Futatabi: Multi-camera instant replay with slow motion

Steinar H. Gunderson

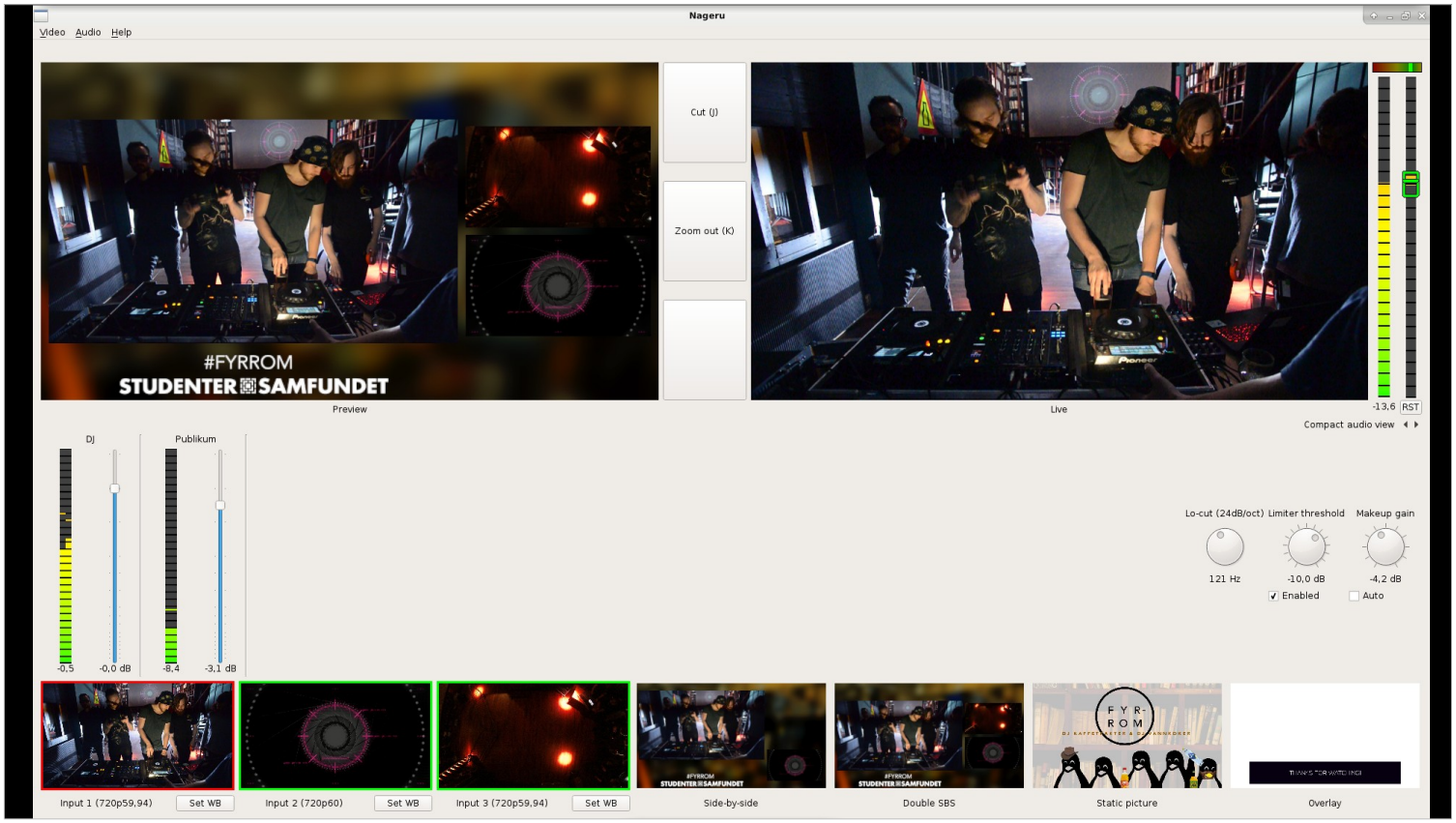
FOSDEM, February 2nd 2019

Hi! Welcome to the speaker notes for my presentation about Futatabi, my instant replay and slow motion system. As you might notice from seeing the slides, they don't always make a lot of sense without the notes—they don't necessarily match exactly what I'm going to say, but they should hopefully be of good use.

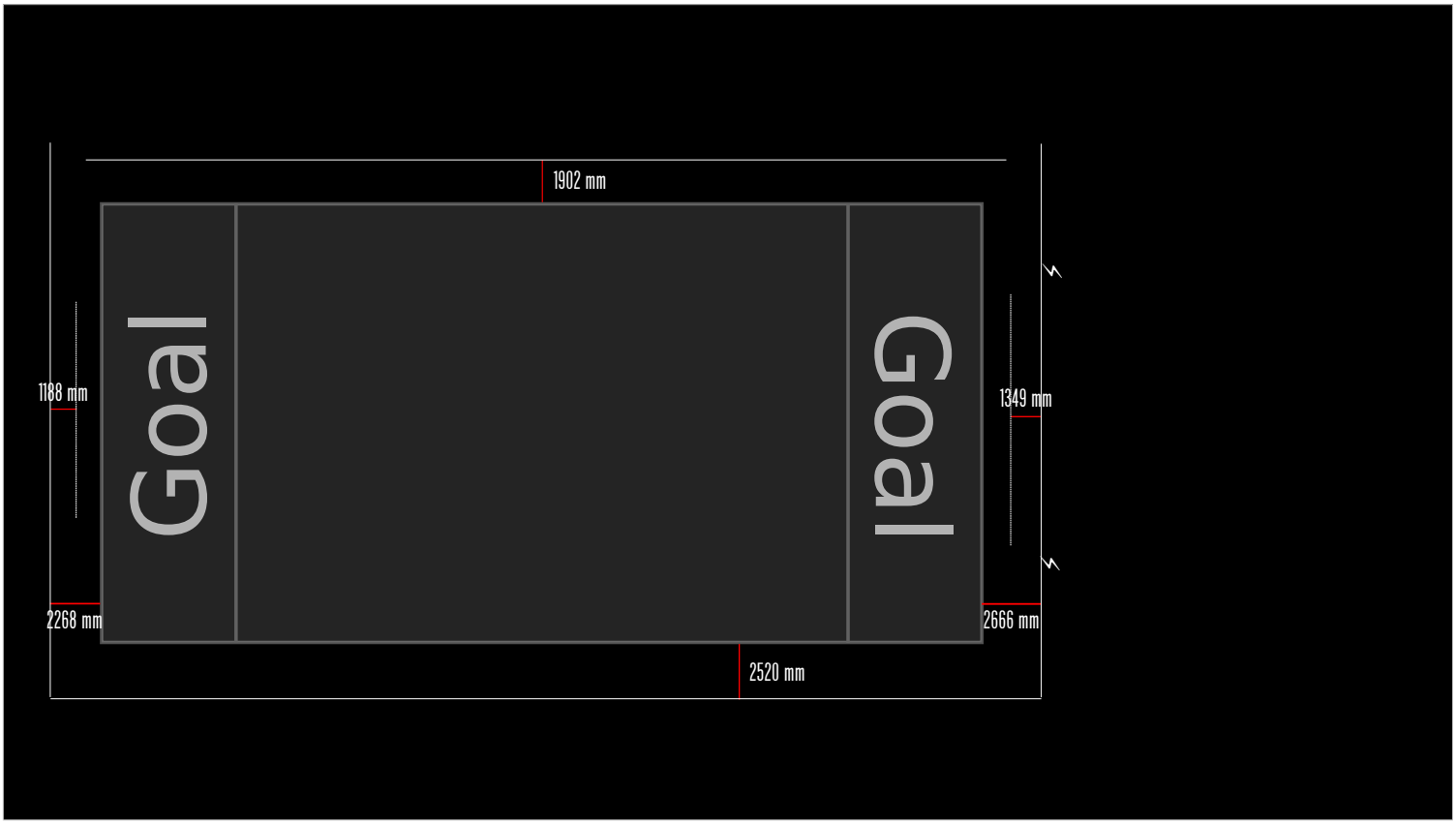
This talk is going to touch on two of my big passions, namely programming and ultimate (also known as ultimate frisbee). Ultimate isn't a big sport in Norway, so when I talk about it, I often have to explain that it's a team sport with a frisbee, where the goal is to pass to a teammate in the end zone.

Unfortunately, when I moved back home to Norway and started playing tournaments, the only available stream looked like this. Running on Xsplit in Windows, it was a very uneven 15 fps or so, and worse, everything was smudged together. There's supposed to be a goal line there (it's at the six-meter line), but you can't see it due to the poor image quality—it's just very hard to explain a game to people when you can't even see whether a pass was caught in or out.

