Aristotle University Of Thessaloniki
Computer Science Department

Creating an optimized 2D physics simulation engine

Dissertation

Student: Tsamis Emmanouil
Supervisor: Nikolaidis Nikolaos

02-11-2020

Abstract

Physics engines are used in an ever increasing number of applications, with multiple high-quality open-source simulators being available. For physics in two dimensions, Box2D is the most well-known and influential library, offering a rich feature set and bindings in all major programming languages. Box2D employs many optimizations to provide solid performance, but after careful inspection we discovered multiple fronts where improvements can be made. Our contribution is a new physics engine, based on Box2D, which offers superior performance and enables simulating much larger scenes in real time. In many scenarios, our library is up to an order of magnitude faster, without making compromises in the stability and accuracy that Box2D provides. In this dissertation, we present how we optimized the algorithms and data structures involved in order to achieved this performance enhancement. For each stage of the Box2D physics pipeline, we initially study its structure to identify potential bottlenecks and then design and implement our optimized alternative. Our results are backed by a new benchmark application we developed that can reliably compare the efficiency and scalability of the simulators.

Contents

List of Figures

Listings

1 Introduction

1.1 Rigid body simulations

Physics engines are computer programs that simulate physical phenomena. Due to the complexity and computational requirements of this process, a simulation is usually approximating the actual phenomenon. Physics engines have applications in many fields including robotics, video games, animation, aerospace and constructions.

There are different kinds of physics engines, depending on the type of the simulated primitives. Some common types are particle-based, soft body and rigid body simulators. The type of primitive determines the capabilities and limitations of the system. Our focus will be on rigid body simulators. A rigid body simulator is a physics engine that simulates solid, non-deformable objects.

Physics engines can be either real-time or offline. Real-time simulators use simpler and less precise models to increase the performance of the engine. They're used in interactive applications when the accuracy of the simulation is less important than performance, for example, in video games. For applications that have high precision requirements, usually scientific simulations, more complex models are used that cannot be simulated in real-time.

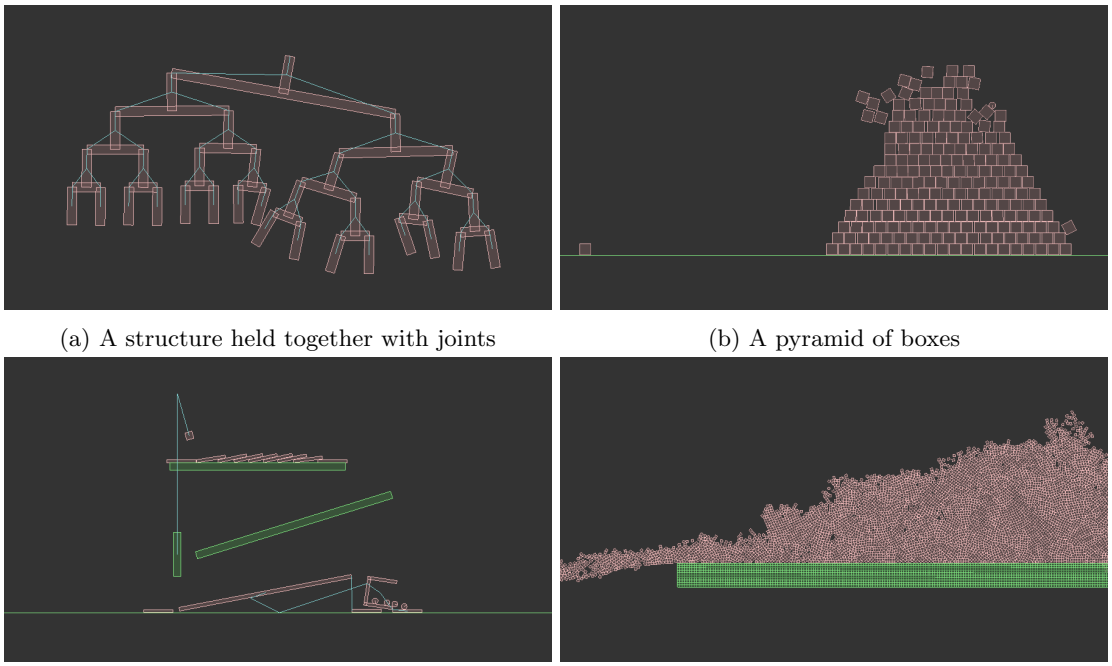
1.2 What is Box2D

Box2D is a real-time, rigid body physics engine for two dimensions, developed and maintained by Erin Catto. It is written in platform independent C++. It can be used for collision detection, raycasting, constraint solving and continuous physics. It supports modifiable restitution and friction for bodies, collision filtering and processing, bodies with multiple shapes, a wide variety of joints and constraints, rope physics and more. The library successfully combines stability, accuracy and performance. It can efficiently handle simulations of thousands of bodies and joints.

Box2D has been ported and used in almost all major programming languages and operating systems. It is used in iconic games such as Angry Birds and Shovel Knight and is the default physics engine in many popular game frameworks such as Unity, Cocos2D and LibGDX. Erin Catto's work in 2D physics engines has been highly influential and with his library being free and open source, Box2D became the de facto standard for real-time 2D simulations. To day, the library is actively improved with new features, performance enhancements and bug fixes.

1.3 Our work

While Box2D is a highly optimized simulator, we found that it is possible to greatly improve its efficiency by redesigning it's core data structures and algorithms. In this dissertation, we study the internals of the library and evaluate how to improve the performance of the engine. We provide insight on how we designed new efficient algorithms and how they affect performance. Because small scenes can be simulated even with naive algorithms, our main focus is how to develop a physics engine that can efficiently scale to multiple thousands of bodies.



(a) A structure held together with joints

(b) A pyramid of boxes

(c) A mixture of joints and bodies

(d) A simulation with thousands of squares

Figure 1: Example scenes simulated with Box2D

We developed a new physics engine, based on Box2D[1], that incorporates all our new algorithms and changes to provide greater performance. We optimized the core of the library in a way that the public API remains very similar to Box2D. This makes it easy to port existing applications and understand the structure of the project. Breaking changes where made only when it was required to achieve significant performance benefits.

1.4 Why Box2D

Although several other 2D rigid body simulators exist, a significant number of them are derived from or use Box2D's algorithms. Using Box2D as the base for our library enables many other projects to benefit and adopt the changes that we propose. Moreover, of all the available libraries, Box2D is one of the most performant and optimized. All other libraries we evaluated had inferior performance and did not scale well. The fact that it is developed in C++ is necessary for low-level programming and to enable advanced compiler optimizations. Higher level languages are handicapped when it comes to CPU-intensive code involving numerical calculations.

Moreover, Box2D is licensed under the permissive MIT licence, making it easy to modify and distribute its source code. We also use the same licence for our work, to make our contributions available to a broader audience.

[1]The official Box2D repository does not accept third-party contributions that change the library's functionality. It is encouraged by its creator to fork the library instead, which is why we created our own physics engine.

1.5 Results

The source code of our library can be found in our repository here. The source code for all the benchmarks that we developed to measure the performance of our library can be found here.

All the optimizations and changes we implemented in our library had the desired result. In almost all cases, it has much better performance than Box2D and can scale efficiently to tens of thousands of objects. In multiple scenes we simulated, our library is up to an order of magnitude faster. Moreover, the time spent in each simulation step is more stable, resulting in reduced application lag. Apart from improved performance, our library also uses less memory during execution.

A more detailed performance comparison between the two engines can be found in section 8.

2 Introduction to Box2D

In this section we introduce some key concepts and entities in a Box2D simulation, which will be used in the following sections. We also list two simple code examples about creating simple simulations.

2.1 Worlds

In Box2D, the top-level entity that controls a simulation is called *World*. A world instance manages all the information required to simulate a scene and perform queries. An application can dynamically create, destroy and use multiple world instances. Different worlds are disjoint, and their objects do not interact with each other.

The world class provides a function named *step* that advances the simulation by some given time. The step method accepts two arguments that control the number of iterations performed by the solver. More iterations mean greater stability and accuraccy, at the cost of increased execution time.

Listing 1 shows how to create an empty world instance and simulate it for a fixed amount of time.

Listing 1: Creating worlds in Box2D

```
// Create a world with 10 m/s^2 gravitational acceleration
b2World world(b2Vec2{0.0f, -10.0f});

// Advance the simulation by 1/60 seconds for 100 times
for (int i = 0; i < 100; i++) {
  // 8 velocity iterations and 3 positional iterations in the solver
  // This is a good default value for stable and efficient simulations
  world.Step(1 / 60.0f, 8, 3);
}
```

2.2 Bodies

Bodies are rigid objects comprising of one or more parts, named *Fixtures*. The body class contains all the information needed to represent and manipulate a rigid structure. It provides many functions to query information, mutate the body dynamically and alter it's properties.

Each fixture corresponds to a convex shape in a body. All fixtures in a body are simulated as a single solid entity, making it possible to create complex structures. The shapes provided by Box2D are polygons, circles and edges, where the latter is an infinitely thin line segment.

There are three types of bodies

- **Dynamic bodies** have positive mass, and their velocity is determined by all the forces acting upon them (gravity, collisions, user interaction). Their velocity and position are controlled and updated by the physics engine.
- **Static bodies** have infinite mass and zero velocity. This type of object is used to create the static scenery in a world. They can be manually moved by the user.

- **Kinematic bodies** have infinite mass but non-zero velocity which is set by the user. Kinematic objects are moved by physics engine.

2.3 Creating Shapes and Fixtures

The shape class represents a single rigid shape that can be used in the simulation. A fixture is an attachment of a shape into a specific body. A fixture stores additional information for a shape that describe its physical properties: density, restitution and friction.

A body is added in a world by constructing a *body definition*. A body definition is a reusable mutable structure that contains all the necesaary information to create a body. The body definition is handled to the world class, which uses this data to allocate and initialize a body instance.

Listing 2 shows how to create bodies with fixtures in Box2D by using body definitions.

Listing 2: Creating and simulating rigid bodies in Box2D

```
// Create a world with 10 m/s^2 gravitational acceleration in the y axis
b2World world(b2Vec2{0.0f, -10.0f});

// create a temporarily body definition struct and populate some of it's fields
b2BodyDef bodyDef;
// Set the body's type to be dynamic
bodyDef.type = b2_dynamicBody;
// Move the body's center to a particular point
bodyDef.position.Set(0.0f, 4.0f);

// Pass the definition struct to the world and create a new body. We get a body instance
    in return
b2Body* body = world.CreateBody(&bodyDef);

// Our body currently has no shapes attached. We'll add a square to it

// Create a new polygon shape and set it to be a square.
b2PolygonShape square;
square.SetAsBox(1.0f, 1.0f);

// Create and attach a fixture with that shape in our body with a density of 1.0f.
// Shapes are deep-copied so we can safely use a pointer to our soon-to-be-deallocated
    polygon shape instance
// Note: If we can modify more properties of the fixture being created we have to use a
    fixture definition struct similar to how bodies are created
body->CreateFixture(&square, 1.0f);

for (int i = 0; i < 100; i++) {
  // Advance the simulation by 1/60 seconds, using 8 velocity iterations and 3 positional
      iterations in the solver
  world.Step(1 / 60.0f, 8, 3);
}
```

2.4 Joints

In addition to bodies, it is possible to create more complex simulations by using *joints*. Joints are constraints that act on one or two bodies and introduce additional limits on their movement. For example a joint can model a spring like constraint that binds two bodies with a spring force. There are multiple joints types available in Box2D, including the Motor, Gear, Rope, Revolute and Distance joints. Joints are created by creating a corresponding joint definition structure.

2.5 Testbed and documentation

Box2D contains a useful testbed application with many scenes and settings. It showcases the different capabilities of the library and eas. It provides a graphical user interface and supports the rendering of bodies, joints and additional simulation information. This makes it a valuable resource to better understand and visualize how Box2D works.

In addition, development with Box2D is aided by the comprehensive documentation it provides. There is a manual that contains implementation details, limitations and common pitfalls. There is also documentation for all the public functions and classes in the library.

3 Broad-phase Collision Detection

3.1 Introduction

Collision detection is the act of computing the pairs of colliding objects for a given set of objects. Collision detection is an early and important stage of the physics pipeline and comprises a performance bottleneck. Due to the overall high cost of computing the exact collision information between two shapes, collision detection is usually split into two sequentially executed phases: the *broad-phase* and the *narrow-phase*. A broad-phase algorithm computes an initial set of *possible* collision pairs by using bounding volumes that enclose and approximate the objects. By using a simple shape for bounding volume, the broad-phase can quickly discard pairs that are guaranteed to not collide, essentially pruning the collision pair set. After the broad-phase computes the set of potential collisions, the narrow-phase algorithm performs exact collision detection and computes additional information needed to resolve the collision later. Any false-positives reported in the broad-phase are discarded in the narrow-phase.

Carefully choosing the algorithms and data structures used in the broad-phase is essential to optimize performance and reduce the work done in subsequent stages. We will examine some well-known algorithms to understand their benefits and weaknesses and finally design the new broad-phase collision detection stage for our library.

Problem statement We define the broad-phase problem

Given a set of bounding volumes BV compute the set $P = \big\{\{\alpha, \beta\} : \alpha, \beta \in BV \wedge \alpha \text{ collides } \beta\big\}$

Note that the collision pairs are sets of two elements rather than tuples. This is done deliberately to state that the order of the two colliding bounding volumes is irrelevant.

3.2 Broad-phase algorithms

Brute force broad-phase We first examine the naive brute force algorithm for broad-phase collision detection, which is useful to evaluate the performance of other algorithms. The brute force approach involves enumerating all the possible pairs of objects and then testing whether their bounding volumes collide. The pseudocode below is a possible implementation of the brute force algorithm.

Listing 3: Creating worlds in Box2D

```
Algorithm brute force collision detection

Input: BV a list of bounding volumes
Output: P a set of pairs {a, b} where bounding volumes a and b collide

P = {}

for i in the range [0, |BV|) do
  for j in the range (i, |BV|) do
    if BV[i] intersects with BV[j] then
      P = P ∪ {{BV[i], BV[j]}}
    end
```

```
  end
end

return P
```

The inner loop of the algorithm begins from i instead of 0 in order to not report unnecessary duplicate pairs. The time complexity of this algorithm is $O(n^2)$, where $n = |BV|$ is the number of objects.

In the worst case, the number of colliding pairs is also $O(n^2)$, when all bounding volumes collide with each other. This worst-case output of size $O(n^2)$ means that $O(n^2)$ is a lower bound for the broad-phase problem; All algorithms will perform at least $O(n^2)$ operations in this case. The brute force approach could be considered optimal in this sense, but in reality it has weaknesses that make it unsuitable to use. The major issue of this algorithm is that it will perform the same number of steps, $O(n^2)$, regardless of how many collisions there are. Even when there aren't any collisions it still has to check all the pairs.

The worst case of $O(n^2)$ collisions arises only in degenerate scenes of little interest. In typical scenes, there are much fewer collisions and they can be computed efficiently. For the broad-phase, worst case complexity analysis in terms of the number of objects is insufficient to study the problem. We need to use *output-sensitive* analysis instead, where both the input and output sizes are taken into account. For example, an algorithm with performance $O(nlogn + k)$, where k is the output size (collision pair count), is much preferred over the brute force. In practice, we can usually expect $k = O(n)$ rather than $k = O(n^2)$, which is what makes other algorithms much faster and interesting.

Sweep and Prune broad-phase Sweep and Prune [3] [11] is an efficient broad-phase collision algorithm that is widely used and has many applications. The algorithm works by projecting all the bounding volumes in a single dimension and then scanning for collisions in the projections of the objects. If the projections of two objects do not overlap, then neither the objects can collide. As a result, collision tests can be limited to objects whose projections collide. By first sorting the projections intervals, the above procedure can be efficiently performed with a single scan. We present a simplified implementation of the algorithm below.

```
Algorithm Sweep and Prune

Input: BV a list of bounding volumes
Output: P a set of pairs {a, b} where bounding volumes a and b collide

P = {}
active = {}

proj = {[b, e] : [b, e] is the projection of a bounding volume from BV }
svalues = a sorted list of all the values from the intervals

for each number i in svalues do
  b = the bounding volume corresponding to the value v

  if v is the beggining of an interval do
```

```
    for each bv in active do
      if the b collides bv then
        P = P ∪ {{b, bv}}
      end
    end

    active = active ∪ { b }
  else if v is the end of an interval do
    active = active−{ b }
  end
end

return P
```

The reason this algorithm is more efficient in practice is that the size of the active set is much smaller than the total number of objects. By using an appropriate data structure for the active set that supports fast insertions and arbitrary removals, most likely a linked list, we can support those operations in $O(1)$ time. An improved but more complex version of the algorithm is to maintain the output set and perform individual operations on it instead of recomputing it in each call.

The algorithm requires the interval points to be sorted before the scanning phase begins. There are $2n$ numbers to sort at the beginning of each step. This can be a performance bottleneck for the sweep and prune algorithm, especially if a classic sorting algorithm like quicksort or mergesort is used to sort the array from scratch. The key observation to reduce the sorting time is that when advancing the simulation time we expect most objects to not move far from their previous position. When this property holds we say that the simulation exhibits *temporal coherence*. Due to the temporal coherence, it is a better choice to use an *adaptive* sorting algorithm for the sorting between consecutive steps. An example of an adaptive sorting algorithm is Insertion sort. When each object is no more than k places away from its sorted position the time complexity for insertion sort is O(kn).

In [23] the runtime of the Sweep and Prune algorithm is carefully analyzed, with assumptions about the distribution of objects and their velocity which apply to typical scenes. From their analysis, we can claim that in the two-dimensional case the complexity is close to $O(n^{3/2} + k)$ for scenes that satisfy the assumptions.

Sweep and Prune is a good broad-phase algorithm, but it has some weaknesses that make it unsuitable to use in our physics engine:

1. Far-away objects may still be tested for intersection if their projections happen to overlap. This means that there will still be a lot of unnecessary collision tests for simulations with a high number of objects.

2. Dynamic scenes with a lot of fast-moving objects can greatly reduce performance, because of the reduced temporal coherence.

3. It can not be easily used to speed up other types of queries, for example, raycast queries.

There are improvements that can be made to address some of these issues, as the ones presented in [23].

