Gold level sponsor

Bloomberg

Engineering

THANK YOU

for playing an instrumental role in
making the NYC++ Meetup possible

THANK YOU

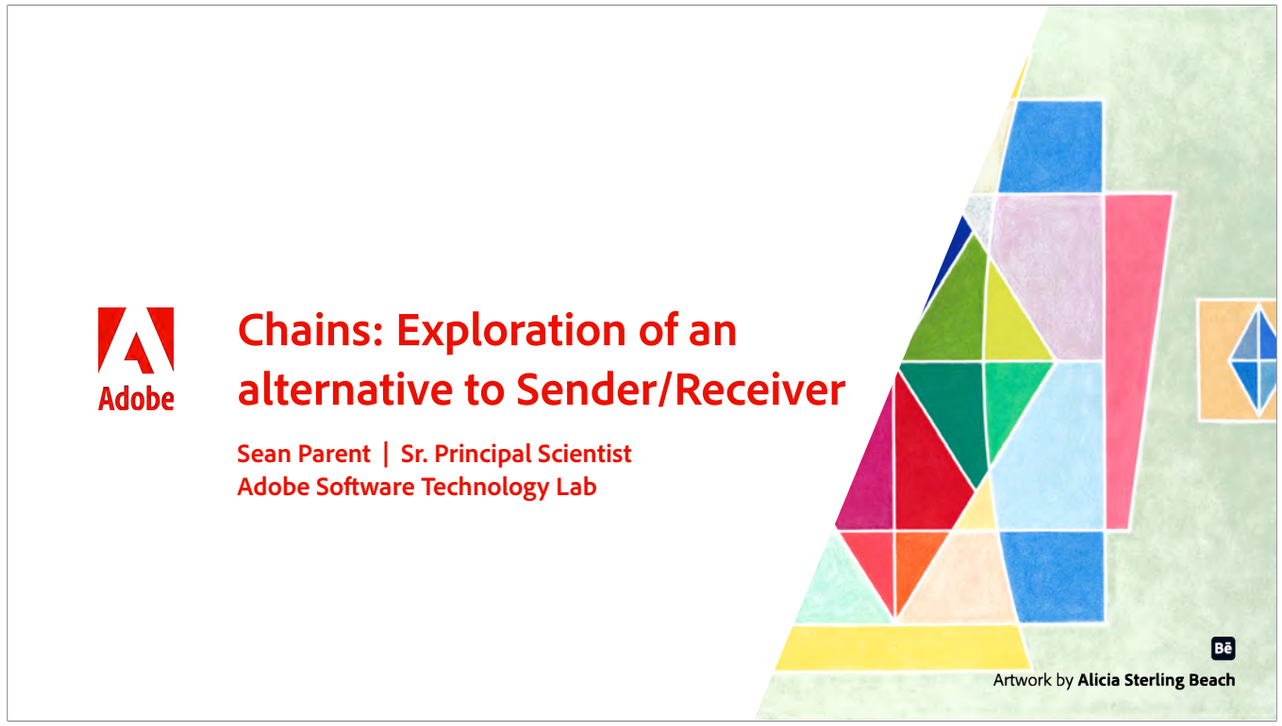
to our sponsors…

Bronze

🙏 **Monochrome Search**

🙏 **Undo**

for supporting the NYC++ Meetup



Concurrency is difficult because of mutation and efficiency.

History

- In 2015, I was involved in bringing Lightroom to the browser
- This was under asm.js - a single-threaded model
- Lightroom is a multi-threaded application
- The code had to be transformed so it could run single-threaded
  - and scale to run efficiently on many cores

Adobe Lightroom is a professional image editing tool targeting photographers.

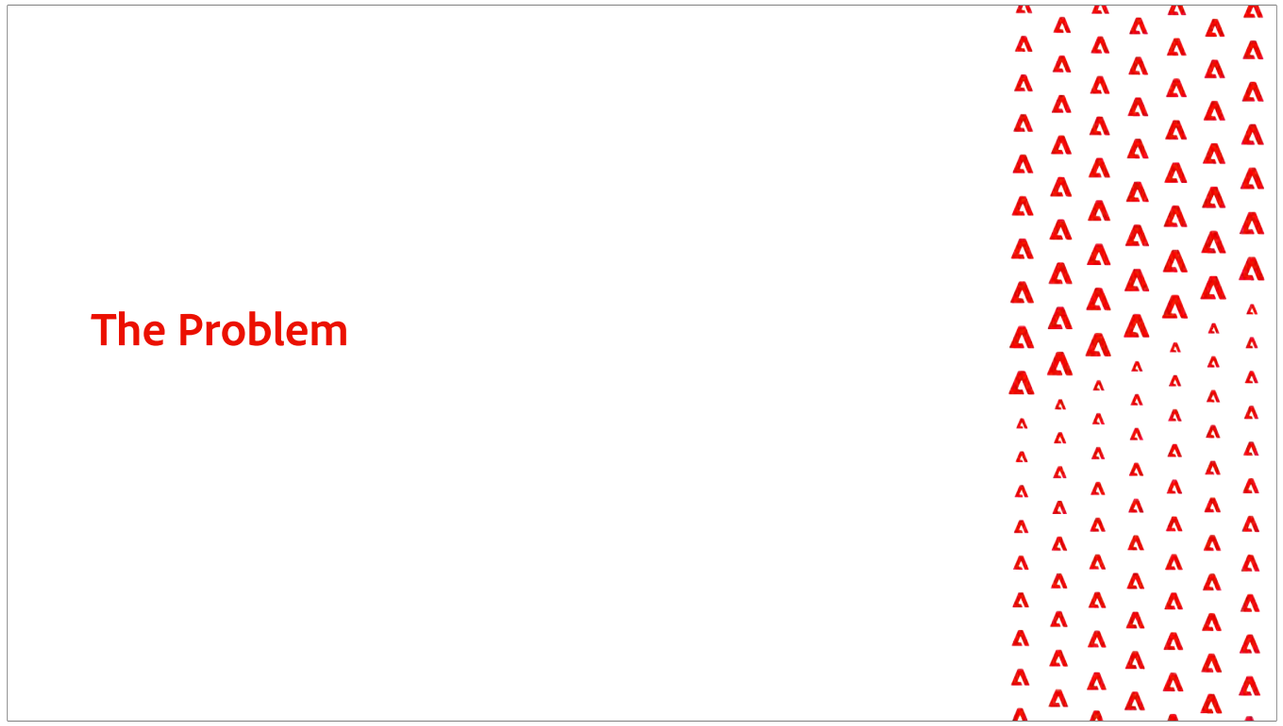
History - Boost futures with continuation

```
auto f = make_ready_future(42);

auto f0 = f.then([](auto a){ return a.get() + 2; });
auto f1 = move(f).then([](auto a){ return a.get() + 3; });

print("{}\n", f0.get());
print("{}\n", f1.get());
```


Futures with continuations are a reasonable model for transforming synchronous code to asynchronous code and async code to non-blocking concurrent code. They don't handle all concurrency patterns - they are just part of the solution, but an important part we are focused on today.

My first thought was to use boost futures with continuations. I wrote a test piece of code equivalent to this. The second line crashed - with boost futures, `then()` is a consuming operation. If it had relied on C++11, it would have required a move. You could write a "split" operation (although, at the time, I didn't see how), but there were more issues that led to me starting what became the stlab concurrency library.

The Problem

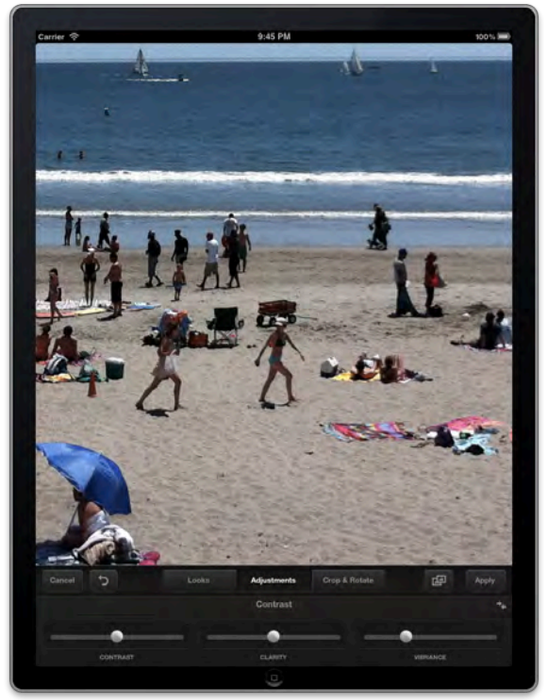


Let's back up a bit and look at one of the reasons concurrency is used in an interactive application—emphasis on interactive. Concurrency is a tool to improve performance with a fundamental tradeoff between latency and throughput. An interactive application strives for specific (real-time) latency guarantees. You want the application to feel fluid and responsive, regardless of how long the processing takes.