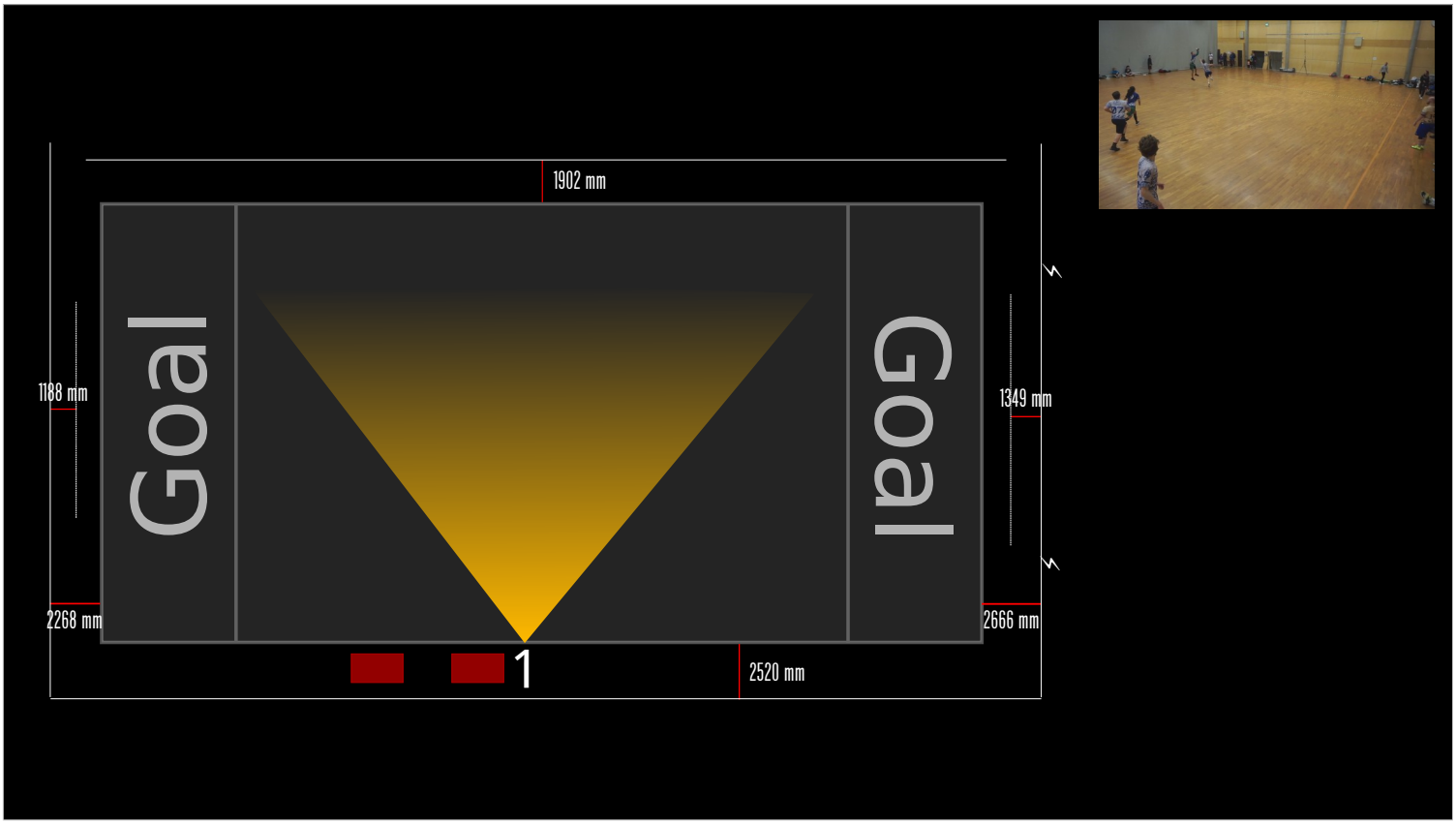
Naturally, the question became: Could we do better with free software?



It turns out we can. This is Nageru, my live video mixer. I presented it at FOSDEM 2016, and it's grown a lot since then. We'd never done sports before, but it sounded like a fun challenge, so I brought a few friends to see what we could do.



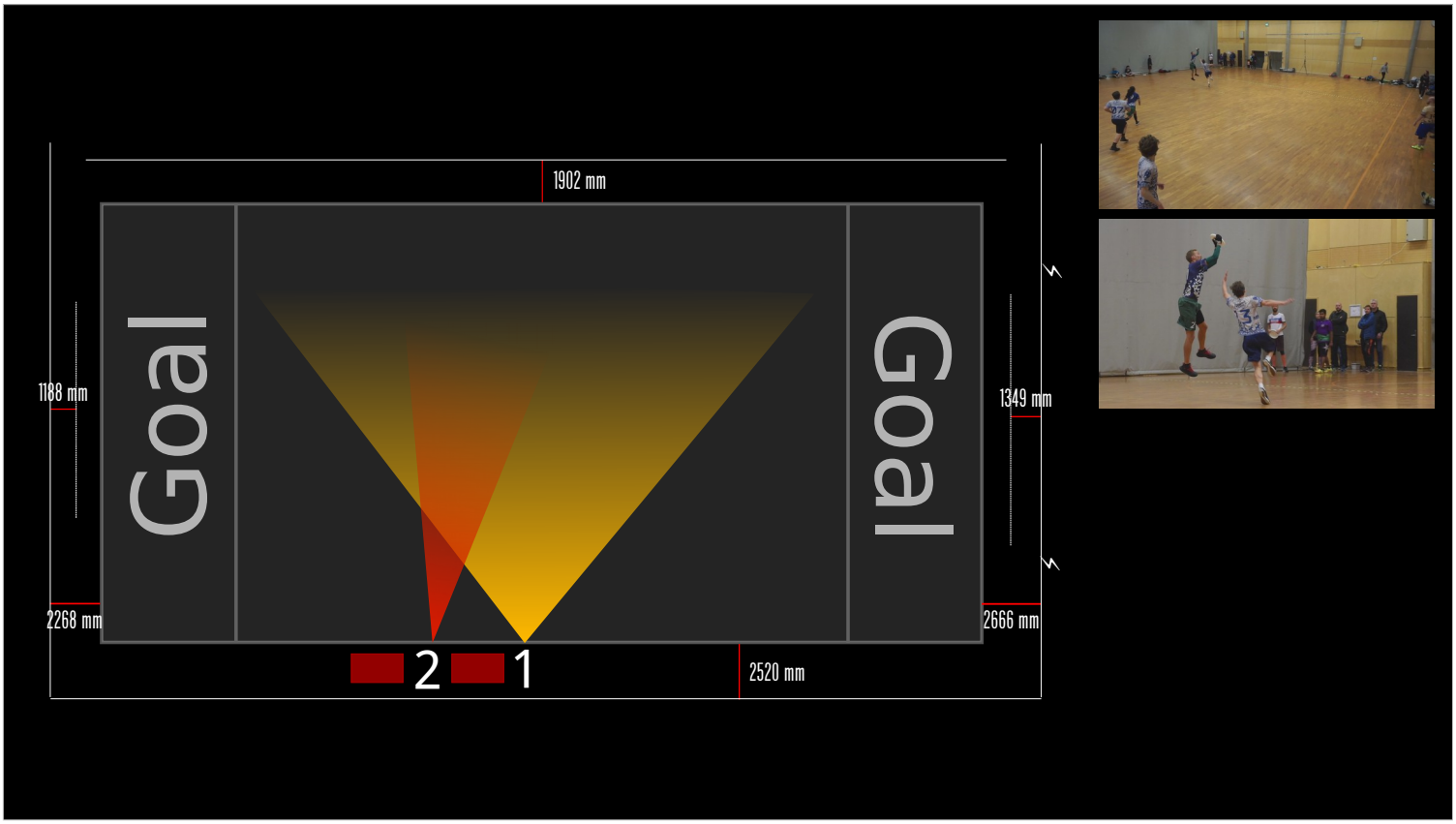
Let’s have a look at what the physical setup looks like. This is a standard indoor ultimate field, 40x20m (ultimate is mostly played outdoors in most parts of the world, but Norway primarily has an indoor tradition, using handball fields) plus a tiny bit of space on all sides.