Spatial subdivision broad-phases Spatial subdivision algorithms are a family of broad-phases that work by dividing space into sub-regions. Each time space is further divided, fewer objects lie inside each region and hence fewer objects must be tested against each other for collision detection. There is a wide variety of spatial subdivision data structures and algorithms that have been researched such as grids, binary space partition trees, k-d trees [26] and hierarchical spatial hashing [12].

One complication with spatial subdivision is that a single object can reside in multiple regions. Objects that exist in multiple regions are hard to handle efficiently and lead to decreased performance.

Bounding Volume Hierarchy broad-phase A Bounding Volume Hierarchy (BVH) is a tree-like structure that forms a hierarchy of bounding volumes. All the objects reside in the leaves and internal nodes maintain a bounding volume that is the union of the bounding volumes of their children. BVH trees are a form of object partition instead of a spatial partition. The benefit of using a BVH tree is that query operations initiate in the tree root and only visit subtrees whose bounding volume overlaps with the query object. This can be used to efficiently implement collision detection, raycasting and point queries.

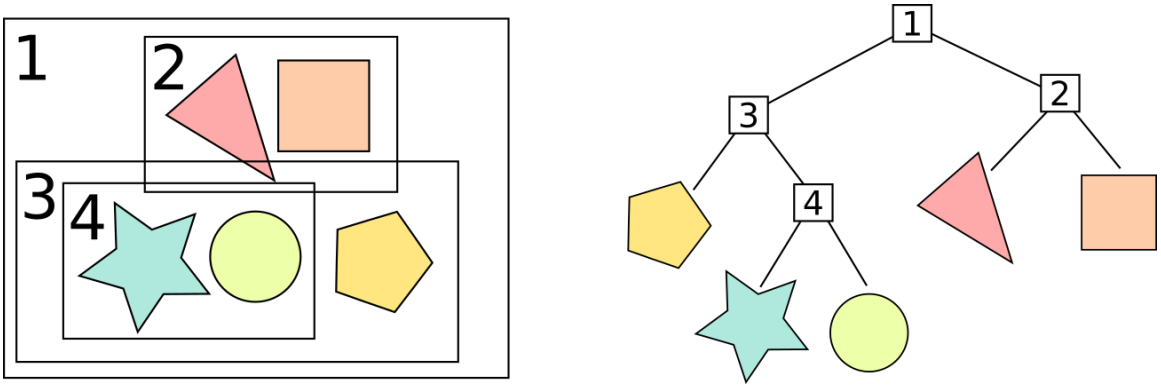


Figure 2: An example of a Bounding Volume Hierarchy

In order to actually speed up operations, a BVH tree must have some desired properties:

1. The overlap of sibling bounding volumes in the tree should be minimal.
2. The total area of tree should be minimal.
3. Objects that are close to the leaf layer should be close to each other in space.
4. The tree should be balanced to some extent to avoid degenerate list-like trees.

We need to maintain a tree of good quality, that is a tree that approximately satisfies those properties. There is a variety of algorithms and associated techniques to build a good BVH tree from a given set of objects [24] [17]. We can make a distinction between top-down, bottom-up and insertion-based construction algorithms. Bottom-up algorithms produce high quality trees but are very expensive. A greedy area-minimizing bottom-up algorithm has a complexity of $O(n^2 logn)$, which is prohibiting to use. Top-down algorithms do not produce as good trees but are much faster, while maintaining decent quality. Insertion algorithms build the BVH by inserting the objects one by one. The most commonly used algorithms use the top-down construction approach.

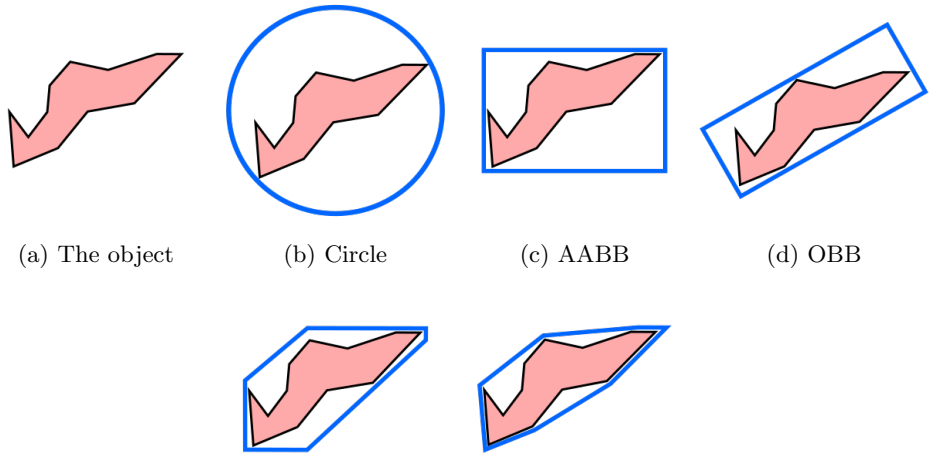
There is a variety of top-down BVH-building algorithms that produce good trees and have a running time $O(nlong)$. A widely used method for building high-quality trees is the Surface Area Heuristic (SAH) that was introduced by Goldsmith and Salmon in [14]. Another choice is to split the objects into two subtrees based on the barycenter of the object centers [15]. This technique is very efficient but produces BVH of slightly inferior quality. Below is an implementation in pseudocode for the top-down barycenter split algorithm.

Listing 4: Top-down BVH construction

```
Recursive Algorithm for top-down BVH construction

Input: BV a list of bounding volumes
Output: A BVH subtree for those nodes

if BV consists of a single volume then
  return that volume
end

unionBV = a bounding volume initialized to center(BV[0])
for each bounding volume bv in BV do
  unionBV = unionBV ∪ center(bv)
end

axis = maximum axis of unionBV
split = center(project unionBV to axis)

leftSubtree = {}
rightSubtree = {}

for each volume in BH do
  if center(project volume to axis) < split then
    leftSubtree = leftSubtree ∪ volume
  else
    rightSubtree = rightSubtree ∪ volume
  end
end

current = a new BVH tree node
current.left = Top-Down-BVH(leftSubtree)
current.right = Top-Down-BVH(rightSubtree)

return current
```

The BVH tree is a general acceleration structure for spatial data and serves other purposes apart from broad-phase collision detection. There are a lot of ways to accelerate collision detection using a BVH, with the most obvious being to just sequentially test each bounding volume for collision with the tree itself. We will explore faster and more advanced collision detection schemes in subsection 3.5.

An crucial choice regarding a BVH is the type of bounding volume it uses. Common choices include Circles, Axis Aligned Bounding Boxes (AABBs), Oriented Bounding Boxes (OBBs), Discrete Oriented Polytopes (k-DOPs) and convex hulls. Two qualities affect our choice of bounding volume: how good approximation it is and what is the cost of performing geometric operations with it (especially the intersection and union operations).



(a) The object (b) Circle (c) AABB (d) OBB

(e) k-DOP (f) Convex hull

Figure 3: Usual choices for bounding volumes

Both Box2D and our library use AABB bounding volumes for the broad-phase. AABBs are good approximations in most cases and their geometric operations can be implemented very efficiently. They usually provide the best performance compared to other primitives. The core weakness of AABBs is that for shapes resembling diagonal line segments, the AABB is a bad approximation. We will evaluate the performance of this particular case later in the benchmarks section.

Aside from AABBs we also evaluated and experimented with using circles as a possible replacement, but they proved to be slower in all cases. Even in scenes made up entirely from circles, where circle bounding volumes minimize the number of reported broad-phase pairs, the AABB-based broad-phase performed better due to its faster primitive operations. The rest primitives are too complex and even basic operations such as a point test require an order of magnitude more CPU-time, hence they were not considered for our broad-phase.

3.3 Broad-phase collision in Box2D

Box2D uses a broad-phase based on a BVH tree. In order to correctly represent the scene as the simulation progresses, a number of structural modifications operations are applied to the tree in each time step: Insert, Update and Remove. These operations are implemented to act on individual objects. For example, in order to initially build the tree, each object is inserted one by one (insertion-based BVH construction). The BVH tree is height-balanced in order to ensure logarithmic height with respect to the object count. This is important because it directly affects the performance of Insert and Delete operations as with most types of trees.

One particularly important operation is the Update operation. Because most bodies have non-zero velocity (except for those who are static or idle), the Update function will be called a significant number of times in each step. It is important that it is efficient and does not affect the quality of the tree negatively.

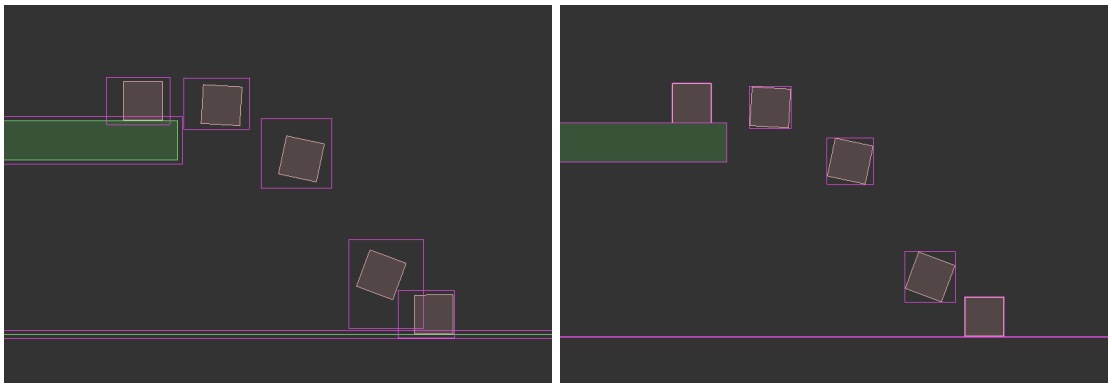
In Box2D the Update operations equals to first removing the object from the tree and then re-inserting it. After the new insertion both the updated object and the bounding volumes of the internal tree nodes will be in new valid positions. This is a straightforward approach but unfortunately has high cost. In order to make up for the high update cost, Box2D employs an additional strategy to reduce the amount of needed Update calls: *AABB enlargement*. It does not use bounding volumes that are a tight fit for the fixtures but instead enlarges them. As long as a fixture stays inside its *enlarged* bounding volume there is no change in the broad-phase level and hence no Update operation needs to be performed.

This bounding volume enlargement scheme is based on an enlargement strategy that defines how the volume is enlarged. The current implementation applies a small constant omni-directional enlargement plus a larger one based on the object's displacement relative to the previous simulation step. This is an heuristic that attempts to predict the body's movement in the future and as a result reduce the required Updates. Only when the object crosses the enlarged volume an Update happens again.

It's worth noting that the enlargement strategy is of core importance to the performance of the library. Tweaking the enlargement factor and heuristic immediately changes the performance characteristics of the simulation. It is possible to make different trade-offs by changing the enlargement algorithm and there is potential for further optimizations if extra assumptions can be made about a simulation.

One additional effect of enlargement is that static bodies don't need to be tested at all in the broad-phase except from the first frame they're added to the world and if explicitly moved by the user. Thus static bodies on average have a reduced impact on performance compared to dynamic bodies. It is common to simulate scenes with a large number of static bodies, for example, a platformer game where all the level is static except from the enemies that can move.

After the BVH tree is updated, broad-phase collision detection can be performed. The collision detection algorithm employed by Box2D works as follows. Because Box2D caches the collision pairs from previous steps (see section 4) collision detection for an object need only run each time it crosses its enlarged volume boundary. The broad-phase keeps track of the movement of the fixtures and



(a) Box2d (b) Our implementation

Figure 4: Enlarged AABBs in Box2D versus tight fitted AABBs

when the fixture is no more contained in the enlarged bounding volume of the BVH tree it is added, among others, in a queue of fixtures that must be tested for collision. When it's time to update the collision pairs, all fixtures in that list will be queried against the BVH tree one by one, in order to find the new contacts. This is the core function of the broad-phase where new pairs are found and a quite expensive operation that should be optimized as much as possible. In a dynamic scene, the CPU time for this stage takes up a considerable fraction of the total CPU time for the simulation step.

3.4 Evaluating Box2D's broad-phase

We reviewed the structure of the default Box2D broad-phase and what optimizations it employs to enhance collision detection performance. We will now evaluate it's performance characteristics and weaknesses in order to gain insight and improve upon that.

No batch-operations As we described previously, all operations in Box2D's broad-phase are implemented to operate with a single object at a time. Yet, the library needs the broad-phase to operate mostly in batch; once per step all the objects are Updated and Queried at once. This limitation to operate on each fixture individually means that useful global information is lost. This information could be used to improve the quality of the BVH and reduce the necessary checks. We already examined more efficient algorithms for batch operations, for example, top-down BVH building algorithms.

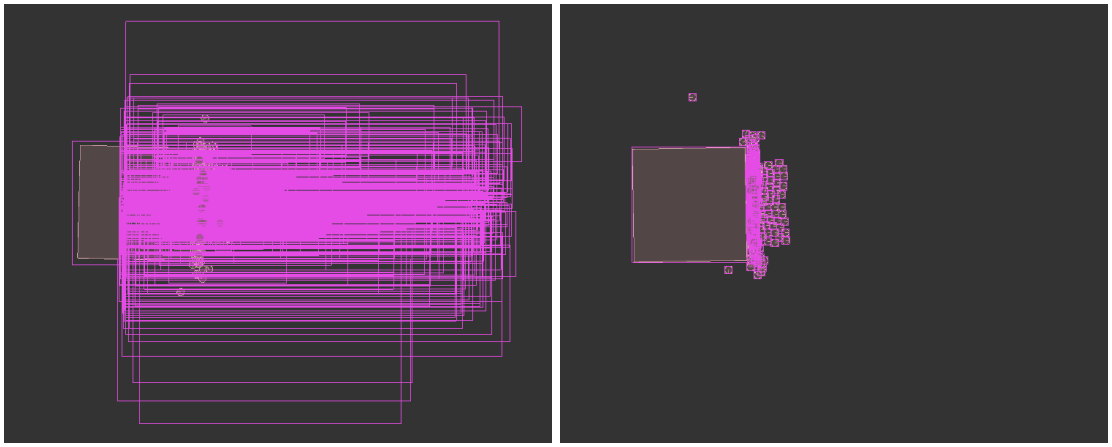
Bounding volume enlargement Although enlargement is used as an optimization and achieves specific advantages, it comes at a great cost. In a high-quality BVH, we want to minimize surface area and sibling overlap, yet enlargement directly increases both of these metrics. This negatively impacts performance and makes the broad-phase collision detection phase much slower. At the same time, since we're performing collision detection on the enlarged volumes, the number of reported collision pairs greatly increases. More pairs need to be stored and processed which slows down subsequent stages in the physics pipeline and increases memory usage. In scenes with fast-moving objects, this is detrimental because their predicted AABB is very large and the number of false-positives skyrockets. We examined some typical scenes with fast-moving bodies or collision of large

groups and found out that the number of contacts can become up to an order of magnitude larger than the real collision count.

Unpredictable running time Since Box2D is used in a lot of video games one important weakness of the specific broad-phase algorithm is that its running time is highly unpredictable in each step. Video games usually have a very limited time budget in order to achieve a desired framerate. A sudden spike in the physics engine will produce a very noticable and unpleasant lag in the application. There are two sources of unpredictability to consider that create this effect.

The first problem is that the number of fixture updates and queries per step is very unpredictable. It equals the number of objects that happened to cross their enlarged bounding volume, which in turn can be anything from zero to all. Because the update and query operations are a time consuming part of the broad-phase, this leads to great instability. The timing of those operations can be considered semi-random as it depends on object position, velocity, creation step, and usually cannot be controlled. In the best case, a similar number of objects will happen to cross their enlarged volume in each step hence providing stable performance. Unfortunately, a lot of times this does not happen, leading to a sequence of spikes in the running time of the broad-phase. In the worst case, if we add similar bodies with similar speeds in a single step then most subsequent steps will have almost zero cost and a single one will have to perform all the costly broad-phase operations.

The second problem is that sudden collisions create a very large amount of false-positives. This can be considered a frequently occuring case, for example, a fast body colliding a group of idle bodies or a group of bodies in free fall hitting the ground. At the critical time step where multiple collisions suddenly happen, the enlarged AABBs create many more additional pairs and the physics pipeline greatly slows down. The result is considerable lag. Box2D does not limit the maximum amount of enlargement that is applied to bounding volumes. The result can be similar to what is seen in Figure 5.



(a) Box2d (b) Our implementation

Figure 5: Massive overlap due to enlargement at the critical time step of a simulation

Reduced tree quality The update strategy used in Box2D is to remove and re-insert bounding volumes from the BVH tree. This strategy is not particularly efficient. Due to the temporal and spatial coherence that simulations exhibit between steps, when an object crosses its enlarged AABB it usually is still nearby. Although the update could be done locally in the tree, the current implementation completely removes and re-inserts the object. This is wasteful and much better alternatives exist. For example, in [16] efficient updates are done locally with few tree rotations while also preserving the quality of the tree. At the same time, because the tree is balanced by height, each update operation can create lasting inefficiencies in the tree. BVH trees are not very suited to height balancing because it does not take into account if the overlap and surface area increase. Over time, the continuous restructuring and rotations decrease the tree's quality without any good means to recover.

3.5 Designing new collision detection algorithms

We explained why the broad-phase is a crucial stage in a physics engine and how the default Box2D broad-phase and collision detection algorithms have inefficiencies. In our first step towards our optimized physics engine, we will completely redesign and re-write the broad-phase. Our end goal is to design a broad-phase that is both faster and has more stable performance.

Based on our observations, the first important decision was to not use fixture enlargement. While enlargement is an interesting concept, it introduces hard limits to the maximum performance that can be achieved. On average, it greatly increases the number of collision pairs that the algorithm must compute. At the same time it leads to a BVH tree of lower quality. This means that the collision detection and raycast operations will be slower on average.

Choosing not to use enlargement means we will have to update and query all the fixtures at each step, because even the slightest fixture movement will result in the invalidation of its AABB. We'll have to design very efficient algorithms for those operations to overcome the amount of work that needs to be done.