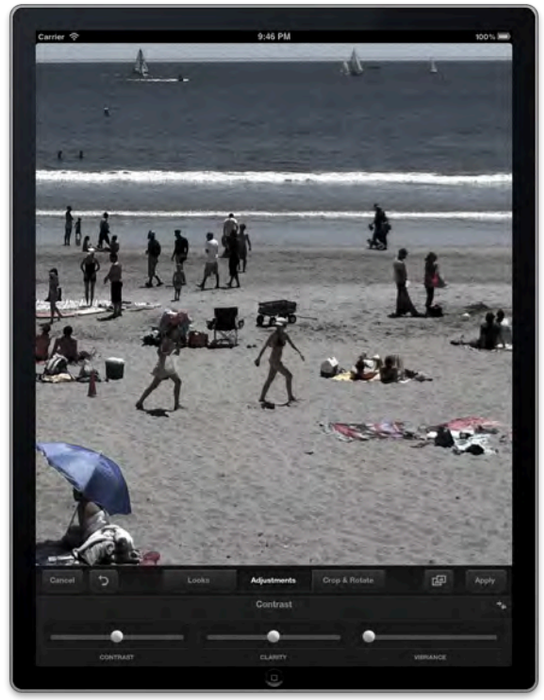
These are slides from an old product I worked on, Adobe Revel, this was built on Adobe Camera Raw, the same rendering engine that powers Adobe Lightroom and the Camera Raw component in Photoshop. The code is optimized for latency over throughput.

To provide smooth pan and zoom without seeing black the code keeps a GPU texture of the entire image at a low resolution.

With a high resolution image overlaying it to fill the current view port.



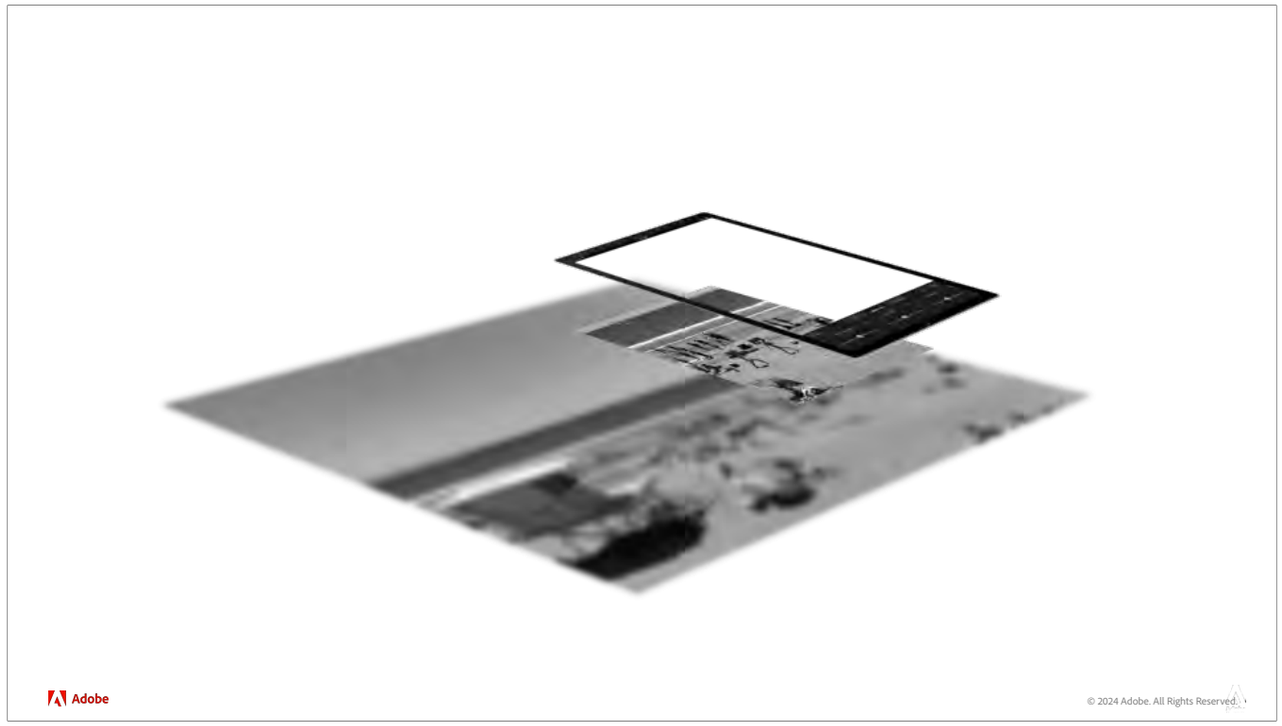
When you make an adjustment, like sliding a slider...



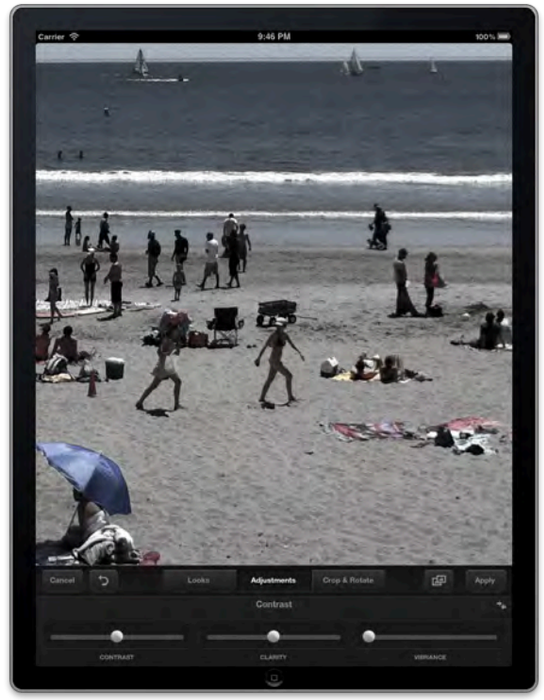
We render a draft of the foreground texture at a low quality.

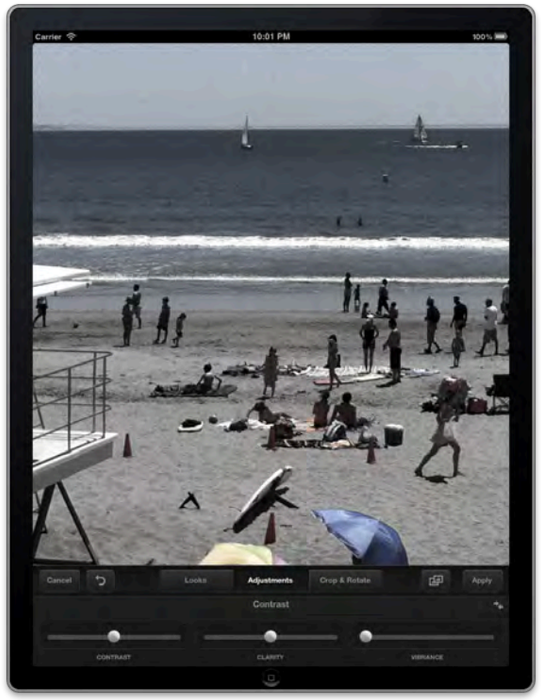
Then, if there are no pending render requests, we render at a higher quality...

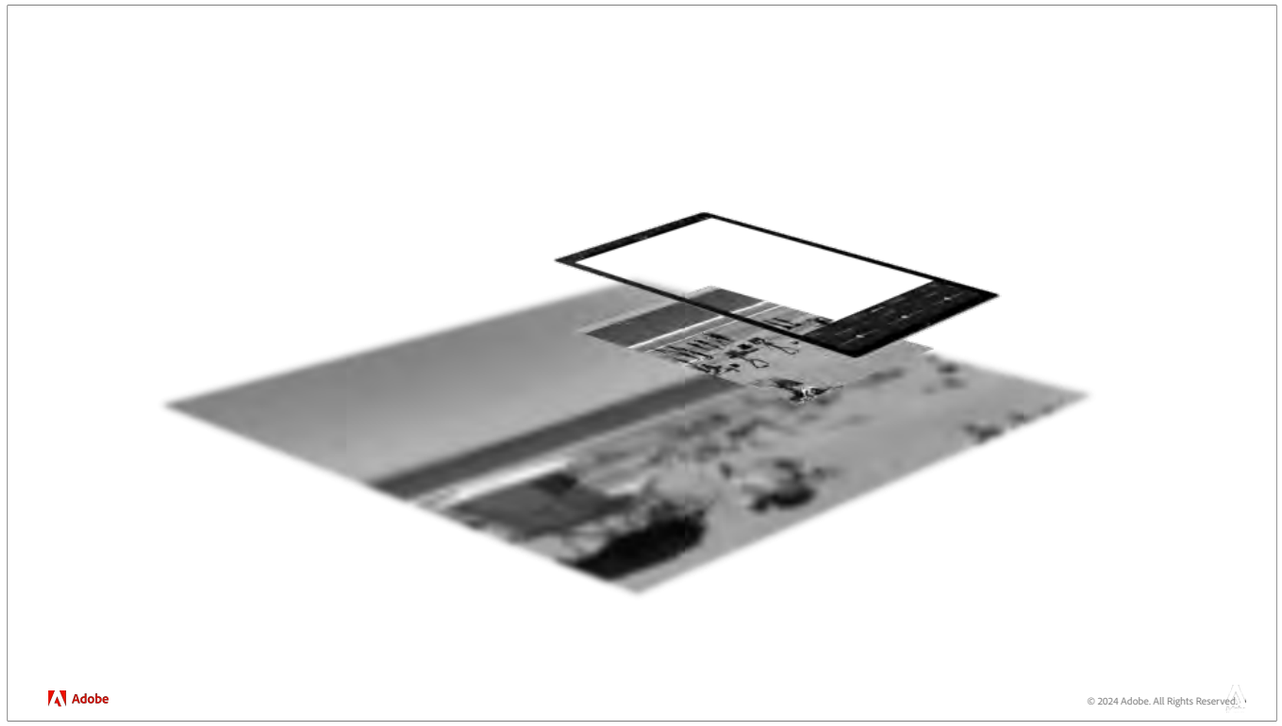
And then again at full quality.