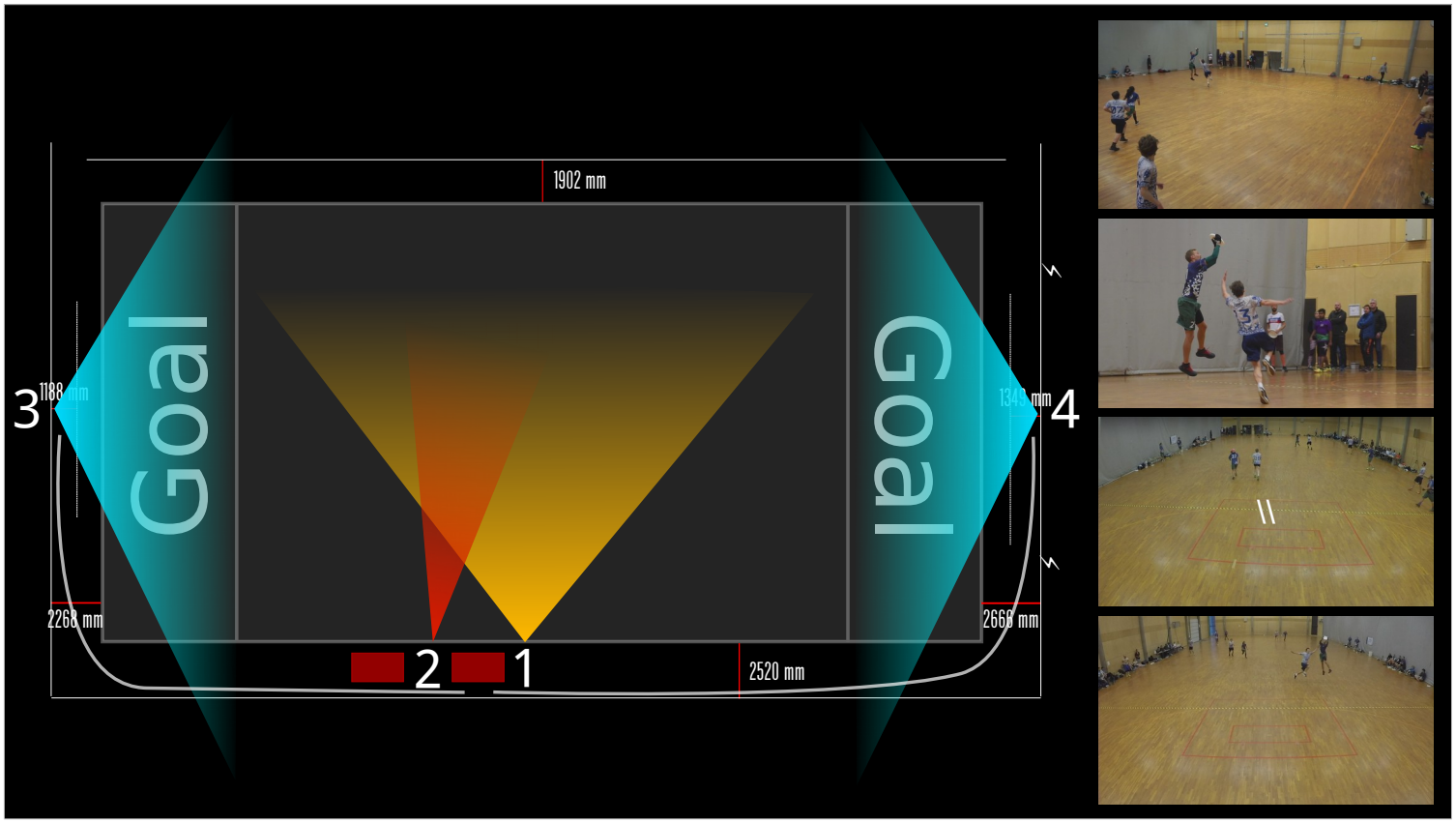
We don’t have a large team. The producer operates Nageru with one hand and operates camera 1, mostly panning, with the other. They also mix audio and control the overlay graphics.

The stream is pointed to camera 1 maybe 80–85% of the time. It’s a general do-everything angle that shows most of the active parts of the field as the play progresses.

You can see the table for the commentators, who have a comfort output of the stream.



Camera 2 is run by a separate operator. While camera 1 shows the entire play, camera 2 is usually zoomed in and following the player with disc. It can be a challenging task for the operator, but it allows for detail shots that show what camera 1 can’t.



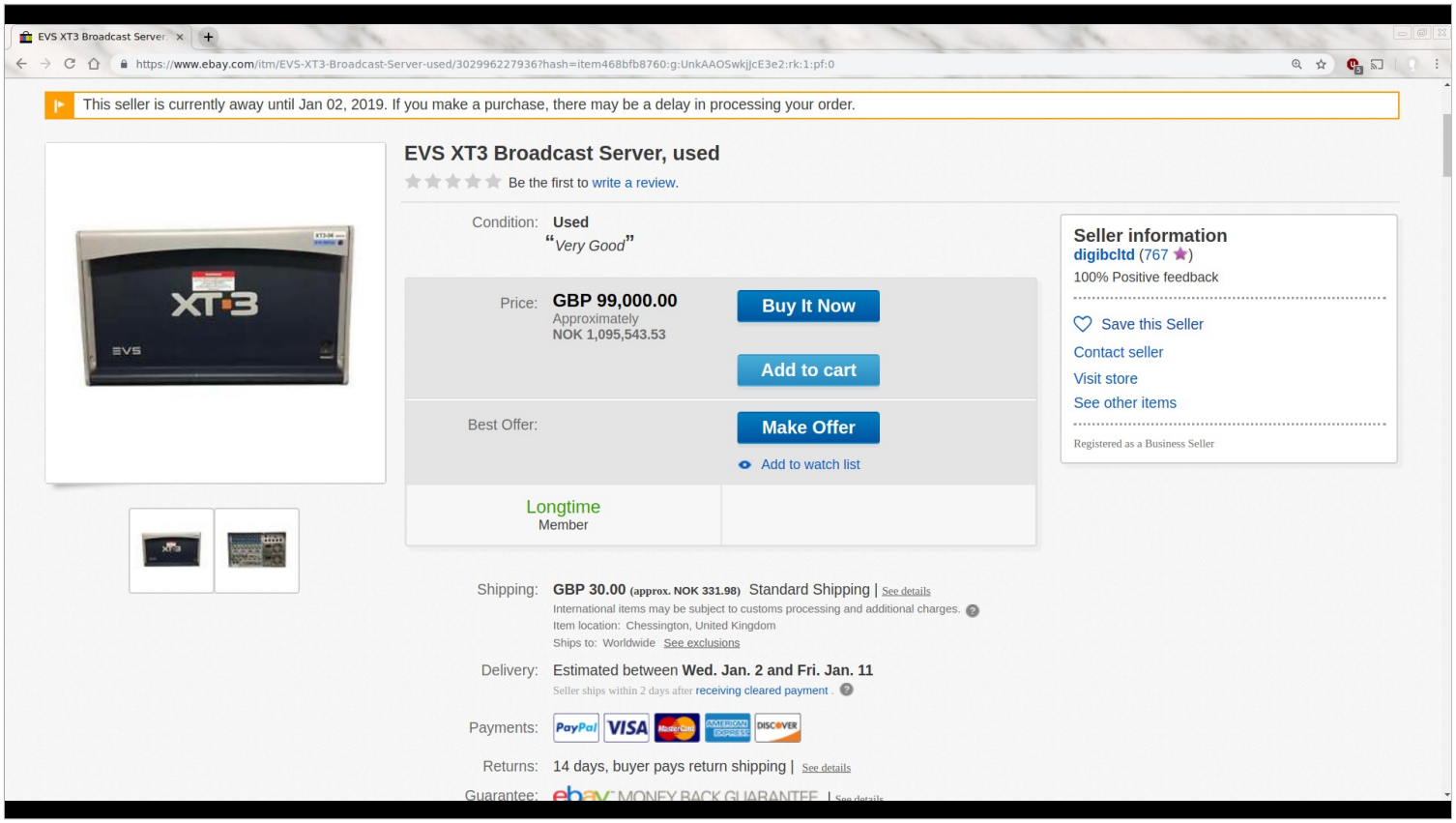
Finally, we have two goal cameras (3 and 4) for “beauty shots”—we generally don’t use them a lot during the actual point, but they can be used in-between. They are GoPros, mounted high up above each end zone on cheap, static tripod extenders. Depending on the venue, we also sometimes have an ambience cam for other fields, a camera on the commentators, or a smartphone as a mobile IP camera.

As a trick, we run these and their associated SDI converters on power banks—they last more than 12 hours, so we don’t have to run power to them.

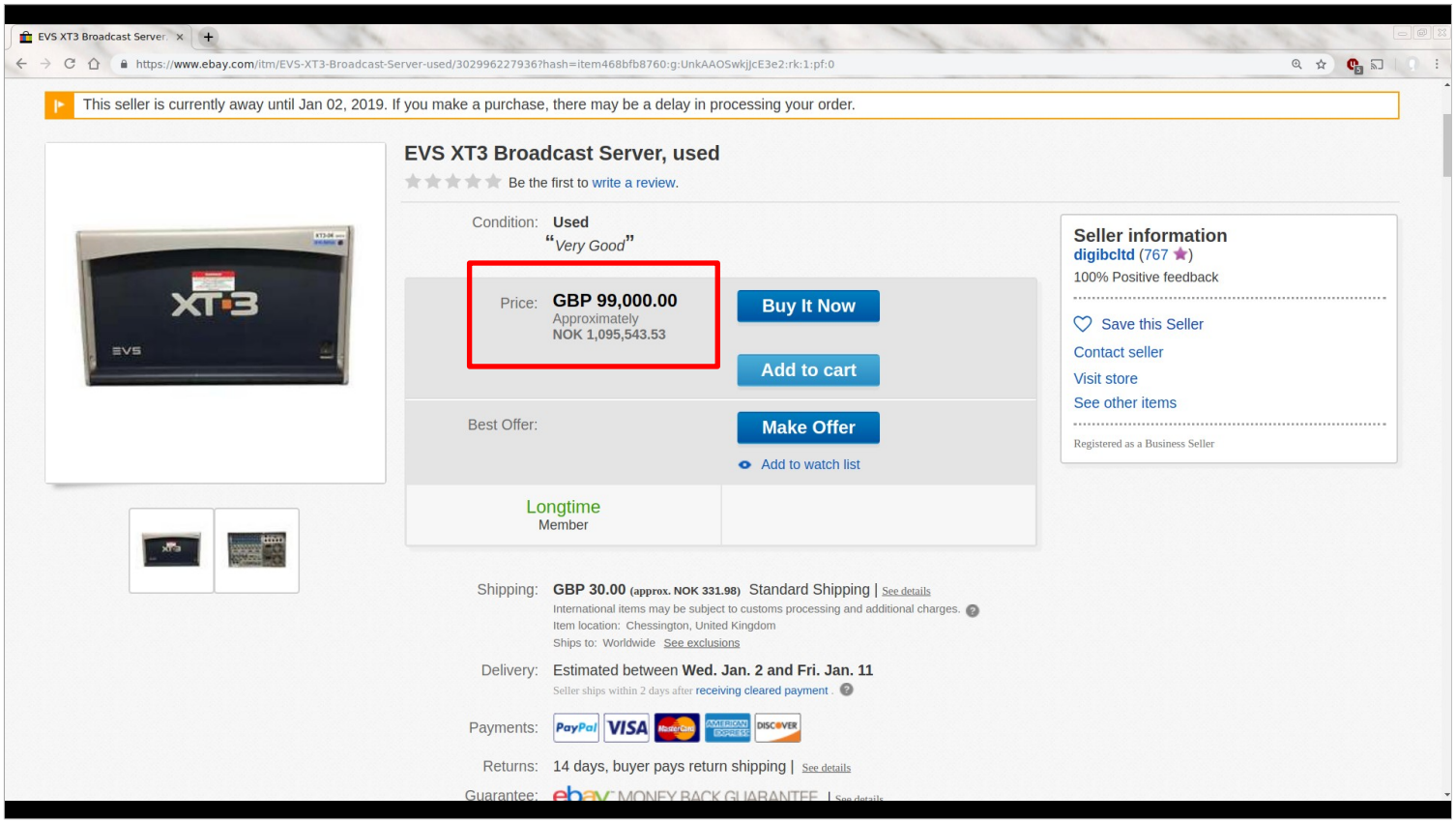
Most equipment is either stuff we had already, borrowed or bought second-hand on eBay, so it’s really a shoestring budget. And it’s nearly 100% free software, all the way to the JavaScript on the player page.

