GDC | March 21-25, 2022
San Francisco, CA

# Knockout City's Parallel, Deterministic, Rewindable Entity System

#GDC22

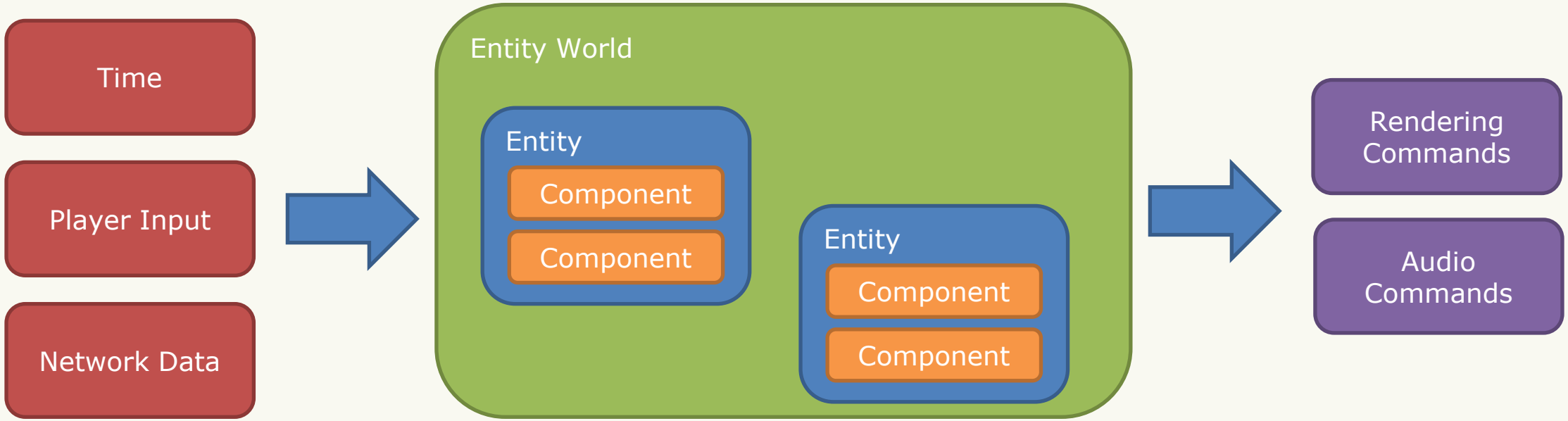
@confusionattack

VELAN STUDIOS

@velanstudios


KNOCK CITY

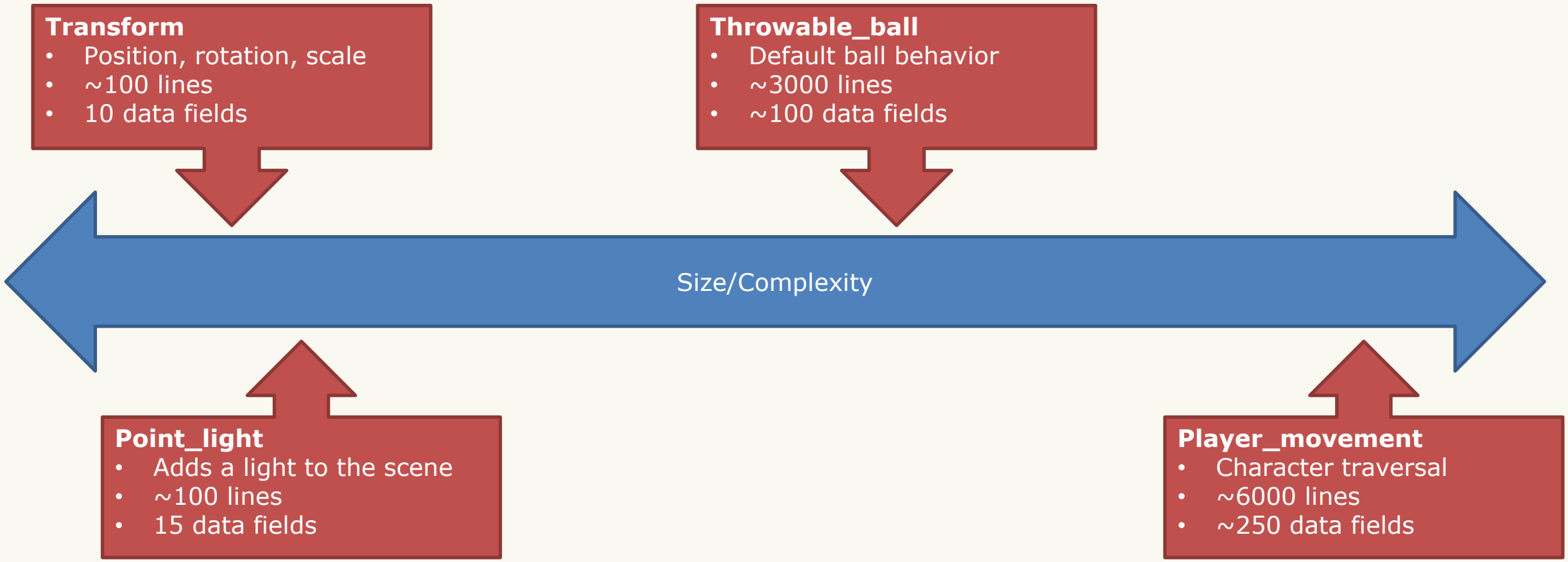
# What's An Entity System?



# What's An Entity System

- Knockout City's typically has 4000+ active components
- Components written in proprietary scripting language

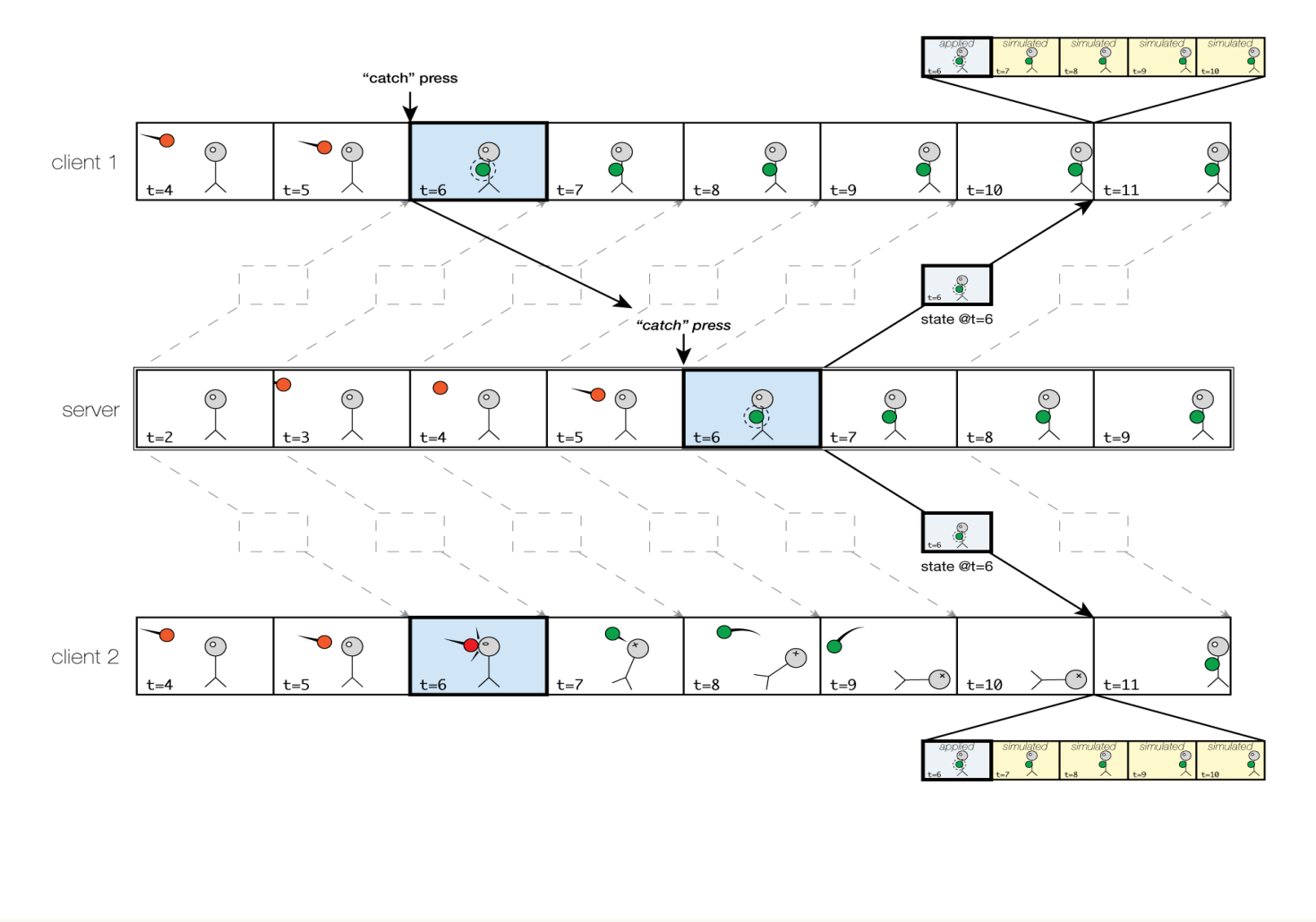
# Range of Components



# So Why Entity+Component+Script Anyway?

- Why entities and components?
  - Familiar pattern for game designers
  - Benefits of composition
- Why script?
  - Reduce cognitive load
    - Making good gameplay is hard enough
    - Also want parallelism + determinism + reversibility + replication
  - Lever for global optimization (or where framerate goes to die?)
  - Live-update / hot-reload is nice

# Motivating Architecture

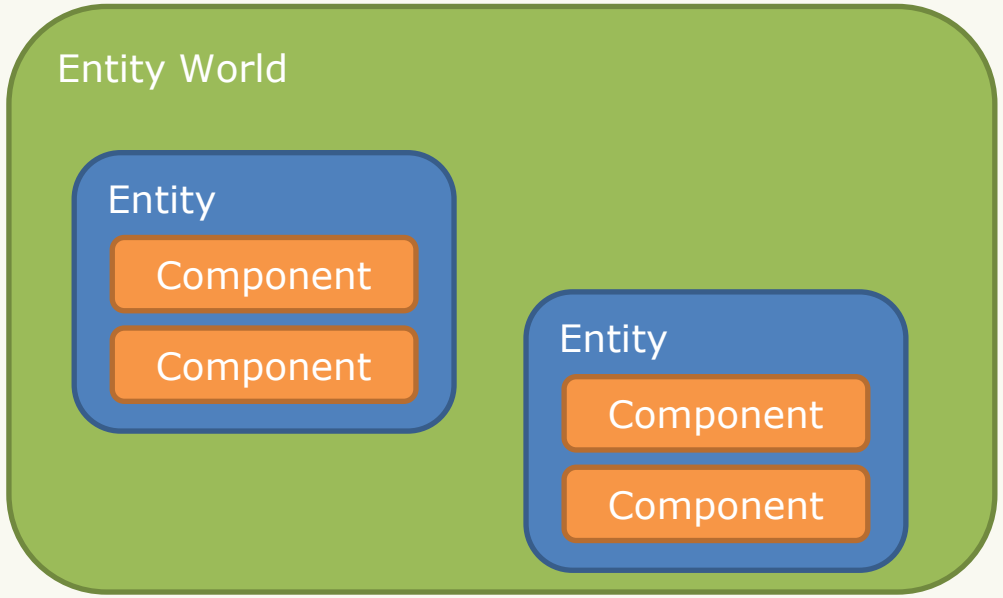


# The Rest of This Talk

- Data structures
- Components as scripted jobs
- Cross-component reads and writes
- Entity spawn & destroy
- Optimizations
- Tools

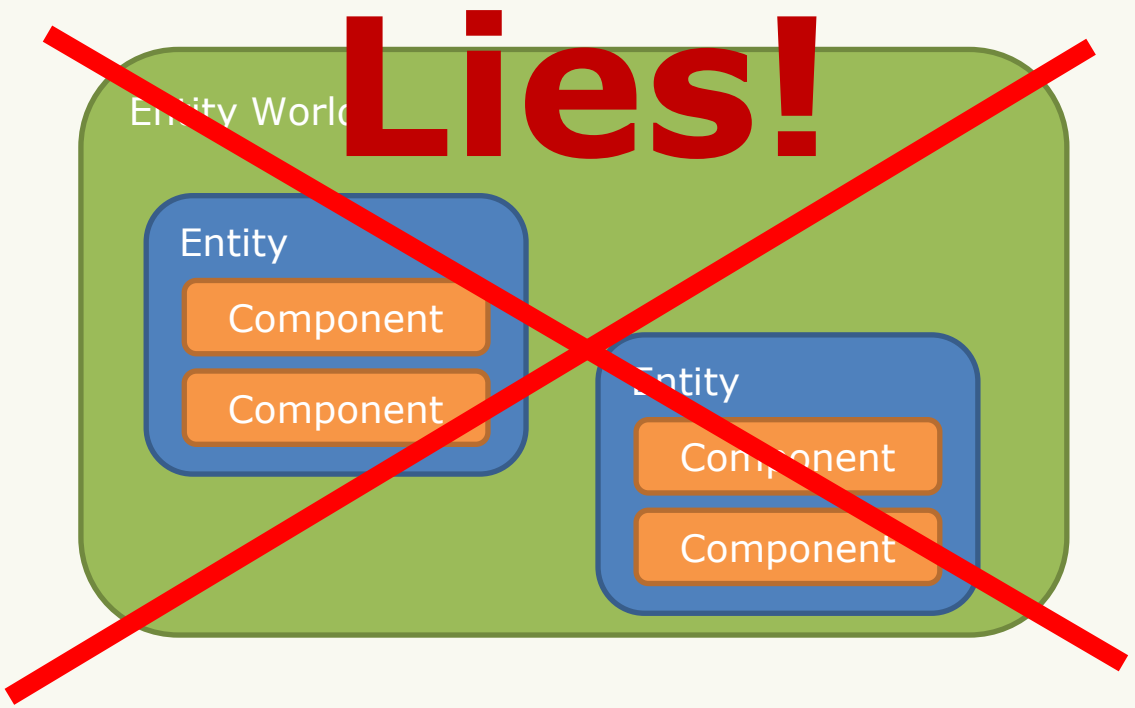
# Let's Build It!

