

From C# Scripts to Packed C Zero-Indirection Physics: A Cache-Optimized 2D Game Engine Architecture with Intrusive Execution

Brent S. Hartshorn , and Gilead Cosman

September 27, 2026

Abstract

Unity’s built-in 2D physics is an integration of the Box2D engine [12]. We build the same kind of 2D game runtime from a different direction. Game scripts are written in a subset of C#, translated to a subset of C++ and then to C, and compiled together with a fork of Box2D v3 that we call Box2D-Packed. Our goal is a 2D runtime that is faster than Unity’s. This paper describes the two techniques we use to get there and measures each of them. The first is data packing: public handles shrink from 8 to 4 bytes, and the records the engine reads on every step are laid out by measured access patterns. Measured with Cachegrind’s exact cache simulation, last-level data cache misses fall by up to 65% on query-heavy scenes relative to Box2D v3, with unchanged instruction counts and bit-identical simulation results. The second is an *intrusive engine*: game logic is compiled into the physics engine at documented points, so it runs while the engine’s data is still in cache, instead of through callbacks and event arrays. Injected pre-solve logic saves 1.3% of instructions, and injecting a platformer’s event handlers cuts L1 data misses by 2%, with identical game results. We also report where injection does not help: a per-body gravity hook that reads game data costs 7.65% more L1 misses, and event handlers followed by game work on the same objects add last-level misses. We compare against Box2D v3, not yet against Unity, and describe the measurements that remain.

1 Introduction

Game engines such as Unity let developers write gameplay in C# and rely on built-in engines for rendering and physics. For 2D games, Unity’s physics is Box2D [12], the open-source engine by Erin Catto [2]. Between the game code and Box2D, Unity adds a managed runtime, component objects, and callbacks such as `OnCollisionEnter2D`.

Our project, `crust` [7], takes a different path to the same kind of game. Its `unity_pack` tool reads a Unity-shaped project, meaning C# scripts and `.unity` scene files, and produces a small C program:

1. The scripts, written in a subset of C#, are translated to a subset of C++ by `cs2cpp`.
2. The C++ subset is translated to C by `cpprust`, and checked by `crust`’s C compiler.
3. The C is compiled by `gcc` together with scene data tables and the physics engine.

Nothing in the result is a general object runtime. Each script class becomes an array of small packed structs, sized from the scene, and object references become array indices. Box2D-Packed is the 2D physics engine of this pipeline. Our goal is a 2D game runtime that is faster than Unity’s, while accepting the same kind of C# the game developer already writes. This paper covers the physics side of that goal. We have not yet run a comparison against Unity itself, and we say so where it matters.

Box2D-Packed makes two kinds of changes to Box2D v3:

- **Data packing.** Smaller public handles and structures, and a layout of the engine’s internal records that follows measured access patterns (Sections 5 to 7).
- **The intrusive engine.** A build pipeline that compiles game code into the engine at marked points, so game logic runs where the engine produces events (Section 8), and its use as the physics backend of `unity_pack` (Section 9).

We measure everything with deterministic cache simulation and require simulation results to stay bit-identical to Box2D v3, unless a change is meant to alter behavior. We report negative results in the same detail as positive ones, because several of them changed our design.

2 Background and Concepts

This section introduces the ideas the rest of the paper relies on, for readers who have not worked on game engines or low-level performance.

2.1 What a physics step does

A 2D physics engine advances the simulation in fixed time steps, typically 60 per second. Each step has roughly four phases:

1. **Broad phase.** Find pairs of shapes whose bounding boxes overlap. Box2D keeps the boxes in a tree (a bounding volume hierarchy) so it does not test every pair.
2. **Narrow phase.** For each overlapping pair, compute the exact contact: whether the shapes touch, where, and along which normal. The result is a *contact manifold*.
3. **Solver.** Compute the forces that keep touching bodies from passing through each other, and integrate velocities and positions.
4. **Events.** Report what happened to the game: which pairs started or stopped touching, which shapes entered sensors, which impacts were hard.

Games attach their own logic to these phases: a *custom filter* decides whether two shapes may collide at all, and a *pre-solve* callback may disable a contact just before it is solved, which is how one-way platforms work.