This is a video of what it looks like. Note the HTML5 graphics in the top left—the data is fed directly from the scoreboard over serial port. (If there's two fields, we show both scores.) We also pull data live from Google Sheets for lower thirds, realtime updated tables, and so on. And you can see the gray score line!

However, we soon noticed that it's not enough. People would make marginal catches or step really near the line, and even after all this work, we *still* don't know whether they were in or out. We needed some form of instant replay, and it needed to be in slow motion.

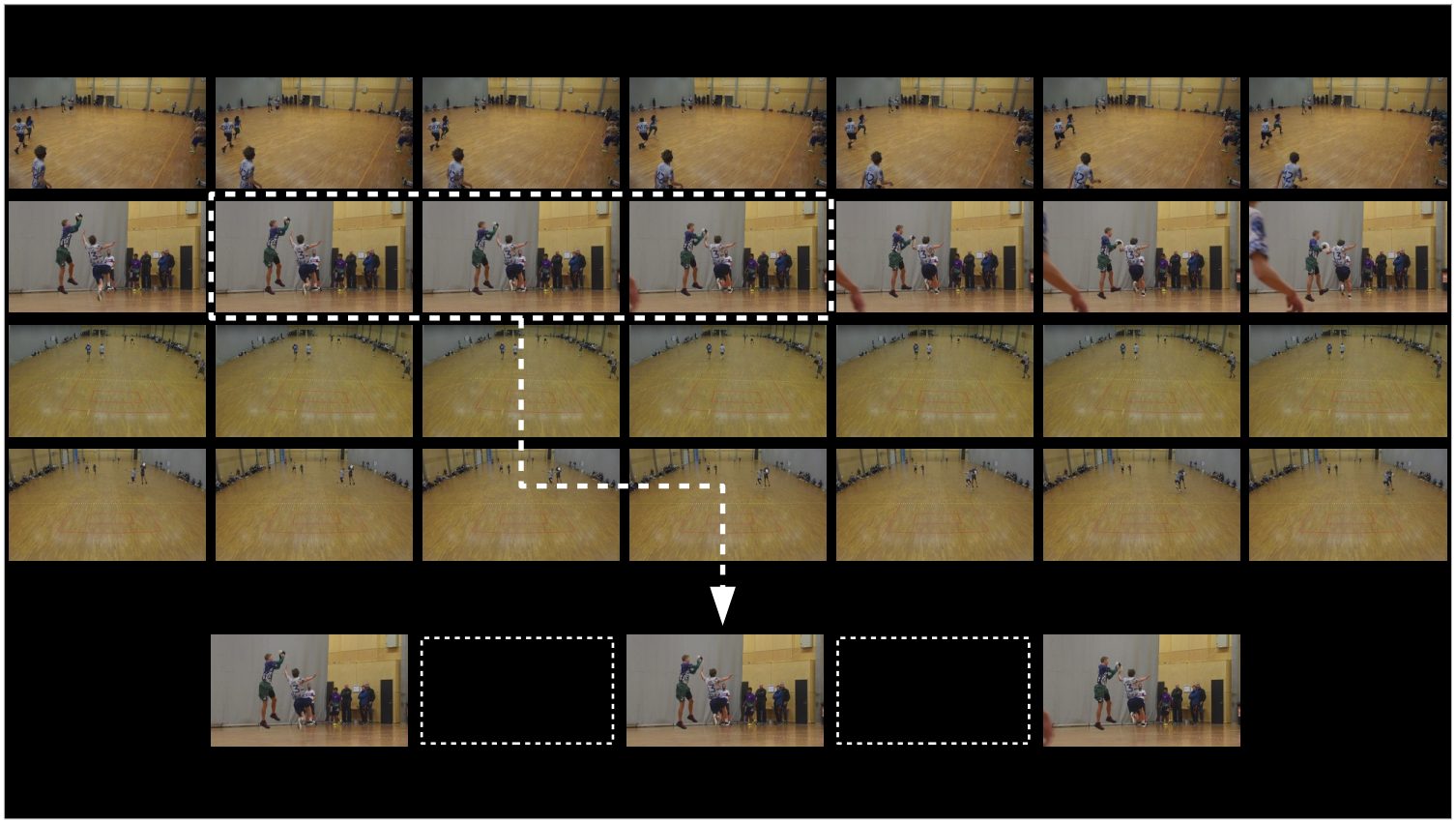


So we went online to see what a slow motion appliance would cost. This is the EVS XT3, which is the standard choice in broadcast. Pretty much every time you see a major sports event on TV, there's an XT3 in the chain. Or multiple ones.



Unfortunately, it's £99,000. Used. (Plus £30 in shipping to Norway. YMMV.) And to add insult to injury, the seller's on vacation. Even the remote control is thousands of Euros!

There are cheaper devices available, but you're generally in the €10k+ range. It's clear that we can't go down this route—we need a software-based solution.



Enter Futatabi. It works as a multi-track recorder—everything is recorded, all the time. It gets the frames from Nageru over a standard network; I've chosen MJPEG as the codec, as it's reasonably high-quality and has hardware encoders/decoders in recent Intel GPUs. We have roughly 75 Mbit/sec/camera in 720p, or 125 GB/hour for four cameras.

You can then make clips, put the into playlists and play them back, again over TCP/IP. A nice bonus is that if we don't want to modify the frames, we can echo them right back over the socket, byte for byte, giving the best possible quality *and* speed. (Futatabi works to maximize the usage of these frames, but if you're playing back at non-integral rates, e.g. ramping, it's not always possible.)

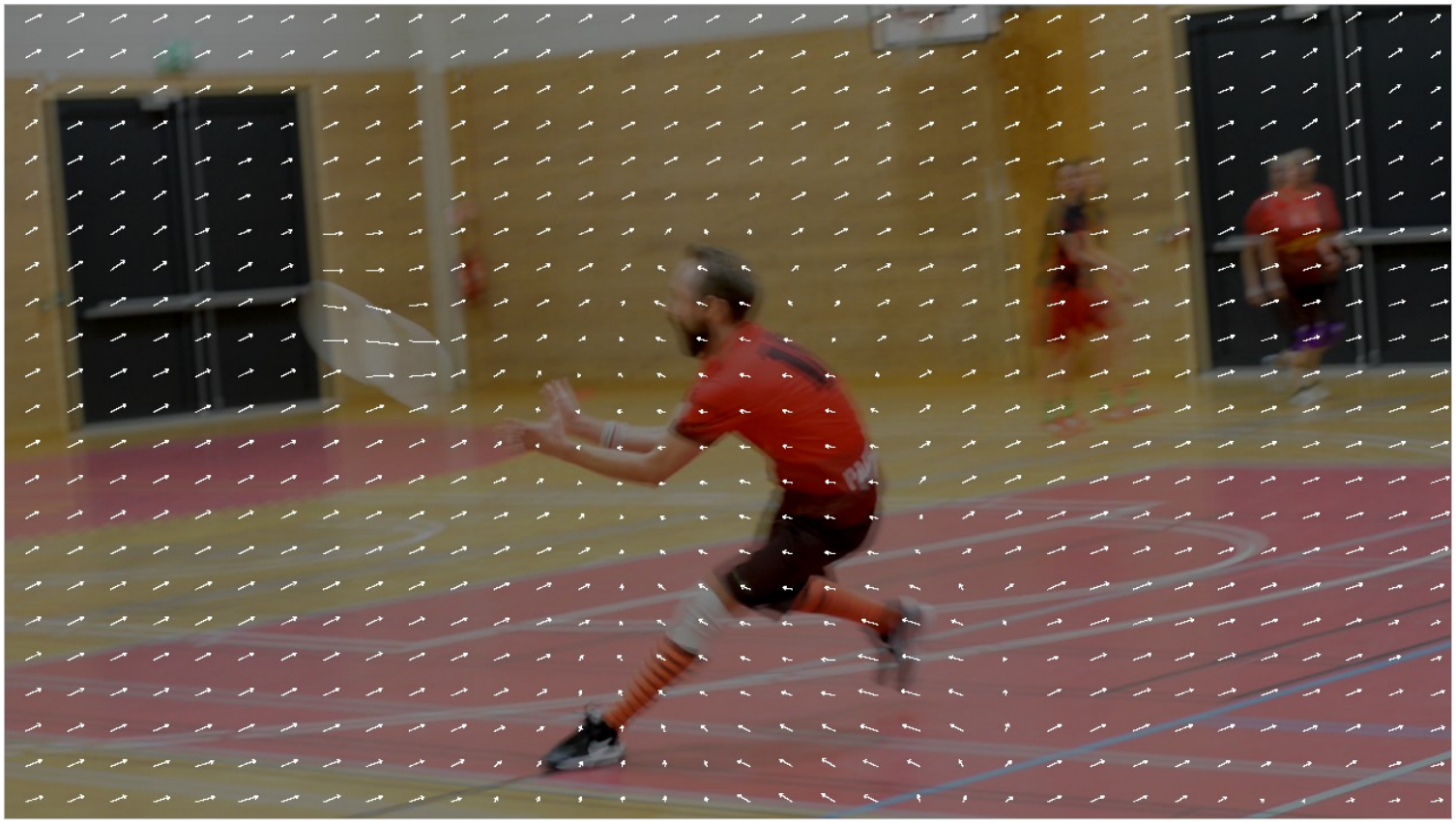
But when we slow down, how do we fill the gaps between the frames? Let's look at the alternatives.