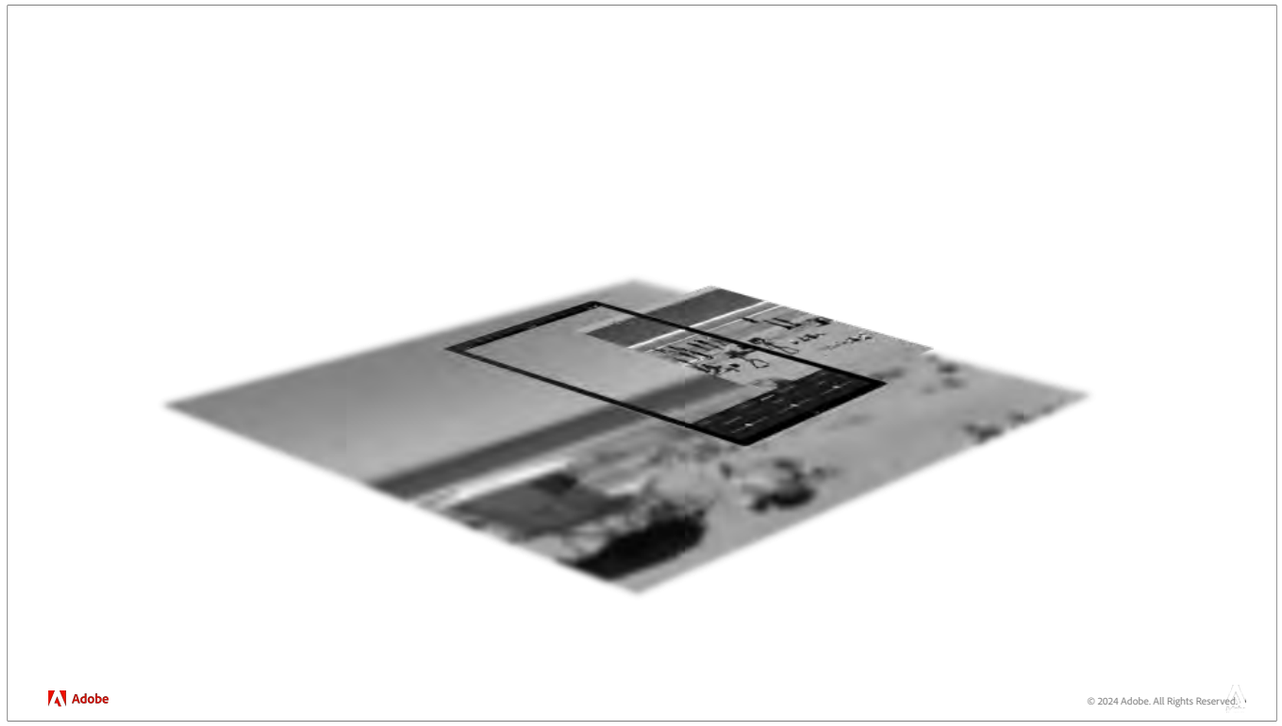
And then we update the background layer.



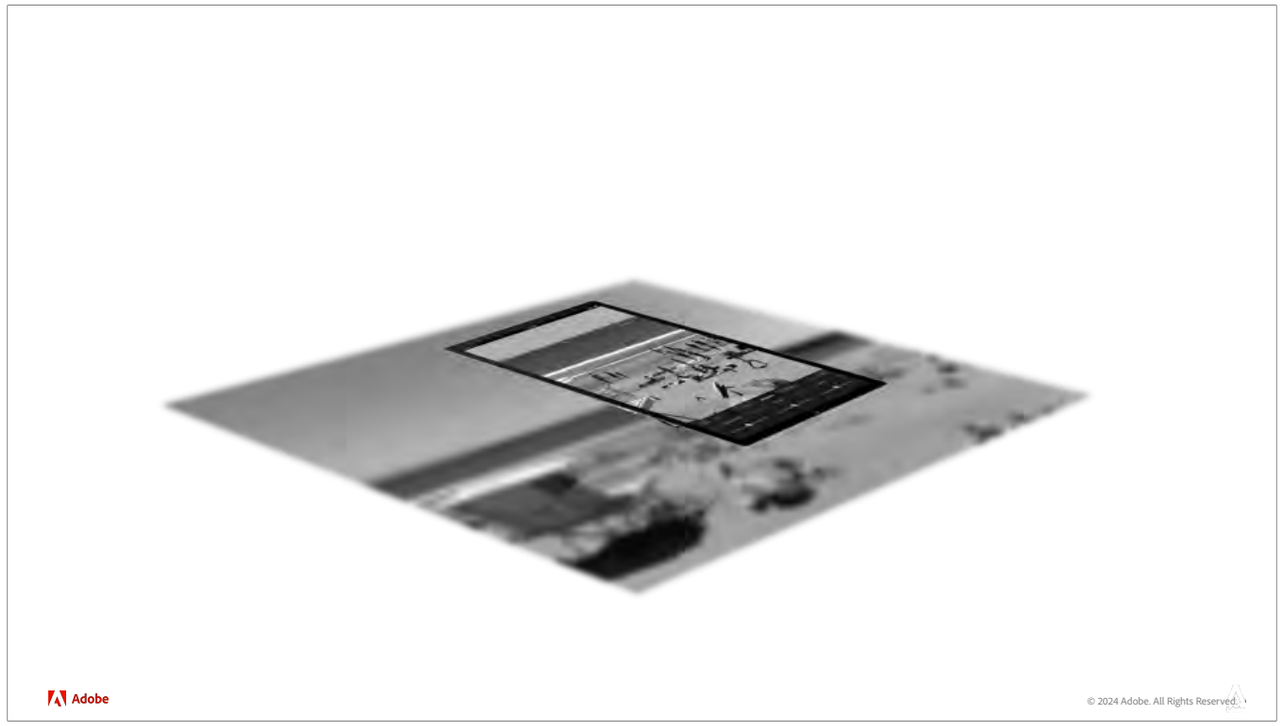




When panning or zooming we animate the two layers in the viewport at full device frame rate (120Hz on an iPad Pro).

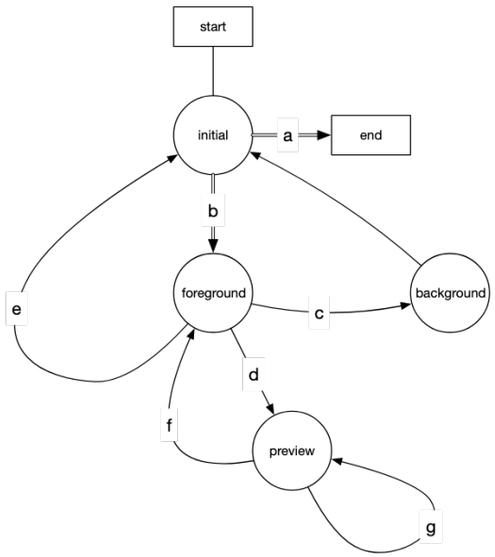


But we chase the animation, starting a render for a higher quality image at the, current, position we are animating too.



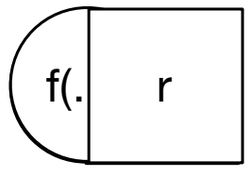
With the goal that by the time the motion stops, you are looking at full quality pixels. The human visual system is poor at discerning detail of objects in motion or objects that are chaning. The effect is it appears to be rendering high quality at full frame rate, even though it may only be capable of rendering full quality at a frame-rate of 1/4 or less.