2.2 Handles instead of pointers

Game code needs to refer to bodies and shapes that live inside the engine. Box2D v3 does not give out pointers. It gives out *handles*: an index into the engine’s internal array, plus a *generation* number that is incremented whenever a slot is reused. If the game keeps a handle to a destroyed body, the generation no longer matches, and the engine can detect the stale handle instead of corrupting memory. The size of a handle matters because games store many of them.

2.3 Caches, cache lines, and misses

A CPU does not read single bytes from main memory. It reads 64-byte blocks called *cache lines* into a hierarchy of caches: a small, fast level-1 (L1) cache per core and a large, slower last-level cache (LL). Reading data that is not in L1 is an *L1 miss* and costs roughly tens of cycles. Reading data that is not in any cache is a *last-level miss* and costs one or two hundred cycles, because it goes to main memory. Two consequences shape this paper:

- Fields read together should sit in the same cache line. A record whose hot fields are spread over three lines costs three misses where one would do.
- A cache is divided into *sets*, and each line can live only in the set its address maps to. If a loop visits records whose size is a power of two, the same field of every record can map to the same few sets, and those sets thrash while the rest of the cache sits idle. This is called set aliasing.

Hardware *prefetchers* detect sequential access and load the next lines early, which hides most of the cost of streaming through an array. Random access, such as looking up an object by index, cannot be predicted and pays the full miss cost.

2.4 Intrusive data structures

The name of our build pipeline comes from a C++ idea. A *non-intrusive* container, such as a `std::list` of objects, keeps its bookkeeping outside the objects: each list node holds a pointer or a copy, and the objects know nothing about the list. An *intrusive* container stores the bookkeeping inside the objects themselves [6]. An intrusive linked list puts the `next` and `prev` links in each element, so walking the list reads only the elements, not separate nodes. The same idea applies to reference counting: `std::shared_ptr` keeps its count in a separate control block, while an *intrusive pointer* (`boost::intrusive_ptr`) keeps the count as a field of the object, so there is no extra allocation and no extra memory read.

The trade is the same in both cases. An intrusive design gives up separation, because the object now knows about its container, and gains locality and fewer indirections. We apply that trade to the boundary between a game and its physics engine. Normally the game sits outside the engine and learns about events afterwards, through callbacks or arrays. In an *intrusive engine*, the game’s handlers are compiled into the engine’s own loops, at the points where the engine already holds the relevant data.

2.5 Callbacks, event arrays, and link-time optimization

A *callback* is a function pointer the game registers with the engine. The engine calls it for every relevant pair or contact. An indirect call through a pointer cannot be inlined by the compiler, and the callback usually has to look up the game’s own data by handle. An *event array* avoids the indirect call: the engine writes one entry per event during the step, and the game reads the entries afterwards. By then the engine’s data for that event has often left the cache. *Link-time optimization* (LTO) lets the compiler optimize across source files as one program, which enables inlining between files that are compiled separately.

3 Box2D v3

Box2D v3 is already designed for cache efficiency. It keeps awake bodies in contiguous arrays of body records (`b2BodySim`) and compact 32-byte body states used by the solver. Contacts that touch are distributed over *graph colors*, so that no two contacts in one color share a moving body, and each color is solved in parallel with SIMD, 4 or 8 contacts at a time. The narrow phase (`b2CollideTask`) visits every awake contact each step, updates its manifold or reuses it when relative motion is small (contact recycling), and records which contacts started or stopped touching. The engine then processes those changes on a single thread and creates begin and end events.

4 Methodology

Timing was not precise enough. We first compared wall-clock times across the 14 scenes of the Box2D benchmark suite. On one shared core, run-to-run noise ranged from 0.3–3.7% in a quiet session to 5–10% in

a busy one. Interleaving the builds and pairing runs did not reduce the noise enough to resolve the 2–5% effects that layout changes produce.

Cachegrind. We therefore measure with Cachegrind [10], which simulates the cache hierarchy for every memory access. We fix the simulated hierarchy at a 32 KB 8-way L1 data cache, an 8 MB 16-way last-level cache, and 64-byte lines, so results do not depend on the host. A repeated run reproduced every counter to the last digit, so any difference between builds is real. We report instructions, L1 data misses, and last-level data misses.

Limits of the method. Cachegrind does not model hardware prefetchers. It counts one miss per line for a sequential stream that a real CPU would mostly hide, while it models random access well. All measurements use one worker thread. The results show where memory traffic changes, but they do not predict time on a particular CPU.

