and this can cause stalls as multiple threads are waiting for the lock. At Guerrilla the physics engine was one of the major sources of lock contention.
See, most physics engines are good at simulating 1000s of objects, and they scale that work across many CPUs, but when it comes to integrating with game code, only one thread can touch them at a time.
In 2014 I started writing my own physics engine, Jolt Physics, in my spare time. In May last year, it had gotten to a point where it implemented everything that Horizon was using. I slapped a big #ifdef in our physics wrapper so that we could run both engines. Jolt Physics had been designed to overcome the lock contention, but, to my surprise, not only the waits were gone, but the performance was much better, it used less memory and the simulation quality was better too.

My name is Jorrit Rouwe, I'm lead of the Game Tech Team. I've worked at Guerrilla for approximately 21 years, starting at the very first demo of Killzone 1. Today I will be presenting you how I architected Jolt Physics for Horizon Forbidden West and got rid of those waits.

CONTENTS

- Typical Physics Update
- Waits
  - Streaming Open World
  - Game Object Update
  - Background Queries
- Solutions
  - Lock free broad phase
  - Lock free island building
- Conclusion

2

In this talk we will first do a quick recap of what a physics simulation update looks like, to establish the terms used in this presentation.
After that we will go through 3 different scenarios that caused our code to wait for the physics engine:

1. How we stream in data
2. How we update our game objects
3. How we build nav meshes in the background

When we have established the waits, we will focus on 2 areas where Jolt Physics differs from other physics engines to get rid of the waits:

1. We will look at a mostly lock free broad phase to allow concurrent query and modification
2. We will look at a lock free island building approach that minimizes shared state between bodies

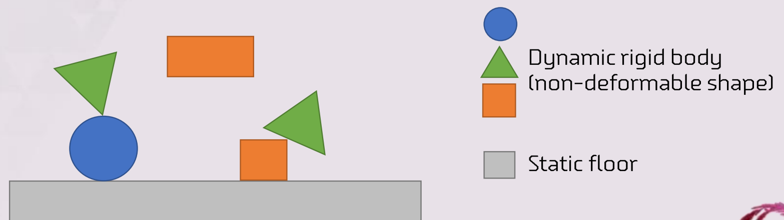
Finally, we will share some of the results of switching to the new physics engine, show that it improved performance, memory and simulation quality all at once.

CONTENTS

- Typical Physics Update
- Waits
  - Streaming Open World
  - Game Object Update
  - Background Queries
- Solutions
  - Lock free broad phase
  - Lock free island building
- Conclusion

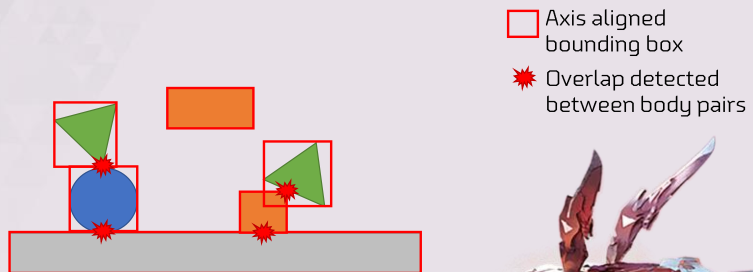
We will first look at a typical physics update.

THE SCENE



We’re going to explain the physics update with a very simple scene. The scene consists of a couple of primitive shapes (triangle, sphere, box) falling onto the floor. Each shape is what is called a ‘rigid body’, a non-deformable shape that can collide and react to other shapes.

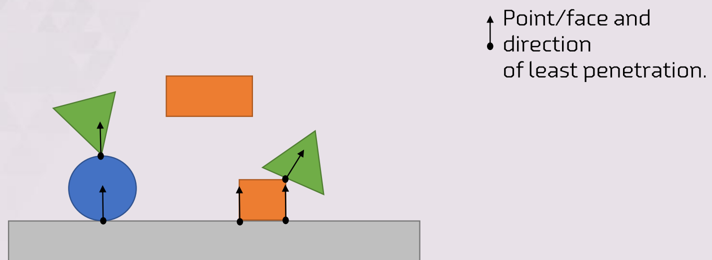
BROAD PHASE COLLISION DETECTION



5

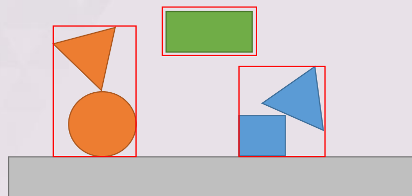
The first step in a physics update is always ‘broad phase collision detection’. Usually, each rigid body has an associated bounding volume (often an axis aligned bounding box). The broad phase is an acceleration structure that allows us to quickly detect overlapping objects. The acceleration structure is often a tree, but other solutions are also possible. In our example you can see which bounding boxes have been found to overlap (called body pairs).

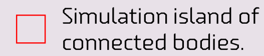
NARROW PHASE COLLISION DETECTION



After the colliding body pairs have been found, we continue to the narrow phase collision detection. At this stage, each body pair will be checked for collision between the actual shapes. When a collision has been found, we calculate a contact point (or polygon as in the case of the orange box on the right) and a direction. This direction is chosen in such a way that it represents the direction along which to the objects can be pushed apart over the shortest distance possible.

GENERATE ISLANDS



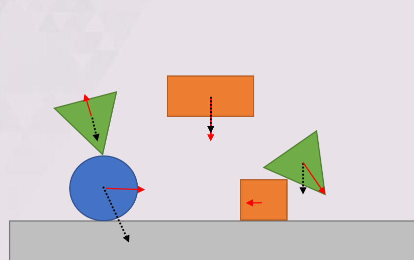


Simulation island of connected bodies.

Floor cannot move so not part of island.

After the contact points have been determined we need to build simulation islands. Simulation islands are groups of connected bodies, be it through a contact point or through a constraint. These bodies will influence each other during the simulation as a change in velocity of one body causes a change in velocity of the other body. Each simulation island can be solved as a separate job on the CPU. Note that since the floor doesn't move, it cannot change velocity and since it cannot change velocity it doesn't need to be part of a simulation island. In this example we now have 3 simulation islands.

SOLVE (CONTACT) CONSTRAINTS



- Velocity at beginning of step
- Velocity after gravity and constraints solved

- Jolt uses Sequential Impulse Solver
- See Erin Catto at GDC 2009

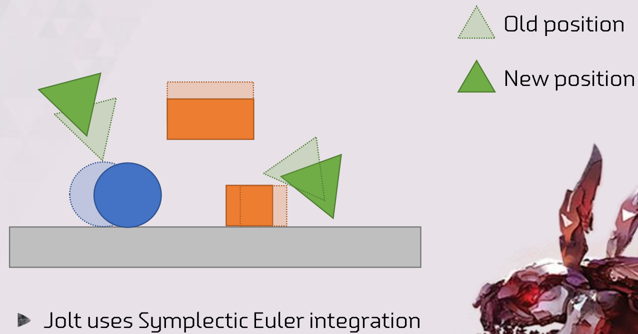
Jolt Physics

=

Box 3D

The next step in a physics simulation is to update the velocities of each of the bodies. First the velocities of all bodies will be updated by the acceleration due to gravity, then all constraints are solved. I've drawn a velocity before gravity and contact solving in black and a possible velocity after solving in red. Note that this is just one of many possible results because the results depend on things like friction and restitution. Constraint solving is a complex mathematical problem which we will not talk about today, there are many GDC presentations that have covered this. Jolt Physics is using the Sequential Impulse Solver as described by Erin Catto at GDC 2009. You can see Jolt Physics as Box 3D.