- Entities are lists of components
- Components are function-specific data



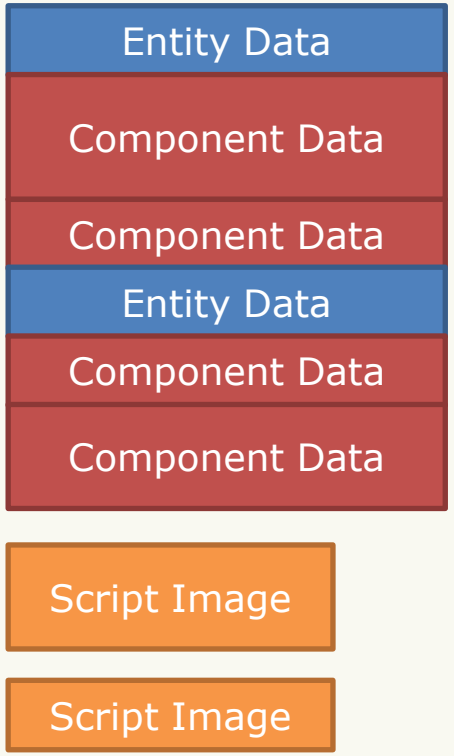
# Let's Build It!

- Entities are lists of components
- Components are function-specific data

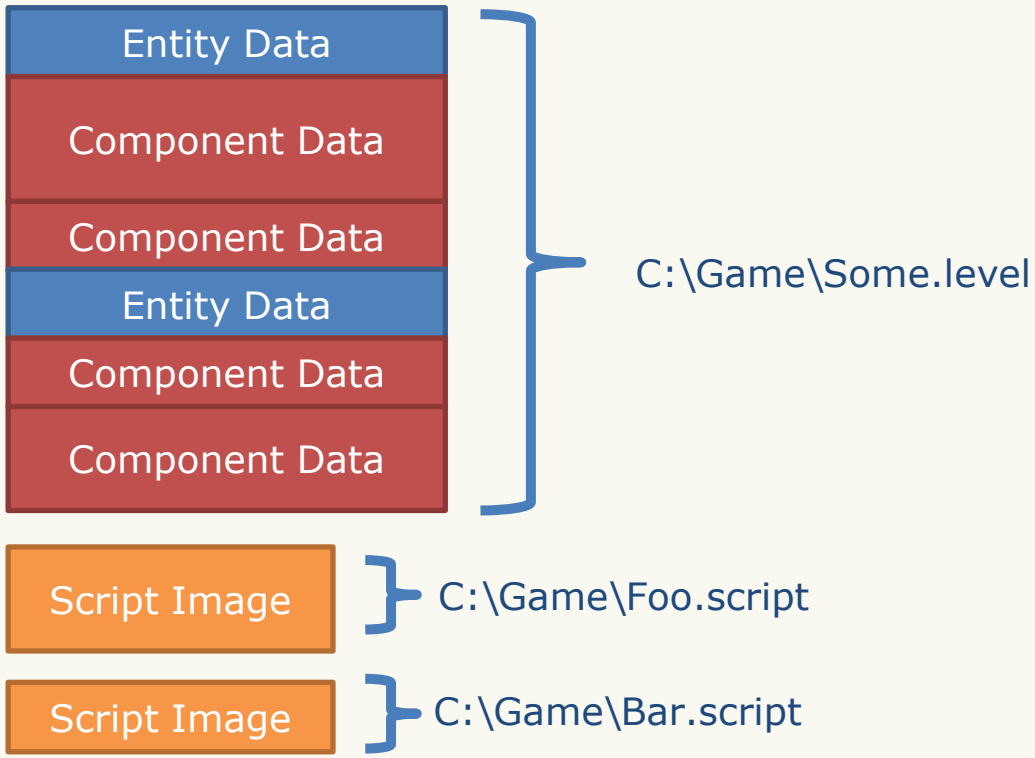


# Let’s Talk About Data Structures

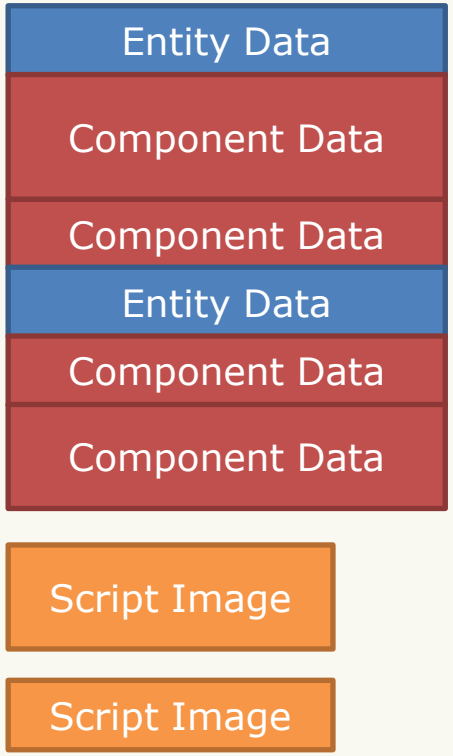
# Let’s Talk About Data Structures



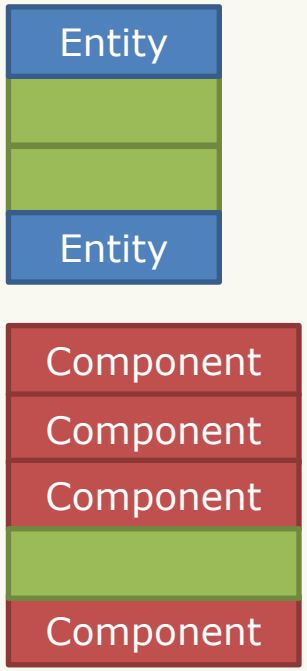
# Let’s Talk About Data Structures



# Let’s Talk About Data Structures

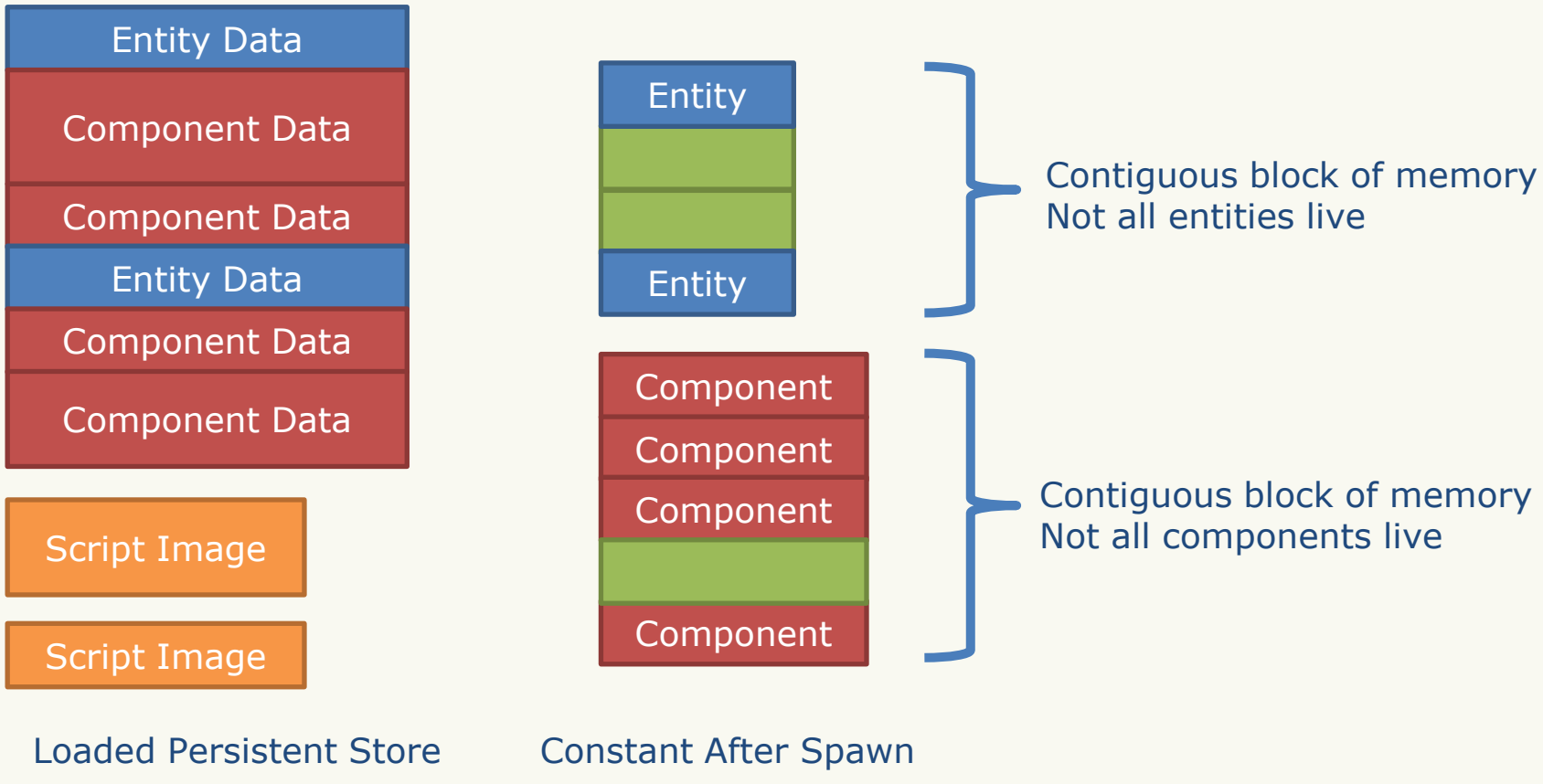



Loaded Persistent Store

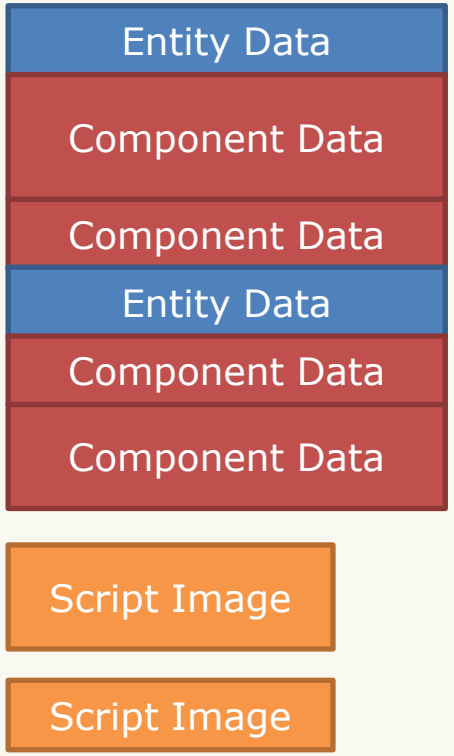



Constant After Spawn

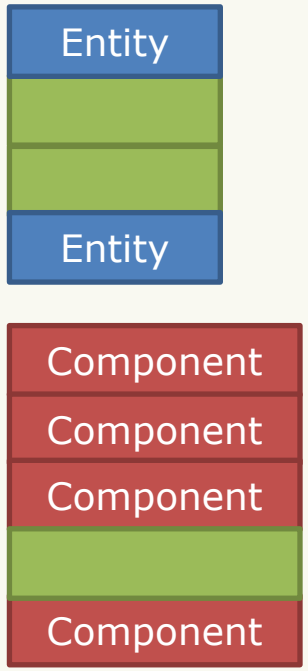
# Let's Talk About Data Structures



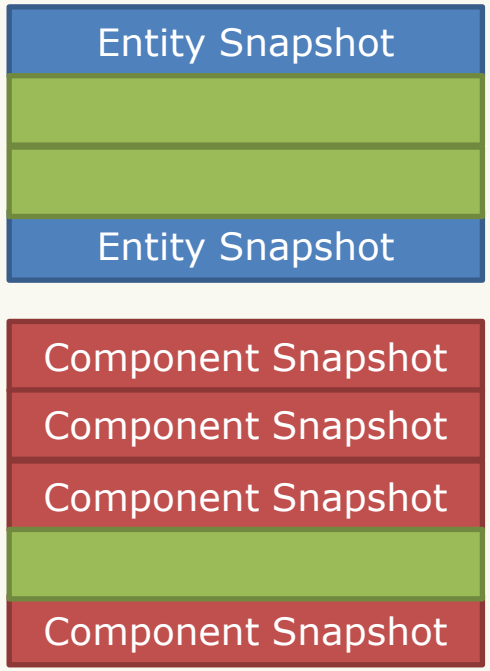
# Let’s Talk About Data Structures




Loaded Persistent Store

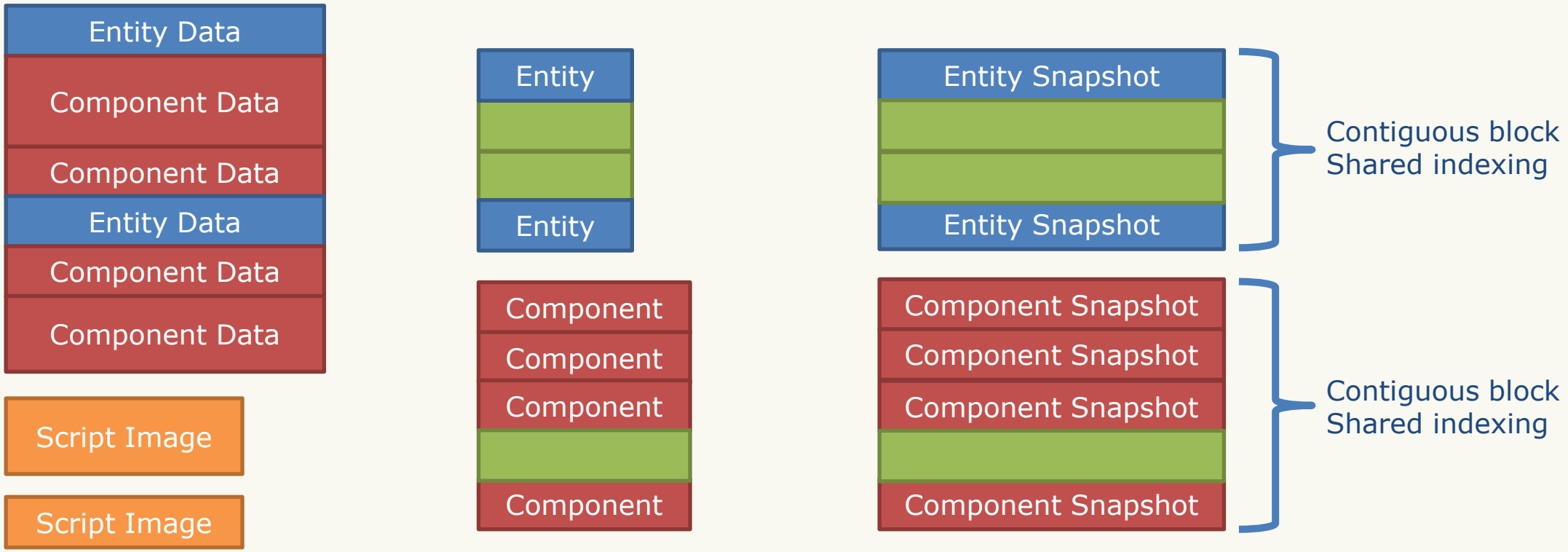



Constant After Spawn

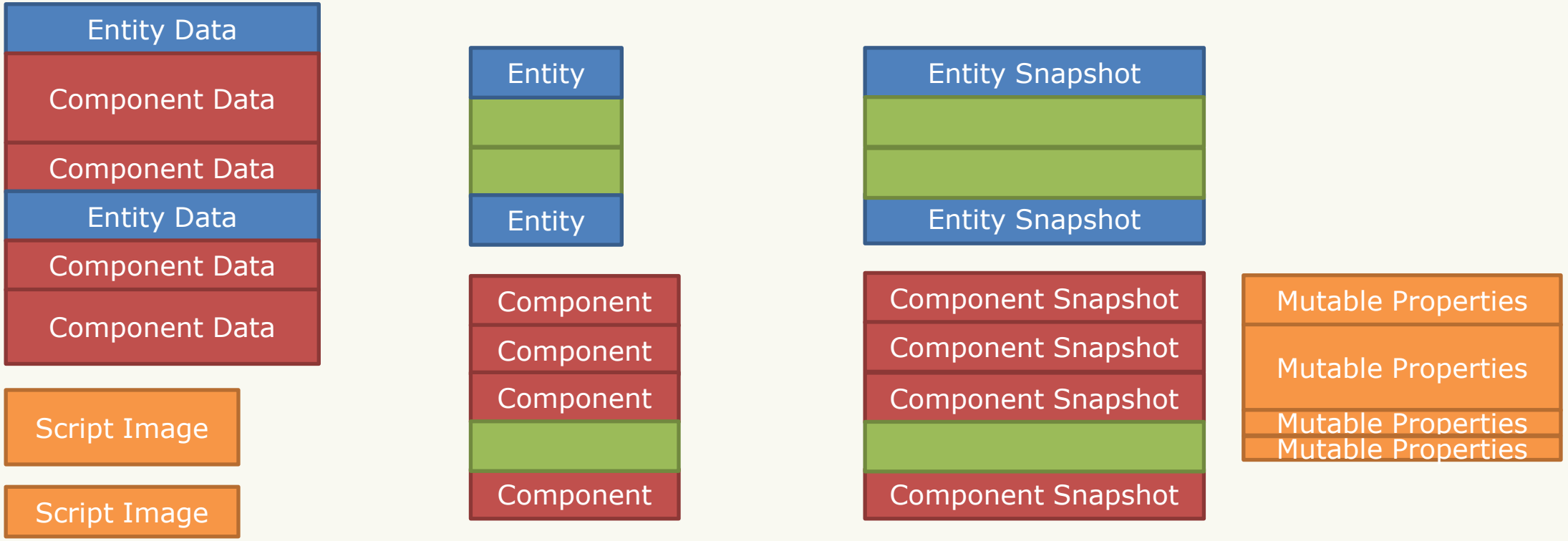



Mutable Over Lifetime

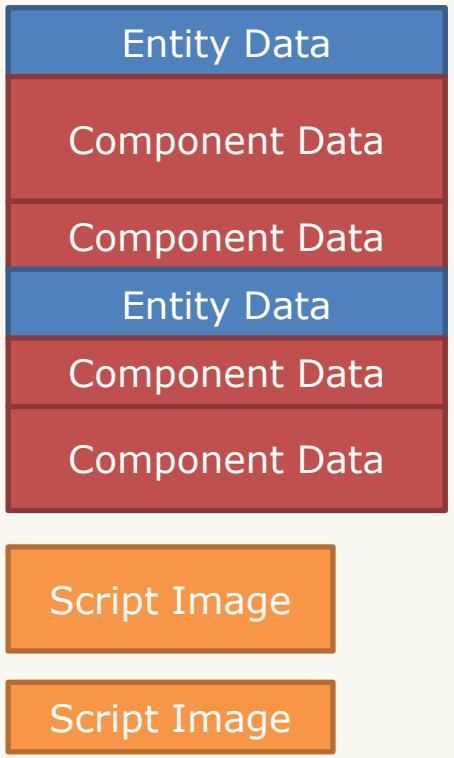
# Let's Talk About Data Structures



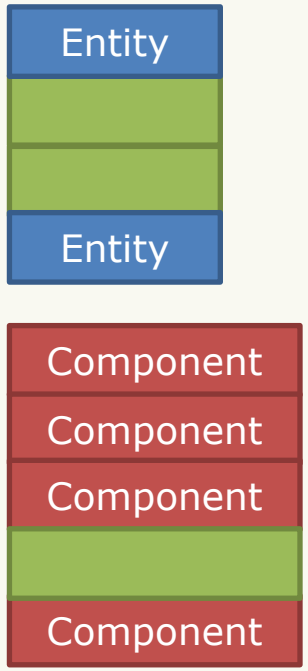
# Let's Talk About Data Structures



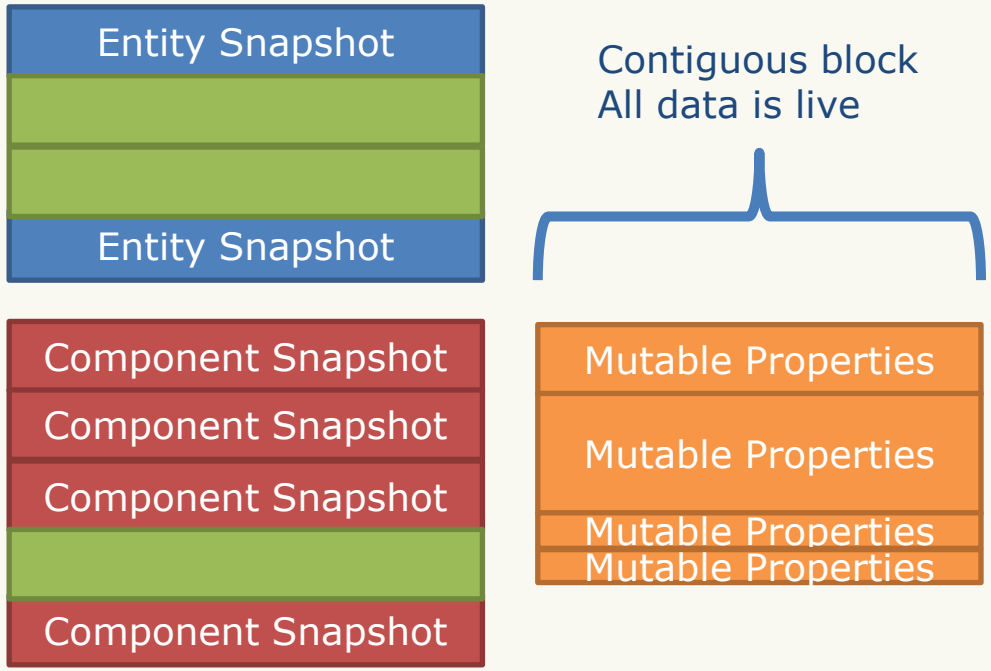
# Let's Talk About Data Structures




Loaded Persistent Store




Constant After Spawn




Mutable Over Lifetime

# Rolling Forward

Every* simulation step memcpy snapshot to form basis of next step

```
void world_rollforth(bitvector_t* live_entities, world_snapshot_t* world_prev, world_snapshot_t* world_next)
{
    entity_snapshot_t* entities_prev = world_prev->entities;
    entity_snapshot_t* entities_next = world_next->entities;

    for (int i = bitvector_ffs(live_entities, 0); i >= 0; i = bitvector_ffs(live_entities, i))
    {
        entities_next[i] = entities_prev[i];
    }

    // Then copy components...
}
```

# Memory Traffic

- Entity snapshot = 16 bytes
- Component snapshot = 16 bytes
- Typical mutable properties size = 75 bytes
- Typical entity count = 1400
- Typical component count = 4000
- Total memcpy/step = 377 kilobytes

# Let’s Build It!

- Entities are lists of components
- Components are function-specific data

# Let’s Build It!

- E s
- C ific data

Entity/component memory layout is complicated.

# Let’s Build It!

- E s
- C ific data
- Events trigger component script code to execute

Entity/component memory layout is complicated.

# <tangent> Script

```
event tick()
  float distance = 5.0 * time.get_gameplay_tick_interval()
  component_ref<transform> transform = entity.get().transform
  float last_x = transform.position.x
  transform.position.x = last_x + distance
end
```

# Script Converted to C

# Script Converted to C

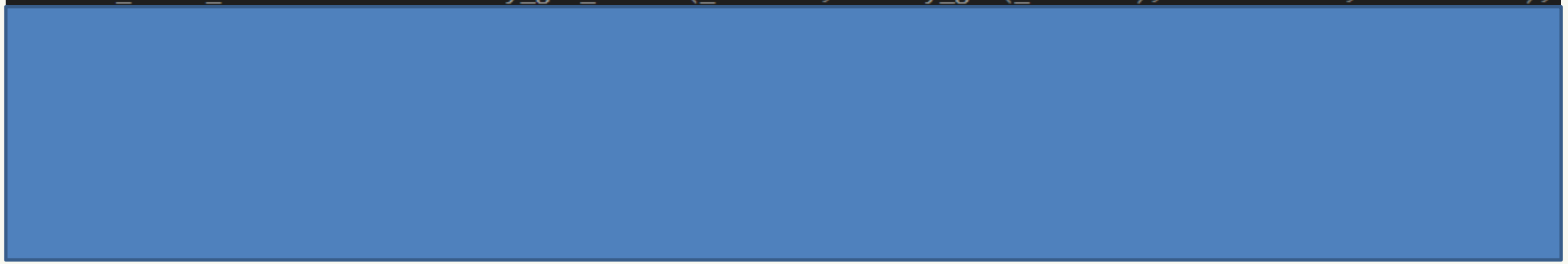
```
static void tick(vvm_native_context_t* _context)
```

# Script Converted to C

```
static void tick(vvm_native_context_t* _context)
{
    float distance = 5.0f * time_get_gameplay_tick_interval(_context);
```

# Script Converted to C

```
static void tick(vvm_native_context_t* _context)
{
    float distance = 5.0f * time_get_gameplay_tick_interval(_context);
    vvm_value_t transform = entity_get_member(_context, entity_get(_context), "transform", 1126380853);