Correctness. Every change passes Box2D’s unit tests in release and debug builds, including a cross-platform determinism test against a fixed state hash, worker-count parity checks, and snapshot and replay round trips. Every injection demo builds the same game both ways, through the standard API and through injection, and requires identical results.

5 Data Packing

Handles. Body, shape, chain, and joint handles store a 16-bit index and a 16-bit generation, 4 bytes in total, where Box2D v3 uses 8 (Table 1). The world index is removed, so Box2D-Packed supports one world at a time, enforced at compile time. Contact handles keep 32-bit fields, 8 bytes instead of 12, because large piles exceed 65,535 contacts and contacts churn fast enough that a 16-bit generation would wrap quickly. Creating more than 65,535 live objects of one kind logs an error and returns a null handle, instead of silently wrapping onto an existing object.

Table 1: Handle sizes in bytes.

Handle	Box2D v3	Box2D-Packed
Body, shape, chain, joint	8	4
Contact	12	8
World	4	4

Definitions and filters. Boolean flags in the creation structures become 1-bit `bool` fields. We use `bool` rather than `uint8_t` bit-fields because C converts any non-zero value to 1 when storing into a `bool`, while a 1-bit unsigned field keeps only the low bit and silently stores 0 for a value such as `0x2`. Fields are ordered by alignment. Collision filters use 16-bit category and mask fields, as in Box2D v2, which reduces `b2Filter` from 24 to 6 bytes (Table 2). None of these structures is read during the simulation step, so, as expected, a timing comparison after this phase showed no difference beyond noise.

Table 2: Structure sizes in bytes, 32-bit float build.

Structure	Box2D v3	Box2D-Packed
b2Filter	24	6
b2QueryFilter	16	4
b2ShapeDef	88	56
b2BodyDef	88	72
b2ChainDef	88	64
b2DistanceJointDef	128	112
b2RevoluteJointDef	120	104
b2Shape (internal)	280	264
b2TreeProxy (internal)	24	16
b2ContactSim (internal)	200	192

6 Hot-Path Layout

We used per-function and per-line Cachegrind annotations to find where misses occur, then changed the layout of the records read there.

Category bits in tree nodes. A tree query tested each overlapping leaf’s category bits through a separate proxy array, a second random access per leaf, even for leaves the filter rejects. A 2D tree node has 8 unused bytes, which hold the z bounds in 3D. We store leaf category bits there, so queries test the node they have already loaded, and only leaves that pass the filter read the proxy. The `tree_cast` scene, which does not touch shapes, isolates this change: L1 data misses fall 2.9%, last-level misses 7.3%, and instructions 0.4%.

A hot cache line in b2Shape. The broad-phase pair filter read fields at offsets 4, 16, 20, 92, and 256 of each shape, three cache lines per shape. We moved every field the filter reads into the first 64 bytes, so the filter now reads one line.

Power-of-two aliasing. Packing the shape’s flags into bits made `b2Shape` exactly 256 bytes, and Cachegrind showed last-level misses in `large_pyramid` rising 26.9%. The increase came from `b2ComputeFatShapeAABB`, which does not read any moved field. With a 256-byte stride, one field of every shape falls into only a quarter of the cache sets, and the loop that sweeps all shapes each step thrashes those sets. Keeping plain `bool` flags gives 264 bytes, shifts each shape by 8 bytes, and spreads the fields over all sets. At 264 bytes, the regression disappeared (+0.1%), and the gains elsewhere roughly doubled: relative to the previous step, `tile_world` last-level misses fell 62.5% rather than 36%, and `queries` 57% rather than 25%. A compile-time assertion now keeps the size from returning to a power of two.

Narrow phase. `b2CollideTask` read `bodyId` from both shapes of every contact, two random cache lines, although the value is needed only in a rare fallback for sleeping bodies. We now read the shape only in that branch, and moved the body fields the narrow phase reads to the first 40 bytes of `b2BodySim`. L1 data misses fell by 1.0–3.4% in scenes with contacts.

A three-line contact record. At 200 bytes, most contact records straddled a fourth cache line. The GJK simplex cache (8 bytes) is used only by chain-segment collisions, so we moved it to the cold per-contact record, which made `b2ContactSim` exactly 192 bytes, three lines.

Table 3: Box2D-Packed relative to Box2D v3 under Cachegrind (32 KB L1, 8 MB LL, one worker). Changes of 5% or more are in bold. Negative is better.