INTEGRATE POSITION: VELOCITY · DELTA TIME



The final step in a physics simulation is to update the position and rotation of all bodies. Simply said, we can multiply the velocity by the delta time to get the delta position. In the image the new positions of all bodies have been drawn as solids, their old positions as dashed shapes. Note that there are different ways in which you can integrate positions, Jolt uses Symplectic Euler, but this topic is out of scope for this talk.

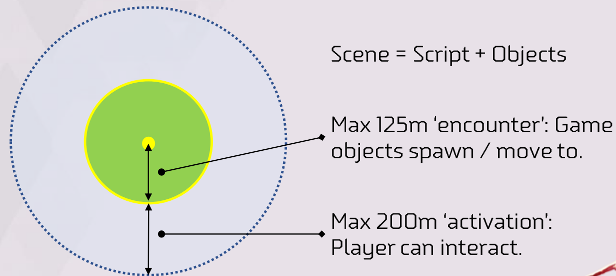
CONTENTS

- The Physics Update
- Waits
  - Streaming Open World
  - Game Object Update
  - Background Queries
- Solutions
  - Lock free broad phase
  - Lock free island building
- Conclusion

10

Now that we've got a high-level idea of the physics update, we will start looking at the requirements that HFW has on a physics engine. We'll start first by looking at streaming. We will show that streaming requires adding and removing lots of objects in as little time as possible.

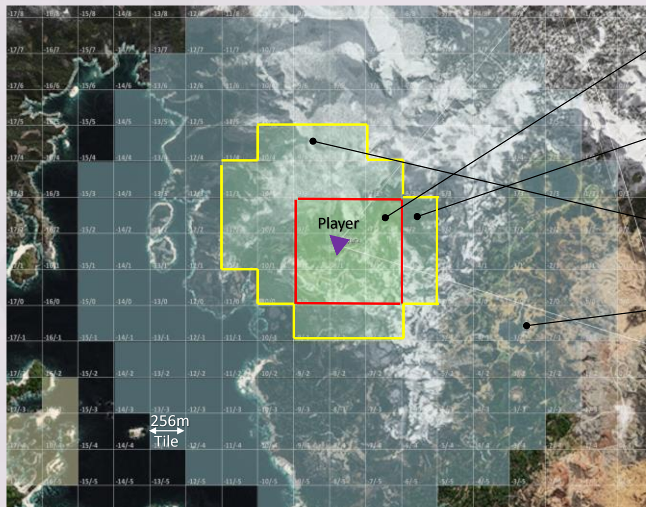
OPEN WORLD - SCENES



11

The main way content is added in the world is through Scenes. A Scene consists of a game script that runs the logic and a set of objects. These objects can be geometry, spawn points for game objects or other objects like collision triggers to detect when the player enters an area.
The Scene consists of a circular encounter area that has a radius of max 125 m. Within this area the Scene script can freely spawn game objects and move them around. When a Scene is active, we guarantee that all collision under the encounter area is loaded.
Around the encounter area, there is a radius of max 200 m. If the player is within this radius, the Scene should activate because the player could interact with it. A game designer can tweak both radii for smaller Scenes.

OPEN WORLD - TILES



9 high LOD:
- Full graphics
- Full collision

19 medium LOD:
- Medium graphics
- Full collision

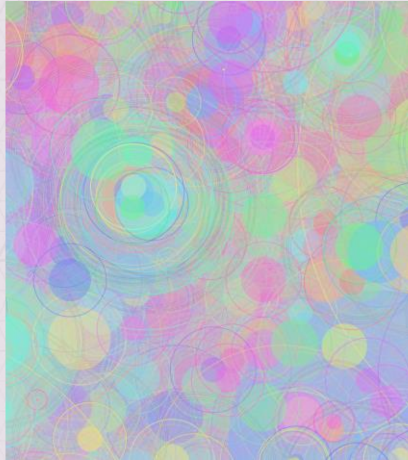
Extra row to cover loading time

Low LOD:
- Low graphics
- No collision

12

Now that we know what a Scene is we talk about streaming tiles. The streaming tiles are 256 x 256 meters. We keep a 3x3 grid of tiles at maximum detail. When the player moves over the tile border, we update the set of active tiles. Around the high detail tiles we have a set of medium LOD tiles which have medium detail baked out version of the graphics and a full detail baked out version of the collision. This is to ensure that scenes that the player can interact with always have collision under their encounter area (remember a tile is only 256 meter but a scene can be active from 450 meter which is twice the encounter radius + the activation radius). We keep an extra row of medium tiles loaded in the direction where the player is moving to cover loading times. Beyond the medium LOD tiles we have low LOD tiles which contain the low LOD graphics but no collision information.

OPEN WORLD - REQUIREMENTS



- 10s of Scenes
- 28 Tiles
- 20K bodies
- Constantly changing
- Situation: Block main thread 5-10 ms
- Want: Insert on background thread!

13

The image shows a debug view of the Scenes of a section of the world. As you can see, we have 10s of Scenes and 28 tiles active at any time. In those Scenes and Tiles there are approximately 20K bodies. When the player walks around, these Scenes and Tiles are constantly changing. Because of this a lot of objects need to be added to and removed from the broad phase. Our previous solution required a full lock when this happened which would first require waiting until other threads were done using the physics world and which would then block all other threads from doing useful work. Adding objects to the world used to be very expensive since it required the broad phase tree to be rebuilt while the physics world was locked. This often caused 5-10 ms stalls that would result in frame rate hiccups. What we want from our physics engine is the ability to do most of the work on a background thread so that there is as little interference as possible with the running game.

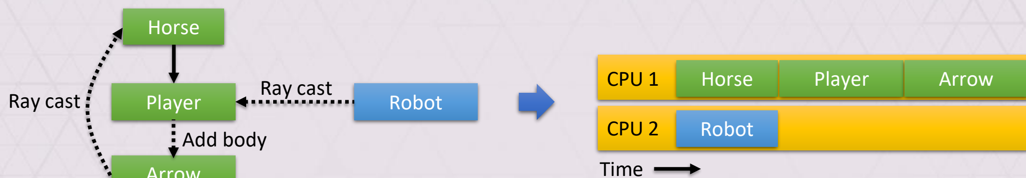
CONTENTS

- The Physics Update
- Waits
  - Streaming Open World
  - Game Object Update
  - Background Queries
- Solutions
  - Lock free broad phase
  - Lock free island building
- Conclusion

14

Now that we've established our first requirement, we're going to look at the next requirement. Our game update is also heavily multi threaded so requires the ability to concurrently query and modify the physics world. This is also something our old engine didn't provide.