Since we'll have to update all the bounding volumes at each step, we will design structures that work in batches of objects. This is in contrast to what Box2D does. We need a performant and scalable BVH building algorithm that produces trees with good quality. The top-down building algorithms fit our needs. We implemented and evaluated both the SAH heuristic and the barycenter split and choose the barycenter split due to its efficiency and simplicity. This BVH construction algorithm produces better quality trees than Box2D's insertion-based algorithm. At the same time, the cost of building the entire tree is a fraction of Box2D's total broad-phase time. We have the margin to implement the collision detection algorithm. With the tree building algorithm in place, we need to look into the harder problem of how to efficiently perform collision detection with it.

Collision detection The initial straightforward algorithm is to just take each AABB and query it against the BVH tree itself, very similar to what Box2D does. Box2D uses an iterative algorithm and performs depth-first search in the tree by using a heap-allocated stack. We found that a recursive algorithm instead is faster in practice while also easier to write.

Listing 5: AABB Query in a BVH

```
Algorithm Query-AABB
```

```
Input: BV an AABB to query against the tree
Output: a set of bounding volumes that collide with BV

result = {}

if BV overlaps with the AABB of the tree root then
  Query-AABB-rec(root, BV, result)
end

return result
```

```
Algorithm Query-AABB-rec

Input: current a BVH subtree,
       BV an AABB to query against the tree,
       result the output set

if current is leaf then
  // we found a collision
  result = result ∪ { current }
else
  left = left child of current
  right = right child of current

  if BV overlaps the AABB of left then
    Query-AABB-rec(left, bv)
  end

  if BV overlaps the AABB of right then
    Query-AABB-rec(right, bv)
  end
end
```

Listing 6: First algorithm for global collision detection

```
Algorithm Global-Collision-Detection1

Output: P a set of pairs {a, b} where bounding volumes a and b collide

P = {}

for each bounding volume bv in the BVH tree do
  query-result = Query-AABB(bv)
  current = { bv }

  P = P ∪ { query-result × current }
end

return P
```

This approach is sufficient to perform global collision detection but it is not very efficient. The AABBs are queried one by one, hence we traverse the tree once for each fixture. It would be better to traverse the tree just once and process all the collisions in a single pass. We need to build a batch-detect algorithm.

Listing 7: First algorithm for batch AABB queries

```
Algorithm Query-All1

Input: current a node in the BVH,
       AABBS a set of AABBs to query

Output: P a set of pairs {a, b} where a is in AABB and b in BVH and a collides with b

if current is leaf then
  output = output ∪ {AABBS × {current}}
else
  left = left child of current
  right = right child of current

  descendLeft = {a in AABBS ∧ a intersects the AABB of left}
  descendRight = {a in AABBS ∧ a intersects the AABB of right}

  Query-All1(descendLeft, left)
  Query-All1(descendRight, right)
end
```

The approach used in the above pseudocode indeed traverses the tree just once and performs collision detection with the given AABBs. By calling this function with input the set of all the AABBs we can perform global collision detection. One issue here is that the pseudocode implies that two sets are created in each recursive call, which is inefficient and a major waste of memory. We can implement this function without creating two new arrays each time. Below is an in-place version of this algorithm that uses no additional memory.

Listing 8: Second algorithm for batch AABB queries

```
Algorithm Query-All2

Input: current a node in the BVH,
       AABBS an ordered list of AABBs to query,
       N the count of AABBs to use from the list

Output: P a set of pairs {a, b} where a in AABBS and b in BVH and a collides with b

if current is leaf then
  output = output ∪ {AABBS × {current}}
else
  left = left child of current
  right = right child of current

  re-arrange the elements of AABBS so that all AABBs that collide with
```

```
  the AABB of left are in the first LC slots (in-place two group sorting)

  Query-All2(AABBS, left, LC)

  re-arrange again the elements of AABBS so that all AABBs that collide with
  the AABB of right are in the first RC slots (in-place two group sorting)

  Query-All2(AABBS, left, RC)
end
```

Listing 9: Second algorithm for global collision detection

```
Algorithm Global-Collision-Detection2

Output: P a set of pairs {a, b} where bounding volumes a and b collide

root = the root of this BVH
all-aabbs = the set of all AABBs in this BVH

Query-All2(root, all-aabbs, |all-aabbs|)
```

This modified version uses no additional space by rearranging the elements and using a size parameter that indicates how many elements in the array to use. This algorithm is faster than the one that does individual queries. We can attribute that to not traversing the tree multiple times and being more cache-friendly. While this is an improvement there are still some important issues that prevent using this algorithm for global collision detection. In order to perform collision detection we have to call this algorithm with initial input the AABBs that make the BVH tree. In that case there are two major drawbacks.

The first problem is that this algorithm reports duplicate pairs. We express pairs in the pseudocode as sets to show that we don't want both a pair (a, b) and its reverse (b, a) to be included in the output. When all the AABBs are given as input, this algorithm will find and report all the pairs and their duplicates. The existing Box2D pipeline has no issue dealing with those duplicates, but this means that unecessary pairs are computed. In this case, the number of unneeded pairs is two times the minimum.

There is a second and more severe problem with this approach. We mentioned that to perform collision detection we call this procedure with the list of all AABBs as input. Because we query the tree with bounding volumes that are at the same time leaf nodes, each AABB will collide all the internal node AABBs in at least one path (the path leading to itself). As a result the number of AABBs in the recursive left and right calls, which determines the running time of this algorithm, will decrease slowly. At each recursive call we'll have at least as many AABBs as the number of objects in each subtree. An AABB that does not collide with any other in a subtree will have to follow that path just because it collides with itself.

All these inefficiencies have to do with the fact that we query the tree against itself. If the query set and the tree bounding volumes are disjoint, then this algorithm is much more efficient and does

not have either of the above issues. We will need this algorithm later and use it for this case. For now, we need to design an algorithm that is fast when the input is the set of AABBs present in the tree. We will use the fact that the input set is the same as the set of bounding volumes in the tree leaves and create a special algorithm that is much faster.

In each level of the tree we can imagine that the two sibling nodes try to separate the AABBs contained in each subtree as much as possible. As a result, given a BVH tree of good quality, there will be few AABBs in each node that overlap an AABB from its sibling node. For the same reason there will be few AABBs from each node that overlap with the union AABB of its sibling node. The AABBs that form possible collision pairs are the few AABBs that overlap the sibling's union AABB. All the other AABBs cannot generate a pair with the sibling sub-tree and are pruned.

The idea is to assign a candidate collision set to each node that contains all the AABBs that may overlap with its leaf nodes. Initially, the root has an empty candidate set. We then perform a pre-order traversal of the tree and at each node we choose one of the two children sub-trees to test if its AABBs overlap with the union AABB of the sibling node. This can be tested with a simple loop in time linear to the number of AABBs in the sub-tree. We then append the AABBs that collide with the union AABB of the sibling to the sibling's candidate collision set; those are the bounding volumes that may collide with some of its children. We do not test both children against each other but only one of them. This removes all the duplicate pairs from the output because if an AABB a is added to the candidate set of the sibling sub-tree that contains the AABB b then b *won't* be added to the sub-tree that contains a nor will it be tested for overlap with it. When we reach a leaf node, its candidate collision set will contain the AABBs that collide with the AABB of that leaf node. Those are the pairs reported by the algorithm.

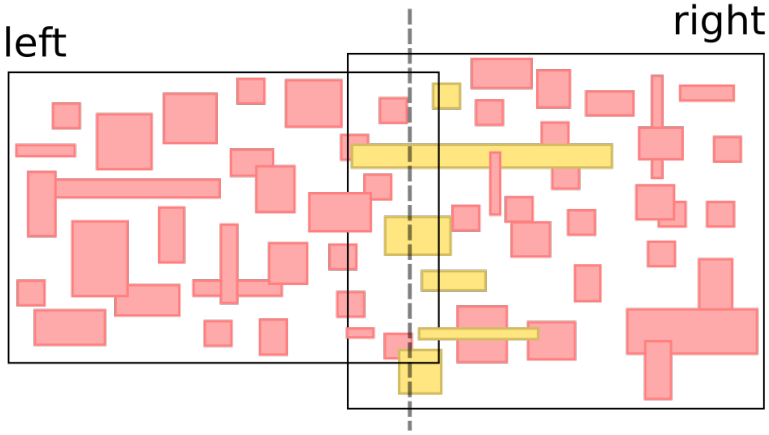


Figure 6: Collision detection between siblings in the BVH tree

Figure 6 depicts a graphic representation of this idea. The red AABBs are the AABBs contained in each sub-tree while the black surrounding rectangle is the union AABB of each child. The dashed line is the separation line by which the AABBs were split into a left and right sub-tree. In this case, the right child was selected and its AABBs were tested for collision against the union AABB of the

left child. The yellow highlighted AABBs are the ones that collide with the union bounding volume and hence are added to the candidate collision set of the left child. In top-level nodes, where each sub-tree contains a high number of children, we can expect the yellow nodes to be a small fraction of the total AABB count, similar to what can be seen in the figure. This follows from our observations about the minimized overlap of sibling nodes.

We design the following algorithm which implements the above idea and propagates candidate AABB collisions to one of the sub-trees.

Listing 10: Specialized algorithm for batch collision detection

```
Algorithm Query-All3

Input: current a node in the BVH,
       AABBS the candidate collision set of the node current

Output: P a set of pairs {a, b} where a is in AABB and b in BVH and a collides with b

if current is leaf then
  output = output ∪ {AABBS × {current}}
else
  left = left child of current
  right = right child of current

  descendLeft = { a in AABBS ∧ a intersects the AABB of left }
  descendRight = { a in AABBS ∧ a intersects the AABB of right }

  propagatedAABBs = { a is an AABB of left ∧ a intersects with the union AABB of right }
  augmentedRight = descendRight ∪ propagatedAABBs

  Query-All3(descendLeft, left)
  Query-All3(augmentedRight, right)
end
```

Listing 11: Third algorithm for global collision detection

```
Algorithm Global-Collision-Detection3

Output: P a set of pairs {a, b} where a is in AABB and b in BVH and a collides with b

root = the root of this BVH
initial-candidates = {}

Query-All3(root, initial-candidates);
```

Note how the input in the initial recursive call is the empty set this time. This is an important point on the efficiency compared to the previous global collision detection algorithms. In the top-level nodes which have a lot of children AABBs that need to be tested the candidate collision set contains very few elements that will be propagated.

One important detail is that the above pseudocode again implies that new sets are created in each recursive call, which is wasteful and needs to be improved in a similar fashion that we improved the 'Query-All2' algorithm. Unfortunately, because the algorithm both removes and appends items in the candidate set, it is not possible to implement it in-place. We still applied the same technique so only one new set is created in each call and used a number of tricks in the actual implementation to reduce the overall memory consumption. The upper bound for temporary memory used is $O(nlogn)$ but our implementation uses much less memory for all the tested simulation scenes.

3.6 A fast unified collision detection algorithm

At this point, we have an efficient BVH construction algorithm and an efficient algorithm for global collision detection. We did implement those two algorithms in our library and managed to achieve a measurable speedup over Box2D's broad-phase. Still, there are some rough edges and potential to improve our algorithm.

The key insight to further optimize the broad-phase is to merge the construction and the detection phase into one unified collision detection algorithm. Both algorithms process the data in the same way and can be merged in a straightforward manner. One issue is that the detection algorithm uses the union AABBs of the subtrees to find which AABBs to append in the candidate collision set. Because the union AABBs are normally computed after the recursive calls to the BVH construction algorithm, if we merge the algorithms we won't have the union AABBs computed when they're needed. The solution is to compute the union AABBs before the recursive calls. There is already a loop that splits the nodes into two groups. At the same time, we compute the union AABBs for each group and then use them. Although this results in some redundant work done (the same can be computed with a single union operation if it was after the recursive calls) the overall gains outweigh this minor overhead.

This unification of the algorithms results in another measurable performance boost to the broad-phase. It is more cache-friendly since the data is used when it is produced and there's no need to read the tree structure from the memory in the collision phase. Moreover, by doing both the construction and the detection at the same time we need to perform just one traversal of the data. There is more work to do per node but modern superscalar CPUs can take advantage of this and execute independent instructions in parallel. As a result, it is more efficient to do both tasks at once. One interesting point is that the new algorithm does not need the tree stored in memory, since it operates on it in a signle pass. In our case, we do store the tree because it is used for possible AABB or raycast queries, but in other scenarios it could be used without persistent storage of the tree structure.

3.7 Loose ends

3.7.1 Fast broad-phase for partially static worlds

Careful profiling and benchmarking revealed that the new broad-phase is much faster than the existing Box2D counterpart, except from one case: worlds that contain only or mostly static bodies. We remind that the Box2D broad-phase performs collision detection and updates only on objects that move outside their enlarged bounding volumes. In contrast, our broad-phase computes the collision pairs from scratch in each step. The latter is very inefficient in the specific case where the world is mostly static. It is apparent that to achieve similar performance we need to modify our design.

Our approach to achieve comparable performance with our design is to split the BVH tree in two BVH trees, one with only static bodies and the other with the rest. By doing this we can persist the static-body BVH into memory; there will be no need to update the tree in each step. We also remind that Box2D does no report static versus static collision pairs because those have no effect. This means that we need not perform collision detection at all in the static-body BVH, which is a net gain and very important when there is a high number of static bodies. We will just need to perform collision detection between the static-body BVH and the dynamic BVH. At this point, we recall the collision detection algorithm 'Query-All2' that we designed in subsection 3.5. Although inefficient for global collision detection, as we commented, when the set of query objects is disjoint from the set of AABBs in the BVH then this algorithm is efficient and makes sense to use. This fits perfectly our needs here. Since each body is either static or not, the static-body BVH is completely disjoint with the dynamic one. In our implementation, we use this procedure to perform collision detection between those trees.

This technique works as expected and for mostly static worlds the broad-phase performance comes close to what Box2D offers, lagging behind by a small factor. We note that this double-BVH setup introduces an inherent overhead and that splitting a larger BVH into two smaller ones is inferior when compared to a single bigger one. That is because the overlap between the root of the two trees can no longer be controlled or minimized. For this reason, we use a threshold to control when this optimization is enabled. Our choice is to enable it when the number of static bodies exceeds 1/8 of the total object count.

3.7.2 Implementing individual operations on the BVH

Our new algorithm makes up the core of the new broad-phase and it operates in batches of fixtures to achieve good performance. Box2D provides individual operations in its API (Insert, Update and Remove of single fixtures) and we want to provide the same functionality. The BVH tree structure is stored in a very similar way with that of Box2D, so we could implement those operations in similar ways. Instead, we decided to implement them with the existing batch processing algorithms. All the individual operations just invalidate the tree and modify the array containing the tree nodes. This makes all these operations easy to implement and with an (expected) cost $O(1)$. The downside is that if the tree is invalidated, before the next query the tree will have to be rebuilt from scratch. However the construction algorithm is, it can be slower than individual modification algorithms. If there are multiple operations before the next query the cost of the construction algorithm is amortized. In the best case, if there are no queries until the next step the tree will be rebuilt for collision detection at no extra cost. It depends on the workload and the specific order of those operations. We deemed that the sequences that cause measurable performance degradation can be avoided and they do not represent a usual scenario. In any case, it is easy for alternative algorithms to be implemented or for the existing ones to be extended to handle workloads with very specific performance requirements.

3.7.3 Easier to use API

Unrelated to the performance of the broad-phase, we changed the *api* in order to be easier to use.

In Box2D when a fixture is added in a broad-phase an integer is generated that is the handle for this fixture in this particular broad-phase. Function calls (e.g. DestroyProxy, MoveProxy, GetFatAABB) must be given the fixture handle in order to work. This means that the user has to take care and keep track of the handles in order to perform broad-phase operations for a fixture. This also means

that adding a fixture to multiple broad-phases involves keeping multiple handles for a single fixture, which can be inconvenient.

In our library, we decided to remove the use of such handles. The fixture itself is the handle to the broad-phase, making it easy to perform operations. Using the same fixture in multiple broad-phases is easier and does not requires to maintain multiple handles. This is implemented by using a hash table to map fixtures to internal tree nodes. Each fixture has a unique id that is used as a key in the hash table to enable fast lookups. The implementation uses separate chaining and Fibonacci hashing, which is a good combination for serial integer keys. The lookup cost is a tiny fraction of each operation and can be considered negligible.

3.8 Performance evaluation

Below we summarize the changes that we made in the broad-phase:

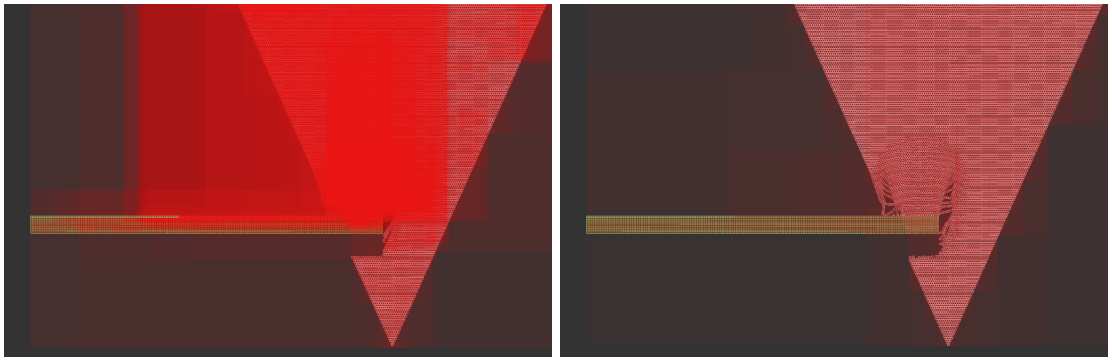
- No AABB enlargement: the number of collision pairs is minimized and the rest of the physics pipeline has fewer pairs to process.
- Efficiency: The new unified collision detection algorithm processes all the fixtures in batch and achieves a significant performance boost.
- Stability: The time spent in the broad-phase is consistent and predictable between steps; it only depends on the number of bodies and collisions. Velocity, position in previous steps, time of creation and iteration order do not affect the performance.
- Memory usage: The broad-phase reports the minimum amount of pairs and thus reduces memory consumption in the contact manager. On the contrary, it uses more temporary memory in the collision detection phase. Because the collision pair structure occupies more memory than the tree node it is still an overal improvement to memory consumption.
- BVH Quality: The quality of the BVH tree that the broad-phase produces is superior. There is less overlap between sibling nodes and the tree nodes combined have smaller surface area. This means that further AABB and raycast queries will execute much faster even with no changes to the query algorithm.
- Fast processing of static bodies: Static bodies are handled in a special way to ensure good static body performance and the ability to efficiently simulate very large static worlds.
- Easier to use *api*.