Scene	Instr.	L1 data	LL data
tile_world	-0.1%	- 25.7%	- 65.3%
queries	+0.2%	- 8.6%	- 56.7%
tree_cast	-0.4%	-2.8%	- 7.3%
smash	+0.3%	-2.7%	- 5.4%
large_pyramid	+0.2%	-3.6%	- 18.0%
many_pyramids	+0.2%	-3.6%	-1.6%
joint_grid	+0.0%	-0.3%	-4.3%

Cumulative result. Table 3 compares Box2D-Packed with Box2D v3. Last-level data misses fall in every scene, from 1.6% in `many_pyramids` to 65.3% in `tile_world`, and instruction counts change by at most 0.4%. The largest gains are in query-heavy scenes. The table was measured before the injection markers of Section 8 were added. The markers compile to nothing in normal builds, and results stay bit-identical, but we have not re-run the table since.

7 Negative Results

Ordering contacts within a color. The wide contact solver accounts for 64% of L1 read misses in `large_pyramid`. Because a body appears at most once per graph color, the order of contacts within a color cannot change results. We sorted each color by body index, expecting the solver’s body reads to become sequential. All determinism tests passed, and the solver’s misses were identical to the last digit (95,726,000). The misses we had attributed to body reads were the first touch of each 592-byte constraint bundle in a sequential sweep, which prefetchers largely hide on real hardware. The sort cost 10.4% more instructions and was dropped.

Removing inverse mass copies. Removing four inverse-mass copies from `b2ContactSim` also produced a 192-byte record with bit-identical results and cut `large_pyramid` last-level misses 23.3%. It raised L1 misses 3.3% and instructions 0.6%, because contact preparation then read body records at random, and `many_pyramids` came out net worse. Moving the simplex cache reached the same size without these costs.

8 The Intrusive Engine

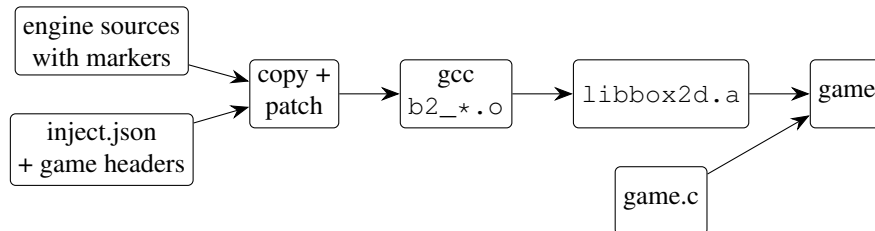


Figure 1: The intrusive engine build pipeline.

8.1 Pipeline

The engine sources contain marker comments of the form `//$scope$POINT` on lines of their own. A normal build ignores them. The script `box2d_pack.py` copies the sources, replaces markers in the copies with C code described by a JSON file, and compiles the engine and the game together (Figure 1). Compiler errors point at the JSON entry, not the patched copy, and the game’s own headers can be included, so injected code can use the game’s types and inline functions directly.

A one-way platform, for example, is injected as follows. The game’s rule is a `static inline` function in its own header, and the pre-solve marker calls it with the two shapes the engine is already holding:

```
{
  "includes": [ "platformer.h" ],
  "defines": { "B2_PACK_NO_PRE_SOLVE_FCN": 1 },
  "inject": [ {
    "event": "pre_solve",
    "code": [ "Game_PreSolve( shapeA->userData, shapeB->userData, ",
              "                  &contactSim->manifold );" ]
  } ]
}
```

8.2 Injection points and their contract

Table 4 lists the injection points. Each documents its thread context and the variables injected code may use. Two run on worker threads and must be thread-safe, and the build tool prints a note whenever code is injected there. Where the engine uses a concept in several places, the marker lives in one inline hook that every place calls, so injected logic applies consistently. The custom filter, for example, applies to new contacts, sensor overlaps, and continuous collision.

Table 4: Injection points.