PARALLEL GAME OBJECT UPDATE



- Independent game objects on different threads
- Global systems = Lock contention
- PS5 CPUs > PS4 CPUs
- See GDCE 2014 presentation

15

In our game, all game objects (or entities) that are not dependent on each other will run concurrently. An example of a dependency between two entities is a rider on a horse. The horse needs to be animated before the rider can be moved along. In this example, besides the horse and the player, there is a robot. This robot is independent of the horse and player so can be updated concurrently. As the player casts a ray to fire an arrow, it expects the horse to be updated or it may accidentally hit it. At the same time, the robot may do a perception check and cast a ray towards the player. We don't care too much about update order between non-dependent entities, so if the ray of the robot sees the player before or after its update is non-deterministic. With this approach, global systems like the physics system are a source of lock contention: Adding objects while moving other objects and doing collision detection at the same time requires a lock on the physics world. This would often cause other threads to stall. With the PS5 this became much worse since it has many more CPU cores than the PS4 and this often resulted in serious performance degradation. Our new solution should support concurrent query and modification. Note that if you're interested in how we multithread our game update, you can look at a GDCE presentation that I did in 2014.

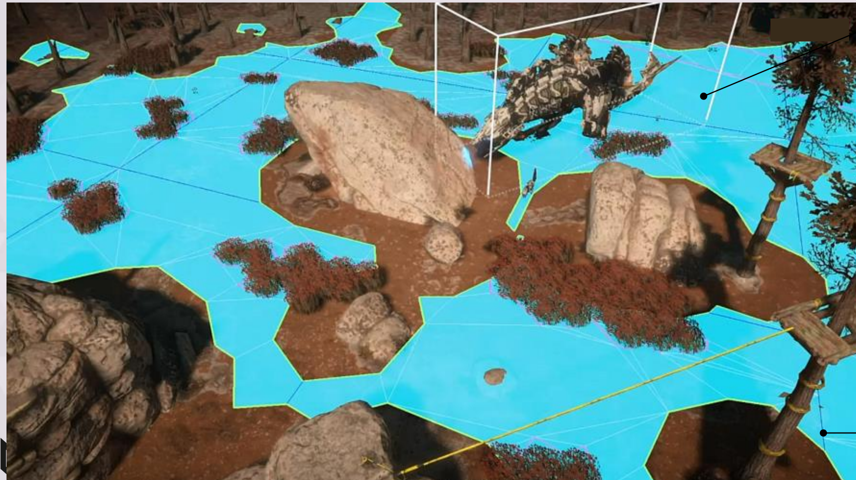
CONTENTS

- The Physics Update
- Waits
  - Streaming Open World
  - Game Object Update
  - Background Queries
- Solutions
  - Lock free broad phase
  - Lock free island building
- Conclusion

16

As an example of the last important requirement on our physics system we'll discuss the Navigation Mesh (or NavMesh) builder. As we will see, this system requires us to do background queries without stopping the physics world.

BACKGROUND NAVMESH JOB



Walkable area.

- Query 12.8x12.8m area.
- Recast on background thread.
- > 1 frame.

Other threads continue = contention!

Tile border

17

The NavMesh system generates a set of connected polygons that follow the terrain and is used by AI to find a path from A to B. We build these navigation meshes on a background thread in 12.8 x 12.8 meter blocks. In the image, you can see the navigation mesh for a Thunderjaw, which corresponds to the area that the robot can walk in. When an AI spawns, it requests NavMesh in its surrounding area. A background job picks up this request and does a query of all rigid bodies that intersect with the area. It queries all triangles of those bodies, voxelizes them and then generates the Navigation Mesh. We're using a customized version of Recast to do this work. Depending on how busy the scene is, a NavMesh job can last for more than a frame. During this time the game code may be modifying, adding and removing objects or perform other collision queries. This again causes waits as mutexes are locked. We require the ability to do a long running query that doesn't block any other threads from doing work.

CONTENTS

- The Physics Update
- Waits
  - Streaming Open World
  - Game Object Update
  - Background Queries
- Solutions
  - Lock free broad phase
  - Lock free island building
- Conclusion

18

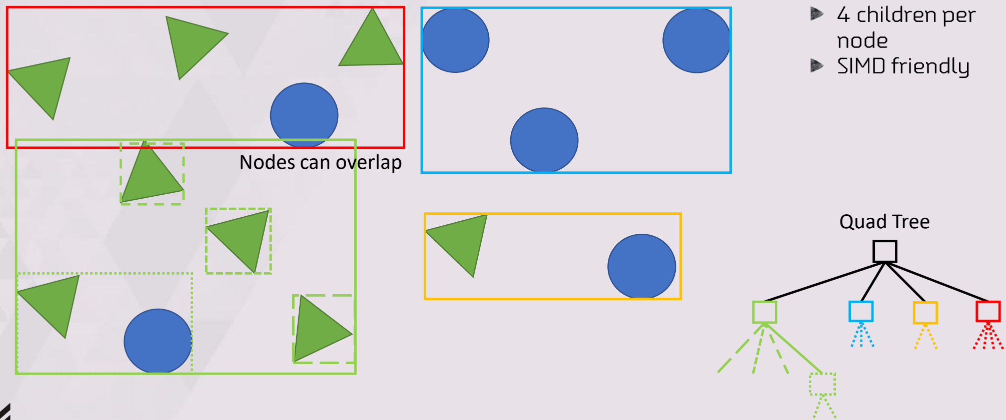
Now that we've established our requirements, we'll move on to discuss a couple of solutions for these requirements. The first solution we'll discuss is a lock free broad phase. To recap: The broadphase detects collisions by checking against the axis aligned bounding box of a body.

BROAD PHASE OVERVIEW

- Waiting on concurrent
  - Add, modify, remove
  - (Background) query
- Solution
  - A Quad Tree
  - Lock free (mostly)

19

First a short recap of what we established over the last couple of slides that cause us to wait: we add, modify and remove objects while running queries on foreground and background threads. The broad phase is a singleton that manages all bodies and that is most likely causing contention between different threads. We chose a Quad Tree structure. In order to avoid lock contention we chose to make it mostly lock free. We will show how it is implemented, the advantages it offers and the disadvantages it has.



Our broad phase is a quad tree, which means each node has 4 children. On the slide you see a random collection of spheres and triangles and a possible way to split the tree. At the highest level we split all objects in 4 mostly disjoint sets. Note that nodes are allowed to overlap, but for efficiency reasons you want the amount of overlap to be minimal. The example split here is indicated by a red, blue, green and yellow box and you can see them appear in the tree on the right. Three out of four nodes: blue, yellow and red, have 4 or less shapes in them, so the tree can directly point at the shapes rather than at a next. One node: green, has more than 4 shapes in it so needs a further split. Three shapes can be added directly to the node and we need to create a new node, dotted green, to hold the last two shapes. The reason why we pick 4 children is that modern CPUs support doing 4 math operations in a single instruction, so when we walk the tree from top to bottom during a collision query, we can handle 4 children at the same time.