```



# Script Converted to C

```
static void tick(vvm_native_context_t* _context)
{
    float distance = 5.0f * time_get_gameplay_tick_interval(_context);
    vvm_value_t transform = entity_get_member(_context, entity_get(_context), "transform", 1126380853);
    float last_x = component_get_mutable_vec(_context, transform, "position", 5249027).x;
```

# Script Converted to C

```c
static void tick(vvm_native_context_t* _context)
{
    float distance = 5.0f * time_get_gameplay_tick_interval(_context);
    vvm_value_t transform = entity_get_member(_context, entity_get(_context), "transform", 1126380853);
    float last_x = component_get_mutable_vec(_context, transform, "position", 5249027).x;
    component_set_mutable_vec_element(_context, transform, "position", 5249027, 0, last_x + distance);
```

# Script </tangent>

```
event tick()
  float distance = 5.0 * time.get_gameplay_tick_interval()
  component_ref<transform> transform = entity.get().transform
  float last_x = transform.position.x
  transform.position.x = last_x + distance
end
```

```
static void tick(vvm_native_context_t* _context)
{
    float distance = 5.0f * time_get_gameplay_tick_interval(_context);
    vvm_value_t transform = entity_get_member(_context, entity_get(_context), "transform", 1126380853);
    float last_x = component_get_mutable_vec(_context, transform, "position", 5249027).x;
    component_set_mutable_vec_element(_context, transform, "position", 5249027, 0, last_x + distance);
}
```

# Let’s Build It!

- E s
- C ific data
- Events trigger component script code to execute

Entity/component memory layout is complicated.

# Let’s Build It!

- E s
- C ific data
- Events trigger component script code to execute
- Typically handle an event on multiple components – parallelize!
  - Ticking the world runs **tick** handler on all components that have one
  - Each component gets its own task or job
  - Ensure the result is deterministic

Entity/component memory layout is complicated.

# <tangent> Job System

- One job per component handling an event
- Job = { data blob, function pointer, completion token }
- Job system has 2 main operations:
  - Run_async – Queue job for execution
    - Returns completion token
  - Wait – Wait for a job to finish execution
    - Takes completion token

# Running a Job

- Run_async pushes the job onto a queue
- Scheduler runs when current job finishes or waits
  - Pop job off a queue
  - Give the job a fiber, if needed
    - Fiber = thread’s stack + registers
  - Switches to the job’s fiber
  - Loop until terminate

# Waiting On A Job

- The scenario: While Job A is running, Job B waits on it
- What happens:
  - Job B adds itself to Job A's wait list
  - Job B switches back to scheduler fiber
  - When Job A completes
    - Adds everyone in its wait list to the ready queue
    - Scheduler pulls B off the ready queue

# Job System </tangent>

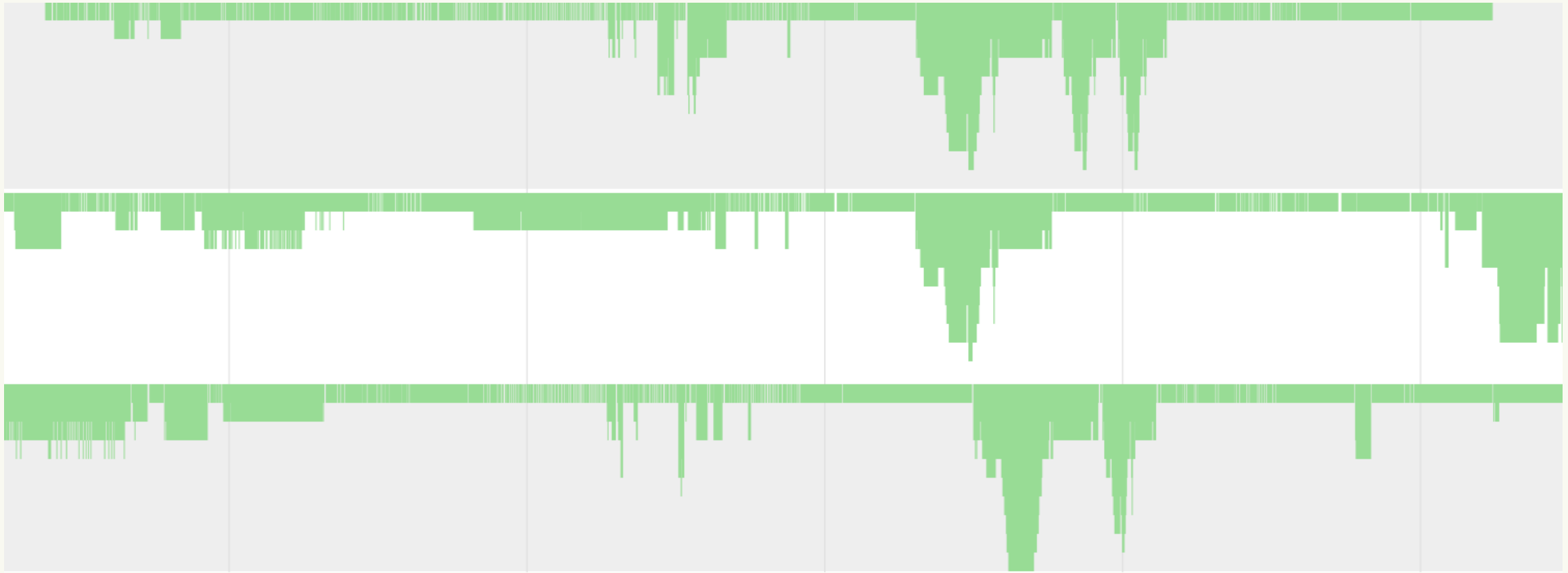
- Christian Gyrling's "Parallelizing the Naughty Dog Engine Using Fibers"
  - Our starting point
  - Added a few operations, one of which we will talk about later