Following all the changes in the broad-phase we will now measure and evaluate the difference between our broad-phase and Box2D's. We will compare the time spent in the broad-phase and the quality of the generated BVH trees for each of the first 100 steps of two sample simulations. In all cases, the measurement of execution time were made in a consistent environment, exactly as described in section 8.

The first scene is a variant of the Box2D benchmark "Tiles". A huge inverted pyramid of squares collides with a thick ground made up of small static squares. This scene comprises of more than 22 thousand fixtures and is a designed to stress the broad-phase. The bodies fall slowly and gradually generate more collisions.

The second scene is the old Box2D benchmark named "Add pair". A big body with very high velocity collides with a group of idle circles in a zero-gravity environment. The scene is made up of only 400 bodies. The critical time for this test is when the big object first collides with the idle objects. A big number of contacts is suddenly created and lasts for the next few steps.

We will evaluate the quality of the BVH trees visually, by rendering the union AABBs of all the internal nodes with a semi-transparent color. In places where there is a lot of overlap the opacity of the rendered AABBs adds up. The more overlap there is the less transparent that area becomes. We implemented this type of BVH rendering on top of the Box2D testbed application to examine overlap in the various scenes. In all simulations, we use the same opacity for the coloring. The default settings of the Box2D testbed were used but with continuous collision detection disabled.



(a) Box2d (b) Our implementation

Figure 7: BVH Overlap visualized in the *Tiles* scene.

Tiles Rendering the BVH nodes in the *Tiles* scene makes it easy to see the excessive overlap in the BVH built using Box2D's broad-phase (Figure 7). In both cases, the BVH trees have the same height, yet with our algorithm the splitting of nodes is much better and the whole scene has a consistent light red shade. In Box2D, we can see a huge opaque red square even in places with little to no bodies. This means that far away fixtures were grouped in the same subtree multiple times, which is the source of major inefficiencies in a BVH tree.

Enlargement is not very important in this case because the bodies are moving slowly. The low quality of the tree is the result of the individual updates and the height balancing.

In Figure 8 we can see the gross time spent doing collision detection in each step. There are multiple observations to make here. In the beginning we can see Box2D performing collision detection in near to zero time for multiple steps in a row, which is remarkable for such a scene. Soon after that, there is a huge spike going well into $20ms$ just for the broad-phase. This can be attributed to the enlargement heuristic and the slow initial speed of the bodies. Having near-zero speed, the bodies stayed for multiple steps inside their enlarged AABBs but then crossed them at the same time. This pattern continues, but with the cost gradually increasing throughout the simulation. This is a good demonstration of how spikes in the execution time are detrimental for the simulation even if the time spent in some steps is low.

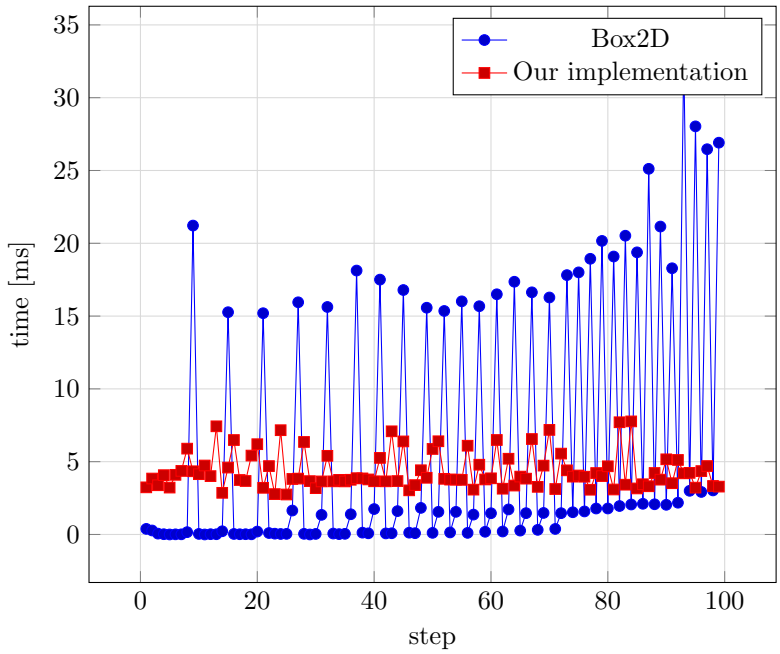
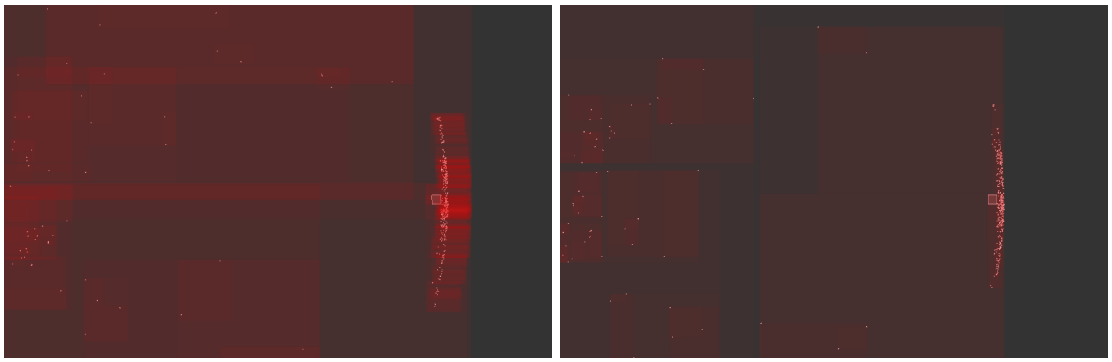


Figure 8: Box2D's broad-phase time is very unstable and unpredictable

Compared to that, our algorithm's running time is much more stable and predictable. The variation between subsequent steps is low and can be attributed to the fluctuations of the number of collisions as the simulation progresses. The increase in collisions after step 70 affects much less the performance, at which point our broad-phase is at least five times faster. This continues throughout the rest of the simulation and the total time spent in the broad-phase in our library is a small fraction of Box2D's.

Add Pair In the *Add Pair* scene, it is possible to examine how enlargement and group collisions can affect the broad-phase. In the left image of Figure 9 each small circle has a very wide AABB extending towards its moving direction which is a result of AABB enlargement and the circles's high velocities. Even if the AABB is sufficient to predict the movement of the bodies in a future step, the performance loss from maintaining the much larger bounding volumes is very high.

The result of the excessive enlargement and the slower collision detection algorithm is very visible in the time plot. The critical step of the simulation is when the fast-moving square collides with the idle circles for the first time, which happens near the $20th$ step of the simulation. Even with just 400 bodies participating in the collision we can observe a huge spike in Figure 10. At the critical step, more than $15ms$ are spent just for collision detection. A very high amount of collision pairs are reported at the same time, greatly slowing down the rest of the pipeline. This produces visible lag in the simulation as it runs in the Box2D testbed. In subsequent steps the running time and the size of the spikes decrease again but that happens slowly. It is probable that lasting inefficiencies are present in the broad-phase which cannot be recovered without a full re-build of the BVH tree.



(a) Box2d (b) Our implementation

Figure 9: Enlargement in the *Add Pair* scene generating a large number of false-positive pairs when bodies have large velocities

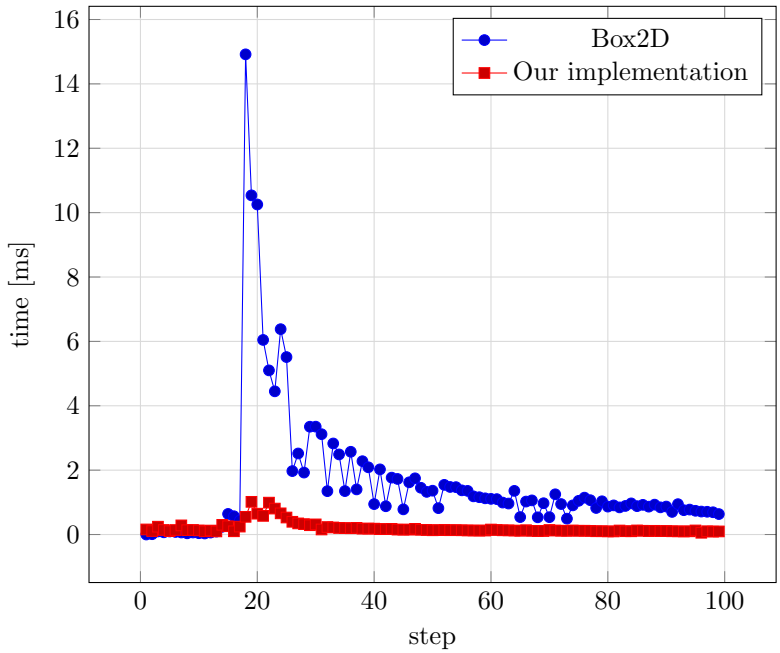


Figure 10: A huge spike created from a momentary collision

Instead, our broad-phase performs collision detection in near-zero time, for as long there are few collisions, and has only a minor spike at the critical step when the collision initiates. Even then the total time spent in the collision detection phase hardly exceeds $1ms$. When the collision happens the number of reported collision pairs in our library is far smaller and doesn't slow down the rest of the pipeline.

4 Contact Management

After a collision pair is found in the broad-phase, it is converted into a *contact*. A contact represents a potential collision between two fixtures and stores additional information that will be used in subsequent stages for collision resolution. The sub-system that creates contacts from collision pairs and manages their life cycle is called *Contact Manager*. The contact manager is responsible for:

1. Accepting new collision pairs from the broad-phase and creating contacts.
2. Maintaining a set of all the contacts that are currently alive.
3. Finding which contacts have persisted from the previous step.
4. Finding which fixtures have ceased to collide and remove their corresponding contacts.
5. Performing narrow-phase collision detection and storing the collision information for each contact.

4.1 Contact management in Box2D

In Box2D, all the contacts are stored in a doubly linked list which is maintained by the contact manager. A doubly linked list is used because it supports efficient insertion and removal of elements in arbitrary positions. This, in turn, is necessary for efficient contact creation and destruction.

The contact manager maintains contacts for all the fixtures that have their enlarged AABBs colliding. It is necessary to keep track of fixtures whose enlarged AABBs collide, even if their AABBs do not. Although this increases the number of live contacts, this is how the broad-phase can perform collision detection only for objects that moved out of their enlarged AABB. The contacts of fixtures that were not queried in a particular step are already stored in the contact manager from previous steps. By extension, contacts are removed when their corresponding enlarged AABBs stop colliding.

Contacts are not only stored in the global contact list but also in individual lists in each body. Each body maintains a linked list with contacts where one of the two fixtures belongs to that body. These lists are used to detect which contacts persisted from a previous step and later to build the body-contact graph which is used by in the solver.

Most of the contact manager's operations are implemented in two functions:

The first function is responsible for finding new contacts. This is where contact creation and persistence is implemented. It uses the broad-phase for collision detection and processes the incoming collision pairs. When a fixture collision is reported by the broad-phase, it is first checked if a contact with these fixtures already exists. This is implemented by searching if the contact list of either body contains a contact with these fixtures in it. If such contact exists, then we're done processing this pair. Otherwise, a new contact is created, initialized and appended to the global and per-body contact lists. From a performance perspective, searching for existing contacts requires few iterations on average because the per-body contact lists are usually small. After this procedure completes, the contact manager is up-to-date with the new contacts and broad-phase collision detection has finished.

The second function performs narrow-phase collision detection. At this stage, the global contact list is iterated and contacts eligible for narrow-phase collision detection are found. Eligible contacts are those whose fixtures are active, not sleeping[2] and have their tight-fit AABBs collide among other things. Finally, the appropriate narrow-phase collision algorithm is chosen and used to compute the exact collision information for this contact. While the contacts are iterated, those that no longer have their enlarged bounding volumes colliding are destroyed.

4.2 Optimizing the contact manager

The contact manager in Box2D is overly efficiently implemented. Nonetheless, since this stage accounts for an important fraction of each simulation's execution time, we will further optimize it.

4.2.1 Implicit improvements

Because the contact manager is coupled with the broad-phase, its performance is also directly affected by the broad-phase. In our library, we completely changed the broad-phase implementation and this resulted in some immediate changes in the contact manager.

The number of broad-phase pairs is considerably reduced due to not using enlarged AABBs. As a consequence, the number of contacts stored in the contact manager is equally reduced. Since the contact manager has to iterate through all the contacts in each step, this reduction translates to decreased contact processing. In a similar fashion, because the contact class takes up to 200 bytes, the memory required by the contact manager is much less.

At the same time, because the number of reported collisions is lower, the size of the per-body lists is on average smaller. This makes it faster to check if a contact is already present in the contact manager.

4.2.2 Contact flagging for persistence

One crucial difference in our engine is that the contact manager receives *all* the contacts in each step and not just the new ones. Although this results in increased load for the contact creation phase, we can use it to our advantage. Since the broad-phase will find all the current colliding pairs, it also implicitly detects which contacts stopped colliding. In Box2D, the contact manager has to test all the AABBs of the pairs for collision again in each step (even if they were just reported to be colliding by the broad-phase) which is a non-trivial amount of work over the course of the simulation. Moreover, because the enlarged AABBs are used, some contacts will reach the narrow-phase even though they could have been pruned based on their tight-fit AABBs. We implemented a contact-marking strategy that solves both issues:

1. Mark all contacts of the previous step as not-persistent.
2. When an incoming collision pair is matched to an existing contact, mark that contact as persistent.
3. After the broad-phase, all contacts with the persistent flag not set are to be deleted; the rest are sure to be colliding.

[2]In Box2D, bodies that stay idle for multiple steps are put in a sleeping state. When asleep, the required operations to process a body are greatly reduced.

The contact class already has a field named *flags*, that stores boolean flags as single bits. This means we can add a new flag for persistence without requiring any additional memory.

4.2.3 Contact memory reduction

Contacts are the main primitive used in the physics pipeline after the broad-phase. In turn, the performance of subsequent stages depends on the volume of contact data that needs to be processed. We made changes to reduce the amount of memory needed by the contact struct. Reducing memory footprint is a sometimes overlooked optimization opportunity that results in better performance, reduced memory consumption and cache pressure.

As we previously described, each contact must be stored in the linked list of its two bodies. In order to add the same contact in two separate lists, a wrapper structure is needed. This structure holds a pointer to the contact and two pointers that implement the doubly linked list in the body. Box2D also stores a pointer to the other body participating in this contact. We need to add each contact to two body contact lists, so each contact contains two such wrapper structs. A total of eight pointers per contact are required, which translates to 64 bytes in a typical 64-bit computer.

We implemented contact lists in bodies as arrays instead of linked lists, where contact pointers are stored directly. This eliminates the need for the wrapper structure and its memory overhead, while also simplifying contact storage. The pointer to the other body is no longer present; we replaced it with a simple conditional check in the places it was needed. Body contacts lists are iterated many times to persist contacts and build the contact graph. Iterating over an array of continuous memory is easier to predict by the caching mechanisms of the CPU compared to a linked list.

This change results in some complications regarding the removal of contacts from the body lists. For a contact to be removed it must first be located, which requires performing a linear search in the contact array. Although the contact arrays on average have very few elements, we wanted to avoid searching the contact's position for each removal. This was achieved by implementing a more efficient and complex approach, where contact removal happens in a batch post-processing pass. This pass is executed after the broad-phase and efficiently removes all the dead contacts at once. It does so by iterating the contact lists and replacing every dead contact. In the case where there are multiple dead contacts, this method performs much less work that iterating the array once for each contact destroyed. The individual linear scan is only used when it is known that few contacts have to be removed and the batch algorithm is costly in comparison. This happens, for example, when a body is removed from a world and its contacts must be destroyed.

The contact class also has two integer fields needed to implement continuous physics. We replaced those two fields with one because we are using an on-demand data structure when continuous physics are enabled. Additional details can be found in section 6.

At the time of writing, the above changes reduced the size of the contact structure from 208 bytes (Box2D) to 128 bytes (our library).

4.2.4 Merging contact classes

In Box2D, there is a large number of classes to handle contacts. Specifically, each supported combination of collision between two shapes has its own contact class. For example, the contact class that

models a collision between a circle and an edge shape is found in the files ‘b2_chain_polygon_contact.cpp‘ and ‘b2_chain_polygon_contact.h‘ accordingly.

This is design has drawbacks:

- Adding a new shape is error prone and tedious. It requires creating an increasing number of classes and files. Adding a new shape in Box2D currently requires 5 new classes.
- There is excessive code duplication and potential for subtle bugs when creating or altering those contact classes.
- The size of the executable is increased unnecessarily.
- Some compiler optimizations are possibly hindered by the complexity of this approach.

The primary difference between the contact classes is the implementation of a different method for narrow-phase collision detection depending on the shapes participating in the contact. Other differences include the types used in the code and the allocation functions.

We merged all the classes into a single contact class by adding a new function pointer to the existing base contact class. Instead of overriding a function in each class to provide the correct collision algorithm, we assign a function pointer that points to the correct implementation for the specific collision type. The function pointer for each combination is precomputed and stored in a lookup table for fast access during contact creation. With all the contacts implemented as a single class, contact allocation and deallocation is also faster and simplified.

We added a new pointer in the contact class, but it actually does not increase the required memory. In most cases, when a class has virtual functions a hidden pointer is introduced by the compiler that points to the class's *virtual table*. The new contact class has no virtual functions and hence the combined memory will be the same as before.

4.2.5 Cyclical contact lists

Inserting and deleting contacts from the global contact list happens very frequently. The double linked list structure used requires a number of conditional checks for each operation. The corresponding source code for removing a contact from the list is as follows:

Listing 12: Contact removal in Box2D

```
if (c->m_prev != NULL) {
    c->m_prev->m_next = c->m_next;
}

if (c->m_next != NULL) {
    c->m_next->m_prev = c->m_prev;
}

if (c == m_contactList) {
    m_contactList = c->m_next;
}
```

All these conditional checks are necessary to check for null elements and the special case of removing the root element. In our library we made the contact list *cyclical* and eliminated all the branches from the insertion and deletion procedures.