QUAD TREE - NODE

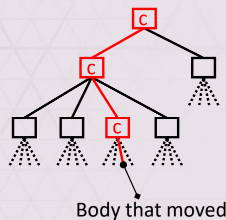
Type	Name
AtomicFloat	BoundsMinX[4]
AtomicFloat	BoundsMinY[4]
AtomicFloat	BoundsMinZ[4]
AtomicFloat	BoundsMaxX[4]
AtomicFloat	BoundsMaxY[4]
AtomicFloat	BoundsMaxZ[4]
AtomicUInt32	ChildNodeIdx[4]
AtomicUInt32	ParentNodeIdx
AtomicBool	Changed

- Child bounding boxes in SIMD format
- Initialized to [LargeFloat, -LargeFloat]
- ChildNodeIndex refers to node or body
- ParentNodeIndex refers to parent
- Changed flag set if bounds too large
- Using atomic operations

21

To give some context, this is the layout of a quad tree node. First of all, we have the axis aligned bounding box of 4 child nodes arranged in such a way that we can easily load the same quantity for 4 children at the same time: MinX, Y and Z then MaxX, Y and Z. They are initialized so that they are inside out and are filled with a value, 10^30 to be exact (we'll call this LargeFloat), that is large enough to cover the entire world but not too large to create overflows to infinity when we work with the values. As the bounds are initialized inside out, the collision queries that we do on a node will not find a hit. Next, we store the node indices of the 4 children. These can be an index in the node array or (when a specific bit is set) they indicate that they point to a body index instead. Each quad tree node points to its parent node through an index. Finally, each node tracks if any changes were being made to it or its children since the last rebuild of the tree. As we will see later, the bounding boxes will become larger over time and parts of the tree need to be rebuilt. All members are stored as atomics to make it possible to modify them concurrently from multiple threads.

QUAD TREE – MOVING BODY



Node flagged as 'changed'

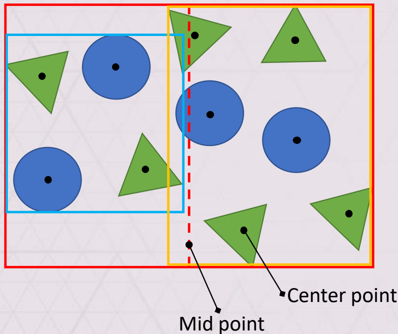
Widened bounding box

- Body stores (NodeIndex, ChildIndex)
- Body moves
  - Calculate bounding box
  - AtomicMin / AtomicMax to update node
  - Mark 'Changed'
  - Go to ParentNodeIndex, repeat
- Query: No collisions missed!
- Performance degradation, need tree rebuild!

22

Let us first assume that we have already built a tree. We'll look at moving a body first as it will influence how we add bodies to the tree. Each body that is in the tree will store its location in the tree through a node index and a child index 1 .. 4, this means that we can find the correct node in the tree in O(1). If a body moves, we recalculate its bounding box and then use AtomicMin and AtomicMax to enlarge the bounding box of the specific child to also contain the new bounding box. If another thread at the same time makes a modification, we will end up with the widest bounding box that will encapsulate the writes of all different threads. Next, we mark the body as 'changed' to indicate that we've enlarged the bounds and that this sub-tree needs to be re-built. We then use the parent index to go to the parent node and widen its bounds in the same way and mark it as 'changed' too. We keep repeating this until we get to the root of the tree. Any query that is in flight while the tree is being modified will never miss a collision because the tree was halfway through being rebuilt simply because we only widen bounding boxes. The downside of this approach is obviously that the tree will become less and less efficient over time, so we need to rebuild the tree at some point.

QUAD TREE – ADDING BODIES

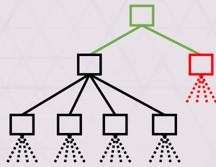


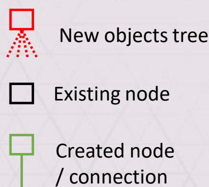
- Loading thread: Sub-tree of objects to add
- Fast tree construction
  - Calculate bounds all objects
  - Split at mid point of longest axis
  - Classify 'Center Point' of AABB
  - Repeat to get 4 leaves
  - Recurse to create tree

23

Next we will look at adding a set of bodies. In order to make insertion fast, we collect all bodies of a tile or Scene and insert them at once. We do this by building a sub-tree of the quad tree on the loading thread first.
We use a very simple splitting strategy to avoid too much load on the loading thread. We begin by calculating the bounding box that encompasses all objects that we want to insert. We calculate the longest axis and split halfway at the 'mid point'. Then we look at each object's axis aligned bounding box and use the center point to classify if the object should go to one side or to the other. Once this is done, we can repeat the same process on the 2 sets of objects to end up with up to 4 sets of objects. These sets will become the first children of our new sub tree. We repeat this process recursively until we have a full tree.

Quad Tree – Adding Bodies – Case 1





- Insertion in tree in O(1)
- Case 1: Root Full?
  - Create new root
  - Mark as 'Changed'
  - Child 1 is old root, AABB [-LargeFloat, LargeFloat]
  - Child 2 is 'new objects' & set parent
  - Compare exchange old root with new root
    - On failure, delete new root and go to case 2
  - Set old root parent to new root

24

Once the loading thread has finished building the new sub tree, lets call it the new objects tree, we insert the bodies into the broadphase in O(1) on the main thread. In order to make this lock free, we always insert at the root. There are 2 possible cases. The first case assumes that the current root of the tree, aka old root has no more empty children available. If this is the case, we need to create a new root and replace the old root with the new root. We first construct a new root, immediately mark it as 'changed' as it is not going to be a very efficient root so we want it rebuilt when possible.

Next, we make child 1 of our new root point to the existing root of the tree and give it a very large bounding box. This ensures that any collision query will always hit the old root. We need to do this because other threads may be modifying the old root and widening its bounding box, we have no way of synchronizing this. The old root already contained the bounding box of the 'entire world' so arguably this does not degrade the whole tree by much.

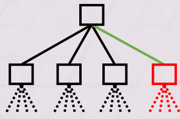
Child 2 becomes the 'new objects tree' and again because no-one is aware of these objects we can simply set the bounds to the correct value.

Next, we can set the parent of our 'new objects tree' to this new root, nobody is yet aware of these new bodies so we can do this without affecting other threads.

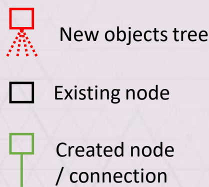
The next step is to exchange the old root of the tree with the new tree, we use

compare exchange for this to make it an atomic operation. If the atomic operation succeeds we only need to set the parent index of the old root and we're done.
If some other thread replaced the root before us, we delete our newly created root and we hope that there is room for us in the new root that the other thread created.

QUAD TREE – ADDING BODIES – CASE 2



- Case 2: Root has slot?
  - Set parent 'new objects' to root
  - Compare exchange on root: Invalid Index → Node Index
    - On failure go to case 1 or 2
  - Update child bounds in root (Max then Min).
  - Mark root 'Changed'
- Inserting 1 body is bad!



25

Case 2 is actually the first check we do when inserting bodies. We check if there are any children available in the root by checking if the Child Node Index is invalid. If this is the case we're going to try to insert our tree into that location. We tentatively set our parent index to the root and then use compare exchange to try to swap the invalid node index with our node index. If this fails, then we try the next child until we've ensured that there are no slots available in this root and then go back to case 1 to create a new root. If it succeeds however, we've successfully linked ourselves to the existing tree, but no collision query will be able to collide with us yet since the bounding boxes are still invalid. We can simply set the Max X, Y and Z values first. This will keep the Min values at LargeFloat which means that the bounding box is still inside out and nobody will collide with us. Then we will update the Min X, Y and Z and only when the last coordinate is set, the bounding box is valid and other threads can find the newly added bodies. The root is marked as 'Changed' again since we want the tree to rebuild to find a more efficient place in the tree for our newly added nodes.

So why do we insert at the root? We could find a free slot anywhere in the tree and insert there, but this would involve a search and some heuristic to see if the node is a good fit. To insert halfway in the tree, you would often need to create a new node. In order to do this, we need to atomically change both the child node index and the

parent node index to ensure that threads that are querying the tree / modifying the tree see a consistent state. This is not possible on current hardware.
One thing to mention about this way of inserting in the broad phase is that it is very bad to add bodies 1 at a time. If you do this, you end up with a linked list instead of a tree and you'll get very bad performance.