# Let’s Build It!

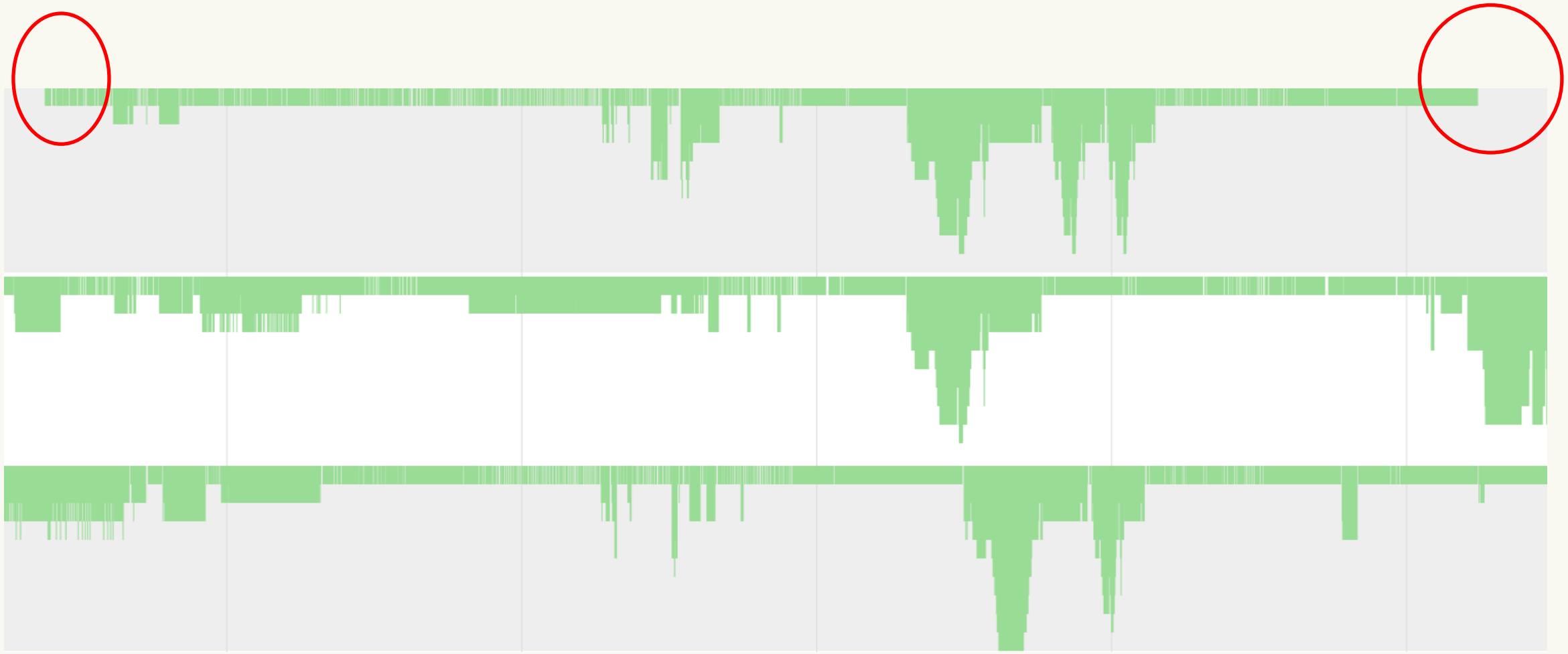
- E s
- C ific data
- Events trigger component script code to execute
- Typically handle an event on multiple components – parallelize!
  - Ticking the world runs **tick** handler on all components that have one
  - Each component gets its own task or job
  - Ensure the result is deterministic

Entity/component memory layout is complicated.

# Ta Da!



# Ta Da!



# So… We're Done?

# Nope.

# Let’s Look At That Example Script Again…

```
event tick()
  float distance = 5.0 * time.get_gameplay_tick_interval()
  component_ref<transform> transform = entity.get().transform
  float last_x = transform.position.x
  transform.position.x = last_x + distance
end
```

# Let's Look At That Example Script Again…

```
event tick()
  float distance = 5.0 * time.get_gameplay_tick_interval()
  component_ref<transform> transform = entity.get().transform
  float last_x = transform.position.x
  transform.position.x = last_x + distance
end
```

# Cross-Component Reads + Writes

- Reading from **transform**
  - What if **transform** has its own **tick** handler?
  - Will our component see **transform** before or after it mutates state?
  - Seems like we’ve got a race on our hands
- Writing to **transform**
  - Same thing differently
  - Will **transform** see its state before or after we mutate it?

# Solving the Read Problem

- Good news: we have job wait and a scripting language!
- When reading **transform**, script compiler adds a job wait
- We don’t run until **transform** is done
- Determinism achieved, data race gone

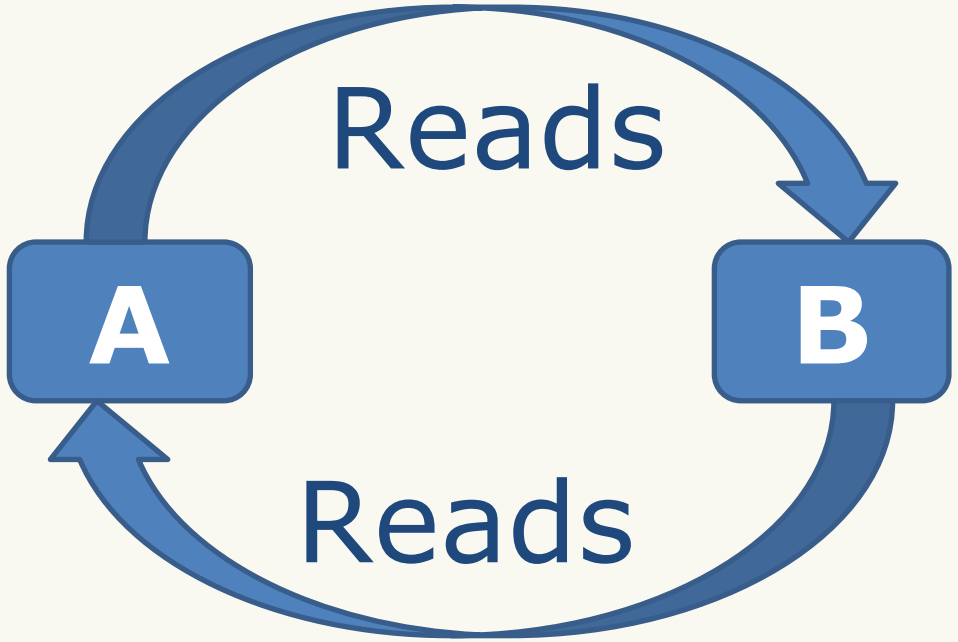
```
event tick()
  float distance = 5.0 * time.get_gameplay_tick_interval()
  component_ref<transform> transform = entity.get().transform
  // Implicit wait on transform here!
  float last_x = transform.position.x
  transform.position.x = last_x + distance
end
```

# That seems too easy.

# Yep.

# Cycles

- What's stopping A from reading B, and B reading A, in the same event? *Nothing.*
- What happens if that happens? *Deadlock.*



# The Synchronize Operation

- Introduce the **synchronize** operation
  - **synchronize(A,B)** means **A** “waits for” **B**
  - Invoke this operation on every cross-component read
- Without considering cycles, **synchronize** = **job_wait**
- With cycles, **synchronize** must avoid deadlock

# Synchronize Pseudocode

```
void synchronize(component_t* a, component_t* b) {
    if (!is_running(b)) {
        return;
    }

    while (true) {
        component_t* next_to_run = get_next_to_run(a, b);
        if (next_to_run == a) {
            break;
        }
        job_yield(a->job);
    }
}
```

# Job_Yield

- Recall **job_wait**
  - If **A** waits on **B**, **A** puts self on **B**’s wait list
  - **A** switches back to scheduler
  - When **B** completes, it pushes all its waiters on ready queue
- Yield operation is similar-ish
  - **A** flags self as yielding
  - **A** switches back to scheduler
  - Scheduler sees **A** is yielding and pushes **A** onto yield queue
  - When scheduler exhausts other work, pull from yield queue

# Job_Yield + Job_Wait?

- **Job_yield** spin cores on yield queue, but no deadlock
- **Job_wait** idle cores if there is no work, but can deadlock
- Can we combine the concepts and do better?

# Introducing Job_Yield_Wait

- **Job_yield_wait(A, B)**
  - **A** flags self as yield-waiting
  - **A** switches back to scheduler
  - Scheduler sees **A** is yield-waiting
    - Wakes all yield-waiters on **A**
    - Puts **A** in **B**’s yield-waiter list
  - When **B** completes, push all its yield-waiters onto ready queue
- Only wake jobs when progress is made!

# Now It's Perfect

```
void synchronize(component_t* a, component_t* b) {
    if (!is_running(b)) {
        return;
    }

    while (true) {
        component_t* next_to_run = get_next_to_run(a, b);
        if (next_to_run == a) {
            break;
        }
        job_yield_wait(a->job, b->job);
    }
}
```

# What's Next?

```
component_t* get_next_to_run(component_t* a, component_t* b) {
    if (!is_running(b)) {
        return a;
    }

    component_list_t cycle;
    push(&cycle, b);

    bool has_cycle = false;
    for (component_t* waiting_on = b->waiter; waiting_on; waiting_on = waiting_on->waiter) {
        if (contains(&cycle, waiting_on)) {
            has_cycle = true;
            break;
        }
        push(&cycle, waiting_on);
    }

    if (has_cycle) {
        return get_next_to_run_in_cycle(cycle);
    }
    return last(&cycle);
}
```

# What's Next?

```
component_t* get_next_to_run(component_t* a, component_t* b) {
    if (!is_running(b)) {
        return a;
    }

    component_list_t cycle;
    push(&cycle, b);

    bool has_cycle = false;
    for (component_t* waiting_on = b->waiter; waiting_on; waiting_on = waiting_on->waiter) {
        if (contains(&cycle, waiting_on)) {
            has_cycle = true;
            break;
        }
        push(&cycle, waiting_on);
    }

    if (has_cycle) {
        return get_next_to_run_in_cycle(cycle);
    }
    return last(&cycle);
}
```

# What's Next?

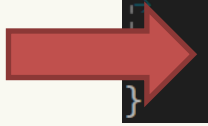
```
component_t* get_next_to_run(component_t* a, component_t* b) {
    if (!is_running(b)) {
        return a;
    }

    component_list_t cycle;
    push(&cycle, b);

    bool has_cycle = false;
    for (component_t* waiting_on = b->waiter; waiting_on; waiting_on = waiting_on->waiter) {
        if (contains(&cycle, waiting_on)) {
            has_cycle = true;
            break;
        }
        push(&cycle, waiting_on);
    }

    if (has_cycle) {
        return get_next_to_run_in_cycle(cycle);
    }
    return last(&cycle);
}
```

# What's Next?

```
component_t* get_next_to_run(component_t* a, component_t* b) {
    if (!is_running(b)) {
        return a;
    }

    component_list_t cycle;
    push(&cycle, b);

    bool has_cycle = false;
    for (component_t* waiting_on = b->waiter; waiting_on; waiting_on = waiting_on->waiter) {
        if (contains(&cycle, waiting_on)) {
            has_cycle = true;
            break;
        }
        push(&cycle, waiting_on);
    }

    if (has_cycle) {
        return get_next_to_run_in_cycle(cycle);
    }
    return last(&cycle);
}
```

# What's Next?

```
component_t* get_next_to_run(component_t* a, component_t* b) {
    if (!is_running(b)) {
        return a;
    }

    component_list_t cycle;
    push(&cycle, b);

    bool has_cycle = false;
    for (component_t* waiting_on = b->waiter; waiting_on; waiting_on = waiting_on->waiter) {
        if (contains(&cycle, waiting_on)) {
            has_cycle = true;
            break;
        }
        push(&cycle, waiting_on);
    }

    if (has_cycle) {
        return get_next_to_run_in_cycle(cycle);
    }
    return last(&cycle);
}
```



# Selecting What To Run From A Cycle

- Want determinism
- No other criteria? Just pick one already!
- Our system compares:
  - Entity GUIDs, then
  - Component index on entity

# Synchronization Costs in Practice

- On a “typical” KO City simulation step with 4 job workers
  - 280 synchronizes where we check for a cycle
  - 75 cycles broken
- ~100 ns overhead per job switch on Xbox One X
  - Time to pop job from queue, switch to job, back to scheduler
- More waiting/yielding jobs = more fibers
  - 64kb / fiber for “normal” jobs
  - For KOCity: 512 normal job fibers preallocated = 32 MB

# Ok, Now We’re Done.

# With Read.

# What About Write?

```
event tick()
  float distance = 5.0 * time.get_gameplay_tick_interval()
  component_ref<transform> transform = entity.get().transform
  float last_x = transform.position.x
  transform.position.x = last_x + distance
end
```

# Making Write Safe

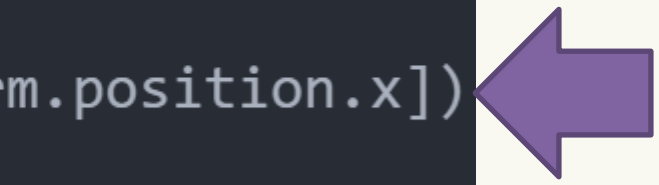
- Defer cross-component writes
  - A does not write to B immediately
  - A queues write on B to happen at the end of the current event
  - Now A writing to B does not race with B (or other readers of B)
- Good news: we have a scripting language!
  - We can transparently defer writes