In order to remove the check for when the contact list is empty, we used an auxiliary dummy contact. This contact is never removed from the list, so the list it can never be empty. Initially, this node is linked with itself and creates a loop. This signifies an empty contact list. To insert a contact in the list, it is linked directly after the auxiliary root node. Removal is implemented like a normal linked list. In both cases, the conditional tests are not needed because there are never links to null nodes and the dummy node is never removed.

With the new approach, the equivalent code for removing contacts is branch-free and minimal:

Listing 13: Faster contact removal with cyclical lists

```
c->m_prev->m_next = c->m_next;
c->m_next->m_prev = c->m_prev;
```

The code for inserting new contacts is simplified in a similar way. The implementation and modifications needed for this change are minimal, but they alter how the contact list has to be iterated. To iterate through all the contacts, we start with the element after the auxiliary root and loop until we encounter the auxiliary root again. We introduced new function calls that make it easy to modify existing code that needs to iterate through the contacts. We also measured the effectiveness of this particular change with the benchmark simulation programs and concluded that it improves performance.

4.3 Performance evaluation

During the development and testing of the optimizations for the contact manager, we needed a way to reliably measure the effectiveness of our changes and compare them to Box2D.

Measuring whether a particular change improved performance in our library alone was easy. We evaluated changes by measuring the total time spent in the contact manager with the benchmark suite found in section 8. On the other hand, comparing the performance to Box2D proved to be challenging for two reasons:

1. The combined time spent in the contact manager per step is not a fair measurement. Due to the changes we implemented in the broad-phase, the contact manager has to process much fewer collision pairs in our library compared to Box2D.

2. The time spent per contact is not a meaningful comparison either because it is also affected by the difference in the broad-phase output. In Box2D there are many contacts to manage but most of them are "easy" due to enlargement; most do not collide and narrow-phase collision detection terminates quickly. In our library the contacts are fewer but most of them do collide and require more operations to complete the pipeline.

The combined time spent in the contact manager is less in our library but sometimes Box2D has lower per-contact processing time. This is nonetheless a satisfactory result, as the total simulation time is lower with our engine. Moreover, it demonstrates how well the new broad-phase benefits the rest of the physics pipeline.

5 Constraint Solver

5.1 Box2D's solver

When the contact manager has processed the collision pairs from the broad-phase and performed narrow-phase collision detection, all the necessary information for collision resolution is present. The *constraint solver* (or just solver) uses this information and modifies the position and velocity of bodies to solve the position and velocity constraints of the simulation. These constraints prevent body penetration and apply friction. The solver algorithm does not find a global solution, which would be computationally infeasible to perform real-time, but instead uses an iterative solver to approximate that solution, as described in Erin Catto's paper [9]. Due to its iterative nature, it is possible to tune the relation of accuracy and performance of the simulation by increasing or decreasing the iterations of the solver.

The solver requires only linear time and space $O(n)$, where n is the number of contacts. The specific implementation found in Box2D is carefully designed and very efficient. It can handle tens of thousands of bodies in real-time. The solver's performance is mostly limited by memory latency and cache misses. Because of it's efficiency, we kept the solver almost intact in our library. We only performed two minor optimizations that results in additional efficiency benefits in specific configurations.

5.2 Minor refinements

5.2.1 Fast path for bodies without contacts

The bodies, joints and contacts of a simulation form a graph of constraints. The bodies are the graph's nodes and the constraints (contacts and joints) are the edges. In most cases, the graph is disconnected and consists of many smaller isolated subgraphs that can be processed individually without affecting each other. In Box2D, those isolated subgraphs are called *islands*. In order to maximize performance, the library invokes the iterative solver on individual islands instead of the whole graph. At the beggining of the constraint solving stage, the body graph is traversed with a Depth First Search (DFS) to build up islands. Every time an island is built, the solver is invoked on it.

When an island is built and before the solver is invoked, a number of operations are performed. This is where:

- Position and velocity of bodies is integrated.
- Damping and external forces are applied.
- Auto-sleeping is implemented.
- The solver is initialized and allocates various structures needed for its operation.

At this point some of the body data, specifically position, rotation, linear and angular velocity, is copied into seperate continuous arrays. These elements are accessed very frequently inside the solver and this dense packing greatly reduces cache misses and increases throughput.

There is a single case where most of these operations in islands are superfluous. This is when a body has no contacts nor joints, what we'll call an *isolated* body. There are many scenarios where isolated bodies exist: bodies in free fall, kinematic bodies scattered through a world, a low gravity scenes where bodies float in space or even momentarily between steps. In this case, the generic island code performs a high number of unecessary operations and allocations. We deemed useful to optimize the handling of isolated bodies by creating a new specialized implementaion in the island class. When such a body is found during island building, all other operatinos are bypassed and the efficient specialized function is called. This change improved the total execution time for a wide selection of scenes.

5.2.2 Static body processing

Throughout the library, there are some code fragments that need only execute for bodies that are not static. Since static bodies are interleaved with other types of bodies in the body list, all the bodies are iterated and the static ones are skipped. To reduce iterations in these cases, we decided to keep the body list sorted into two groups. The first group, at the beggining of the list, stores only non-static bodies. The second group, which is the rest of the list, stores only static bodies. In this way, loops that need only process non-static bodies can terminate when they encounter the first static body. The bodies are connected in a doubly linked list so it is easy to maintain the groups sorted. We insert static bodies in the list's tail and all other bodies in the head.

6 Continuous Collision Detection

6.1 The problem

Both Box2D and our physics engine use discrete time steps and hence the movement of objects is also discrete. At each step, the new position of each object is calculated and afterwards the object is 'teleported' there. As long as the change in position relative to the previous step is small, there are no major drawbacks with this approach; Box2D is designed to allow and gracefully handle object penetration. But when the change in position becomes large enough, a problem can arise. If an object has high enough velocity, it can teleport through another object without creating any impact or collision. This unwanted artifact is called *tunneling*.

Even if tunneling does not occur completely and the collision is detected, a body can move fast enough and penetrate another object more than the solver can handle effectively. It can then take several steps before the solver manages to bring the object to its correct position. It is even possible for the object to be moved in an invalid position, if the solver does not have enough information for the collision. The tunneling effect can lead to incorrect and unstable simulations in many ways.

In general, depending on the simulated scene and the needs of the application, tunneling may be of varying importance. If a group of dynamic objects collides with another and some collisions are missed, it may be hard to notice. On the other side, if a single object passes through a static wall it is much easier to notice. Apart from being a source of visual artifacts, tunneling can lead an application to invalid configurations that are hard to resolve. In a platformer game, for example, a character falling fast enough can pass through the ground, from which there are no meaningful ways to recover the simulation.

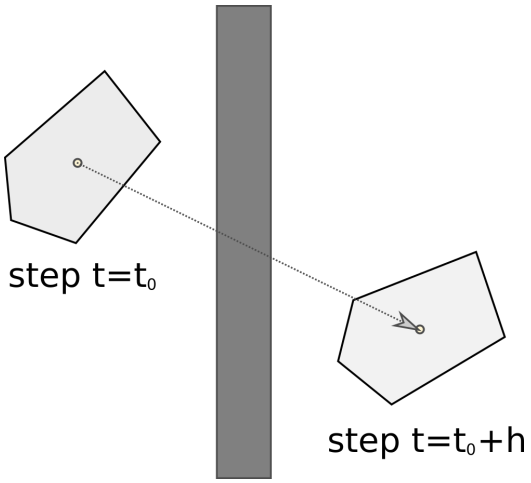


Figure 11: A scenario of object tunneling

A number of tricks can be used to prevent, or at least limit, the possibility of such event happening. The application can:

- Limit the maximum velocity of objects.
- Increase the thickness of walls and objects that must not be penetrated.
- Decrease the time delta that is used to advance the simulation.
- Use speculative contacts that slow-down objects when they approach other objects that must not be penetrated.

If the above techniques are correctly combined, it may be possible to combat the tunneling effect to some extent. While this can be enough for certain applications, this in not a solution to the general problem. Those techniques have a limited scope and it is not always possible to apply them:

- Limiting the maximum velocity or increasing object size may not be a choice if the application requirements cannot be changed accordingly.
- Decreasing the time delta is effective but requires running multiple steps per frame which is usually prohibitively expensive.
- Speculative contacts help prevent tunneling but can lead to artificial slowdown and unnatural behavior of objects.

The problem remains that these techniques cannot *reliably* prevent tunneling. None of these techniques can prevent penetration from happening at all.

6.2 CCD in Box2D using Time of Impact

Fortunately, there exist robust methods to solve the problem of tunneling and Box2D has such functionality implemented. Box2D supports *Continuous Collision Detection (CCD)*, which is a collection of methods that make simulations appear to be continuous. In Box2D, using CCD prevents both the tunneling effect and object penetration from happening.

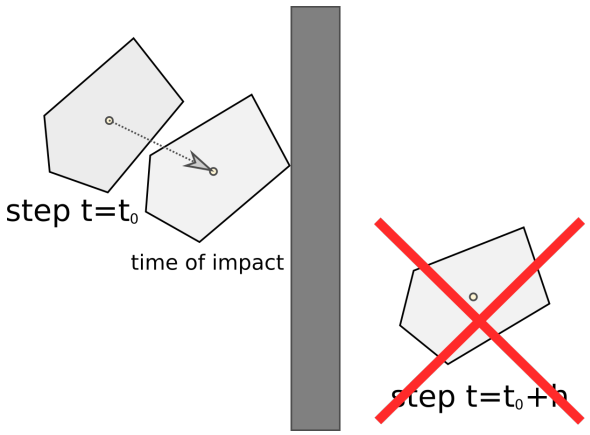


Figure 12: Tunneling prevented by computing the Time of Impact

In most cases, continuous physics are achieved with a combination of interpolation, raycasting and root finding. Specifically in Box2D, continuous physics are supported by computing the *Time of Impact (TOI)* between two colliding bodies. The time of impact is some time between two discrete time steps were the collision of two objects happens. The library already provides an implementation of an algorithm to compute the time of impact based on *Conservative Advancement* [18] and which we will also use. To prevent tunneling, it also uses *swept AABBs* in the broad-phase. For a particular object, its swept AABB is the union of its AABBs from the previous and current step. Although this increases the number of false-positive collision pairs and the work that needs to be done in the broad-phase, it is necessary for detecting when an object teleports through another.

Continuous Collision Detection is a powerful tool that can solve the tunneling problem, but it must be used with caution. The algorithms used for continuous physics are very CPU-intensive and using them excessively can noticeably slow down the simulation, even with a few hundred of objects. By default, when CCD is enabled in Box2D, the library only checks for dynamic versus static body collisions to apply the TOI algorithm in order to decrease the runtime cost. Dynamic versus dynamic TOI tests are enabled only for certain bodies, called *bullet bodies*, that are designated by the user.

Below is pseudocode for the CCD procedure, as found in Box2D.

Listing 14: Continuous Physics in Box2D

```
Algorithm Solve-TOI

Input: Contacts the set of all collision pairs from swept AABBs (from the broad-phase)

// TOI(c) is the time between the steps were the collision of
// the two bodies happens. It is a value in the interval [0, 1]

loop forever
  minContact = null
  minTOI = 1

  for C in Contacts do
      if TOI is not cached for C then
          calculate TOI with Conservative Advancement
      end

      if TOI(C) < minTOI then
          minTOI = TOI(C)
          minContact = C
      end
  end

  if minTOI > 1.0 - epsilon then
      // The minimum TOI is close to 1; we can stop here
      end loop
  end

  advance minContact to its TOI

  build a body-contact graph starting from minContact and
```

```
    use the solver to correct the positions of the bodies in the graph

    for each body in the graph do
        broadphase.Update(body)
    end

    for each contact in the graph do
        invalidate the cached TOI of contact
    end

    // Find possible new contacts from the above changes
    Find-New-Contacts()
  end
```

It is noted that the above algorithm does not preserve time, since the time of impact happens between two time steps. In addition, the implementation found in the library has some limitations on how many collisions and how complex scenarios it can handle. These relaxations exist because otherwise the computational cost would be prohibitive to use in real-time applications.

6.3 Rebuilding the CCD pipeline for performance

Some parts of the current CCD implementation in Box2D are implemented naively and result in much increased running time when continuous physics are enabled. Only worlds with few bodies and collisions can be simulated in real time, which is a limiting factor for applications. In our library, we redesigned some parts of the CCD pipeline and used accelerating structures which greatly improved performance.

The main loop of the algorithm presented in Listing 14 finds the contact with the minimum TOI in each iteration. We need to process the event with the minimum TOI each time in order to preserve the order of the collisions. The current implementation in Box2D finds this contact by iterating over the entire contact list. Because most contacts remain unchanged after resolving an event, the TOI computed for a contact is cached and used until it is invalidated by a change. In general, the Conservative Advancement algorithm is expensive and must be called as few times as possible.

Although Box2D caches the computed TOI for the contacts, it is still quite inefficient. At each loop it iterates linearly through all the contacts and the number of contacts can be high. The number of loops can also be high because at each iteration resolves few contacts. This algorithm may do up to $O(n^2)$ operations, where n is the number of contacts, which is not acceptable for a real-time simulation.

In our library, we first replaced the inner loop that scans all the contacts to find the minimum TOI. Throughout the algorithm, the two operations required are *find minimum TOI* and *update TOI of contact*. These operations can both be efficiently implemented with a min heap data structure.

We implemented a binary heap, tailored to the specific needs of our use case. In order to support fast access to the heap for the *update key* operations, we introduced a new field in the contact class that stores the index of the contact in the min heap. The heap operations are implemented to modify

this field accordingly, so it is always valid. Listing 15 shows the modified algorithm that uses the heap to find the minimum TOI contact.

Listing 15: Improved implementation for Continuous Physics

```
Algorithm Solve-TOI2

Input: Contacts the set of all collision pairs from swept AABBs (from the broad-phase)

Heap = empty min heap for contacts, using TOI as the key

for C in Contacts do
  Store TOI(C) in C
  Append C in Heap
end

Build-Min-Heap(Heap)

loop forever
  minContact = Heap.ExtractMin()

  if TOI(minContact) > 1.0 - epsilon then
      // The minimum TOI is close to 1; we can stop here
      end loop
  end

  advance minContact to its TOI

  build a body-contact graph starting from minContact and
  use the solver to correct the positions of the bodies in the graph

  for each body in the graph do
      broadphase.Update(body)
  end

  for each contact in the graph do
      Compute the new TOI(C) for this contact
      Heap.UpdateKey(C)
  end

  // Find possible new contacts from the above changes
  Find-New-Contacts()
end
```

The introduction of the min heap drastically speeds up the first phase of the CCD algorithm. Yet, because of the changes we made to the broad-phase and the contact manager, there are some new regressions in the last part of the algorithm.

After performing one iteration of the algorithm, some fixtures are moved and as a result the broad-phase must be updated. There are also potential new contacts that must be discovered and used in subsequent iterations.

In Box2D this is implemented in a straightforward manner while also being efficient. The broad-phase is updated by calling the corresponding update function for each body and at the end there is a contact manager call to find new contacts. The few update calls will be performed locally and leave the rest of the tree intact. Because the broad-phase keeps track and performs collision detection only on fixtures that moved, finding new contacts will require few queries in the tree.

In our library, using the broad-phase in this way results in poor performance. The fixture updates will require no work to be performed, but they will invalidate the broad-phase tree in turn. As a result, when the contact manager will run the broad-phase tree will be built from scratch and collision detection will be performed on all bodies. Needless to say, it is infeasible to perform global collision detection in each iteration of the algorithm. We have to modify this last part to achieve the desired performance speedup.

We need an algorithm for the broad-phase that updates individual fixtures without invalidating the whole BVH tree. It would be possible to implement an algorithm for individual updates similar to the one in Box2D, but we followed an alternative approach that delivers better performance. We move the fixture's AABB and update the union AABBs of its parent nodes. This is done without modifying the structure of the tree. The result is a valid BVH tree, but with slightly worse quality than before. We trade tree quality for a fast update operation that does not invalidate the tree. Because the change of body positions during the CCD algorithm are small, the decrease in quality is also small. The tree will be rebuilt later and so the decreased quality is only temporary. After measuring the execution time and quality decrease, we concluded this is an efficient way to implement updates for the CCD algorithm.

Now that the broad-phase update is efficient and doesn't invalidate the tree, we must also find the new contacts efficiently. Because in each iteration a body-contact island is built for the minimum TOI contact, we know which bodies were possibly moved. We perform individual broad-phase queries with the fixtures of those bodies. Before the queries, the persistence flag of the island's contacts are also cleared. This is needed to find possible new or dead contacts. If new contacts were created, we calculate their TOI and insert them in the heap. If some contacts were destroyed, we delay their removal until the end of the CCD algorithm because removing them at every iteration is expensive. In order to not use destroyed contacts in the next iterations, we perform a liveness check on contacts before using them.

6.4 Evaluating performance

We have redesigned the Continuous Collision Detection phase and as a result it is much faster, while maintaining the same functionality as in Box2D. Below, there are plots comparing the running time of Box2D and our library at increasing world sizes, with CCD enabled and disabled respectively. More information regarding the structure of these scenes and their interpratation can be found in subsection 8.1.