Rendering State Machine Structure



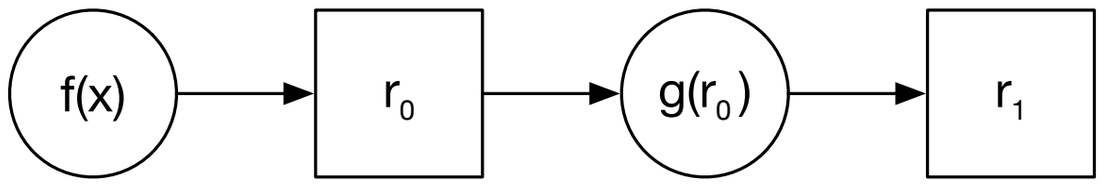
The state machine that drives the rendering for display is implemented as a coroutine. The operations at each state are asynchronous. As the state machine transitions, it may _cancel_ the operations from prior states as they are no longer required, typically canceling a "stale" high-quality render request in favor of a lower-quality, faster request. But lower-quality renders may be a delta-render of a high-quality render, or there may be caches shared between renders that may be built. So when canceling, the question becomes, cancel what exactly?

Futures



Futures are a building block for asynchronous code. They provide a simple way to transform synchronous code into asynchronous code. Conceptually, a future is a token for the result of a function executed in a different context. Futures carry the dependencies—destructing the future cancels the task.

Futures: Continuations

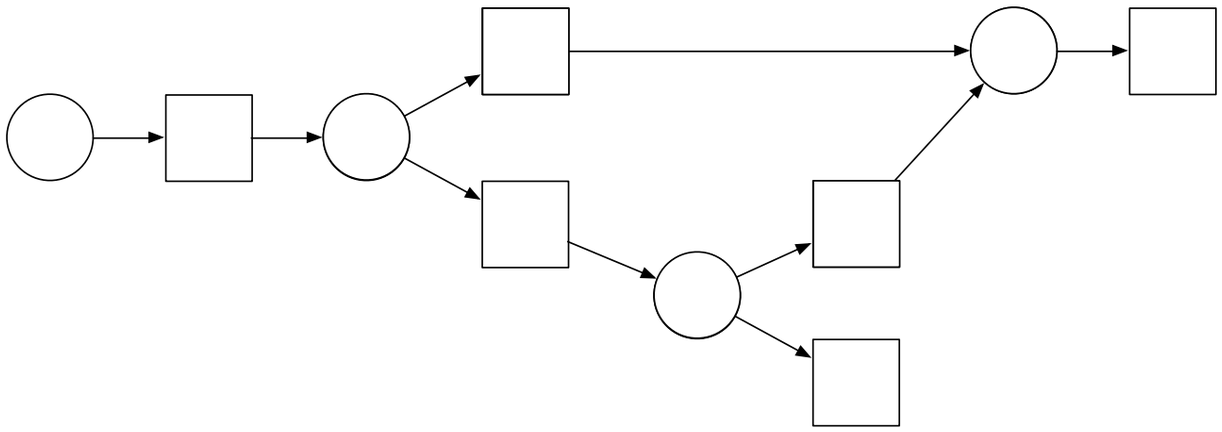



```
g(f(x));
```

```
auto future = async(f, x) | g;
```


Continuations are just function composition. Again, dependencies are carried by the futures. If you don't want a result, then destructing it "unravels" the dependencies.

Cancelation

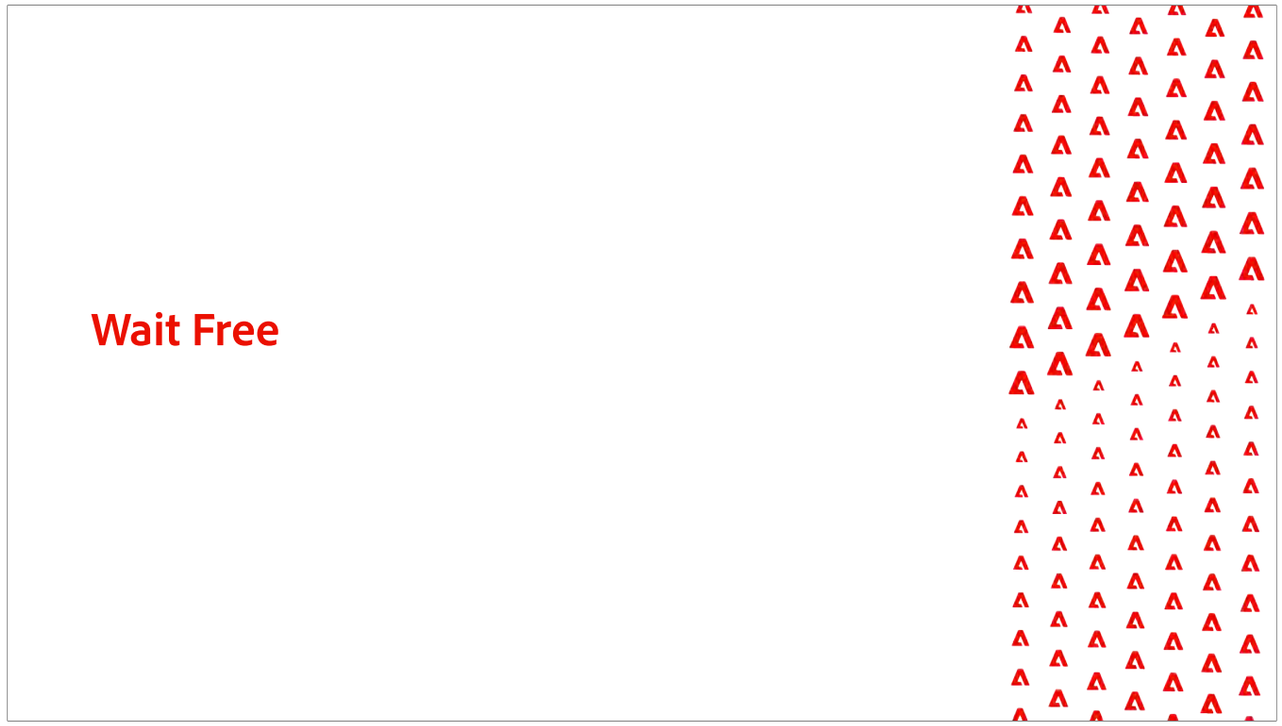


With joins (when all) and splits (multiple continuations on the same future), the graph becomes a DAG. Canceling an operation may cancel may in turn cancel other operations, but not those that are required to satisfy other futures.

Efficient Cancelation

- A dependency graph is a bipartite DAG of operations and their results
- Operations may be scheduled for execution, active (executing), or completed
- If a result is no longer needed (future is destructed), any **uniquely** contributing operations are *canceled*
  - Active operations receive a stop request
  - Scheduled operations will not start, and their resources (which may include results of other operations) are released
- Cancelation is *efficient* if every active uniquely contributing operations immediately receives a stop request and no scheduled operations start after the cancelation.

We can define efficient cancelation. Cancelation is also asynchronous - the canceling thread should not await active operations. Of corse the stop message propagation takes time - and is inherently racy. We want to minimize the message propagation time and avoid serializing cancelation between active tasks.



Let's back up a bit and look at one of the reasons concurrency is used in an interactive application—emphasis on interactive. Concurrency is a tool to improve performance with a fundamental tradeoff between latency and throughput. An interactive application strives for specific (real-time) latency guarantees. You want the application to feel fluid and responsive, regardless of how long the processing takes.

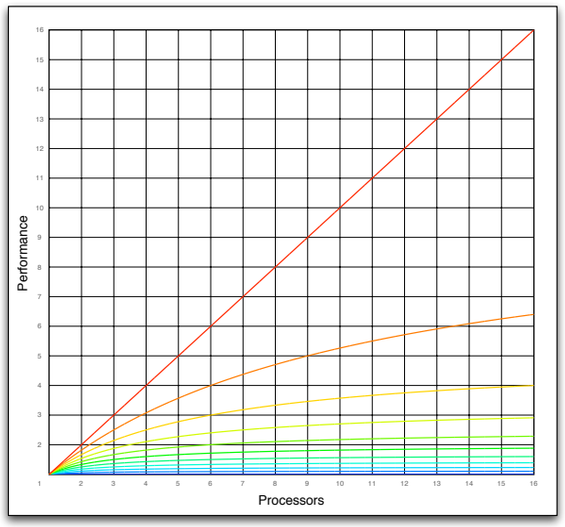
No Waiting

- On a single-threaded system, tasks are queued to the main run loop
- This allows work to be interleaved with responding to events
  - The amount of work done by each task needs to be small enough (a small grain size) to not block user events from coming through and being responded to
- Waiting for a task to complete (a synchronous wait or sync wait) is a deadlock in a single-threaded system
- In a multithreaded system, a sync wait is also *very bad*

Waiting for a task to complete... such as on a condition variable or a semaphore

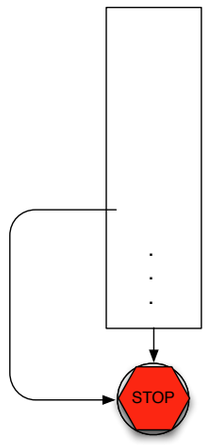
Amdahl's Law

Each line represents 10% more synchronization



The first reason is Amdahl's. This graph shows the effect of serialization in 10% increments up to a 16 core machine. If only 10% of our code is serialized, our performance with 16 cores is only slightly more than 6x and will never exceed 10x no matter how many processors we throw at the problem. Every synchronization point is expensive.