# Are We There Yet?

```
event tick()
  float distance = 5.0 * time.get_gameplay_tick_interval()
  component_ref<transform> transform = entity.get().transform
  float last_x = transform.position.x
  transform.position.x = last_x + distance
end
```

# Are We There Yet?

```
event tick()
  float distance = 5.0 * time.get_gameplay_tick_interval()
  component_ref<transform> transform = entity.get().transform
  float last_x = transform.position.x
  transform.position.x = last_x + distance
  debug.print_line(text: "new x={}", args:[transform.position.x])
end
```



# Snooping Your Write Queue

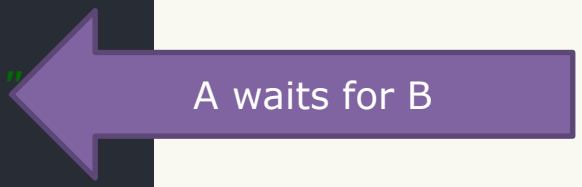
- When deferring a write to another component, record it
- Before we read another component, check pending writes
  - Pending write list is component-job-local
  - “Typical” frame average pending write list has ~1.2 items
  - Just do a linear search
- Return the last pending write, if any

# Summary of Read/Write Semantics

```
// Script A
event tick()
  debug.print_line(text: "A {}", args: [B.value]) // Prints "A 3"
end

// Script B
event tick()
  value = 2
  debug.print_line(text: "B {}", args: [value]) // Prints "B 2"
  value = 3
end

// Script C
event tick()
  B.value = 1
  debug.print_line(text: "C {}", args: [B.value]) // Prints "C 1"
end
```

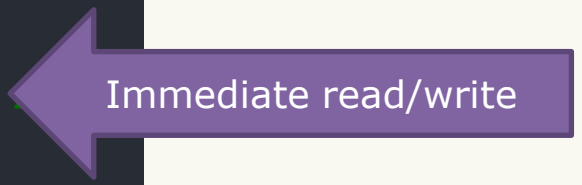
# Summary of Read/Write Semantics

```
// Script A
event tick()
  debug.print_line(text: "A {}", args: [B.value]) // Prints "
end

// Script B
event tick()
  value = 2
  debug.print_line(text: "B {}", args: [value]) // Prints "B 2"
  value = 3
end

// Script C
event tick()
  B.value = 1
  debug.print_line(text: "C {}", args: [B.value]) // Prints "C 1"
end
```



# Summary of Read/Write Semantics

```
// Script A
event tick()
  debug.print_line(text: "A {}", args: [B.value]) // Prints "A 3"
end

// Script B
event tick()
  value = 2
  debug.print_line(text: "B {}", args: [value]) // Prints "B 
  value = 3
end

// Script C
event tick()
  B.value = 1
  debug.print_line(text: "C {}", args: [B.value]) // Prints "C 1"
end
```



# Summary of Read/Write Semantics

```
// Script A
event tick()
  debug.print_line(text: "A {}", args: [B.value]) // Prints "A 3"
end

// Script B
event tick()
  value = 2
  debug.print_line(text: "B {}", args: [value]) // Prints "B 2"
  value = 3
end

// Script C
event tick()
  B.value = 1
  debug.print_line(text: "C {}", args: [B.value]) // Prints "C
end
```

C defers write & snoops

# Surely There's More?

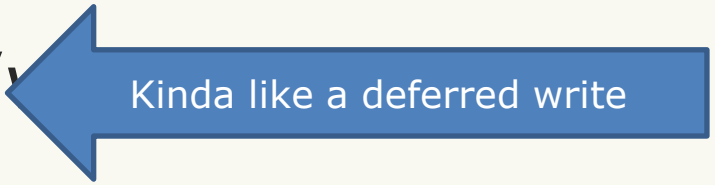
# Yep.

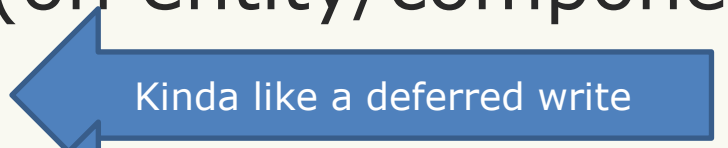
# What Other Operations Do We Want?

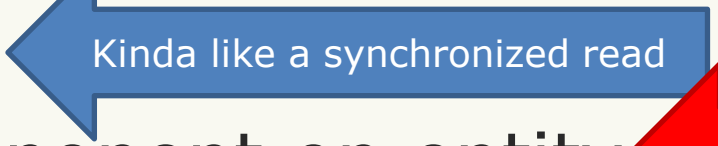
- Spawn an entity
- Destroy an entity
- Queue an event (on entity/component/world)
- Set entity parent
- Get entity sibling
- Add/remove component on entity

# What Other Operations Do We Want?

- Spawn an entity
- Destroy an entity
- Queue an event (on entity/component/
- Set entity parent
- Get entity sibling
- Add/remove component on entity

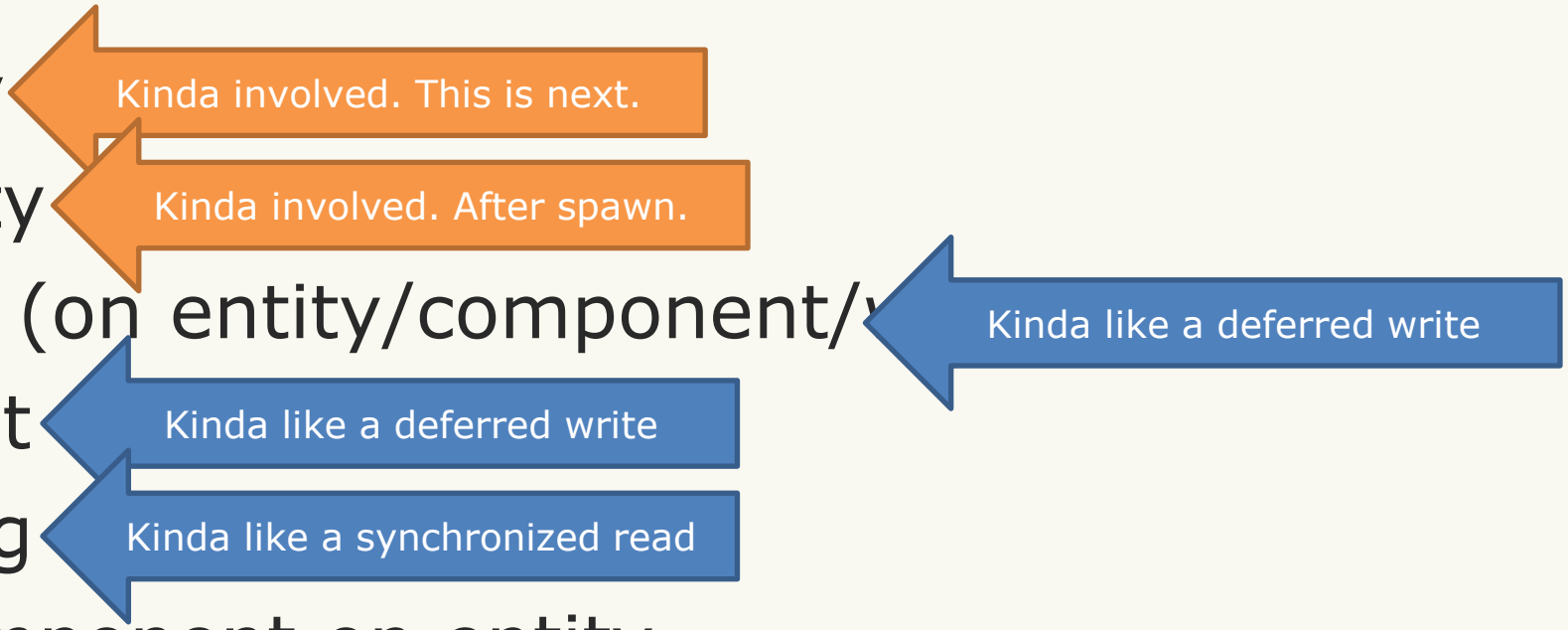



# What Other Operations Do We Want?

- Spawn an entity
- Destroy an entity
- Queue an event (on entity/component/
- Set entity parent
- Get entity sibling
- Add/remove component on entity



# Spawning Entities

- Split spawn operation into 2 steps
  - Allocate space + Initialize values
  - Run **created** event
- Do alloc+init immediately
  - Need address for new entity
- Defer the **created** event
  - Handle **created** in parallel for all components in new entity
  - Resolves create time ordering issues between components

# Spawn In Action

```
entity_ref child = entity.spawn(entity: prototype, parent: entity.get())
// Child is initialized to default values here
child.transform.position.x = 42.0
// Child's transform's position is now modified
```

# Spawn In Action

```
entity_ref child = entity.spawn(entity: prototype, parent: entity.get())
// Child is initialized to default values here
child.transform.position.x = 42.0
// Child's transform's position is now modified
```

But there’s a problem. Do you see it?

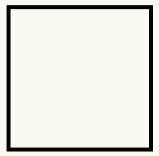
# Entity Visibility

- If component A spawns entity E, when can B see E?
- If B sees E immediately -> nondeterministic
  - For B, visibility to E must be deferred until after **created**
- But A needs to see E immediately to reference it

# Entity Visibility Solution

- When allocating an entity, record the creating component
- All entity query API takes the *calling component*
- Query API is safe to return an entity/component if:
  - Entity/component is fully created, OR,
  - Caller is the creator
- Query API returns entities/components visible to caller

Spawn entity



Destroy entity

# Destroying Entities

- You know the drill
  - Can’t just destroy entities
  - That would be racy and nondeterministic
- Defer the destroy
- Split the operation in two parts
  - Queue destroyed event (to run in parallel)
  - Deallocate the storage

# There’s A Catch, Right?

# Destroying Without Creating?

- Spawn entity in the normal way
  - Created event queued for end of simulation step
- Then destroy entity, flagged to run after current event
- Do we get a **destroyed** event without a **created**?
  - If resource management tied to created/destroyed -> bad
  - Also, just not a cool thing to do

# Destroyed Guarantees Created

- Before processing destroyed event, check if created
- If not, find created events in queue and run them
  - Probably need to check the “other” queue
- Then run destroyed events

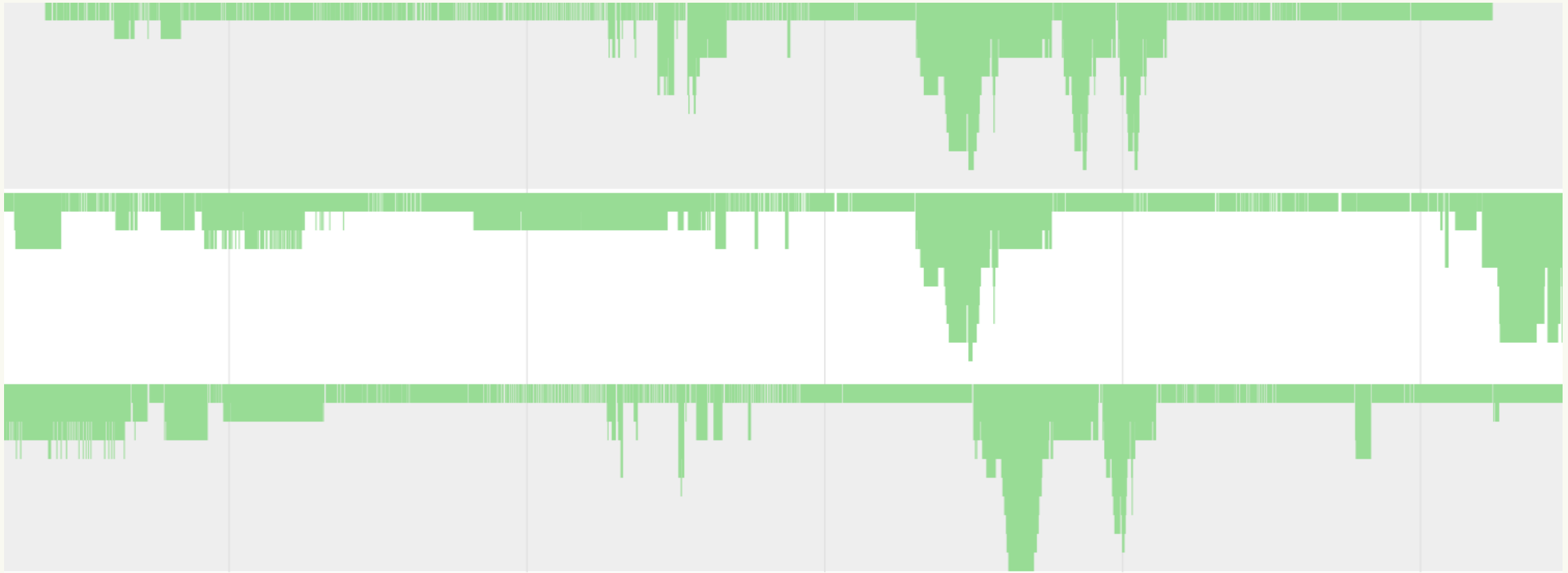
Spawn entity

Destroy entity

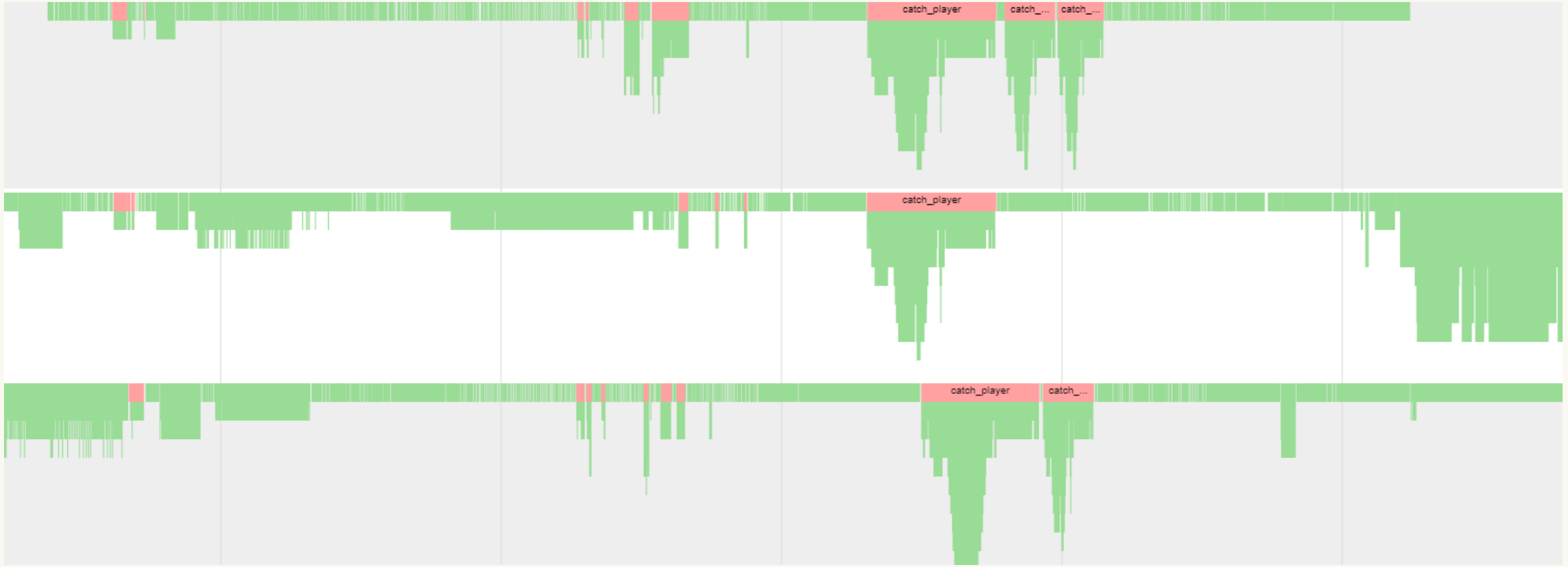
# That’s It?