QUAD TREE – REMOVING BODIES

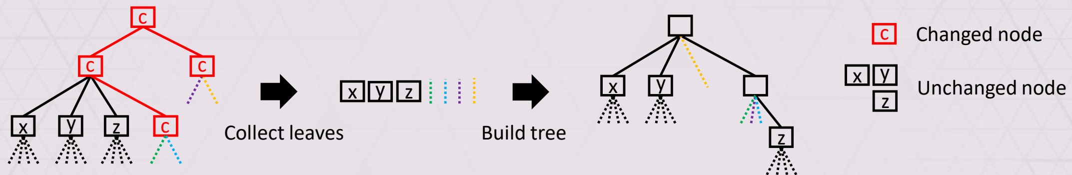
- Invalidate bounding box
  - Set Min to LargeFloat
  - Set Max to -LargeFloat
- Set ChildNodeIdx to Invalid
- Mark node and parents as 'Changed'
- Don't remove entire sub-tree, rebuild mixes bodies
- Remove 1 body in 0.5 μs on PS4

26

Removing a body from the tree is very simple and the reverse of adding a body. We first set the Min X, Y and Z value of the child of the node that the object resides in to LargeFloat. This makes the bounding box inside out, so any new queries will no longer find the body. Then we can continue by setting Max X, Y and Z to -LargeFloat. We set the child node index to invalid and from that point on another thread can re-use the slot and insert a new body there.
As before we mark the node and its parents as 'Changed' so that a rebuild of the tree will rebuild this part of the tree.
It would be possible to remove an entire sub-tree at a time (for example an entire tile or Scene) but when a single object changed in that sub-tree, all the bodies in that tree will have been mixed with other nodes, so we don't do that.
In this way, we can remove a body from the tree in about 0.5 us on PS4.

QUAD TREE – REBUILDING

- Not 'Changed' = Single Object

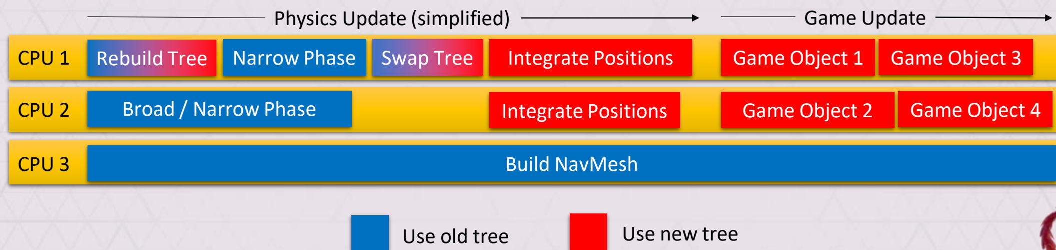


- Use multiple trees (static vs dynamic)

27

When we've added or modified some objects, the bounding boxes of the tree and the structure of the tree becomes less and less efficient. We need to rebuild and refit the tree. We do this in a separate job that will first traverse the existing tree and grab all bodies. We stop at nodes that do not have their 'Changed' flag set because it is too CPU intensive to rebuild the entire tree every frame. Then a new tree will be generated in the same way as when we added objects. To illustrate this, you can see a tree where some objects were removed. After collecting the leaves, we're left with the unchanged nodes x, y and z and we have 4 separate objects, indicated by the colored dashed lines. The resulting tree may look like the image on the right.
An important performance optimization is to use multiple broad phase trees. We have several, but the most important ones are one for static objects and one for dynamic objects. The broad phase for static objects is very big but changes infrequently. When it changes, large portions of the tree are still 'unchanged' so the rebuild can be cheap. The broad phase for dynamic objects is much smaller but changes all the time so needs a constant full rebuild. Because the number of objects is less, this can be done in a reasonable amount of time. When you do a collision query, you potentially need to query multiple trees instead of one, so this does give some overhead there.

QUAD TREE - REBUILDING



- Keep old tree until next physics update

28

So, when do we rebuild the tree? We've chosen the physics update for rebuilding the tree as we're already locking all bodies at that point to ensure that no-one makes changes to the world while we're simulating. The order in our frame is always: physics update first (using all cores), then update all game objects (using all cores). While the broad and narrow phase collision detection is running at the beginning of the physics update, we're only reading from the broad phase, so this is an ideal time to rebuild the tree. Once the tree is built, we wait until all narrow phase jobs complete and then we can simply swap the old tree with the new tree. The NavMesh job that had already started will continue using the old tree, but anything that starts after the swap will use the new tree. Integrate positions will modify objects in the new tree and the game objects that update next will read and write to the new tree. We delete the old tree at the beginning of the next physics update, which gives the NavMesh job almost two full frames to finish its query.

QUAD TREE – COLLISION QUERY

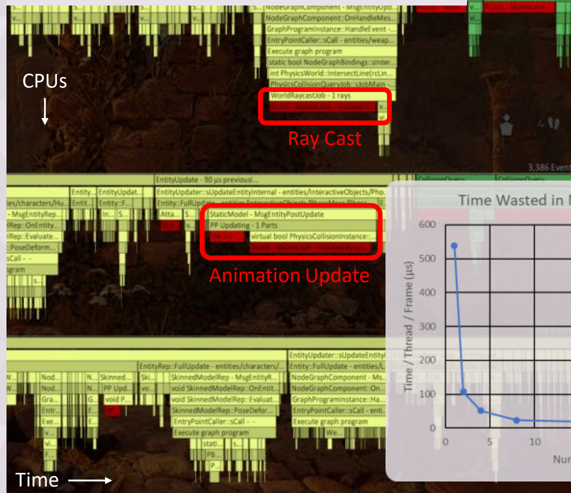
- Lock tree for read (for background jobs)
- Recurse children until leaf (body)
- Lock body mutex for read
  - Get shape (collision volume)
  - Get transform
- Unlock body mutex
- Narrow phase work
- Overhead locks 10-20%, only for Game Object Update