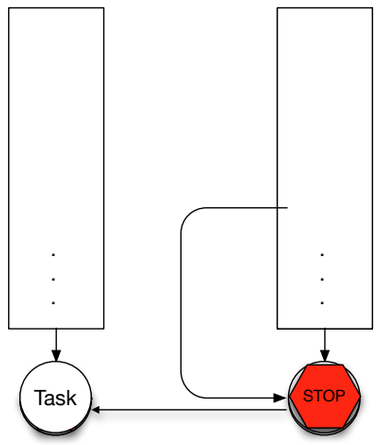
Sync Wait Deadlocks



Even in a multithreaded environment, sync-waits can lead to deadlocks. Here is a visualization of a deadlock on a with a single thread of execution.

A task issues another task which is placed in the queue.
Then stops to wait for the task to complete.
There is only one thread and one queue so the queued task will never complete,
  this is a deadlock.

Sync Wait Deadlocks



With two threads, the first task may spawn a task, and wait for it to complete.
That task is picked up by the second thread it spawns some additional work.
And waits for that work to be completed. Deadlocked.

Any sync wait operation requires non-local reasoning about where the task that is being waited on is scheduled, what tasks that task may spawn, and how many threads are available in the system.

This leads to the "function color problem" - If a function is async, every operation that depends on it is also async. There is no way to rejoin.

To use sync wait safely, you need to know that thread resources are available to execute the item being waited on. Sync wait breaks local reasoning.

History

- I gave a talk on how I solved these problems, *Better Code: Concurrency*
- I cleaned up my code and packaged it as a little library on GitHub, stlab/libraries/concurrency
  - Written almost entirely in the hotel bar at C++Now

- Felix Petriconi found it useful and contributed most of the current code
- The library is now in broad use at Adobe and other companies

Splits with stlab futures

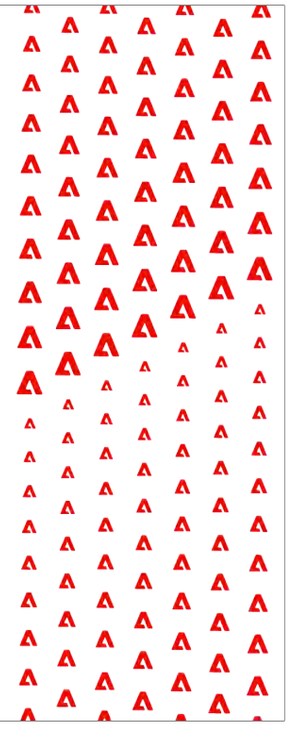
- stlab futures are copyable and support attaching multiple continuations and efficient cancelation

```
auto a = f | g;
auto b = f | h;
```


stlab futures are regular, they can be copied, you can attach multiple continuations, destruction is cancelation. RAII applies to the processor (arguably even more important a resource than memory!).

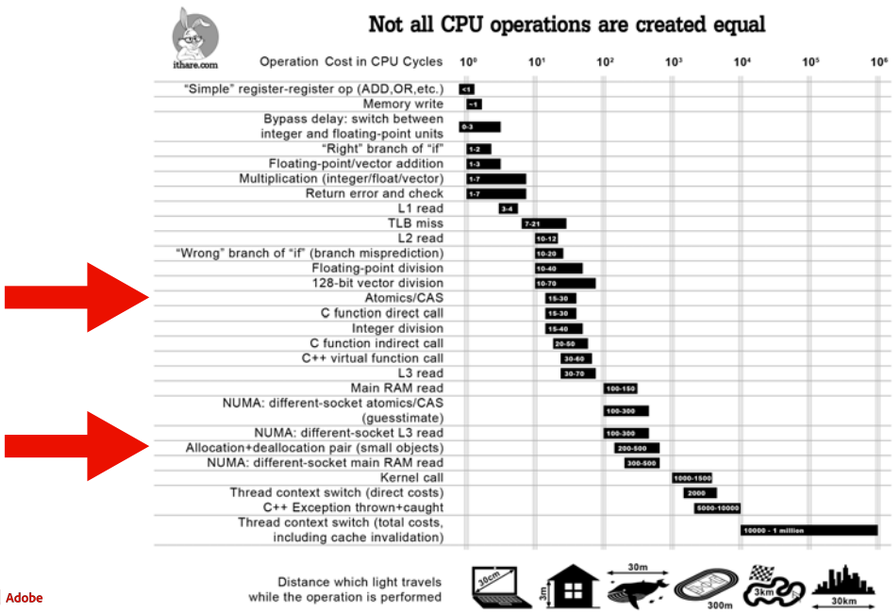
There is more to the library than just futures with continuations. There is a high performance dynamic task stealing executor (that back-ends to the platform executor, i.e. Apple's libdispatch, or Windows thread pool,if available), a "main" scheduler to schedule on the main event loop (if the platform has one), an implementation of channels (needs a major update), and a number of smaller utilities...

The Problem with the Solution



The programming model presented by futures with continuations is deceptively simple -
But every continuation comes at a cost.

Operation Costs are Not Reflected In Code



A small object allocation/deallocation is 200-500 cycles.
Every atomic operation is 15-30 cycles.

The overhead of an async continuation vs sequential function composition can easily be 100-1000x.

We only want to attach continuations when the operation is significantly large or a specific execution context is required.

(Side note: The last line on this graph is a thread context switch. This is why thread pools exist and the importance of not overcommitting.)

Cost of Future Continuations

- Continuations require shared state
  - Memory allocation and deallocation with atomic operations to control lifetime
- Attaching a continuation requires synchronization
- Resolving a continuation requires synchronization
- All of this is under the hood for the simple expression, `f | g`

Futures and continuations work well when the grain size (how much work we have to do) in the continuation is relatively large. The more we reduce the overhead the greater the performance benefit - not just because we reduce the cost per call but because we enable additional parallelism to benefit.

Don't use continuations as expensive function composition

- If you know the continuation in advance, don't write:

```
auto a = async(f) | g;
```

- But instead write:

```
auto a = async(compose(g, f));
```


But this doesn't compose well, an async operation now needs to take a callback so it can compose the operation prior to starting it. The simple syntax of futures with continuations means they get used for simple function composition - at a high (very measurable) cost. Even this function composition has a cost - at each level we are adding another stack from. But what if instead of starting the async operations, we simply described them - enter sender/receivers.

Don't use continuations as expensive function composition

- Async operations need to take a callback and compose it with their work

```
auto op() -> future<T> { /*... */ return async(f); }
```

- Becomes:

```
auto op(invocable<T> auto g) -> future<decltype(g(T))> {
    /* ... */
    return async(compose(g, f));
}
```


An async operation needs to take a callback so it can compose the operation prior to starting it. The complexity grows rapidly.

The simple syntax of futures with continuations means they get used for simple function composition - at a high (very measurable) cost. Even this function composition has a cost - at each level we are adding another stack from. But what if instead of starting the async operations, we simply described them - enter sender/receivers. The complexity grows rapidly.

Sender/Receivers are Function Composition

- `then(f) | then(g)`
  - `compose(g, f)`
- `transfer(s) | then(f)`
  - `bind_front(s, f)`
- `transfer(s) | then(f) | then(g)`
  - `bind_front(s, compose(g, f))`