- We have a workable, parallel, deterministic system
- But it could be faster
- That’s next

# Recall This



# Enhance!



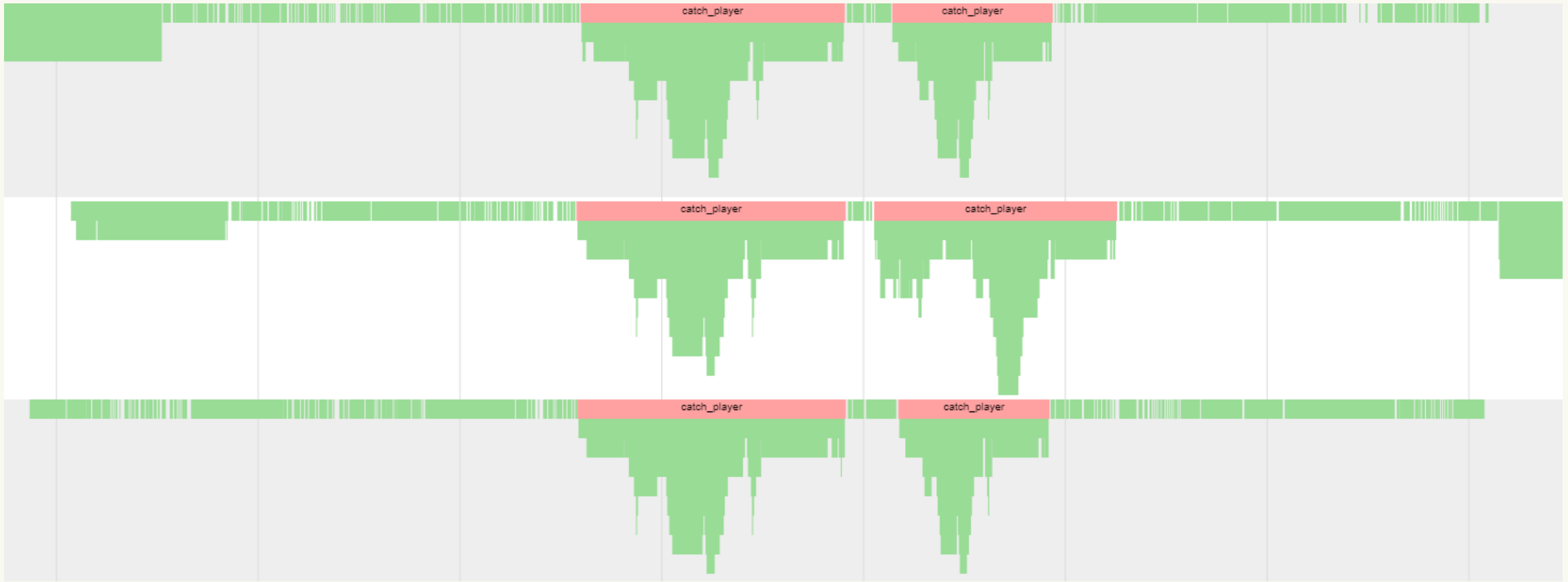
# Observation

- Cross-component reads often have per-type relationship
- Example: move-5-units-per-second-in-X components read **transform** components
- To minimize yields, tick our component *after* **transform**

# Formalizing Our Observation

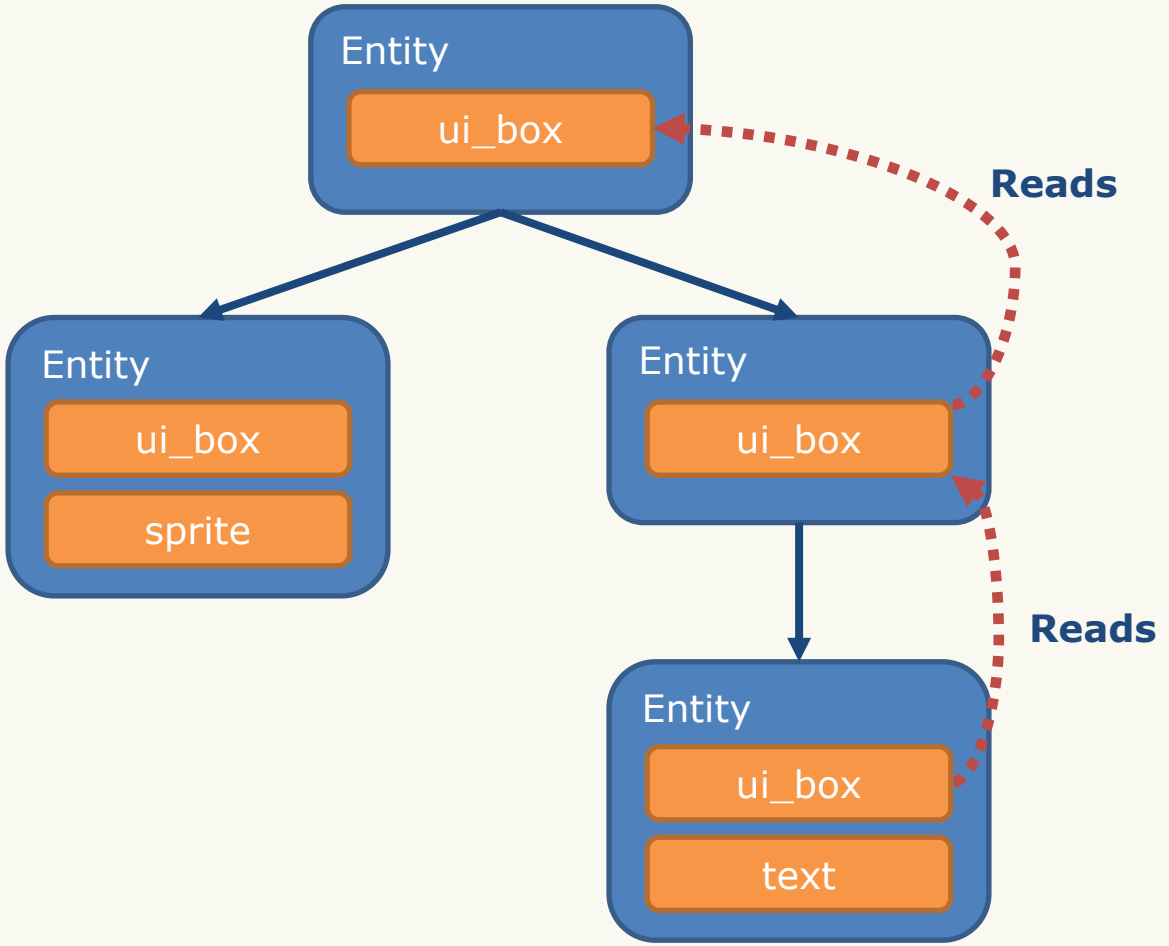
- Let **wait_order** be an integer
  - On each component *type*
  - For each major event
- Top of each core event, sort types by **wait_order**
- Queue component jobs in sorted order
- When **synchronize(A,B)** and B is active
  - Increment **wait_order** for **typeof(A)** and the current event
  - Decrement **wait_order** for **typeof(B)** and the current event
  - Clamp **wait_order** to a reasonable range

# After Enabling Job Ordering



# Another Observation

- Tracking read relationships between types is great
- But what about reads *within* a type?
- Pathological case: same type components that read up the entity tree



# Handling Hierarchies of Reads

- Script specifies “self sort order” –> depth in hierarchy
- Sort components within a type by self sort order
- Components higher up the tree run earlier

# A Third Observation

- Sometimes components of different types work as a unit
  - Example: camera behavior components working as a unit
  - Synchronizing reads is slow
  - Deferring writes leads to latency
- Introduce notion of component “control”
  - Every component has a controller
  - Default controller is self
  - If A controls B:
    - A receives events, B does not
    - Reads and writes are immediate

# A Fourth Observation

- Large quantities of simple components
  - Might read another component, but not part of read cycles
  - Do a single thing
  - Example: component that plays a visual effect
- Queuing a job for each vfx component is wasteful
- Batch up multiple vfx components to run in a single job

# A Fifth (and Final) Observation

- Do we have to snapshot world state each step?
- Maybe not:
  - If we don’t use all snapshots
  - If we know which snapshots we will likely use
- On Knockout City:
  - If latency is relatively stable, we can make good guesses
- Only store snapshots we are likely to use:
  - Remember that 377kb of snapshot state?
  - Save memcpy time!

Build Entity System

Make It Run Faster

# Debugging Tools

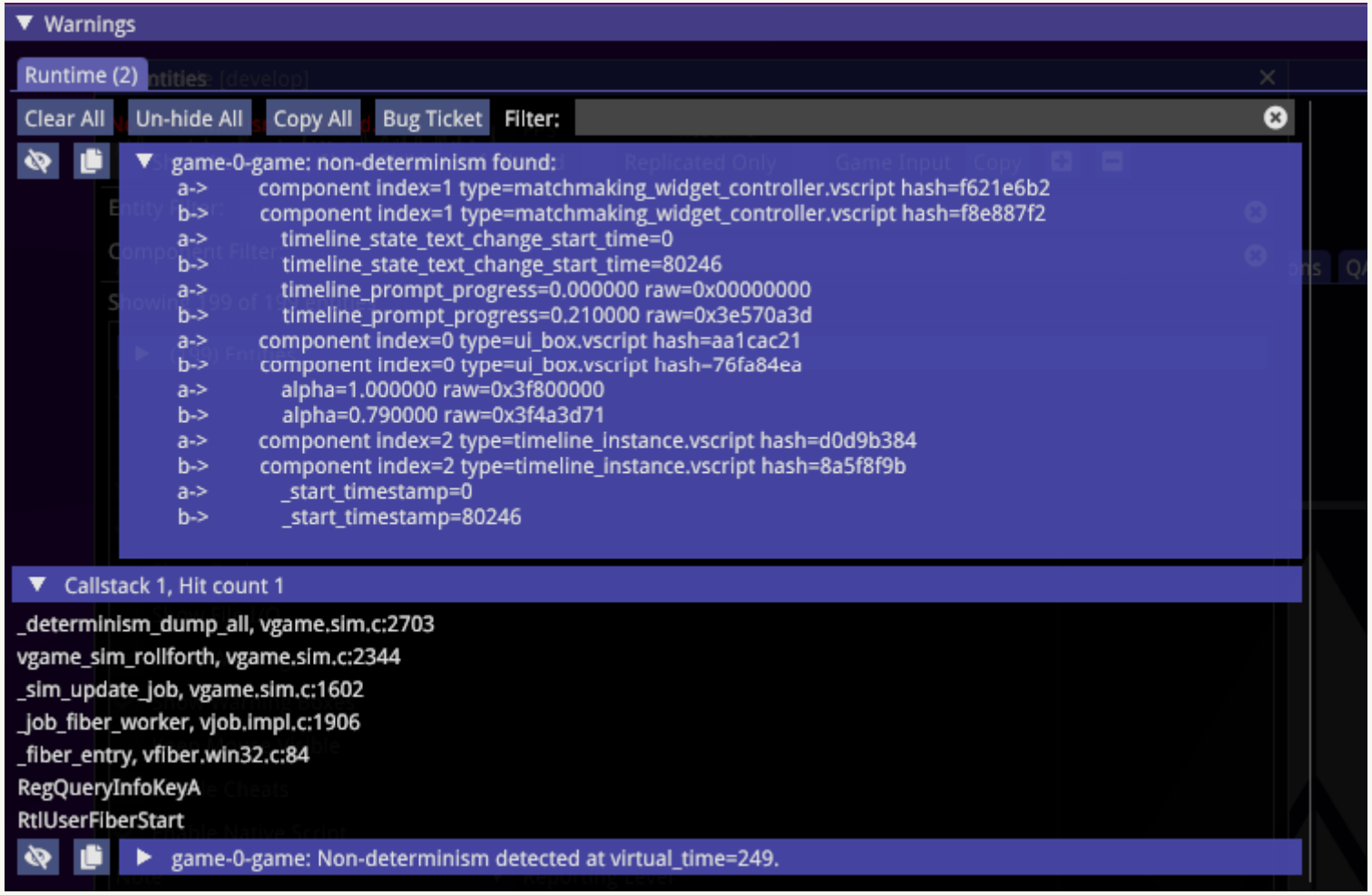
Some useful stuff to help validate and debug issues:

- Snapshot comparison
- Entity view
- Script memory view
- Tracing system
- Lots of unit tests

# Snapshot Comparison

- Run every step 2x
  - Run step
  - Rollback
  - Run step again
- Hash and record all mutable component state
- Assert hashes are equal at end of second run
  - If not, report differing state between two runs