We can try to just repeat each frame the required amount of times, but the result isn’t so pretty—it’s stuttering visibly. (If your PDF viewer supports video, you can click to play it. But not on the notes page—you’ll need to go to the slides PDF. Sorry.)

Note that I’ve purposefully chosen a challenging sample here; there’s lots of motion, and I’m also using 0.25x (“super-slow”) instead of the regular 0.5x. There are two reasons: First of all, it’s a lot more interesting to look at something that goes wrong—who cares about the easy case? Second, if you’re not using to look at these various forms of artifacts, it’s good if they are clear. Also, the FOSDEM stream is downconverted to 25 fps, so it’s for the stream viewers, too.

Fading between each frame isn’t much better. It’s just as choppy, with some extra blur to boot.

To illustrate, I’ve overlaid two consecutive frames from the previous video. It’s obvious that creating good in-between frames somehow has to relate to motion.



This brings us to the basic idea of *optical flow*. For every pixel in the first frame, we can try to estimate where it moved in the second one. It's not a perfect model—perhaps it went to nowhere (occlusion), and with things like blur, maybe half of it went to one place and half of it went to somewhere else. But it's a fairly good model, if we can make a good estimate.

In this example, most of the vectors are slightly up and to the right, since that's how the camera moved. But the disc is moving faster to the right, and the player is moving to the left.

So once we have a flow field between the two frames, we can halve and then somehow invert it. (We won't be discussing this step in detail—it's a bit trickier than it looks at first sight.) This would give us, for the intermediate frame, where to fetch each pixel *from*, which is more directly useful than where it wants to *go*.

Note that this assumes near-linear motion, but for high enough frame rates, it's a fairly good approximation.

Fast Optical Flow using Dense Inverse Search

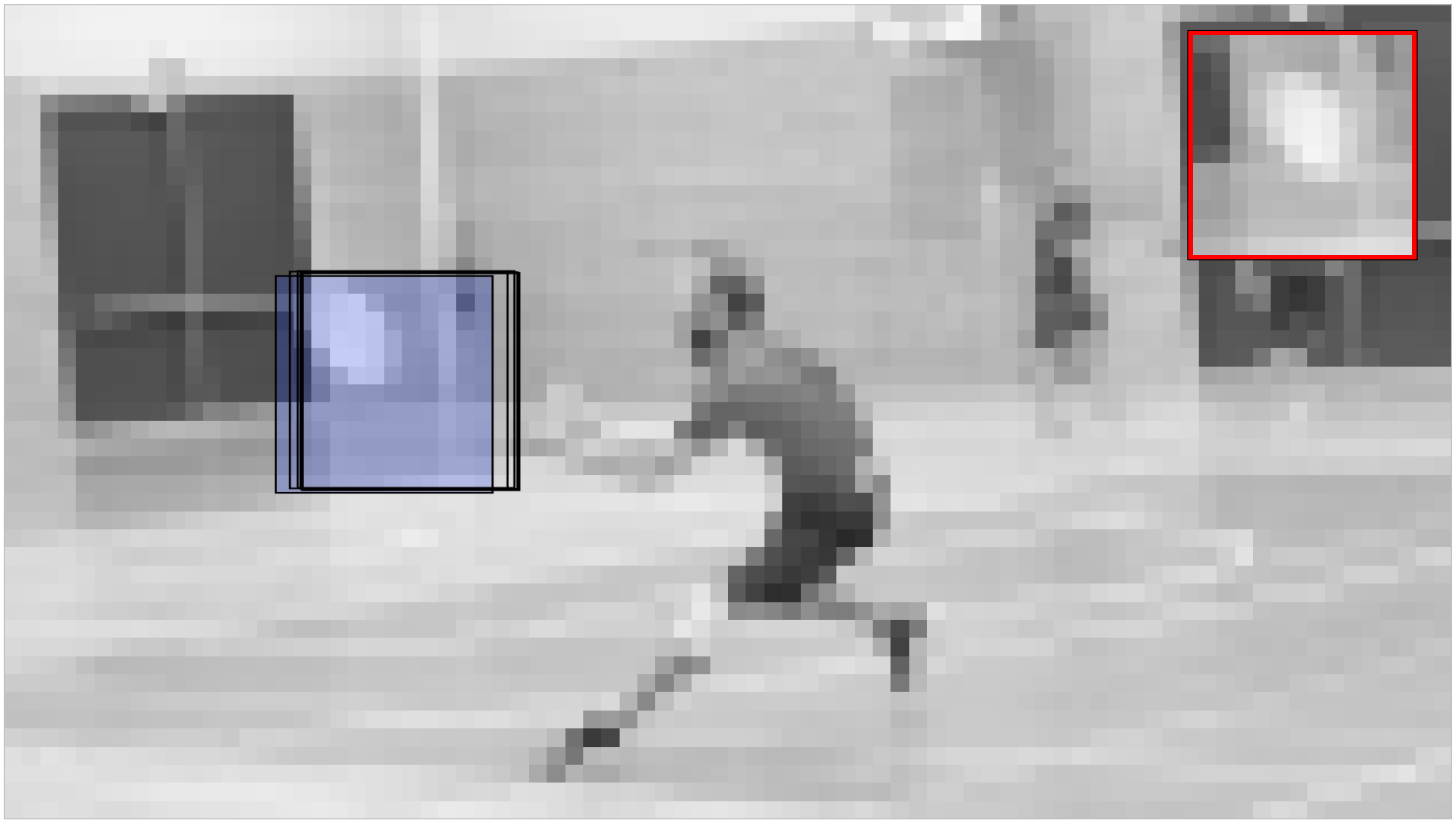
Till Kroeger[1] Radu Timofte[1] Dengxin Dai[1] Luc Van Gool[1,2]

[1]Computer Vision Laboratory, D-ITET, ETH Zurich
[2]VISICS / iMinds, ESAT, KU Leuven
{kroegert, timofter, dai, vangool}@vision.ee.ethz.ch

Abstract. Most recent works in optical flow extraction focus on the accuracy and neglect the time complexity. However, in real-life visual applications, such as tracking, activity detection and recognition, the time complexity is critical. We propose a solution with very low time complexity and competitive accuracy for the computation of dense optical flow. It consists of three parts: 1) inverse search for patch correspondences; 2) dense displacement field creation through patch aggregation along multiple scales; 3) variational refinement. At the core of our *Dense Inverse Search*-based method (DIS) is the efficient search of correspondences inspired by the inverse compositional image alignment proposed by Baker and Matthews [1, 2]. DIS is competitive on standard optical flow benchmarks. DIS runs at 300Hz up to 600Hz on a single CPU core[1], reaching the temporal resolution of human's biological vision system [3]. It is order(s) of magnitude faster than state-of-the-art methods in the same range of accuracy, making DIS ideal for real-time applications.

There are more than 200 different papers about optical flow! (There are also lots of other ways to interpolate frames; in particular, deep learning methods are rapidly becoming popular.) I read through a bunch of the most promising one. Unfortunately, *realtime* optical flow isn't of all that much interest in academia, but I found one that showed a lot of promise. It talks about 300–600 fps on a single CPU core, and it comes with reference code.