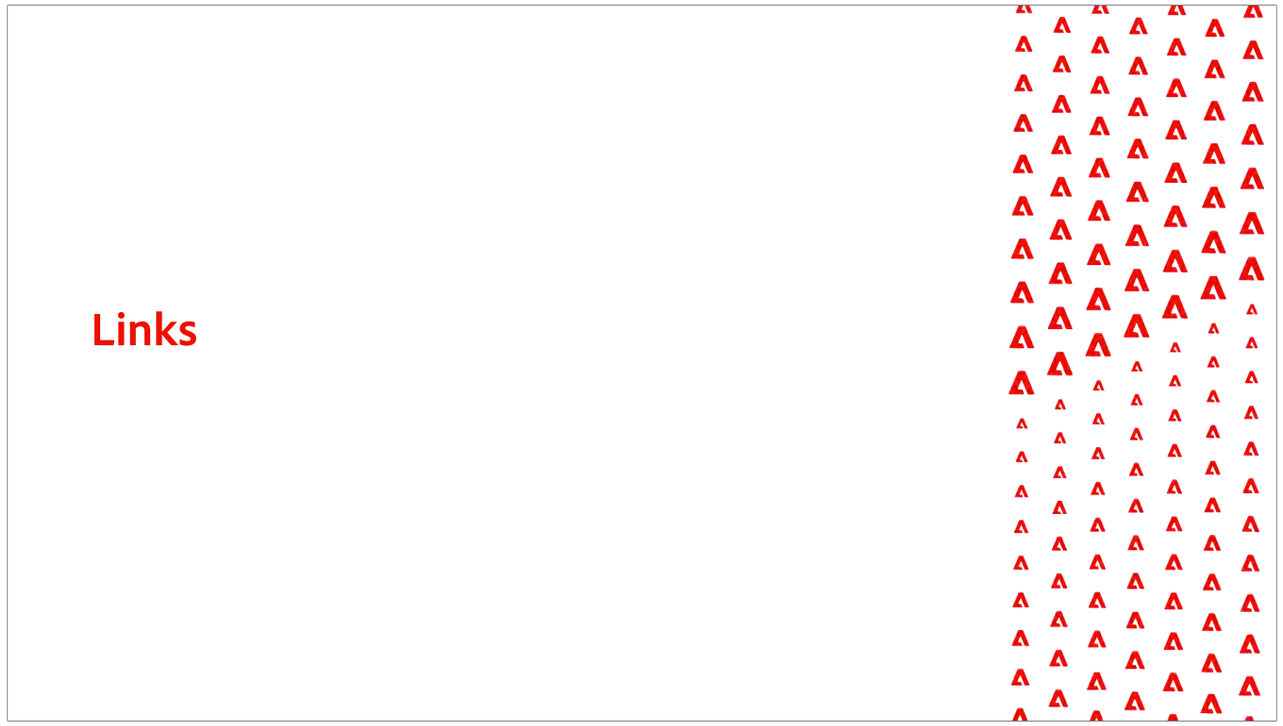
This is the problem that C++ sender/receivers solve. Instead of launching async code and attaching continuation, we build a _program_ (a function) that describes the async operation and then start that. This _program_ is built flat, without heap allocations, with no need for synchronization because the connections are made before it starts.

`then()` is compose logically compose, it doesn't return g(f(x)) it returns the logical function g composed with f. This composition pattern is obscured with a complex sender/receiver interface. We're building up functions that describe an async computation graph, and then we start it. By appending application and composition operations, we can build a computation tree.

In sender/receiver terminology, `then()` returns a sender adapter closure.

In std::exec, "transfer" transfers execution to, for example, a thread, serial queue, GPU, etc.

Links



The programming model presented by futures with continuations is deceptively simple -
But every continuation comes at a cost.

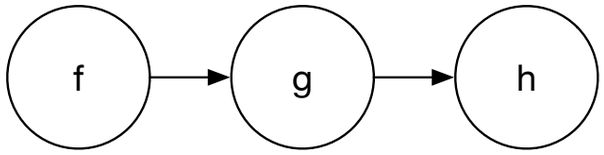
Flat Composition - Links

- … `f | g | h`
- The desired result is, `compose(h, compose(g, f))`
  - But, without the nesting - we don't want to blow the stack

Let's start with flat composition - we will drop the `then`.

We're building a functional model to describe the execution of a program

Links



Links are just functions - stored as a flat sequence. The links form a single logical function.

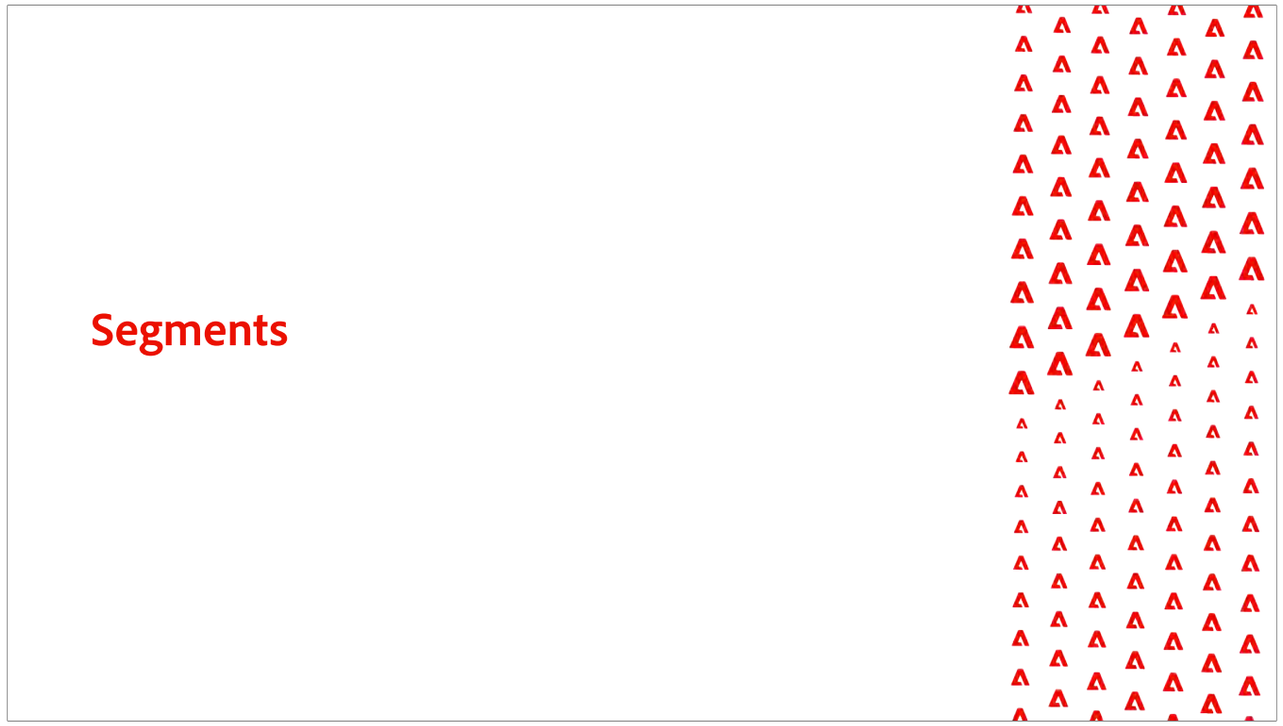
Flat Composition - Exceptions

- Given `f | g | h`, what if an error occurs during the application?
- If we execute `h(g(f(x)))` and f throws an exception, can `g` or `h` catch it?
- The same is true with `std::expected`
  - `f(x).and_then(g).and_then(h)`

With sender receiver we can pass the error from one `then()` to the next, but what do we do if this is just a simple compose? If we were using std::expected, it doesn't change this, instead it changes how the application happens.

We only need a sequence of functions to describe the computation

Segments



The programming model presented by futures with continuations is deceptively simple -
But every continuation comes at a cost.

Application - Segments

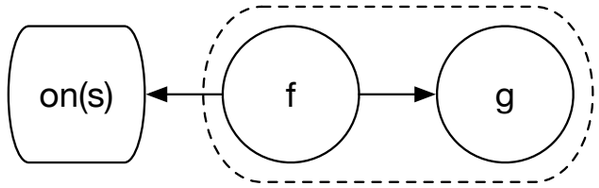
- `on(s) | f | g`
  - `on()` creates a segment, a function that is the applicator function, `s`, bound to the subsequent sequence of functions
    - Additional functions may still be appended

```
bind_front(s, f | g);
```

  - `s` determines the context in which `f | g` is executed

This is binding s to f and g - but we don't actually do the bind directly, we store the structure

Segments



Segments are an applicator and a sequence of links. A segment logically forms a single function.