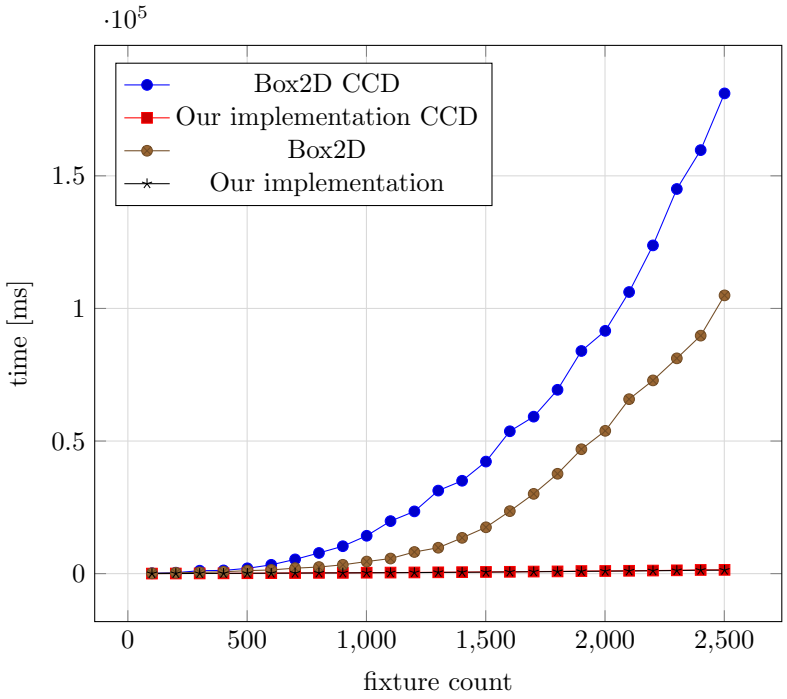


Figure 13: CCD benchmark "Add pair"

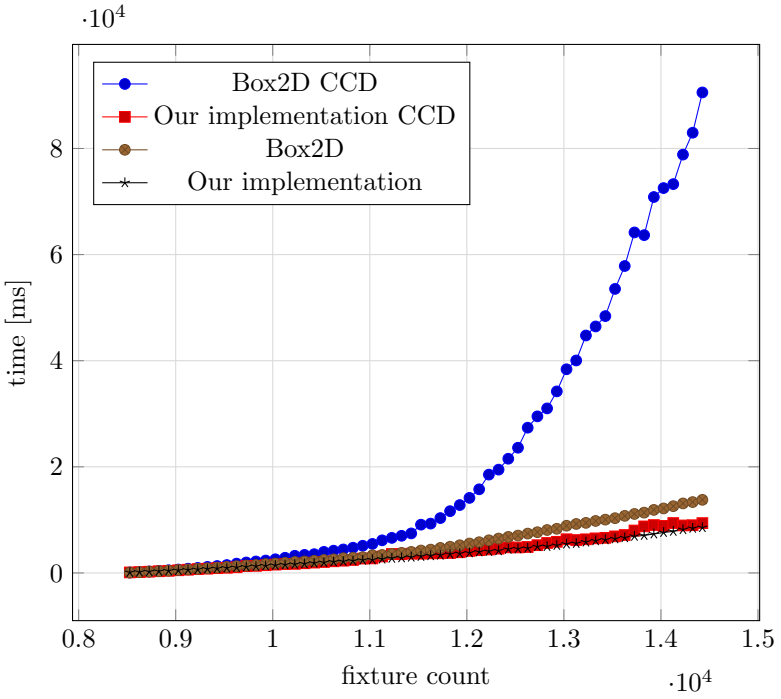


Figure 14: CCD benchmark "Mixed static/dynamic"

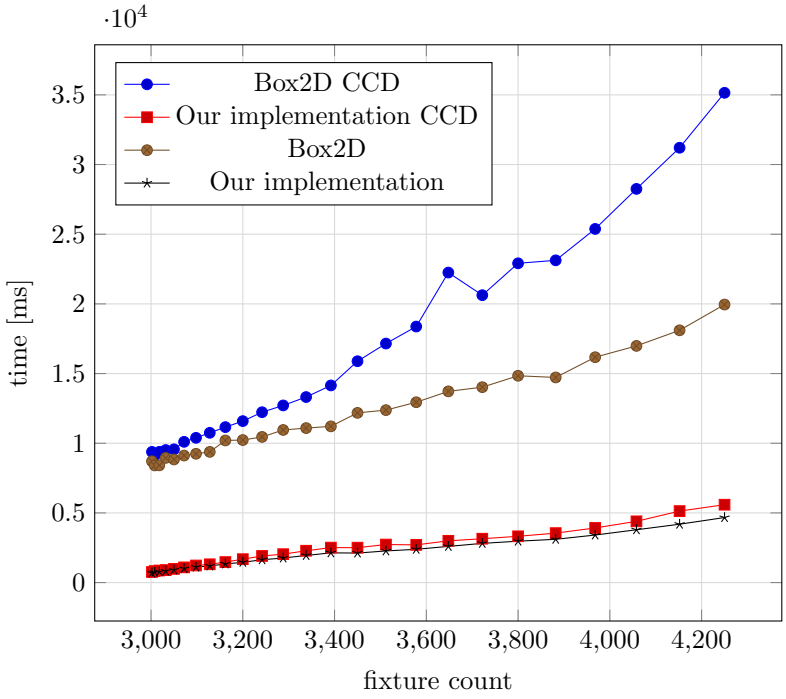


Figure 15: CCD benchmark "Diagonal"

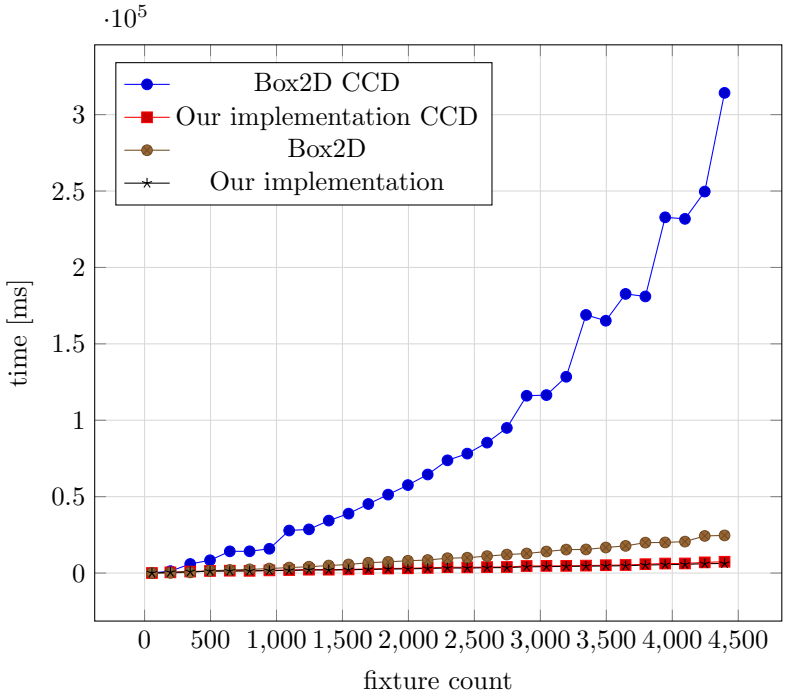


Figure 16: CCD benchmark "Falling squares"

It is evident that enabling CCD in Box2D results in a major performance penalty. The running time is much higher and scales worse, compared to both Box2D and our library with CCD enabled. In constrast, enabling CCD in our library only slightly decreases performance; scaling to thousands of bodies is comparably efficient. In most benchmarks, our library with CCD enabled still performs measurably better than Box2D without CCD.

We conclude that the changes we introduced in the Continuous Physics stage are extremely effective and introduce little overhead compared to the rest of the pipeline.

7 Additional improvements

7.1 Reducing shape complexity

A single body can have multiple fixtures, enabling the creation of complex objects like non-convex polygons or shapes with holes. In Box2D, a single fixture's shape can also have multiple parts. Throughout the library, this functionality is used to implement the chain shape, a type of polygonal path.

Enabling shapes to declare multiple parts is convenient for implementing the chain shape but has drawbacks. Handling shapes requires using a loop to go through all their potential parts, even though most of them have only one. Shape operations, like raycasting and AABB creation, require indices to define which part to use. When the contact manager persists contacts, it has to check both the fixtures and the shape indices that create a collision pair. All these combined, lead to increased code complexity and execution overhead.

The weakness of this scheme is that in most cases this added complexity is unnecessary. In our engine, we successfully removed the ability of shapes to comprise of multiple parts without sacrificing any of the functionality the library provides. This was achieved by implementing the chain shape at the fixture level.

Given that the rest of the shapes don't have multiple parts, we only need to change the implementation of the chain shape. Inspecting its source code reveals that is uses a list of points that form continuous edge shapes. The individual edge shapes are not stored in memory but instead created dynamically when needed. Our approach was to implement chains with multiple edge fixtures. We created new helper functions that create the necessary edge shapes to form an equivalent chain shape. By using them, it is easy to port existing applications.

We identify multiple advantages to this approach:

- Library and user code are simplified. There are fewer shapes and collision types to handle. There is no need to use loops every time a shape operation has to be performed.
- The simplifications, most importantly the removal of loops and handling of indices, lead to additional performance gains.
- The fixture-based implementation offers greater flexibility. Since the individual edge shapes are exposed, the user can modify their friction or restitution which was not possible before.
- User-facing functions are easier to use due to the lack of shape indices.

The only potential drawback of this change is the increased memory consumption of the new implementation. Using one fixture per edge requires more memory for the additional information stored in each fixture. In most cases, this should not be a problem because shapes and fixtures occupy an insignificant amount of memory.

7.2 Compile-time configuration of features

Box2D is a general-purpose physics library and provides a lot of useful features by default. In some cases, these features lead to increased memory consumption and execution time, even if they are not used in the application. For this reason, we implemented conditional compilation, where the user can choose to disable some features at compile-time. We implemented this change entirely in the C++ preprocessor. Without making the codebase much more complex, there are multiple benefits:

- *Reduced memory consumption.* By disabling a feature, the class fields that implement it are removed. By carefully disabling unneeded features, the combined memory savings can be very significant.
- *Improved efficiency.* Depending on which features are disabled, the corresponding code of the library that implements them is disabled. In specific scenarios, this enables more compiler optimizations. Moreover, reduced memory consumption translates to better cache utilization.
- *Fewer possible bugs.* If a feature is not to be used in an application, completely disabling it can help prevent accidental errors. This can be valuable in applications where multiple people collaborate.

8 Benchmarks and Results

In this chapter, we examine the performance of our library compared to Box2D. The source code for our library includes all the improvements and changes we presented in this document. For Box2D, we used the latest available version at the time of writing, Box2D 2.4.1 (commit 95f74a4).

8.1 The benchmark suite

Alongside our physics engine, we developed a program with a set of benchmark scenes. This program helped us identify performance issues in the original library and evaluate the changes we made in our implementation. We deliberately built scenes that need time to simulate (usually tens of seconds) in order to measure execution time with accuracy. Accurate timing is essential to distinguish changes that result in consistent improvements from micro-optimizations and regressions.

In all cases, the benchmark programs were executed in a consistent environment (adequate cooling, power supply) in the same machine without any other applications running. Both libraries were compiled with the same compiler and optimization flags. All benchmarks were run on Ubuntu 20.04 with an AMD Ryzen 3600 CPU running at 3.6GHz and 16GB 2993Mhz DDR4 RAM. The benchmarks were reproduced on Intel processors and other hardware configurations with similar results.

The benchmarks are run with the default settings that the Testbed application has, except that Continuous Collision Detection (CCD) is turned off. Benchmarks with CCD enabled can be found in subsection 6.4;

The benchmark scenes have a parameter that defines the number of objects in the simulation. Each benchmark is executed with increasing object count. This makes it possible to compare the performance at different world sizes and examine how the performance scales as the number of fixtures increases.

There is a wide variety of scenes, each with different characteristics. Some of the benchmarks are specifically built to exhibit worst-case performance. These are used to evaluate the expected behavior when simulating degenerate scenes. Others represent simulations that can arise in real-world applications. A snapshot of each benchmark scene can be seen in figures 17 and 18. The list below has a short description for each scene, explaining its purpose in the benchmark suite.

1. *Add pair*: A fast-moving object hits a group of resting circle objects in a zero-gravity world. This scene tests performance when a very high number of collisions and contacts are suddenly created. (Adapted from Box2D's Testbed)

2. *Multi-fixture*: A scene made of bodies with greatly varying fixture count. Several table objects are created, each built with increasingly more and smaller non-overlapping fixtures. Those tables are thrown into a U-shaped static object.

3. *Falling squares*: A scene with stacked squares of unequal size falling against a ground body. It is made to test the performance when fixtures have a great variety in size.

4. *Falling circles*: The same scene as above but with circles instead.

5. *Slow explosion*: A scene with a high number of dynamic bodies moving slowly in a zero-gravity world. Used to test performance in scenes with slow-moving objects and very few collisions.

6. *Tumbler*: In each step of this scene, new polygon shapes are added inside a rotating box. Used to test dynamic body creation. (Adapted from Box2D's Testbed)

7. *Mild N^2*: A degenerate scene. In this benchmark, many composite objects are created (tables and spaceships as found in Box2D's Testbed) and added at the same place, one on top of the other. This creates $O(n^2)$ contact pairs momentarily and is designed to evaluate worst-case performance.

8. *N^2*: A simpler version of the above. Several single-fixture bodies are created in the same position, offset by a small value.

9. *Mostly static (single body)*: Features a huge static square made from a very high number of small static squares. Only a few dynamic bodies exist in a diagonal opening inside the big square. Used to test worlds with very high static to dynamic body ratio. All the small squares are individual fixtures in a single body.

10. *Mostly static (multi body)*: The same scene as above, but all the small squares are individual bodies with a single fixture.

11. *Diagonal*: A benchmark with dynamic bodies falling through a big group of static diagonal lines. The diagonal lines have a very bad AABB approximation. This scene creates very much broad-phase pairs but with few real collisions. Used to evaluate the effectiveness of the broad-phase.

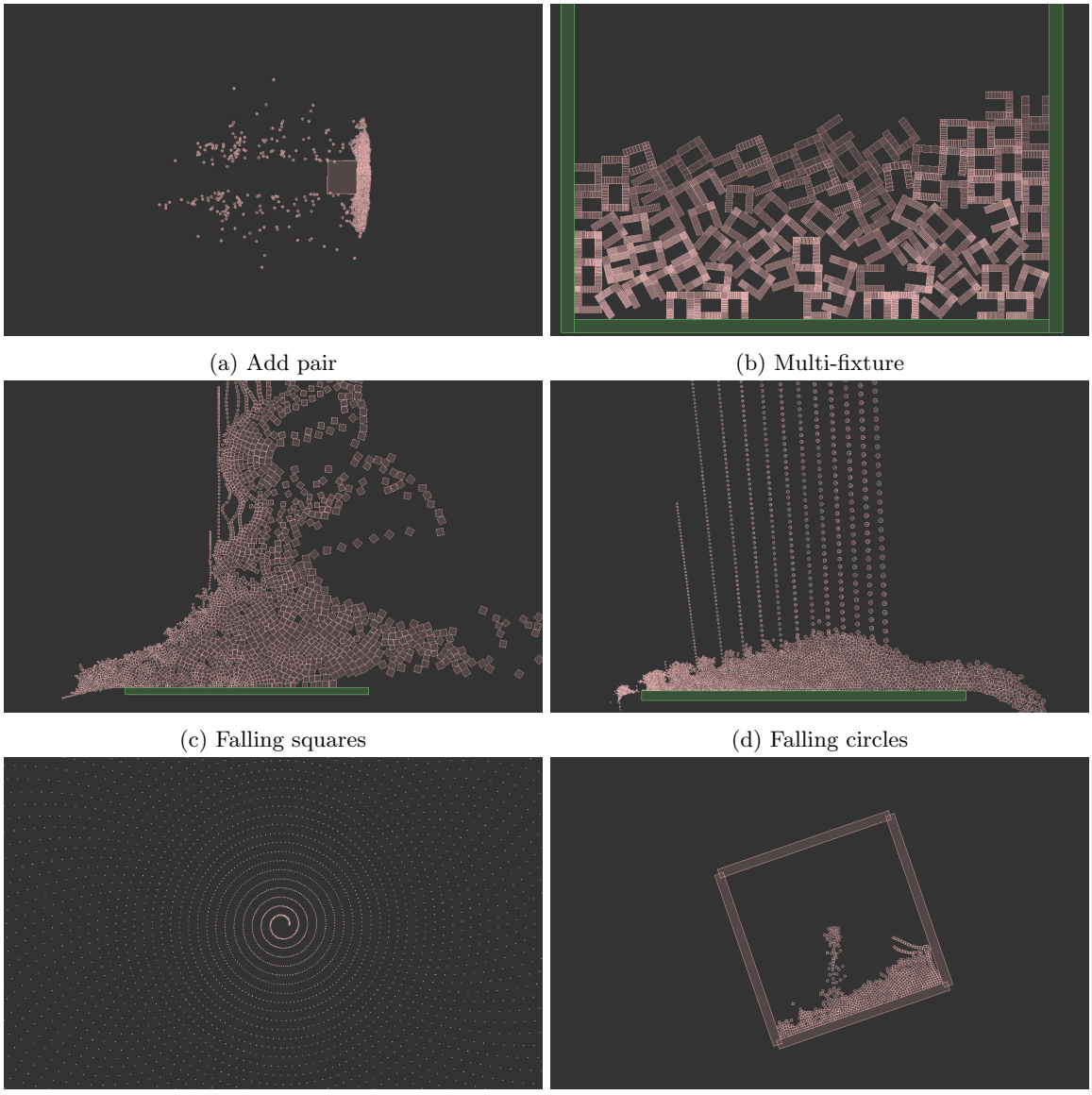
12. *Big mobile*: This scene features a lot of bodies and joints in a very wide structure. (Adapted from Box2D's Testbed)

13. *Mixed static/dynamic*: A casual scene with both static and dynamic bodies, polygons and circles, some movement and collisions.

8.2 Benchmark results

Below are all the comparison plots for the benchmark scenes. For each benchmark, there exists one plot comparing world size (x axis, fixture count) and total execution time (y axis, ms) for N simulation steps. N is an arbitrary number of steps chosen based on each scene's content to correctly measure execution time.

Important note: The total execution time (y axis) for each benchmark **does not** mean something on its own; it is only meaningful for comparison. For this reason, each plot shows the execution time for both libraries, at the same scale. In all plots, a lower time value means better performance.