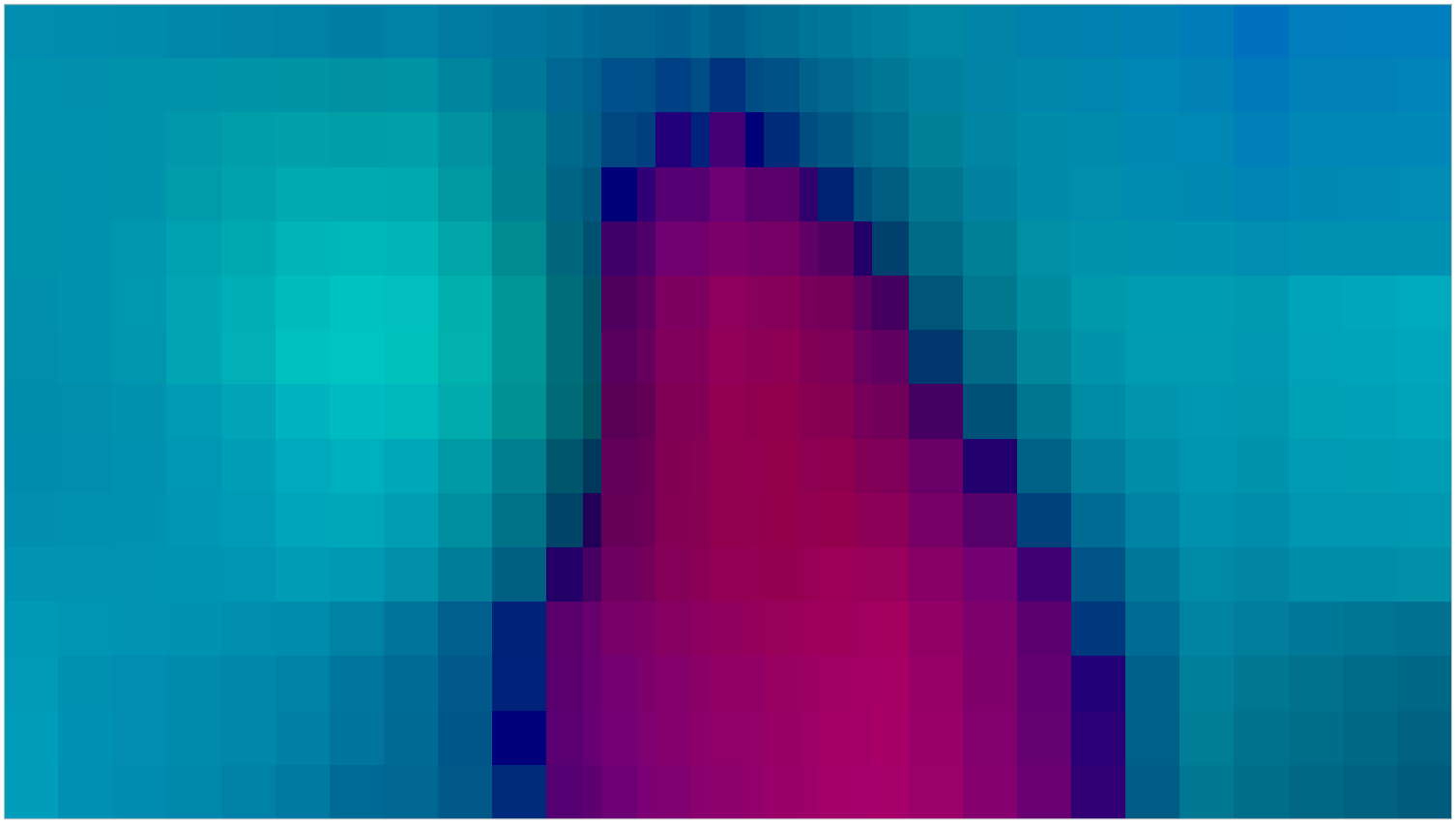
Now, 300–600 fps isn't really what you get in practice; on reasonable settings, it's more like 10 fps. I ended up making a GPU reimplementation from scratch, also for license reasons (the reference code was a bit unclear on licensing).



Let’s go through the algorithm on a high level. We start with a motion search on low resolution; divide up the image in a series of overlapping blocks, and search for a similar block in the other image.

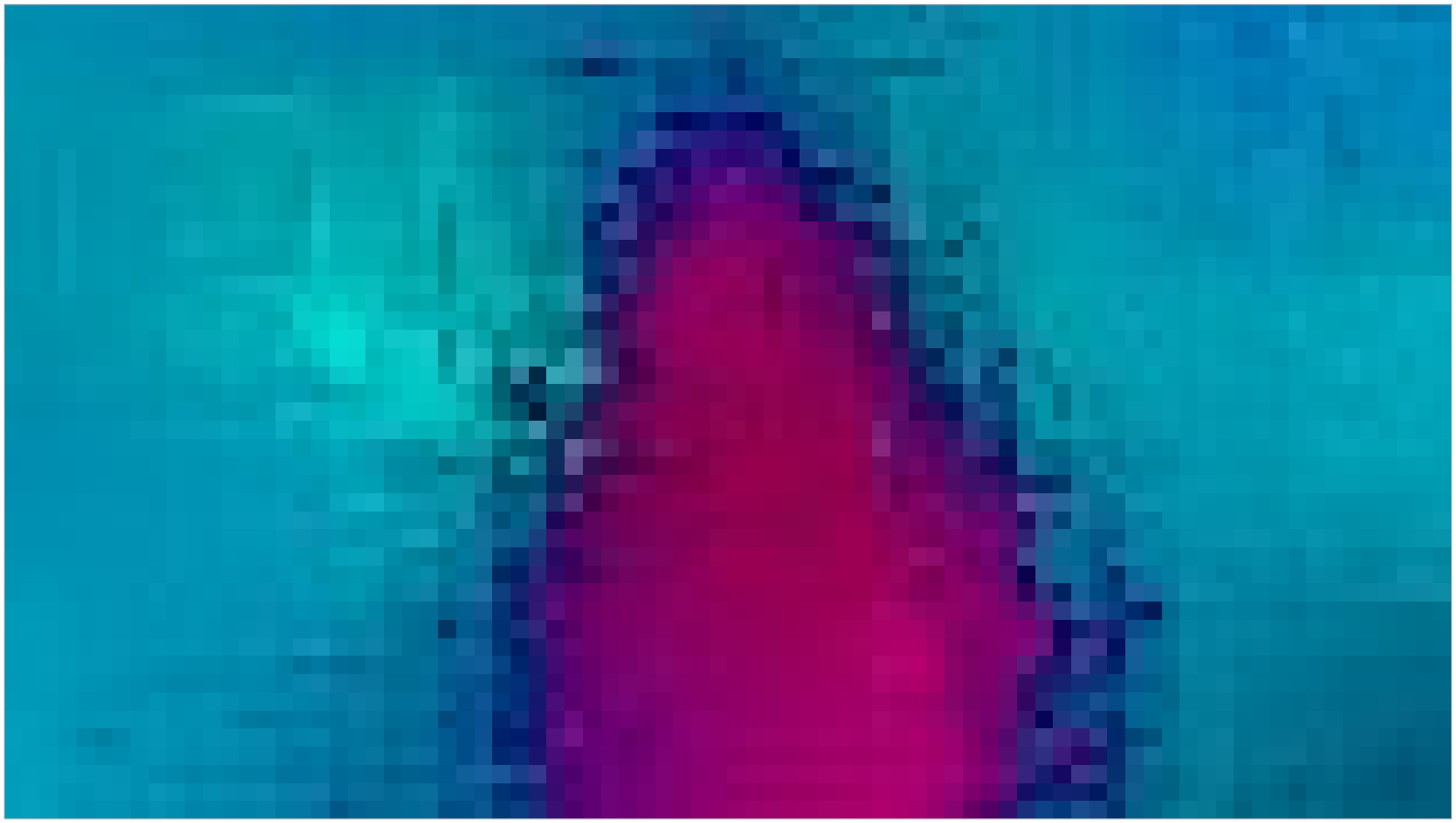
This motion search isn’t quite what you’d see in a video codec—in particular, we’re interested in *real* motion, not the best mathematical match. The DIS paper uses gradient descent: We start with a zero motion vector and then try to move in the direction of the right change. Assuming the motion is small compared to the resolution (which it is!), we have a reasonable chance of finding the right vector.

Note that we’re not drawing the motion vectors as arrows anymore, since it’s hard to read lots of tiny arrows. Instead, we’re using the hue to denote the angle, and the lightness to show the length. In this case, the flow vector is a bit to the right.