Event	Runs	Typical use
<code>pre_step</code> , <code>post_step</code>	main thread, world unlocked	game update
<code>contact_begin</code> , <code>contact_end</code>	single thread, world locked	damage, scoring
<code>contact_hit</code>	single thread, world locked	impact effects
<code>sensor_begin</code> , <code>sensor_end</code>	single thread, world locked	pickups, triggers
<code>custom_filter</code>	worker threads	teams, pass-through
<code>pre_solve</code>	worker threads	one-way platforms
<code>body_gravity</code>	worker threads, every substep	gravity fields

Each marker is verified by an equivalence test that builds one scene with the standard callbacks and event arrays, and again with injection, and requires identical output, including every body’s final position. A control build without the filter and pre-solve rules must produce a different result, which shows the test is sensitive to those rules. The equivalence test found a bug in our first gravity hook: it changed only velocity integration, while continuous collision also uses a body’s gravity scale when it rolls a fast body back to its time of impact. The `unity_pack` integration found a second marker bug, described in Section 9, and the equivalence test now covers it.

8.3 Evaluation

Table 5 summarizes four demos. Each builds the same game both ways, and each pair of builds produces the same checksum over game state and body positions.

Table 5: Injected build relative to the standard build, under Cachegrind.

Demo	Instructions	L1 data	LL data
Arena brawl, contact begin (121 events per step)	−0.16%	−0.50%	0%
Bullet storm, contact begin (2,112 per step)	−0.16%	−0.26%	+0.62%
Platformer crowd, pre-solve (about 40,000 contacts)	−1.33%	−0.20%	−0.84%
Platformer, four event rules	−1.05%	−2.05%	+0.01%
Platformer, event rules + gravity hook	+0.07%	+7.65%	+0.01%

Per-event savings are real but small. Each injected contact event saves about 70 instructions (74 in the arena, 66 in the bullet storm) and a few L1 misses (5.2 and 1.4): the event array write, the array walk, and two random lookups of user data. In the arena and the bullet storm that is 0.16% of instructions, because each event also carries much larger engine work: a contact is created and destroyed, and a recycled bullet’s broad-phase entry moves.

Injection moves game work away from the game’s own hot data. In the bullet storm, last-level misses rose. The handler touches bullet objects in the middle of the step, which evicts engine data, and the game’s post-step pass that relaunches bullets no longer finds those objects in cache. Injection helps when the handler uses engine data that is hot at the marker, or records little, and hurts when the game does follow-up work on the same objects after the step.

Pre-solve is the best fit. A pre-solve callback runs for every touching contact every step, through a function pointer, and usually looks up both shapes’ user data. Injecting it saves 1.3% of instructions and 0.84% of last-level misses. Link-time optimization is a separate and larger effect: compiling the engine and game as one program saves 5.9% of instructions even without injection, and 6.8% with it (Table 6).

Table 6: Platformer crowd pre-solve: link-time optimization and injection.

Build	Instructions	L1 data	LL data
Standard callback	baseline	baseline	baseline
Standard + LTO	−5.86%	−0.01%	−0.07%
Injected	−1.33%	−0.20%	−0.84%
Injected + LTO	−6.77%	−0.20%	−0.92%

A per-substep hook can cost more than it saves. The platformer injects four rules: one-way platforms, coin and feather pickups, stomping enemies, and hard landings. That cuts L1 data misses by 2%, the largest gain we measured for injection. Moving the feather’s low gravity into the `body_gravity` hook as well turns the gain into 7.65% more L1 misses, because the hook runs for every body in every substep and reads the game’s data each time. The standard API stores the gravity scale once in the engine’s own hot body data,

which is the better design for a value that changes rarely. Our guideline is to inject event handlers, and to use per-body, per-substep hooks only for rules computed from the body itself.

9 Box2D-Packed in the C# Pipeline

`unity_pack` started with a physics engine of its own: an integrator and axis-aligned box contacts with a special rule to keep stacks from sinking. It now uses Box2D-Packed for all 2D physics, as Unity uses Box2D. The packed data tables that describe each `Rigidbody2D` and `Collider2D` stay as they are, and a generator in the Box2D-Packed repository produces the glue between them and a Box2D world. Each fixed step, the glue creates bodies for new rigidbodies, including ones added at runtime with `AddComponent<Rigidbody2D>()`, pushes what scripts changed (velocity, teleports, gravity scale, damping, world gravity), steps the world, and writes positions and velocities back. Friction and bounciness combine exactly as in Unity, through Box2D’s material callbacks. `unity_pack` then sends `OnCollisionEnter2D`, `Stay2D`, and `Exit2D` after the step, as Unity does.