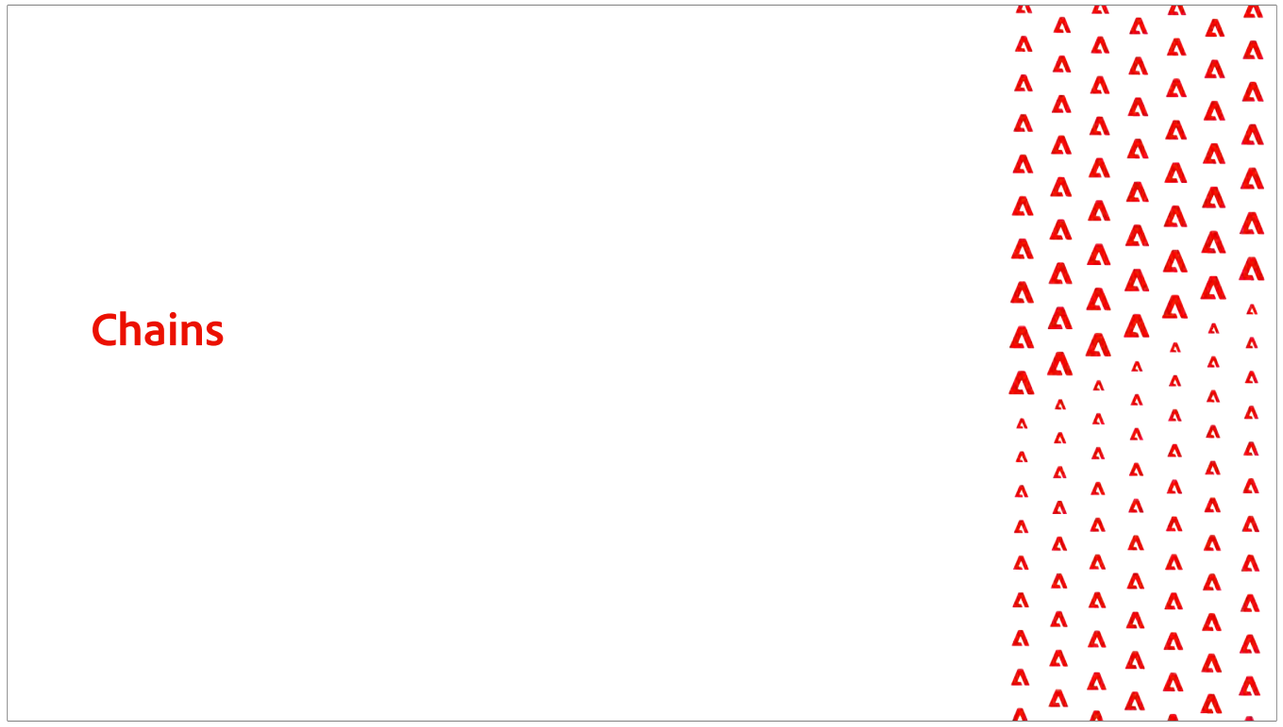
Application - Segments

```
auto expector = [](auto f, auto... args) -> expected<...> {
    try { f(args...); }
    catch { return current_exception(); }
};

auto badd = [](int x, int y){
    if (x == 42) throw runtime_error("bad x");
    return x + y;
}

(on(expector) | badd | [](int x){ return x * 2; })(12, 5);

expected{34};
```


As an example, we can write an apply function that maps exception to std::expected and then use it to compose functions and return expected on an error.

If we pass 42 for x will result in an expected carrying the bad x exception.

Chains



The programming model presented by futures with continuations is deceptively simple -
But every continuation comes at a cost.

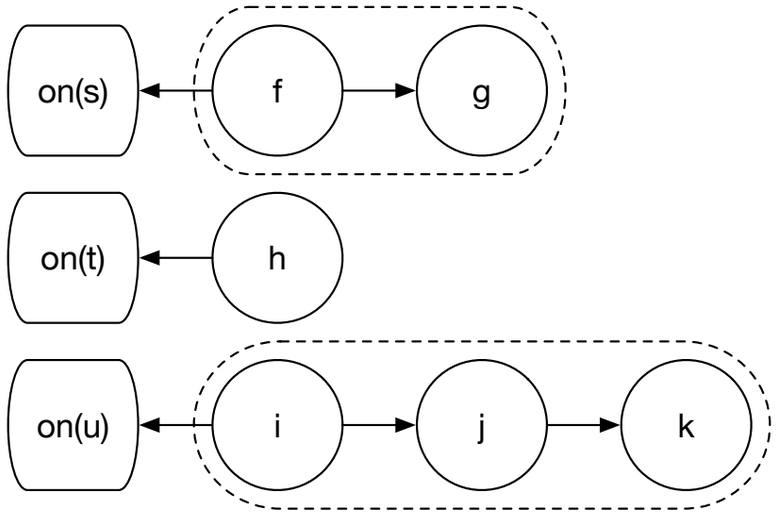
Chains

- A chain is simply a sequence of segments
- on(s) | f | g | on(t) | h
- Segments are linked - the next segment is part of the prior segment

```
[ _f = f | g | (on(t) | h), _s = s](auto... args){
    return _s(_f, args...);
}
```

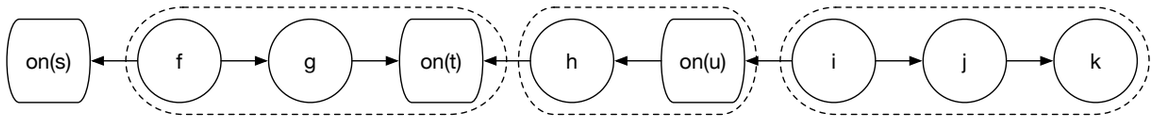

If s sends f | g to another thread, how do you get the result back? Are we back to a future?

Chains



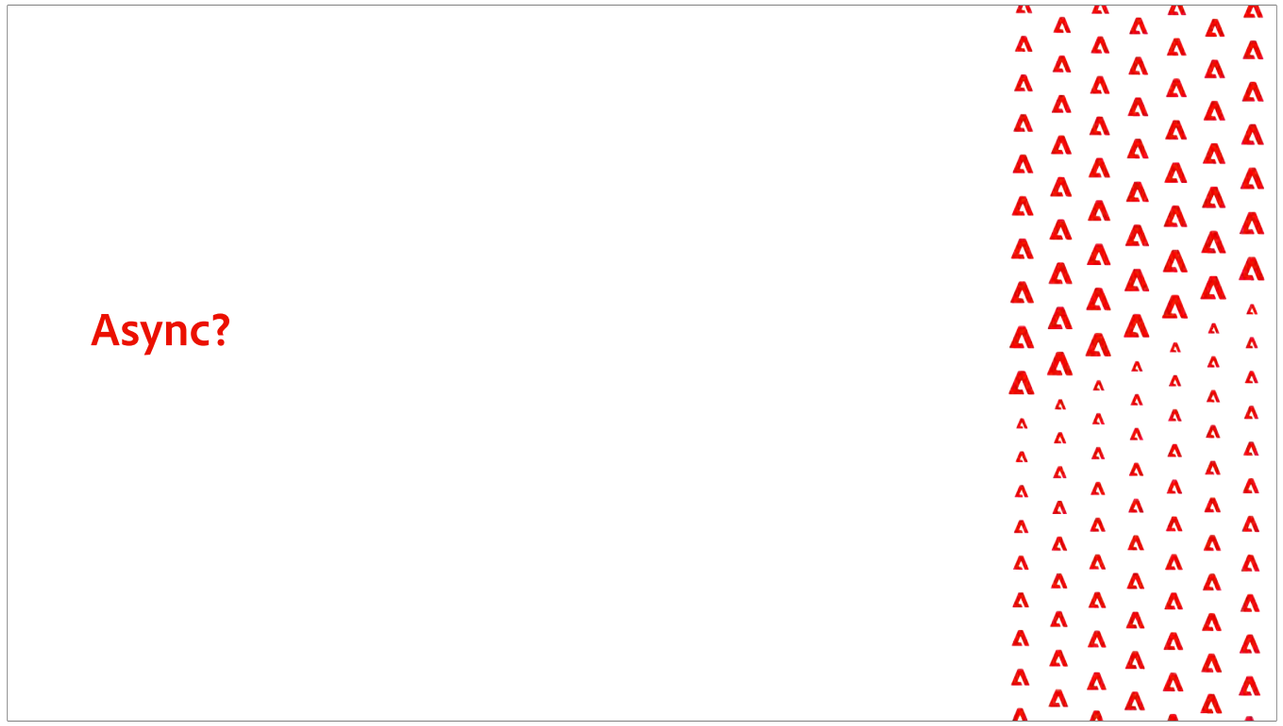
Chains are two-dimensional data structures. We can append functions to the end of the last segment or segments to the end of the chain. They are stored as a tuple of tuples of functions - flat with no heap allocations.

Chains



The chain is executed by appending each segment to the end of the prior segment, so the result of one segment is passed as an argument to the next segment. A chain is logically a single function.

Async?



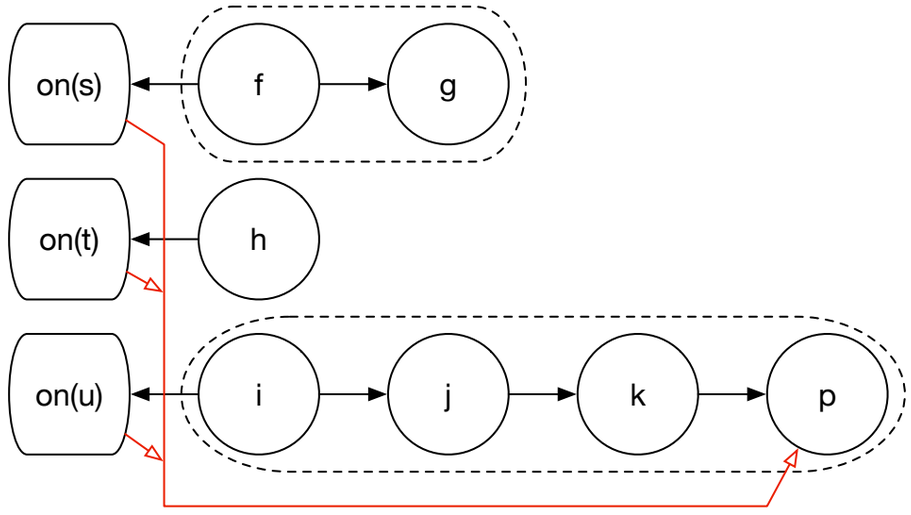
The programming model presented by futures with continuations is deceptively simple -
But every continuation comes at a cost.