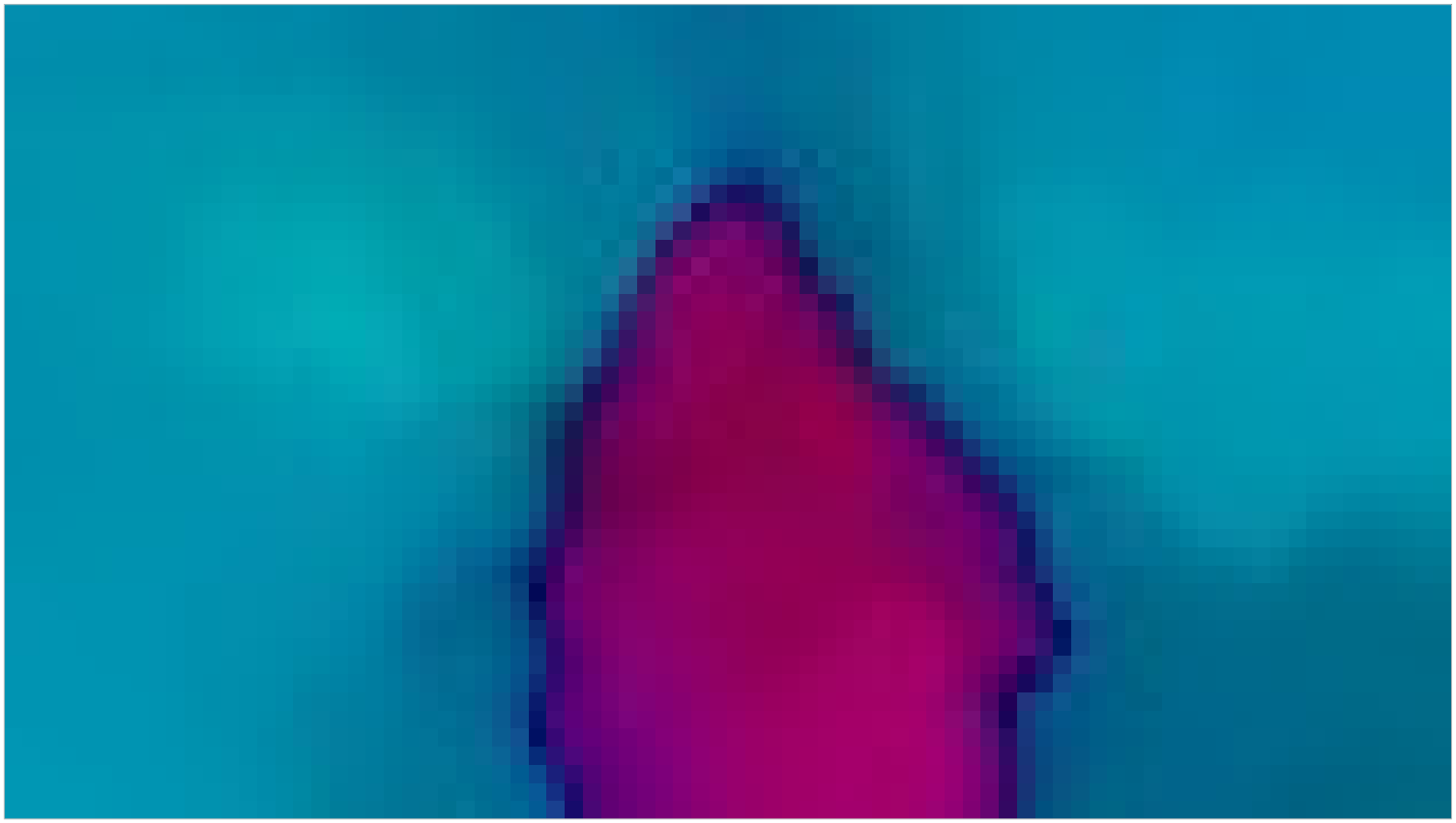


After motion search over all the blocks, we have our initial field. Here the overlapping blocks are all weighted the same.

Note that I’ve brightened up the image of the flow field a bit for benefit of the projector.



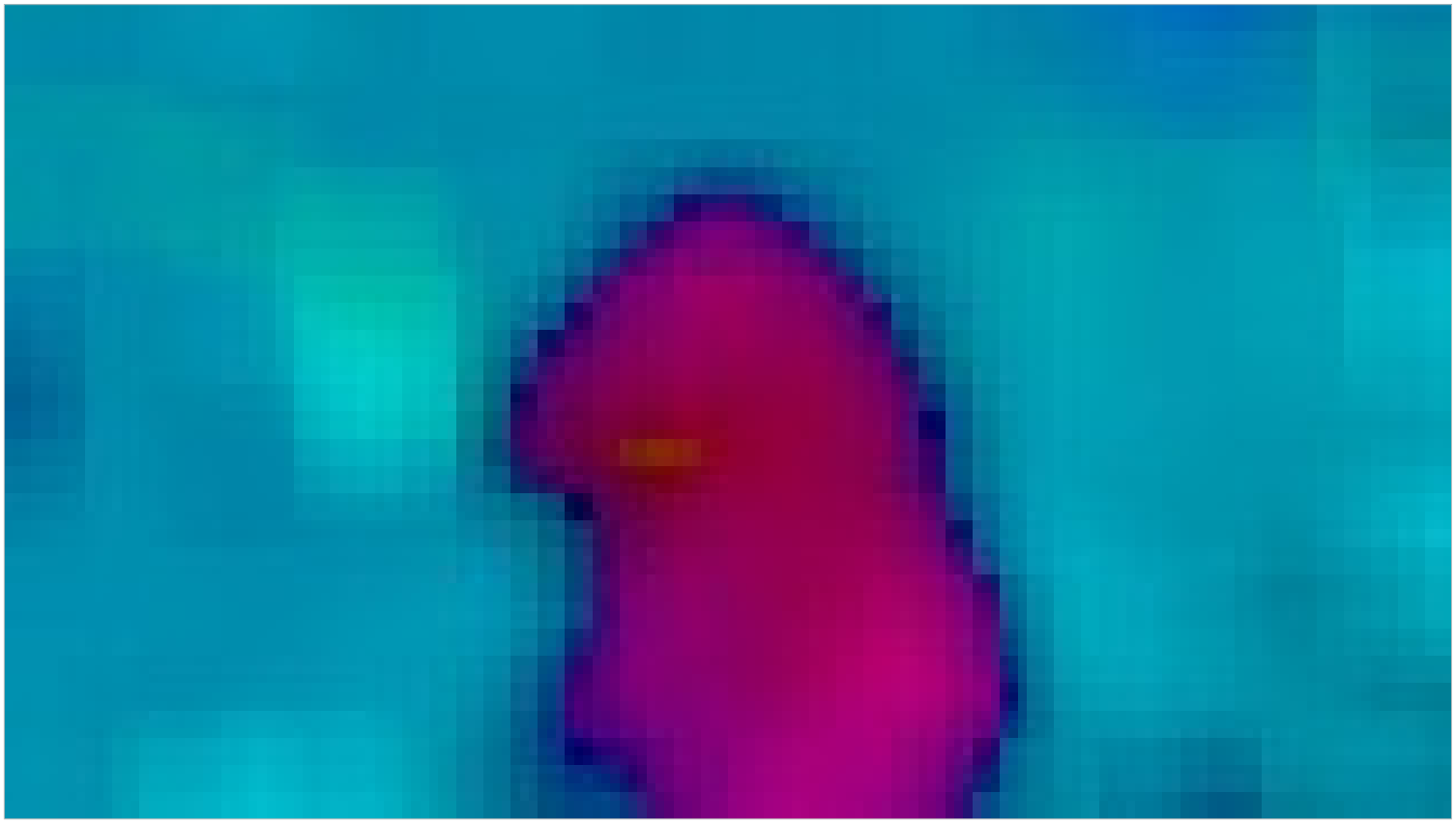
Now we apply a process called *densification*. Instead of weighting all the motion vectors (or hypotheses) equally, we look at how well they actually match for each single pixel. It looks a bit noisy now, but it will soon get better.



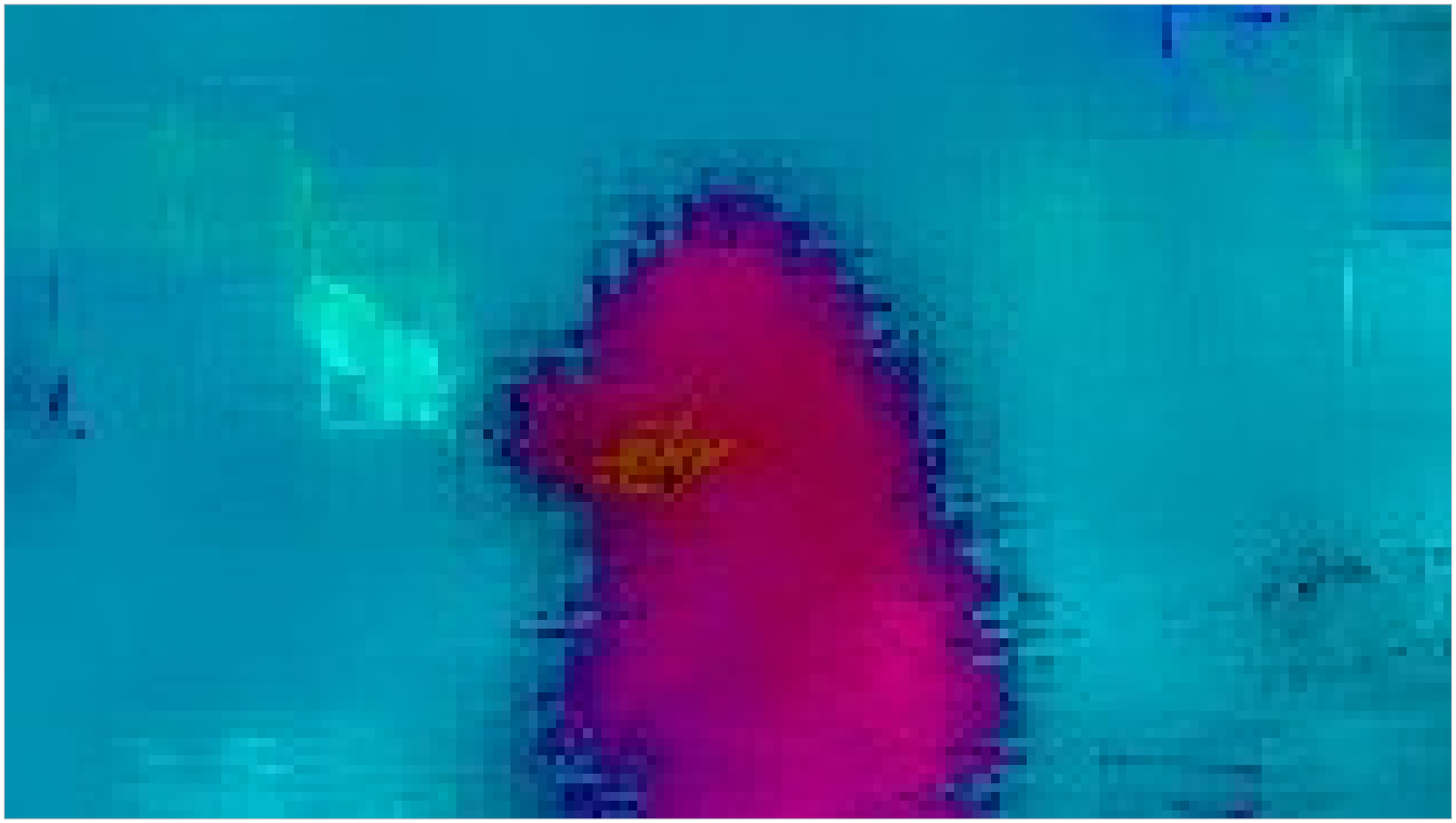
The final step is what the paper calls *variational refinement*. It essentially amounts to setting up a nonlinear differential equation and solving it numerically, and ends up cleaning up the field a lot.