With injection enabled, touching pairs are recorded at the contact markers instead of read from event arrays. The injected code only records the pair, so scripts still run after the step, following the guideline above. On a test scene with a ball that bounces three times and a stack of six crates, both modes give Unity’s four enter and three exit messages, the stack stands, and the injected build matches the standard build exactly. Before its own engine was removed, `unity_pack` gave the same four enter and three exit messages.

The integration found two bugs that our Box2D tests had missed. When a fast body leaves a contact, Box2D destroys the contact rather than reporting that it stopped touching, and that path had no contact end marker, so injected builds never saw those exits. A scene with a rigidbody but no colliders failed to link, because the generated glue assumed collider tables always exist. Both are fixed and covered by tests.

10 Limitations

Box2D-Packed supports one world at a time, 65,535 live bodies, shapes, chains, and joints of each kind, and 16 collision categories, and it changes Box2D’s public API and file formats. All cache results come from simulation on one thread, without hardware prefetching. In `unity_pack`, body rotation is locked, because packed rigidbodies do not model rotation yet, and 3D physics still uses `unity_pack`’s own engine. Most importantly for our goal, we have compared against Box2D v3, not against Unity.

11 Related Work

Cache-conscious data layout is a long-standing technique [3, 4], and data-oriented design brought it into game development [5]. Box2D v3 already applies it extensively [2]. Our layout work continues that approach, guided by deterministic simulation instead of timing.

Intrusive containers [6] store bookkeeping inside the objects they manage. Weaving code into a program at defined points is the core idea of aspect-oriented programming [9], with source-to-source weavers for C++ such as AspectC++ [11], and specializing a program to inputs known at build time relates to partial evaluation [8]. The intrusive engine is a small, explicit instance of these ideas applied to a physics engine’s event pipeline: a few documented injection points, a JSON description instead of a pointcut language, a thread-safety contract per point, and equivalence tests that require the injected build to reproduce the standard one exactly.

12 Future Work

- **A comparison with Unity.** The same 2D scenes built as Unity players and through `unity_pack`, measured for frame time and memory. This is the measurement our goal depends on.
- **Hardware measurement.** Hardware counters and timing on quiet multi-core machines, to test whether the simulated cache reductions become time savings, and how they scale with worker threads.
- **Rotation in `unity_pack`.** Packed rigidbodies with rotation, so Box2D bodies no longer need to be locked.
- **Contact constraint stream.** The wide contact constraint is about 148 bytes per contact. Whether shrinking it helps depends on memory bandwidth, which only hardware measurement can show.

13 Conclusion

Box2D-Packed is the 2D physics engine of a pipeline that translates a C# game subset to C and compiles it together with the engine, with the goal of a faster 2D runtime than Unity, whose 2D physics is also Box2D. Measured with deterministic cache simulation, its layout changes cut last-level data misses by up to 65% on query-heavy scenes relative to Box2D v3, with bit-identical results. The intrusive engine compiles game code into the engine at documented points. It helps most for logic the engine calls through function pointers per contact, and it can hurt when it pulls game data into per-body loops or away from the game's own follow-up work. We report both, because the second is as useful to a game developer as the first. Code, tests, and measurement scripts are available in the Box2D-Packed [1] and crust [7] repositories.

Acknowledgments. Box2D-Packed builds on Box2D by Erin Catto, released under the MIT License.

References

- [1] Box2D-Packed contributors. Box2D-Packed: A cache-oriented fork of Box2D v3. <https://github.com/crustos/box2d>.
- [2] Erin Catto. Box2D: A 2D physics engine for games. <https://box2d.org>, <https://github.com/erincatto/box2d>. Version 3, MIT License.
- [3] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 1999.
- [4] Ulrich Drepper. What every programmer should know about memory. Red Hat, Inc., 2007.
- [5] Richard Fabian. *Data-Oriented Design*. Richard Fabian, 2018.
- [6] Ion Gaztañaga and Olaf Krzikalla. Boost.Intrusive. Boost C++ Libraries, <https://www.boost.org/doc/libs/release/libs/intrusive/>.
- [7] Brent S. Hartshorn and Gilead Cosman. crust: C and C++ subsets, C# translation, and `unity_pack`. <https://github.com/brentharts/crust>.
- [8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

- [9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
- [10] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.
- [11] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, 2002.
- [12] Unity Technologies. Physics integrations in Unity. Unity Manual, <https://docs.unity3d.com/Manual/physics-integrations.html>. Built-in 2D physics uses an integration of Box2D; built-in 3D physics uses NVIDIA PhysX.