Mapping to Async

- A chain is a program that describes the sequence of execution
  - We can create segments that provide an execution context to schedule the segment
  - Functions on chains can append a promise and start the chain returning a future
    - And bind additional arguments to every applicator (i.e., a stop token and set_exception operation)

Instead of wrapping individual operations

Ready for Async Execution



p is a promise associated with a future. Having access to the structure allows operations to make transformations. Errors only need to be caught and propagated from the segment level (same for cancelation).

Mapping to Async

- Chains are a general purpose* facility to build functional descriptions (programs) in C++
- Sender/receivers are a language within which to build asynchronous descriptions**
- So far, I've implemented the equivalents of:
  - `schedule, just, just_error, transfer, then, on, sync_wait`
  - With efficient cancelation and error handling

* How general purpose? I believe chains with an S & K combinator are Turing complete - still working on the proof.

** Kind of - they don't have to be used for asynchronous descriptions, but they carry the complexity of signaling cancelations and exceptions at every step.

Split

- ???
- We shifted from a model of functions with detached results to one of functions *as* results.
  - What does it mean to split a function?
    - Synchronization is required
    - The shared portion should be executed once
    - Canceling should only cancel the non-shared portion
    - Must have no arguments, first invocation starts, and subsequent invocations ignored

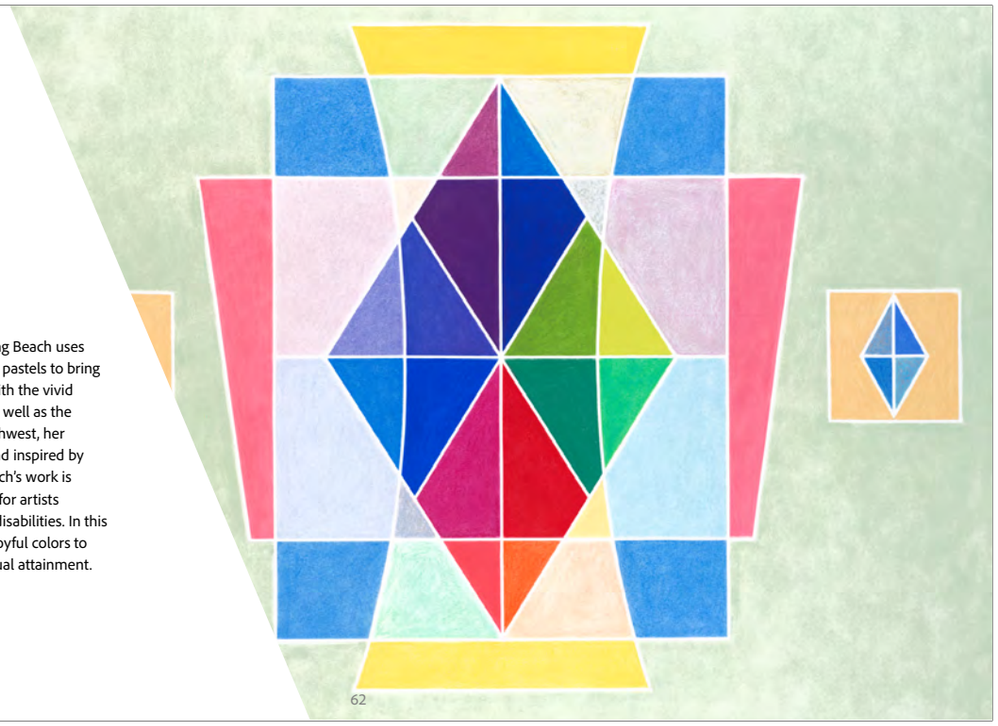
The current implementation with sender/receiver split is broken with respect to cancelation.
let_value() provides a form of split for computation, but split must be contained with a rejoin.
I've managed to spark interest in trying to find a solution on the sender/receiver forum

Next Steps

- Chains can't currently do anything that sender/receivers can't do*
  - But the model is simpler
- Chains are a prototype missing production features:
  - void result types everywhere
  - minimizing copies
  - handling perfect forwarding and move-only types everywhere
  - handling copy everywhere appropriate
  - ...

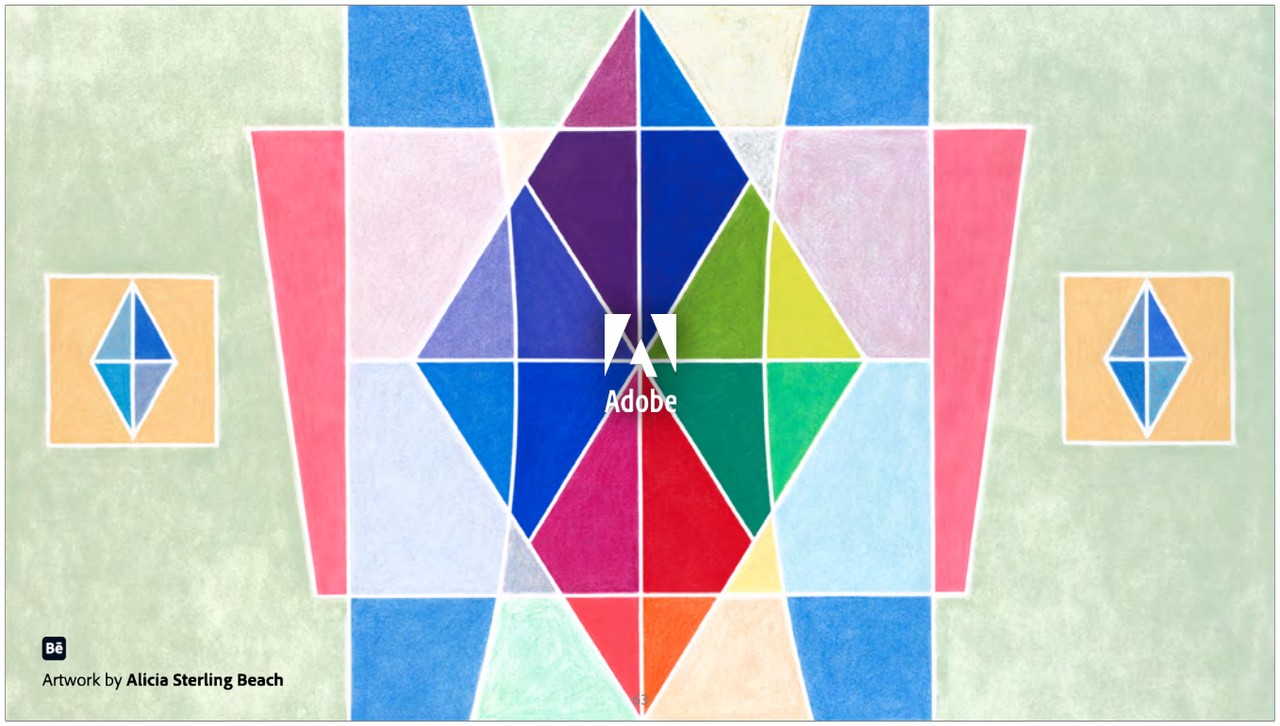
* Maybe. I'm only two weeks into working on chains, and each step has been challenging. I may just rediscover the complexity of sender/receivers.
A structured form for the complete "program" may have advantages for scheduling (especially on GPUs). But also some of the same problems - the entire structure of the program is carried in the type system. I suspect I'll hit implementation limits, however, my types are less complex then sender-receiver types. Eric is currently exploring a form of type erasure through lambda expressions that is questionable.
... And I still have many open questions. Declaring signatures up-front would simplify the code and produce better compiler diagnostics, but it would also limit "generic" code, which would have to build programs on instantiation. I'm 2 weeks in where sender/receivers are nearly 10 years in...
At the very least the process of thinking through and building chains has greatly improved my understanding of Sender/Receivers and where the complexity comes from.

About the artist



Alicia Sterling Beach

Los Angeles-based artist Alicia Sterling Beach uses watercolors, colored pencils, and soft pastels to bring beauty into the world. Growing up with the vivid colors and music of Latin America, as well as the Native cultures of the American Southwest, her artwork is informed by her history and inspired by nature, light, and classical music. Beach's work is featured on *artlifting.com*, a platform for artists impacted by housing insecurity and disabilities. In this piece, she combines symmetry and joyful colors to express balance, harmony, and spiritual attainment.


Adobe
Artwork by Alicia Sterling Beach

Please consider supporting NYC++

https://nycpp.dev/support