# Determinism Fail



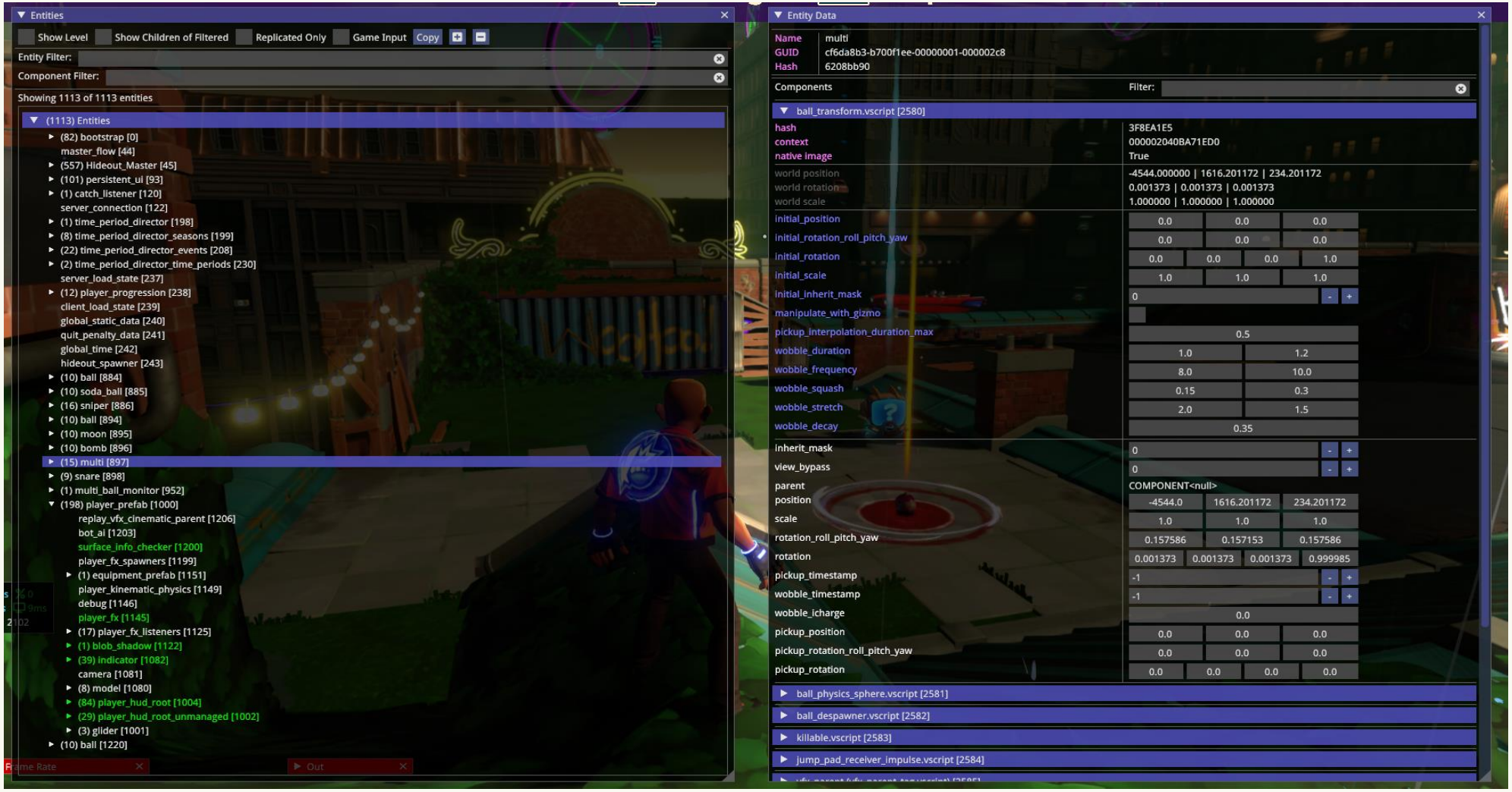
# Snapshot Comparison Part 2

- Store entities and components in contiguous arrays
  - State of entity/components matters
  - Location in array does not matter
  - Use XOR to combine order-independent hashes
- Floating point is hard
  - No floating point determinism across machine architectures
- Allow certain scripts to opt out of snapshot comparison
  - Care about determinism in moment-to-moment gameplay
  - Many situations we do not care

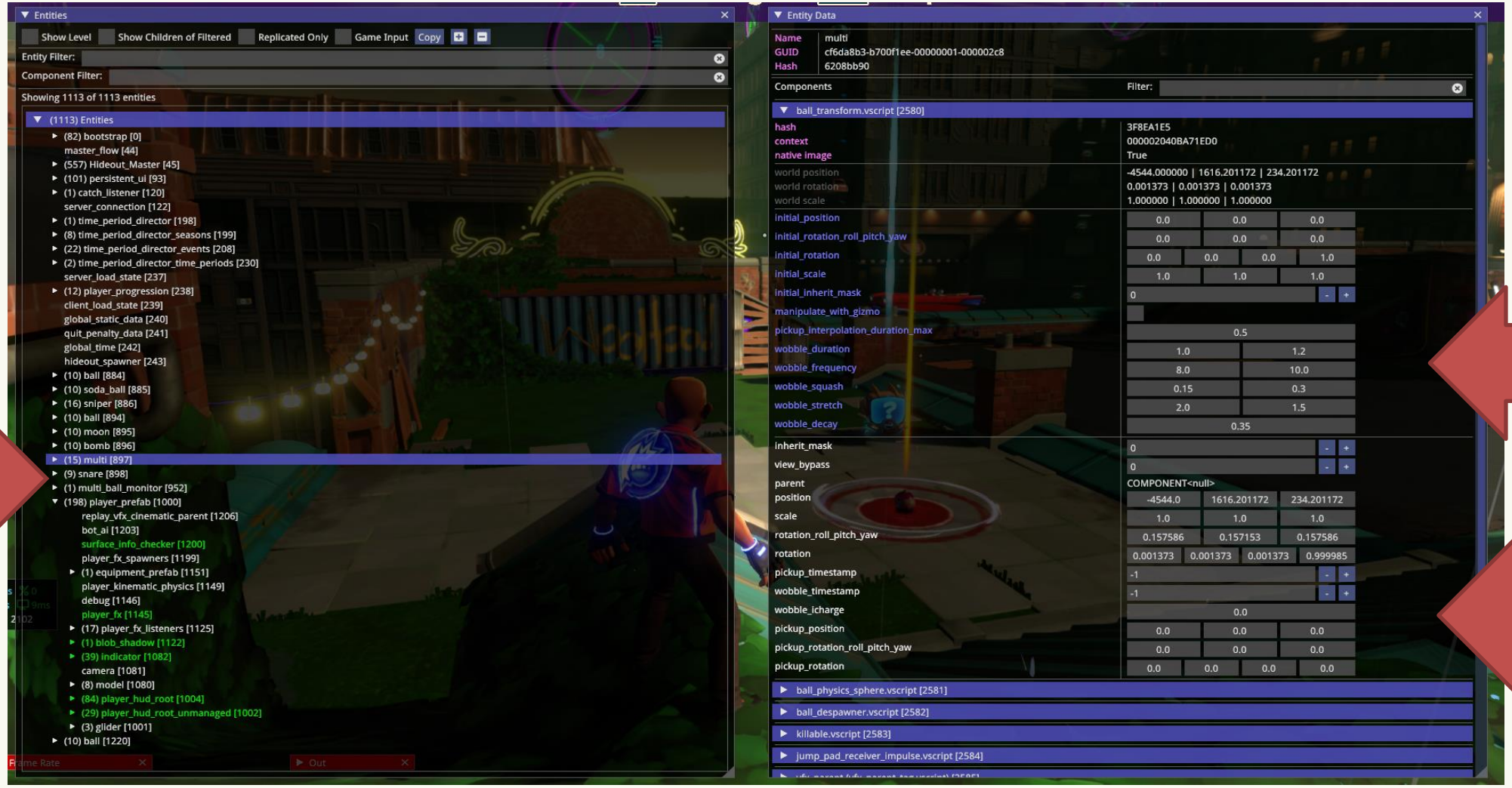
# Entity View Tool

- Primary script debugging tool (with debug.print_line)
- Quickly answer questions like:
  - Is an entity spawned?
  - What’s the mutable state of a component?
  - Who is this entity’s parent entity?
  - Which spawned entities have some component?
- Let a developer perform operations like:
  - Kill an entity
  - Override the state of a component

# Entity View Tool



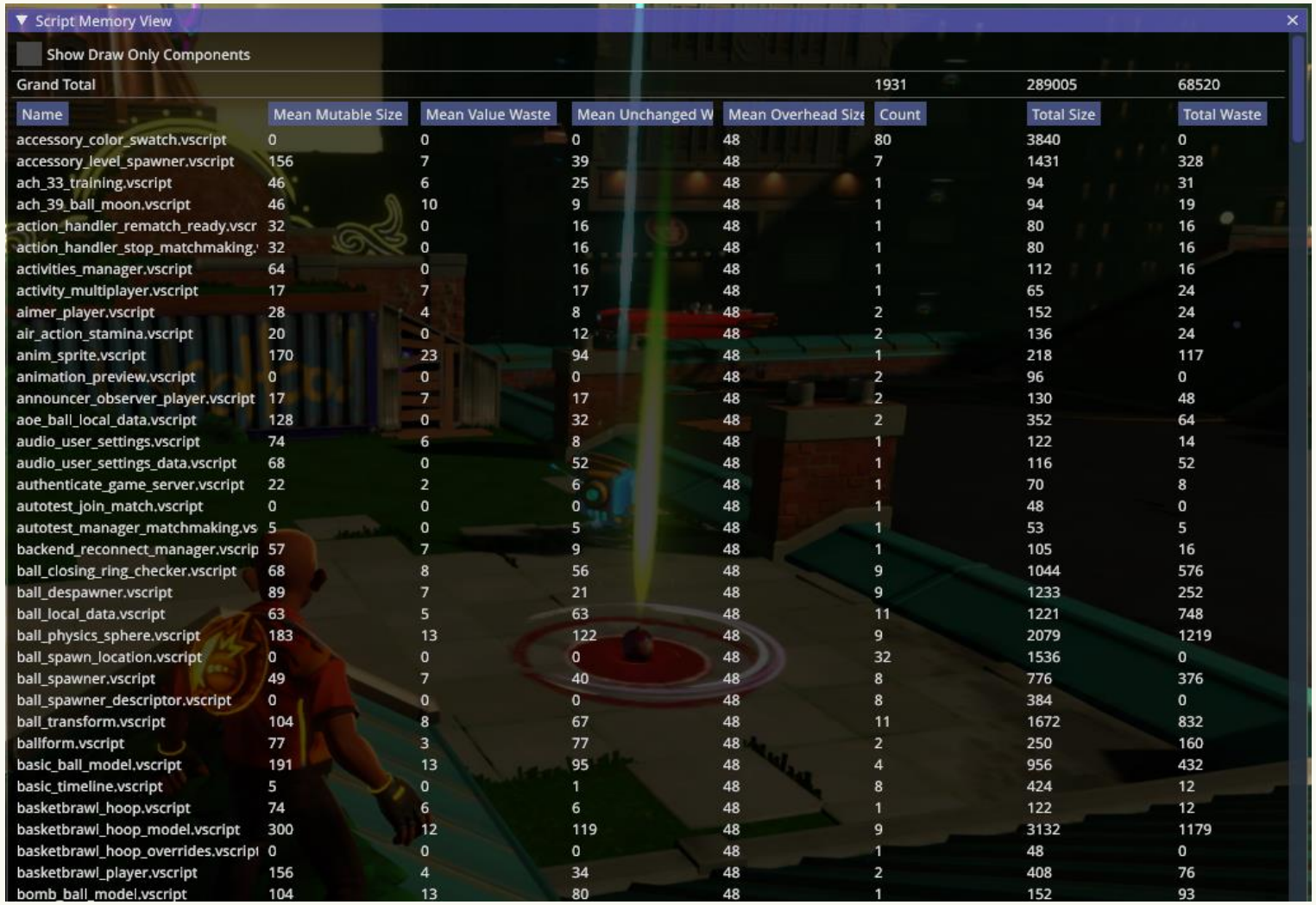
# Entity View Tool



# Script Memory View

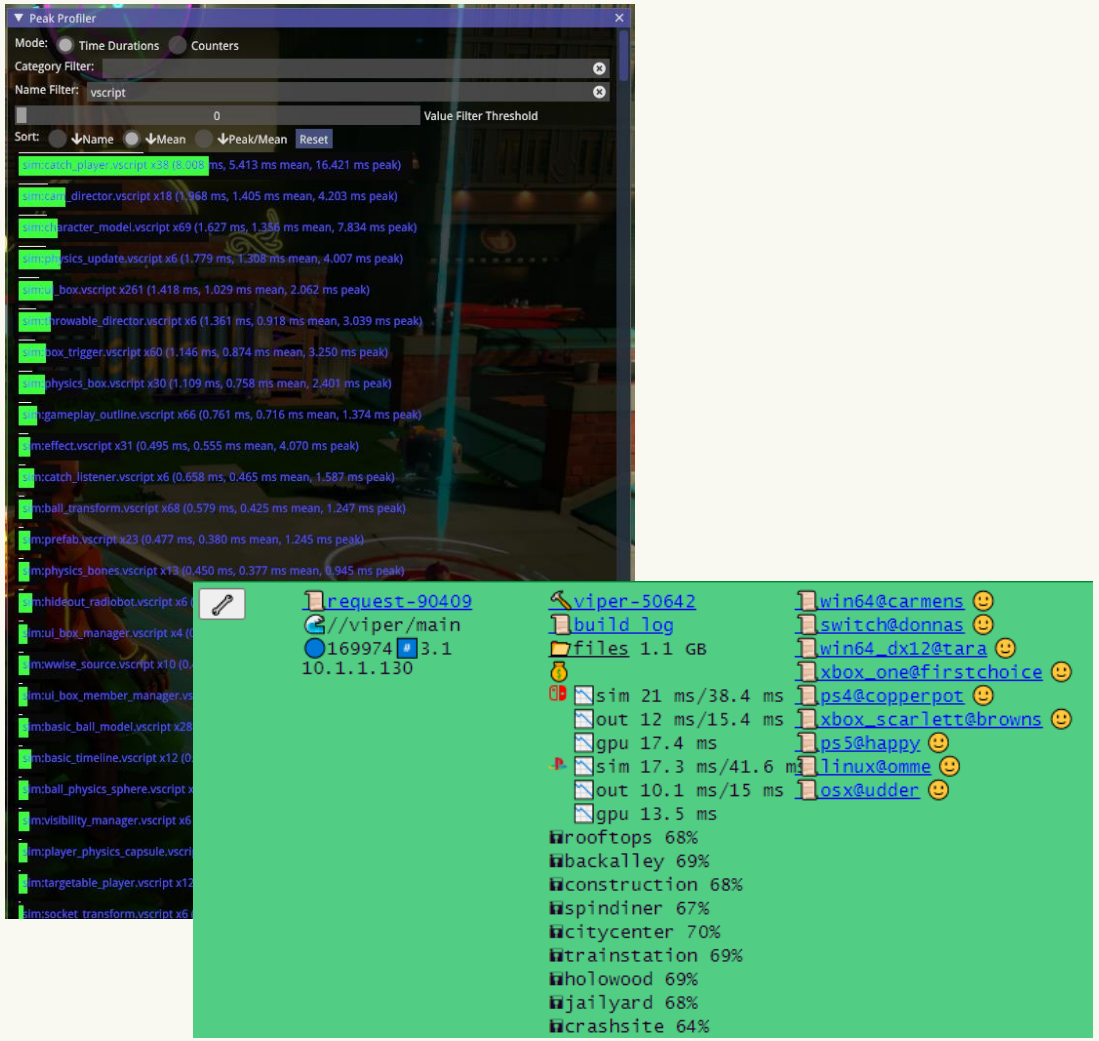
- Mutable script state gets copied forward each step
- Script memory view answers questions like:
  - How many components of some type are currently spawned?
  - How much mutable state is on some type?
  - What is the total amount of mutable state in the world now?