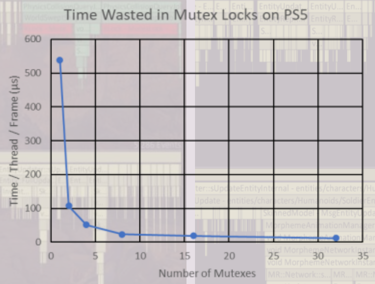
29

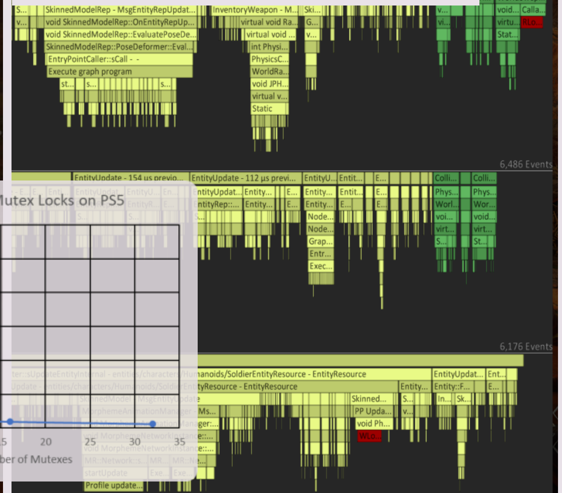
The last thing we're going to cover is querying the tree. This is the part that will require some locking.
If we're in a background job that can span multiple frames, we need to take a read lock on the tree that we're querying. The mutex will only be locked for write just before the tree is deleted the next frame. This ensures that the background job finishes before the tree is deleted. Since that mutex is only locked once for write and can be locked multiple times for read, this lock is never contended so this is fast.
Next, we take the root of the tree, load the bounding boxes of the 4 child nodes and perform a collision test of our collision primitive (a ray, shape or cast shape) against the 4 bounding boxes at the same time. We recurse to the nodes that had a hit and repeat until we collide with one or more bodies. Our broad phase is lock free, but our bodies cannot be, they contain way too much state to atomically change. We have a fixed number of read / write mutexes and associate each body with one of those mutexes. When we want to make a change to the body (e.g. move it or destroy it) we need to lock the body for write and then make the change. When we query, we need to lock the body for read. Once the body is locked we can quickly fetch its shape and transform and then unlock the body again. We now have enough information to do narrow phase collision detection so we don't need to hold any other threads up.
The body locks are an Achilles heel of the whole system, if multiple threads need to

modify the same body they will obviously have to wait for each other. In practice this does not happen as bodies usually have a clear owning game object and only that object will modify the body. Besides that there are so many bodies that the chances of one thread writing to the same body that another thread is reading are low. When querying, usually you only hit a few bodies at a time, so the number of locks is relatively low and we've measured the overhead on the entire query to be 10-20%. Outside of the game object update, we know that the physics world is not changing, so we don't need the locks at all.

Broad Phase - Results







So, what did we gain? On the left you can see the game running on PS5 with 13 worker threads. The screenshot shows 3 threads with time flowing from left to right. Each function call is a bar and the red bars indicate mutex locks. You can see many stalls caused by one entity trying to update the physics state while another thread reads the state. We waste around 0.5 ms per thread waiting. On the right you can see the situation with our new broad phase, we have 32 mutexes in total and waste around 10 us per thread waiting for other threads. The graph shows the trade off between number of mutexes we reserve for the bodies and time we spend waiting for these mutexes, using more than 32 mutexes doesn’t give any further benefits. Background jobs are now free to do collision detection work without disturbing the foreground jobs for the same reason. And finally, we now have a system that allows much faster object insertion / removal than we had before.

CONTENTS

- The Physics Update
- Waits
  - Streaming Open World
  - Game Object Update
  - Background Queries
- Solutions
  - Lock free broad phase
  - Lock free island building
- Conclusion

31

The lock free broad phase was the most important change that allowed us to write efficient game code. The Jolt Physics update itself is also heavily multi threaded. I'm now going to discuss island building and how it is implemented in such a way to keep bodies isolated so that they can be concurrently modified. To recap: The island builder determines bodies connected through a collision that need to be solved in a single CPU job.

Here is a screenshot of the profile of a single physics update. It shows 3 CPU cores from top to bottom (3 red bars) and time flowing from left to right. We can see in yellow the job that rebuilds the tree and we can see the broad and narrow phase jobs in green (they're combined). The dark blue and light green parts are solving constraints. Between these two there is a single job that builds the simulation islands. You can imagine that as we give more cores to the physics simulation, the left and right side will become smaller and smaller as work is divided, but the middle part will always stay the same length as it happens on 1 core only. We need islands to be generated fast!

Generating Islands - Options

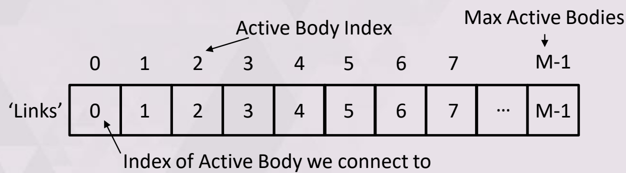
- Build iteratively
  - Problem: Requires locking during update.
- Build from scratch
  - Problem: Needs connectivity = locking.
- Our Solution
  - Like Union Find with Path Compression.
  - Most work during Narrow Phase.
  - Lock free.
  - Finalize islands in O(N).

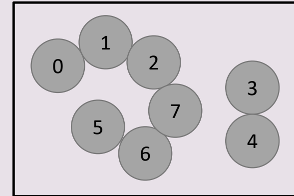
33

In order to efficiently build the simulation islands, the easiest way is to reuse information from last frame. Bodies that touched last frame, most likely, will touch this frame again. So as contacts are added / removed, we can split and merge islands. The problem here is that if a body moves, we may need to update the state of a bunch of other bodies. This goes against our philosophy of being able to independently update bodies and will cause locking. So this option is out.
The other option is to build islands from scratch every frame. A fast way of doing this would involve keeping track of which body contacts which other body and which body connects to which constraint. If you have that connectivity, you can easily do a flood fill to find all bodies that are connected to another body. Box2D uses this approach for example. We do not want bodies to keep track of their contacts and constraints because it creates dependencies between them and thus potential locks. Another problem with this solution is that we still cannot do any work in parallel.
For completeness, I want to mention that there are other methods possible, like grid based islands, that we won't discuss today.
The solution that we picked is very similar to the Union Find algorithm with path compression and allows us to do most of the work during the Narrow Phase, which you saw happens in parallel. In order to make this possible, that part of the algorithm is lock free. When the narrow phase is finished, all that's left to do is 2 very simple

loops over an array with the same size as the number of active bodies to get our islands. This makes the algorithm O(N).

ACTIVE BODY LINKS – INITIAL STATE





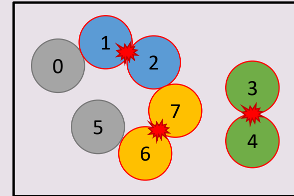
Example

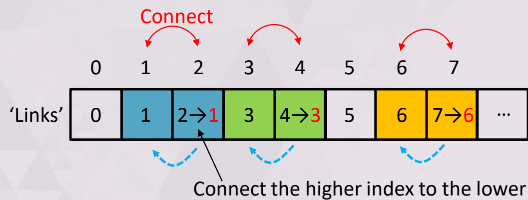
- Initially connect to self.

34

Initially we start with a simple array sized to the max number of active bodies that we can have. As we are using lock free operations and new bodies can be activated as collisions are found, we cannot safely resize and reallocate this array while other threads doing atomic writes because the memory block may move.
Each entry in the array corresponds to an active body and the value in the array corresponds to the body that the active body is connected to. Initially all bodies connect to themselves.
In the top right corner you can see a number of rigid bodies that form 2 islands, we will be using this example to demonstrate the algorithm.

SIMPLE CASE: CONNECT 1 & 2, 3 & 4 AND 6 & 7



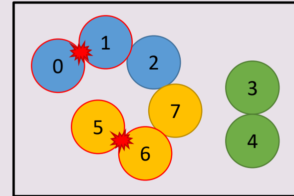


- Connect contacting bodies

35

During the broad and narrow phase update, as we find new collisions, we mark the two affected bodies as connected. In this example the narrow phase finds that 1 and 2, 3 and 4 and 6 and 7 are contacting. We always connect the higher body to the lower body. The image above shows the simple case: We update body 2 to point to 1, 4 to 3 and 7 to 6.