(a) Add pair

(b) Multi-fixture

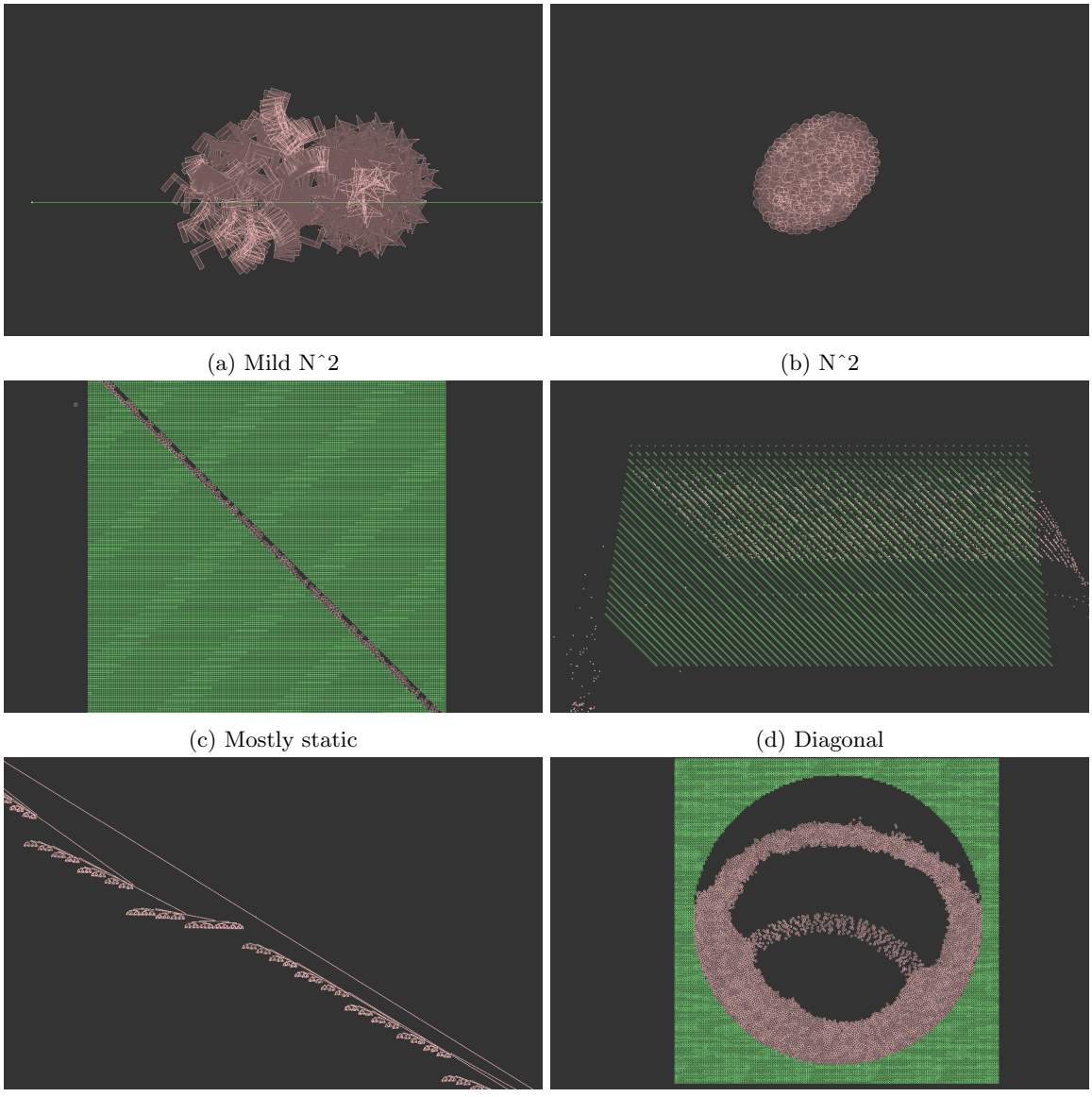
(c) Falling squares

(d) Falling circles

(e) Slow explosion

(f) Tumbler

Figure 17: The benchmark suite (1)



(a) Mild N^2

(b) N^2

(c) Mostly static

(d) Diagonal

(e) Big mobile

(f) Mixed static/dynamic

Figure 18: The benchmark suite (2)

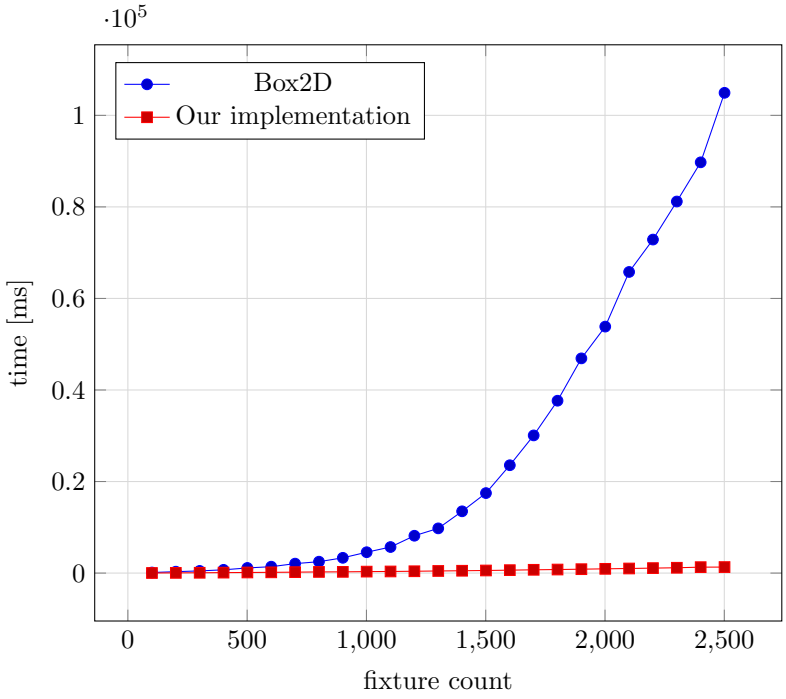


Figure 19: Benchmark "Add pair"

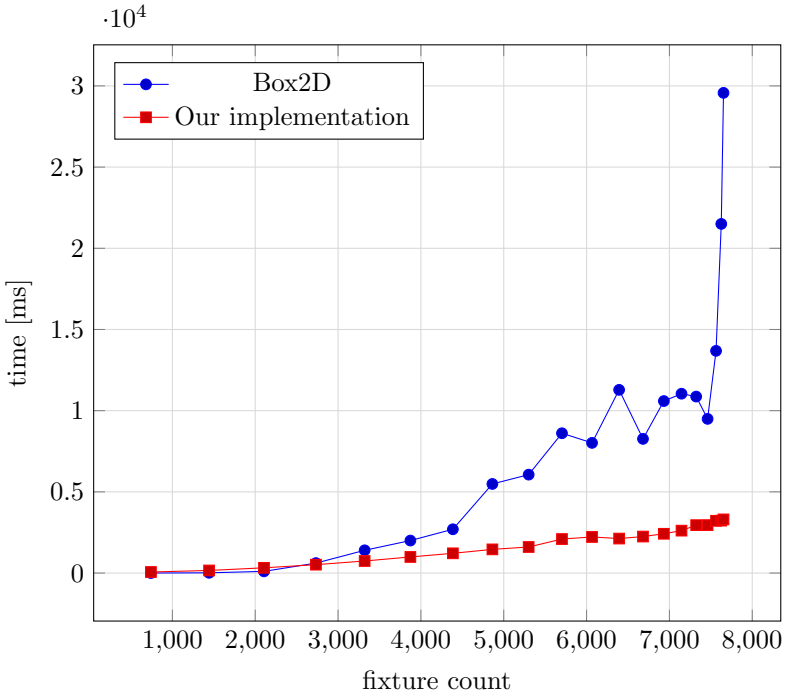


Figure 20: Benchmark "Multi-fixture"

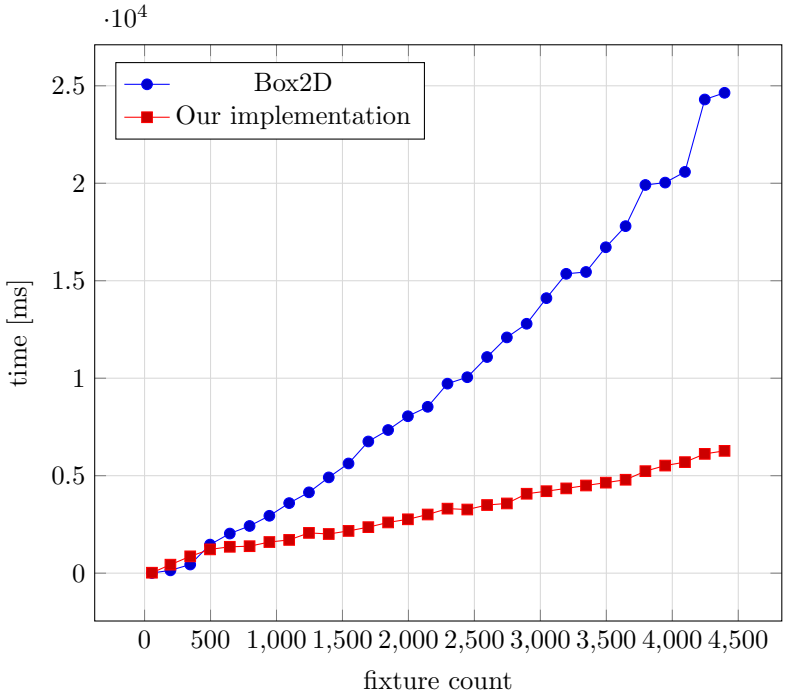


Figure 21: Benchmark "Falling squares"

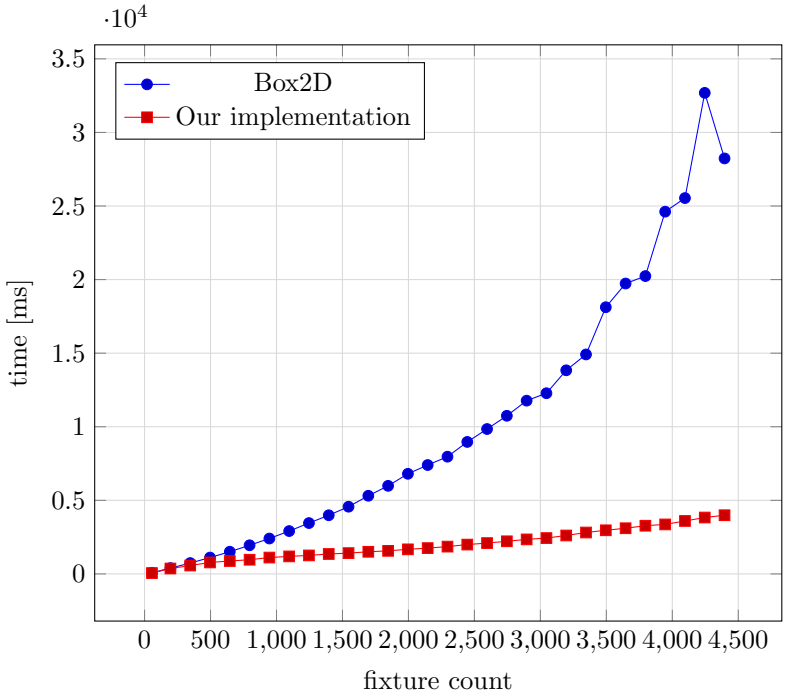


Figure 22: Benchmark "Falling circles"

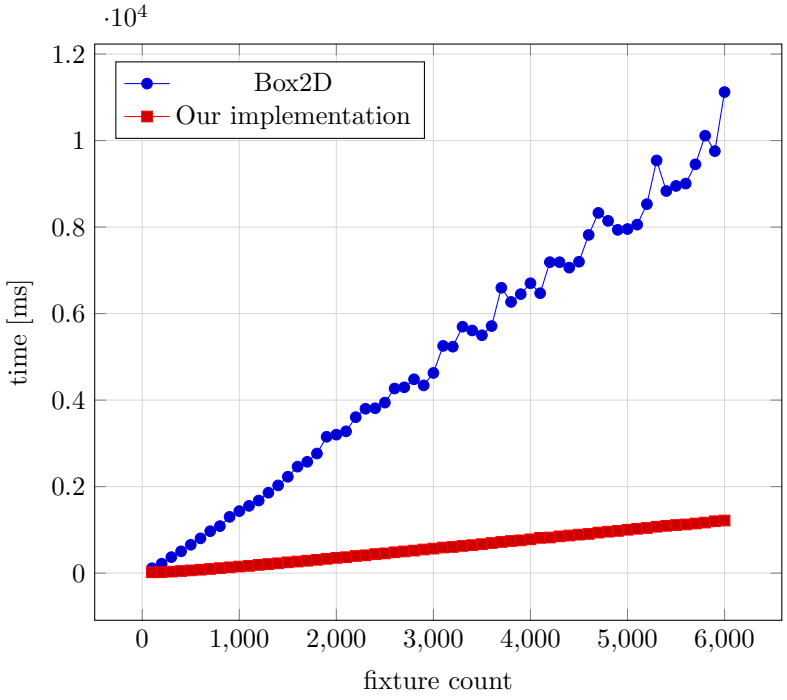


Figure 23: Benchmark "Slow explosion"

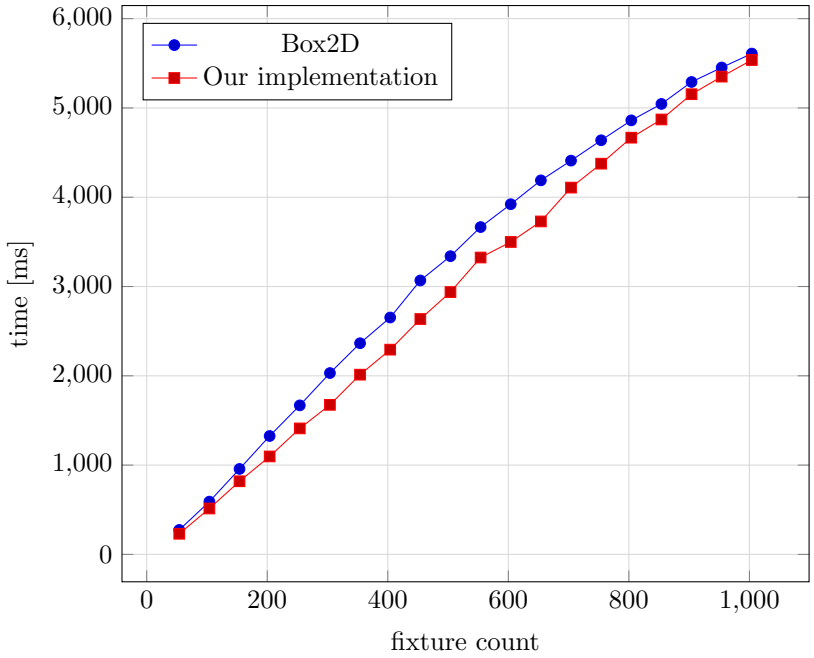


Figure 24: Benchmark "Tumbler"

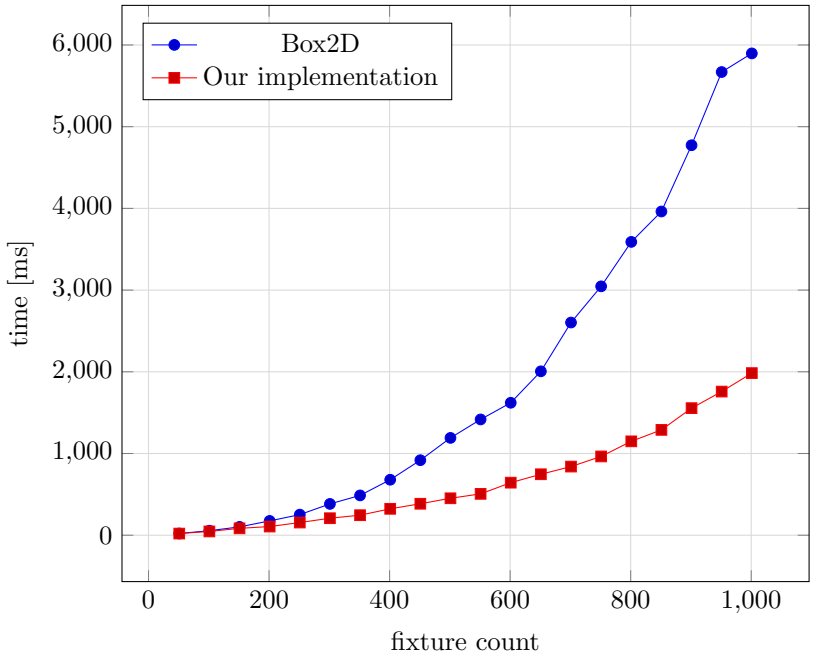


Figure 25: Benchmark "Mild n2"

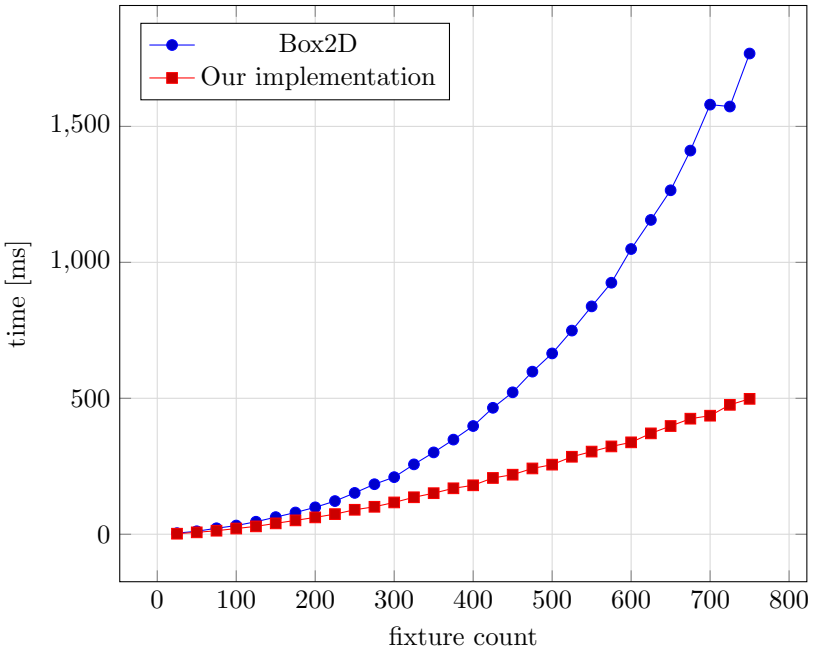


Figure 26: Benchmark "n2"