# Script Memory View




| Grand Total | | | | | 1931 | 289005 | 68520 |
|---|---|---|---|---|---|---|---|
| Name | Mean Mutable Size | Mean Value Waste | Mean Unchanged W | Mean Overhead Size | Count | Total Size | Total Waste |
| accessory_color_swatch.vscript | 0 | 0 | 0 | 48 | 80 | 3840 | 0 |
| accessory_level_spawner.vscript | 156 | 7 | 39 | 48 | 7 | 1431 | 328 |
| ach_33_training.vscript | 46 | 6 | 25 | 48 | 1 | 94 | 31 |
| ach_39_ball_moon.vscript | 46 | 10 | 9 | 48 | 1 | 94 | 19 |
| action_handler_rematch_ready.vscr | 32 | 0 | 16 | 48 | 1 | 80 | 16 |
| action_handler_stop_matchmaking.' | 32 | 0 | 16 | 48 | 1 | 80 | 16 |
| activities_manager.vscript | 64 | 0 | 16 | 48 | 1 | 112 | 16 |
| activity_multiplayer.vscript | 17 | 7 | 17 | 48 | 1 | 65 | 24 |
| aimer_player.vscript | 28 | 4 | 8 | 48 | 2 | 152 | 24 |
| air_action_stamina.vscript | 20 | 0 | 12 | 48 | 2 | 136 | 24 |
| anim_sprite.vscript | 170 | 23 | 94 | 48 | 1 | 218 | 117 |
| animation_preview.vscript | 0 | 0 | 0 | 48 | 2 | 96 | 0 |
| announcer_observer_player.vscript | 17 | 7 | 17 | 48 | 2 | 130 | 48 |
| aoe_ball_local_data.vscript | 128 | 0 | 32 | 48 | 2 | 352 | 64 |
| audio_user_settings.vscript | 74 | 6 | 8 | 48 | 1 | 122 | 14 |
| audio_user_settings_data.vscript | 68 | 0 | 52 | 48 | 1 | 116 | 52 |
| authenticate_game_server.vscript | 22 | 2 | 6 | 48 | 1 | 70 | 8 |
| autotest_join_match.vscript | 0 | 0 | 0 | 48 | 1 | 48 | 0 |
| autotest_manager_matchmaking.vs | 5 | 0 | 5 | 48 | 1 | 53 | 5 |
| backend_reconnect_manager.vscrip | 57 | 7 | 9 | 48 | 1 | 105 | 16 |
| ball_closing_ring_checker.vscript | 68 | 8 | 56 | 48 | 9 | 1044 | 576 |
| ball_despawner.vscript | 89 | 7 | 21 | 48 | 9 | 1233 | 252 |
| ball_local_data.vscript | 63 | 5 | 63 | 48 | 11 | 1221 | 748 |
| ball_physics_sphere.vscript | 183 | 13 | 122 | 48 | 9 | 2079 | 1219 |
| ball_spawn_location.vscript | 0 | 0 | 0 | 48 | 32 | 1536 | 0 |
| ball_spawner.vscript | 49 | 7 | 40 | 48 | 8 | 776 | 376 |
| ball_spawner_descriptor.vscript | 0 | 0 | 0 | 48 | 8 | 384 | 0 |
| ball_transform.vscript | 104 | 8 | 67 | 48 | 11 | 1672 | 832 |
| ballform.vscript | 77 | 3 | 77 | 48 | 2 | 250 | 160 |
| basic_ball_model.vscript | 191 | 13 | 95 | 48 | 4 | 956 | 432 |
| basic_timeline.vscript | 5 | 0 | 1 | 48 | 8 | 424 | 12 |
| basketbrawl_hoop.vscript | 74 | 6 | 6 | 48 | 1 | 122 | 12 |
| basketbrawl_hoop_model.vscript | 300 | 12 | 119 | 48 | 9 | 3132 | 1179 |
| basketbrawl_hoop_overrides.vscript | 0 | 0 | 0 | 48 | 1 | 48 | 0 |
| basketbrawl_player.vscript | 156 | 4 | 34 | 48 | 2 | 408 | 76 |
| bomb_ball_model.vscript | 104 | 13 | 80 | 48 | 1 | 152 | 93 |

# Tracing System




- Track performance
- Track deferred actions
- Track read sync count
- Stream out CSV
- Dump Chrome trace file
- Connect to CI system to record performance each build

# Unit Tests

- Anticipate bugs: Write unit tests for interesting cases
- React to bugs: Whenever there is a bug, write a unit test to expose it
- We have several hundred small unit tests

# Unit Test Example

```
// First test script:
event created()
  component_ref other = entity.get().other_component
  entity.queue_event_on_component(component: other, name: "assert", args: [0])
  other.state = 1
  entity.queue_event_on_component(component: other, name: "assert", args: [1])
end

// Second test script (other_component):
int state = 0
event assert(int value)
  debug.assert_equal(a: state, b: value)
end
```

# Zooming Out

- Started from zero
- Four years later:
  - Built this (and the rest of an engine/toolchain)
  - Shipped Mario Kart Live and Knockout City
  - ~1 programmer working on entities / script for the entire time
- Code written:
  - Job System = ~2 kLOC
  - Script Compiler/Runtime = ~28 kLOC
  - Entity System = ~10 kLOC

# High Level Performance Data

On a typical client…

- Steps the simulation ~6 times per frame
- ~4000 spawned components
- ~250 deferred actions per step
- ~250 synchronizes that do work per step
- ~1500 job switches per step
- ~150us job overhead per step on Xbox One X

# Some Closing Thoughts

- The thing we set out to do is possible
  - ✓ Parallel
  - ✓ Deterministic
  - ✓ Rewindable

# Some Closing Thoughts

- The thing we set out to do is possible
  - ✓ Parallel
  - ✓ Deterministic
  - ✓ Rewindable
- Synchronizing components is a nice tool to have

# Some Closing Thoughts

- The thing we set out to do is possible
  - ✓ Parallel
  - ✓ Deterministic
  - ✓ Rewindable
- Synchronizing components is a nice tool to have
- Support for determinism & rewind goes beyond entities

# Some Closing Thoughts

- The thing we set out to do is possible
  - ✓ Parallel
  - ✓ Deterministic
  - ✓ Rewindable
- Synchronizing components is a nice tool to have
- Support for determinism & rewind goes beyond entities
- Language has a big impact; choosing to create a new language is big

# If I Had To Do It Over Again…

- Support cross-component reads of last step's data
  - Probably good enough much of the time?
  - Avoid synchronization
- Stronger performance focus earlier
  - Parallelism is nice but if my data access patterns suck…
- More flexibility around when we snapshot state
- More flexibility around when a component ticks

# Future Work

- Better support for components not ticking every frame
- Implement more engine systems in terms of components
- Writing a script debugger

# Thanks

- Cory and Joe for foundational work.
- Anton and Patrick for ongoing support and optimization.
- Matt and Neil from Alphablit for their assistance.
- Everyone at Velan for putting up with growing pains.
- Andreas for the feedback.
- Eli and Jenica for everything.

# I Hope We Have Time For Questions

- Velan Studios is hiring!
- https://www.velanstudios.com/careers/

# Bonus Content

# Bonus Content

- Scripting notes
- Faster event dispatch
- Types of entities

# Scripting Notes

- Tour of a simple script
- Interpreted script
- Resource management in script

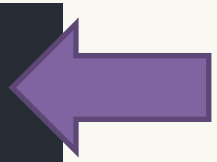
# Tour of a Simple Script

```
// Maximum hit points.
// Set at level editor time. Read-only at runtime.
const int max_health = 100

// Current hit points. Read-write runtime.
int health

// Fired by engine on spawn to all spawned components.
event created()
  health = max_health
end

// Custom event fired by game logic.
event get_hit(int damage)
  health = math.max(value: health - damage, min: 0)
  if (health == 0)
    do_death()
  end
end

// Function that runs when no hit points remain.
function do_death()
  // ...
end
```

A simple script for managing the health of an entity.

# Tour of a Simple Script

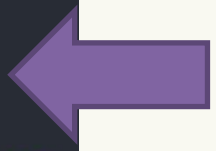
```
// Maximum hit points.
// Set at level editor time. Read-only at runtime.
const int max_health = 100

// Current hit points. Read-write runtime.
int health

// Fired by engine on spawn to all spawned components.
event created()
  health = max_health
end

// Custom event fired by game logic.
event get_hit(int damage)
  health = math.max(value: health - damage, min: 0)
  if (health == 0)
    do_death()
  end
end

// Function that runs when no hit points remain.
function do_death()
  // ...
end
```



"Constants" set at editor time and read-only a runtime.
Component properties that you see in the level editor.

# Tour of a Simple Script

```
// Maximum hit points.
// Set at level editor time. Read-only at runtime.
const int max_health = 100

// Current hit points. Read-write runtime.
int health

// Fired by engine on spawn to all spawned components.
event created()
  health = max_health
end

// Custom event fired by game logic.
event get_hit(int damage)
  health = math.max(value: health - damage, min: 0)
  if (health == 0)
    do_death()
  end
end

// Function that runs when no hit points remain.
function do_death()
  // ...
end
```



"Mutables" are read/write at runtime.
This is the data that forms the component snapshot.
If network-replicated, this is what goes on the wire.

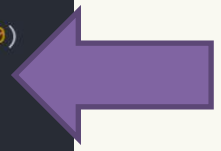
# Tour of a Simple Script

```
// Maximum hit points.
// Set at level editor time. Read-only at runtime.
const int max_health = 100

// Current hit points. Read-write runtime.
int health

// Fired by engine on spawn to all spawned components.
event created()
  health = max_health
end

// Custom event fired by game logic.
event get_hit(int damage)
  health = math.max(value: health - damage, min: 0)
  if (health == 0)
    do_death()
  end
end

// Function that runs when no hit points remain.
function do_death()
  // ...
end
```

An event handler for the created event.

# Tour of a Simple Script

```
// Maximum hit points.
// Set at level editor time. Read-only at runtime.
const int max_health = 100

// Current hit points. Read-write runtime.
int health

// Fired by engine on spawn to all spawned components.
event created()
  health = max_health
end

// Custom event fired by game logic.
event get_hit(int damage)
  health = math.max(value: health - damage, min: 0)
  if (health == 0)
    do_death()
  end
end

// Function that runs when no hit points remain.
function do_death()
  // ...
end
```



An event handler for get_hit event. Presumably sent from some other component.

# Tour of a Simple Script

```
// Maximum hit points.
// Set at level editor time. Read-only at runtime.
const int max_health = 100

// Current hit points. Read-write runtime.
int health

// Fired by engine on spawn to all spawned components.
event created()
  health = max_health
end

// Custom event fired by game logic.
event get_hit(int damage)
  health = math.max(value: health - damage, min: 0)
  if (health == 0)
    do_death()
  end
end

// Function that runs when no hit points remain.
function do_death()
  // ...
end
```

Script can be broken up into functions.

# Interpreted Script

- Stack-based virtual machine with only 14 operations
- Most commonly the VM does this:
  - Push 1 or more values onto stack
  - Call a function defined in C exposed to script
    - Pops values off the stack
    - Does some work (in C)
    - Pushes a new value onto the stack
  - Maybe do an (un)conditional jump
- VM at least 5x slower than script compiled to C
- Useful in development (hot reload)
- Useful in a live environment (small hotfixes)

# Resource Management in Script

- Goal:
  - Avoid lifetime management of external resources in script
- Why:
  - Easy to cause resource leaks
  - Difficult to do in a system that can rewind
- Solution:
  - Expose immediate mode APIs to script for interacting with external resources

# Resource Management in Script

```
event tick()
  vec3 pos = entity.get_world_position()
  physics.box(position: pos)
end
```

- Physics Box Example
  - Avoid create, update, destroy
  - On first frame we see this component declare the box, create it
  - On later frames, update the box
  - When the component stops ticking, destroy the box
- Implementation
  - Requires caching behind the scenes
  - Script authoring is easier
  - Exposing resources to script is harder

# Faster Event Dispatch

- Not all component types handle all events
  - Efficiently know which components handle which events
- Usually* queue event handling jobs in **wait_order** order
  - **Wait_order** is a property of the component type
- Solution
  - Maintain a list of component indices for each component type
  - For core events, maintain a **wait_order** sorted list of component index lists

# Faster Event Dispatch

```
void trigger_event_on_world(int event_index) {
  for (short* b = buckets_by_wait_order[event_index]; *b >= 0; ++b) {
    for (short* c = component_type_buckets[*b]; *c >= 0; ++c) {
      short component_index = *c;
      if (!event_disabled_for_component(component_index)) {
        // Queue job to run event handler.
      }
    }
  }
}
```

# Faster Event Dispatch

- Whenever components are spawned or destroyed, mark the associated component type lists as pending update
- Flush pending updates before processing the next event
- Spawn/destroy are “relatively” rare, maintaining lists is worth the cost

# Types of Entities

- Spent a lot of time talking about types of components
- Are their types of entities? Sort of.
- Motivated by a desire to run faster…

# Types of Entities

- **Logic entities**
  - Have snapshotted component state that can rewind
  - Example: gameplay entities
- **Draw-only entities**
  - Portions of component state are not snapshotted, no rewind
  - Only handles core engine events (e.g. tick) on the leading edge
  - Example: special effects entities
- **Static entities**
  - Not really entities at all
  - At spawn time get baked into faster data structures
  - Component script does not run on static entities
  - Example: map geometry

# Types of Entities

- Logic and draw-only entities can interact in defined ways
  - Determinism preserving (for logic entities)
- Information generally flows one-way; logic -> draw-only
  - Components of logic entities can write to draw-only entities
  - Components of draw-only entities can read from logic entities