Simple case: Connect 0 & 1 and 5 & 6

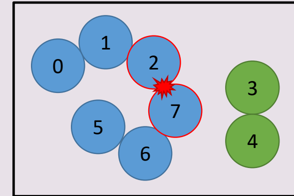


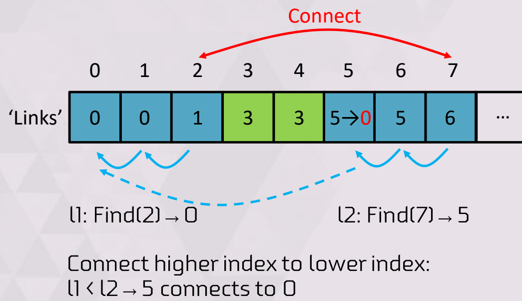
- Forming linked lists
- 'Find' function:
  - Find(7) → 5
  - Find(0) → 0

36

As more and more bodies are connected, we start to form linked lists. When the narrow phase finds that 0 and 1 collide, we have 2 point to 1 and 1 to 0. We express this in the 'Find' function, the 'Find' function will find the lowest index that a body is connected to. As an example, if we pass 7 it returns 5, if we pass it 0 it returns 0 because it connects to itself.

COMPLEX CASE: CONNECT 2 & 7



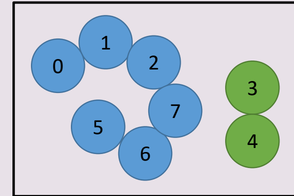


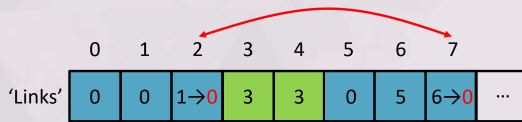

```
Union(b1, b2):
  // Find lowest connected body
  l1 = Find(b1)
  l2 = Find(b2)

  // Ensure l1 < l2
  if l1 > l2: swap(l1, l2)

  // Link the bodies
  Links[l2] = l1
```

37

If we want to connect 2 and 7, we find that their roots are 0 and 5. We can either link 0 to 5 or 5 to 0 to create the union of both trees. We choose to always connect the higher index to the smaller index, so we connect 5 to 0. In yellow this is listed in program form: We first find the roots of both trees, ensure that l1 < s2 and then link them.

WORST CASE: FIND IS O(N)





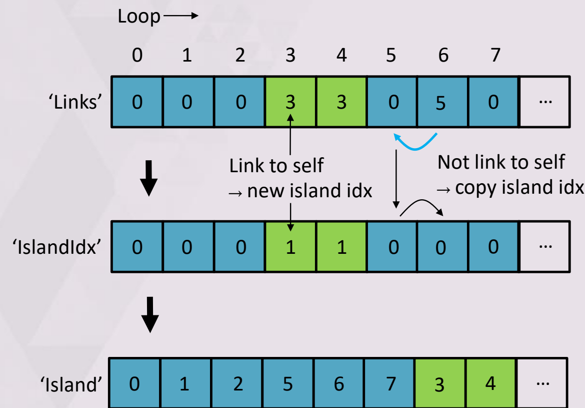
Find = O(N) → Path Compression.

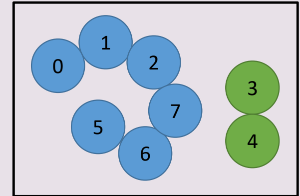
```
Union(b1, b2):
  ... as before ...

  // Link both bodies to minimum
  Links[b1] = l1
  Links[b2] = l1
```

38

When we connect more and more bodies, we can create longer and longer chains and in the worst case we form a chain where each body links to the next. This means that the Find() algorithm becomes O(N). To solve this, we use what is called path compression. We update the bodies that are connected to the minimum value. Normally if you do path compression, you update all bodies along the path to the lowest value, but this would require iterating the entire chain again and in practice doesn't seem to be needed. So, in this case, since we are connecting 2 and 7, we update both to the lowest index 0.

FINISHING UP (ON 1 THREAD)

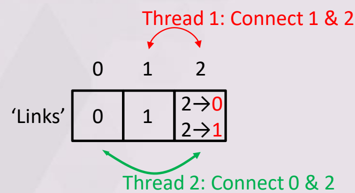




39

Now that all threads have finished connecting bodies, we need to calculate the actual islands. This needs to happen on 1 thread. We go from left to right and calculate the island index for each of the bodies. If a body links to itself, we know that we're at the start of an island so we increment the island index. If a body doesn't link to itself, we just copy the island index from the body that we connect to. After that loop, we loop through 'IslandIdx' and copy the body indices for each island into an array. If you kept a count of the number of bodies belonging to each island in the previous loop, you can store this in a single linear array. We now have all the bodies that belong to an island in a simple array that we can loop over constraint solving and integration.

MAKING IT THREAD SAFE



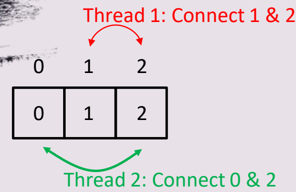
- Race condition:
  - T1: Read l1 = 1, l2 = 2
  - T2: Read l1 = 0, l2 = 2
  - T2: Write Links[2] = 0
  - T1: Write Links[2] = 1
- 2 is **not** connected to 0!

40

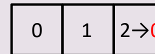
We have one final loose end: We're going to be calling the Union function from multiple threads at the same time, so this function needs to be thread safe. In this example, thread 1 wants to connect 1 and 2 and thread 2 wants to connect 0 and 2. We can break the algorithm if both threads first read the current value, so the first thread will read 1 and 2, the second thread will read 0 and 2. Thread 2 will then decide to link 2 to 0, thread 1 will link 2 to 1. The result is that the first write is forgotten and that 2 is not connected to 0. In order to fix this, we can use compare exchange again.

LOCK FREE SOLUTION

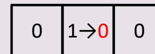
```
Union(b1, b2):
 l1 = b1
 l2 = b2

 while:
  // Search from previous lowest index
  l1 = Find(l1)
  l2 = Find(l2)

  // Ensure l1 < l2
  if l1 > l2: swap(l1, l2)

 // l1 == l2: Already connected
 until l1 == l2 or Links[l2].Exchange(l2→l1)

 // Link both bodies to the minimum
 AtomicMin(Links[b1], l1)
 AtomicMin(Links[b2], l1)
```



- T1: Read l1 = 1, l2 = 2
- T2: Read l1 = 0, l2 = 2
- T2: Links[2].Exchange(2→0)



- T1: Links[2].Exchange(2→1) **fail!**
- T1: Read l1 = 1, l2 = 0
- T1: Links[1].Exchange(1→0)



All done!

41

The algorithm is almost the same as lasts time but with an extra while loop. We start by searching the lowest index to which body 1 and 2 are connected. Thread 1 reads 1 and 2, thread 2 reads 0 and 2. Again, we sort so that l1 < l2. Then either the two indices are equal and the objects are already connected, or we atomically exchange the link at l2 with l1. If this fails, it means that another thread beat us and we need to try it again. In this case we don't need to search from the b1 and b2 anymore, but we can start looking from the previous lowest values. In the example you can see that thread 2 succeeds in changing the link from 2 to 0, but thread 1 fails to change 2 for 1. This time thread 1 reads 1 and 0 and succeeds to exchange 1 for 0. Our path compression step uses an AtomicMin instead of an assignment to update the links for the bodies that we're connecting. If another thread wrote a lower number first, we don't need to update our value, that body must be part of the same island.