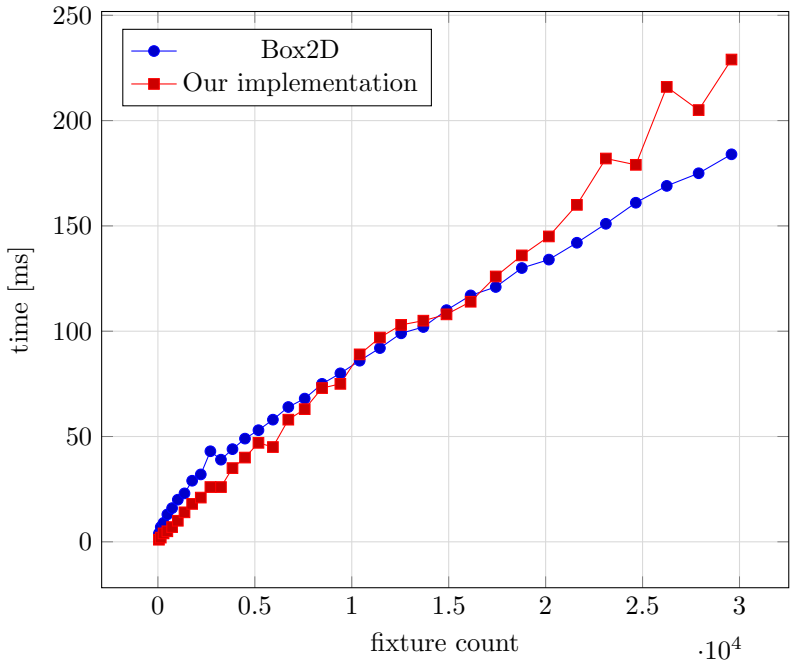


Figure 27: Benchmark "Mostly static (single body)"

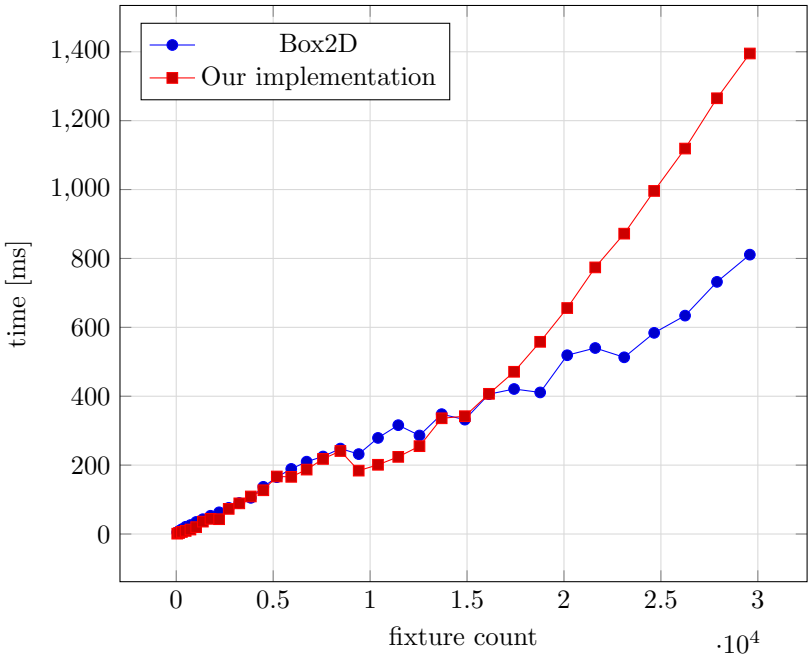


Figure 28: Benchmark "Mostly static (multi body)"

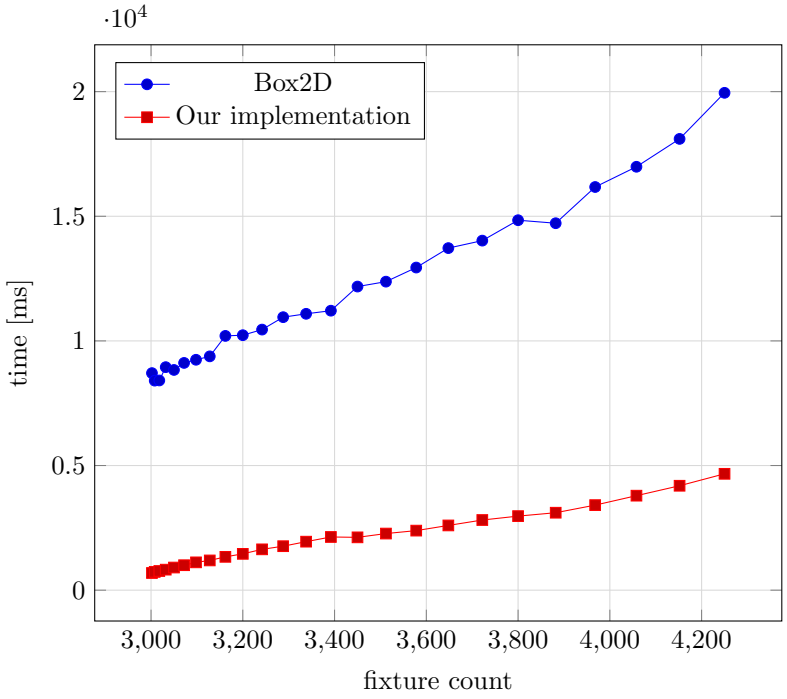


Figure 29: Benchmark "Diagonal"

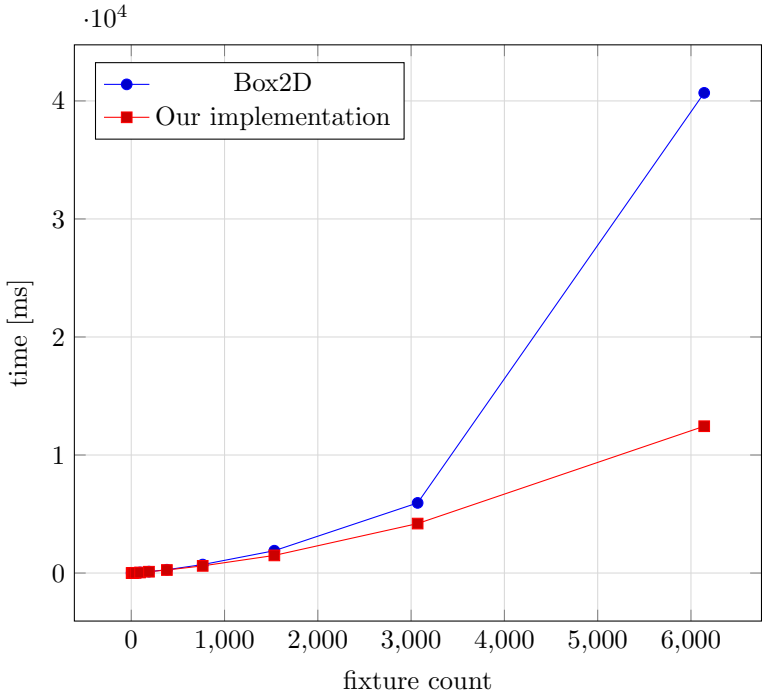


Figure 30: Benchmark "Big mobile"

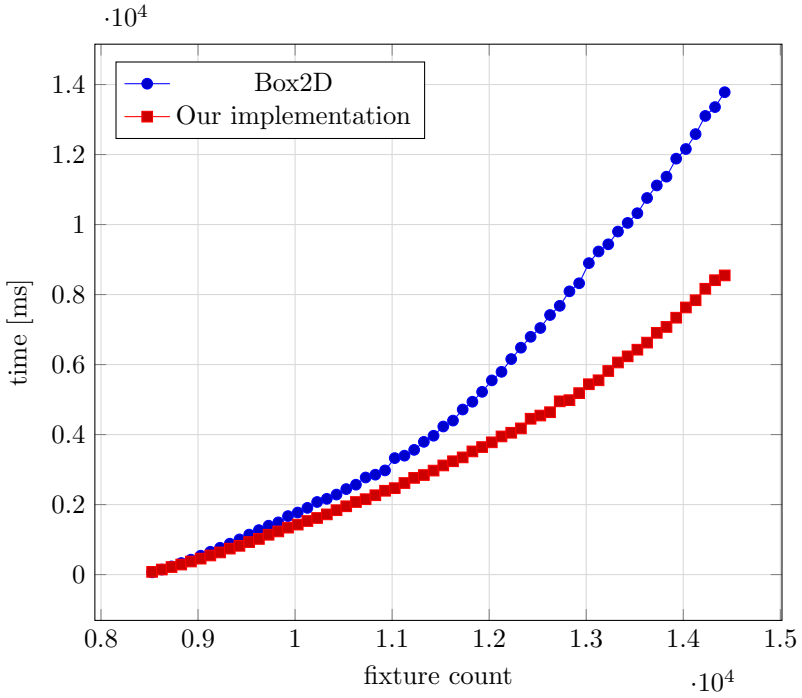


Figure 31: Benchmark "Mixed Static/Dynamic"

Add pair This benchmark is a generalized version of the "Add-pair" benchmark used in the broad-phase performance evaluation section. We have already examined how this benchmark exhibits bad performance due to Box2D's enlarged AABBs. In Figure 19 we see that the running time increases so sharply that comparatively our library's almost looks like a straight line. It is hard to distinguish what happens at smaller world sizes, but the performance is much worse from the begging. Even at 500 fixtures, were the time appears to be very close, Box2D is already close to *10 times* slower ($88ms$ for our implementation versus $725ms$ for Box2D).

Multi-fixture In this benchmark our library delivers stable running time, that increases linearly with the number of fixtures. On the other side, Box2D's performance is unstable after a certain point. The reason for the sudden slowdown is that the enlarged AABBs start creating a lot of unnecessary collisions. At some point, the objects that have the most fixtures slightly penetrate each other, and Box2D's broad-phase has trouble handling the scene. It's running time explodes to $300ms$ *per step*, resulting in the huge vertical spike at the end of the plot.

Falling squares and Falling circles Both of the scenes exhibit similar behavior. Our library is multiple times faster at all fixture counts. The improved performance is a result of the faster broad-phase and the reduced number of contacts.

Slow explosion This scene has slow moving objects in a zero-gravity world, so the future position of objects is easy to predict. It was initially made to favor Box2D's broad-phase, but ended up doing the opposite. Box2D spends much time in the broad-phase to update the bounding volumes and

traverse the tree for queries. Our broad-phase algorithm builds the whole tree and does not perform any queries after that. The few collisions are handled with little overhead. The result is that our library is an order of magnitude faster in almost all world sizes.

Tumbler In this benchmark, the two libraries perform very similarly, with ours being slightly better overall. We found out that our broad-phase performs worse compared to Box2D's in this scene, probably because of AABB enlargement working well. Even with the broad-phase being slower, the combined performance is better due to the other optimizations we made.

N^2 In both degenerate scenes, our library handles the high number of collisions more efficiently. This is the result of our optimized broad-phase and contact manager, which can process a higher number of collisions per unit of time. The number of fixtures in these benchmarks are fewer due to the scaling limitations of handling $O(n^2)$ collision pairs.

Mostly static In both scenes where the world is mostly static, Box2D has the lead for huge worlds. Our library is marginally faster on smaller worlds, but for more than fifteen thousand fixtures, Box2D scales better (Figures 27 and 28). The performance gap is more apparent in the scene were each object is a different body.

Diagonal The fixture count in this scene starts at 3000 because there is a fixed number of static bodies. On average, our library performs four times better. Initially, when only static bodies exist, the running time of our implementation is very low. This is the result of the special handling of static bodies in the broad-phase.

Big mobile Until 3000 fixtures both libraries perform similarly good. With the next size increase, our implementation handles the high fixture count much more efficiently.

Mixed Static/Dynamic In this scene both libraries have good performance, but ours is measurably faster, especially as the fixture count increases.

General remarks:

- For very small worlds (fixture count < 200), the two libraries usually exhibit similar performance and the difference can be considered neligible.
- For the majority of scenes, our library scales much better to thousands of bodies.
- In scenes made up mostly of static bodies, both libraries have similar performance but Box2D scales better to massive worlds.
- Our implementation can cope better with large number of collisions and contacts. By extension, it also better handles degenerate scenes.

9 Future work

9.1 Vectorization

Modern processors have SIMD instructions that operate on vector registers instead of scalars. For CPU-intensive code, these instruction can lead to much higher performance. Although compilers have support for automatic vectorization of programs, it is limited in scope and cannot generate efficient vector instructions for non-trivial code. The best way to reliably generate optimized vector instructions is to use *SIMD intrinsics*, low-level functions that the compiler directly translates to SIMD instructions. We believe that intrinsics can be used to optimize specific core functions that are frequently used.

9.2 Multithreading

The number of processor cores has steadily increased over the past years. Both Box2D and our library do not take advantage of many-core systems. To increase performance, individual stages of the physics pipeline can be enhanced with optional multi-threading support. In particular, we believe that parallel implementations for the broad-phase and the solver can lead to significant performance improvements. In the broad-phase, after the bounding volumes are split in two sub-trees, separate threads can be used to concurrently process them. In the solver, multiple threads can be used to solve body islands concurrently, since by definition the islands are independent.

9.3 Support for new shapes

The set of shapes supported by Box2D is minimal. Other common shapes like ellipses and circle slices are not supported by default and the only alternative is to use polygon approximations. However, polygons are limited to eight vertices, which is not always enough to approximate those shapes to a good degree. To support a new shape, collision detection routines must be implemented for each shape combination. As a result it is increasingly harder to introduce new shapes with the current design.

Most of these issues can be addressed by using a more general narrow-phase collision detection algorithm, for example, the well-known GJK [13] algorithm. It can be used to support arbitrary convex shapes by implementing a single function for each shape. This enables to easily implement new and user-defined shapes while conservating robust performance and numerical stability.

10 Bibliography

References

[1] David Baraff. "An introduction to physically based modeling: rigid body simulation I—unconstrained rigid body dynamics". In: *SIGGRAPH course notes* 82 (1997).

[2] David Baraff. "An introduction to physically based modeling: rigid body simulation II—nonpenetration constraints". In: *SIGGRAPH course notes* (1997), pp. D31–D68.

[3] David Baraff and Andrew Witkin. "Dynamic simulation of non-penetrating flexible bodies". In: *ACM SIGGRAPH Computer Graphics* 26.2 (1992), pp. 303–308. DOI: 10.1145/142920.134084.

[4] Jan Bender, Kenny Erleben, and Jeff Trinkle. "Interactive Simulation of Rigid Body Dynamics in Computer Graphics". In: *Computer Graphics Forum* 33.1 (2013), pp. 246–270. DOI: 10.1111/cgf.12272.

[5] Erin Catto. *Box2D*. https://github.com/erincatto/box2d.

[6] Erin Catto. *Continuous Collision*. https://box2d.org/files/ErinCatto_ContinuousCollision_GDC2013.pdf. 2013.

[7] Erin Catto. *Dynamic Bounding Volume Hierarchies*. https://box2d.org/files/ErinCatto_DynamicBVH_Full.pdf. 2019.

[8] Erin Catto et al. "Fast and simple physics using sequential impulses". In: *Games Developer Conference*. 2006, p. 9.

[9] Erin Catto. "Iterative dynamics with temporal coherence". In: *Game developer conference*. Vol. 2. 5. 2005.

[10] Erin Catto. "Modeling and solving constraints". In: *Game Developers Conference*. 2009, p. 16.

[11] Jonathan D. Cohen et al. "I-Collide". In: *Proceedings of the 1995 symposium on Interactive 3D graphics - SI3D 95* (1995). DOI: 10.1145/199404.199437.

[12] Mathias Eitz and Gu Lixu. "Hierarchical Spatial Hashing for Real-time Collision Detection". In: *IEEE International Conference on Shape Modeling and Applications 2007 (SMI 07)* (2007). DOI: 10.1109/smi.2007.18.

[13] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. "A fast procedure for computing the distance between complex objects in three-dimensional space". In: *IEEE Journal on Robotics and Automation* 4.2 (1988), pp. 193–203. DOI: 10.1109/56.2083.

[14] Jeff Goldsmith and John Salmon. "Automatic creation of object hierarchies for ray tracing". In: *Computer-Aided Design* 19.10 (1987), p. 572. DOI: 10.1016/0010-4485(87)90111-4.

[15] S. Gottschalk, Ming Lin, and Dinesh Manocha. "OBBTree: A Hierarchical Structure for Rapid Interference Detection". In: *Computer Graphics* 30 (1997). DOI: 10.1145/237170.237244.

[16] Daniel Kopta et al. "Fast, effective BVH updates for animated scenes". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 12* (2012). DOI: 10.1145/2159616.2159649.

[17] C. Lauterbach et al. "Fast BVH Construction on GPUs". In: *Computer Graphics Forum* 28.2 (2009), pp. 375–384. DOI: 10.1111/j.1467-8659.2009.01377.x.

[18] Brian Mirtich. “Timewarp rigid body simulation”. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH 00* (2000). DOI: 10.1145/344779.344866.

[19] Brian Vincent Mirtich. *Impulse-based dynamic simulation of rigid body systems*. University of California, Berkeley, 1996.

[20] Joshua Newth. “Minkowski Portal Refinement and Speculative Contacts in Box2D”. PhD thesis. San Jose State University, 2013.

[21] Ian Parberry. *Introduction to Game Physics with Box2D*. CRC Press, 2013.

[22] Stéphane Redon, Abderrahmane Kheddar, and Sabine Coquillart. “Fast Continuous Collision Detection between Rigid Bodies”. In: *Computer Graphics Forum* 21.3 (2002), pp. 279–287. DOI: 10.1111/1467-8659.t01-1-00587.

[23] D.j. Tracy, S.r. Buss, and B.m. Woods. “Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal”. In: *2009 IEEE Virtual Reality Conference* (2009). DOI: 10.1109/vr.2009.4811022.

[24] I. Wald. “Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.1 (2012), pp. 47–57. DOI: 10.1109/tvcg.2010.251.

[25] Xinlei Wang et al. “Efficient BVH-based Collision Detection Scheme with Ordering and Restructuring”. In: *Computer Graphics Forum* 37.2 (2018), pp. 227–237. DOI: 10.1111/cgf.13356.

[26] René Weller. “Kinetic Data Structures for Collision Detection”. In: *Springer Series on Touch and Haptic Systems New Geometric Data Structures for Collision Detection and Haptics* (2013), pp. 49–89. DOI: 10.1007/978-3-319-01020-5_3.