ISLAND BUILDING - BEFORE

42

To remind you, this is what the situation looked like before.

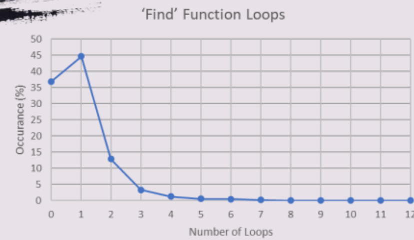
ISLAND BUILDING - RESULT

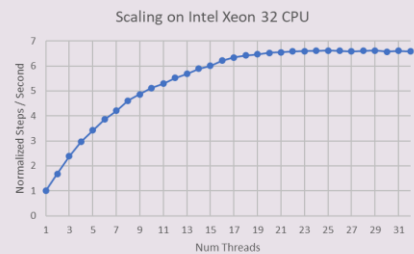
43

And this is after. The result of all of this is that we moved some of the work of generating islands to the multithreaded broad and narrow phase update and that generating islands is a simple loop. You can see that it is much smaller than before!

ISLAND BUILDING - RESULT

- We connect bodies as contacts are detected
- Body unaware of contacts – no locks
- Finishing up O(N)
- Single threaded part 80% less time
- Total 70% less time (4 CPU)
- Find average 0.9 loops
- Union average 0.07 µs
- Not a bottleneck at 16 threads





44

So what did we gain: We connected bodies as contacts are detected. This happened from multiple threads as once.
We achieved our goal of keeping bodies unaware of their contacts, avoiding any shared state between bodies that may require locks.
Our island building is a couple of very simple loops that have a worst case time complexity of O(N).
The measured performance benefit is that the single threaded part now takes 80% less time than before. Obviously, we moved some of the work to other threads but as that was in parallel. Our algorithm still takes 70% less time on a 4 CPU machine (the benefit will become bigger with more cores).
In a stress test scene, the 'Find' algorithm on average only took 0.9 loops (this is excluding reading the first element). In the graph on the right you can see the distribution of the number of loops that 'Find' did. In the worst case it did 12 iterations, but this was only for a tiny fraction, so our limited form of 'path compression' seems to work well enough.
The 'Union' function averaged on my i7-7700HK to about 0.07 µs per call, which is tiny compared to the total amount of time spent in collision detection.
The second graph on the slide shows the scaling of the entire physics update with the number of threads on a 32 core machine. The graph is normalized so that 1 thread is

1. You can see the whole engine scales up well until about 16 threads, at that point the memory bus cannot keep up anymore. Physics is chaos and chaos causes a lot of random memory accesses in the collision detection subsystem.

CONTENTS

- The Physics Update
- Waits
  - Streaming Open World
  - Game Object Update
  - Background Queries
- Solutions
  - Lock free broad phase
  - Lock free island building
- Conclusion

45

We’re going to conclude with the benefits we got from switching to Jolt Physics.

CONCLUSION

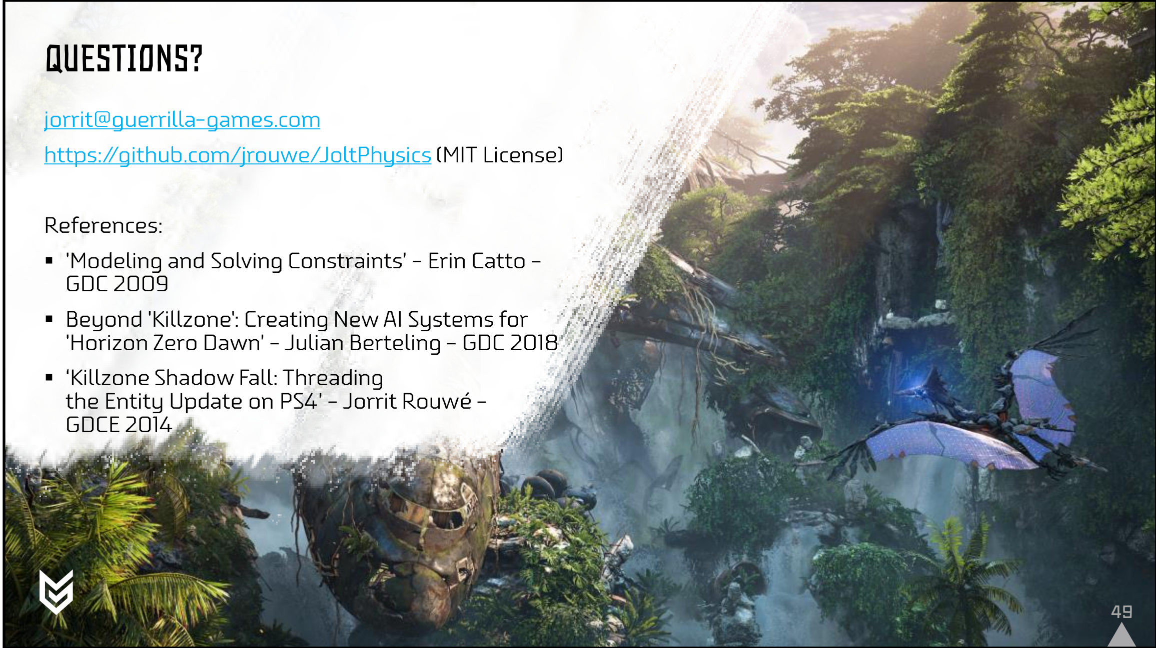
- Simulation frequency 30 Hz → 2 x 60 Hz in less time
- 60 fps mode halves update time
- Increased visual fidelity
- Reduced heap memory 25%, asset memory 30%, executable size 12%
- Eliminated global lock contention in game update, no more waits!

46

So, what did we gain by switching to Jolt Physics? We were able to switch our simulation frequency from the 30 Hz that we were using in all our previous games to two steps of 60 Hz while spending less time on the physics update. In performance mode, which is 60 fps, it means we only need 1 physics update per frame, so we save half of the cost. We increased the visual fidelity of the game, mostly because the physics simulation looks much better at 60 Hz than it does at 30 Hz. We also managed to reduce our run-time memory cost by 25% and reduced the size of all physics assets (meshes, terrain etc.) by 30%. Due to the much lower complexity of the code base, our executable size also dropped by 12%. We eliminated most of the global lock contention during the game update eliminating most of the waits.

I want to show you one of our biggest problem cases before switching to Jolt Physics. This is the Slitherfang, one of the bosses in the game. I will first show you the before state (around May 2021). You will see that in the first couple of seconds, the frame rate is terrible because the physics engine is spiking to more than 20 ms per frame, also the physics engine has a lot of issues handling the long body of the snake, mainly because we used to update at 30 Hz.

Next, I will show you the state after we switched to Jolt Physics. You will see that there is a small hiccup when the snake first dies where the simulation can spike to 10 ms. The simulation recovers much more quickly and the ragdoll of the Slitherfang is much more stable due to the higher update frequency. At this higher update frequency, the total time for a physics update is still 30% lower than it used to be.



Thanks to Chris Zimmerman for mentoring me and giving lots of useful feedback. Also thanks to everyone at Guerrilla for making it possible to integrate Jolt Physics into HFW. Special thanks goes out to Michiel van der Leeuw for making it possible to Open Source Jolt Physics.