Unfortunately, we won’t have time to go through the details, which is a shame, since it’s a very fascinating and powerful technique. (If you’re interested, I’ve made a full tutorial and put in the source code.)

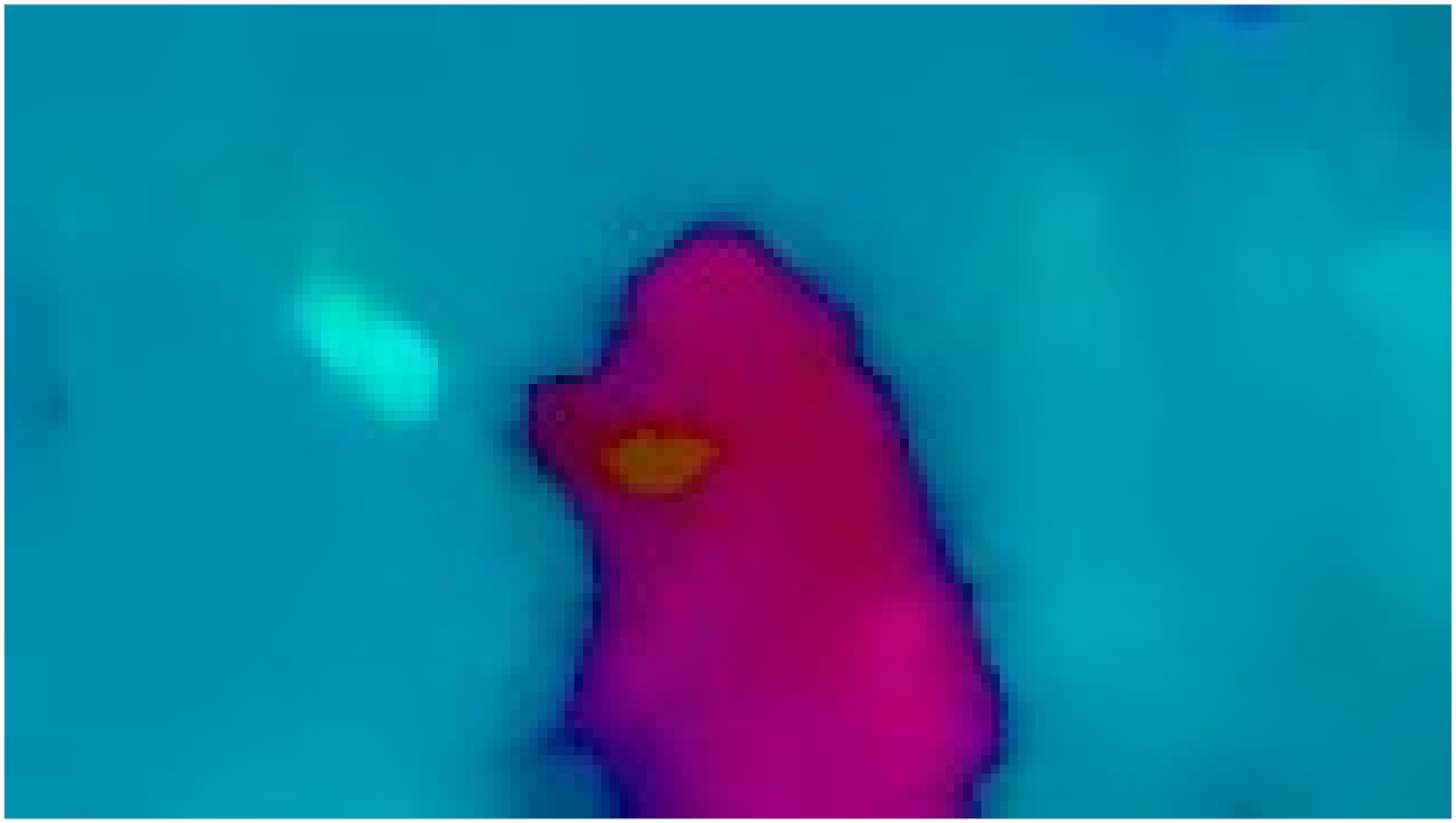
We could have stopped here if we wanted to, and just scaled up the flow field. However, we can do better.



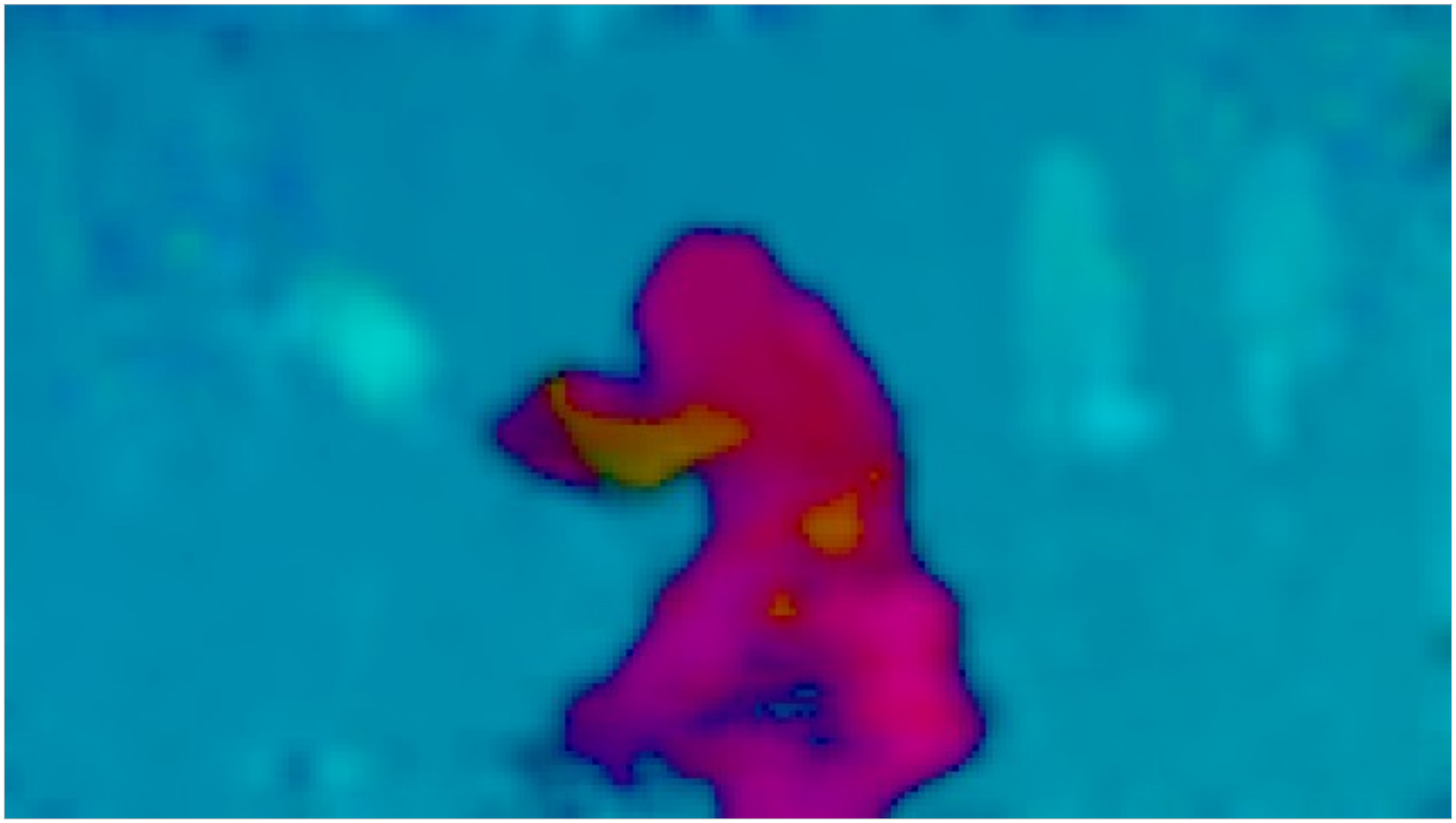
Now we double the resolution, and start over again. However, this time, we start our motion search at the previous flow field. This means that even though the motion is now much larger compared to our block size, we have a good initial hypothesis and still have a good shot at finding the actual motion (now with higher precision).



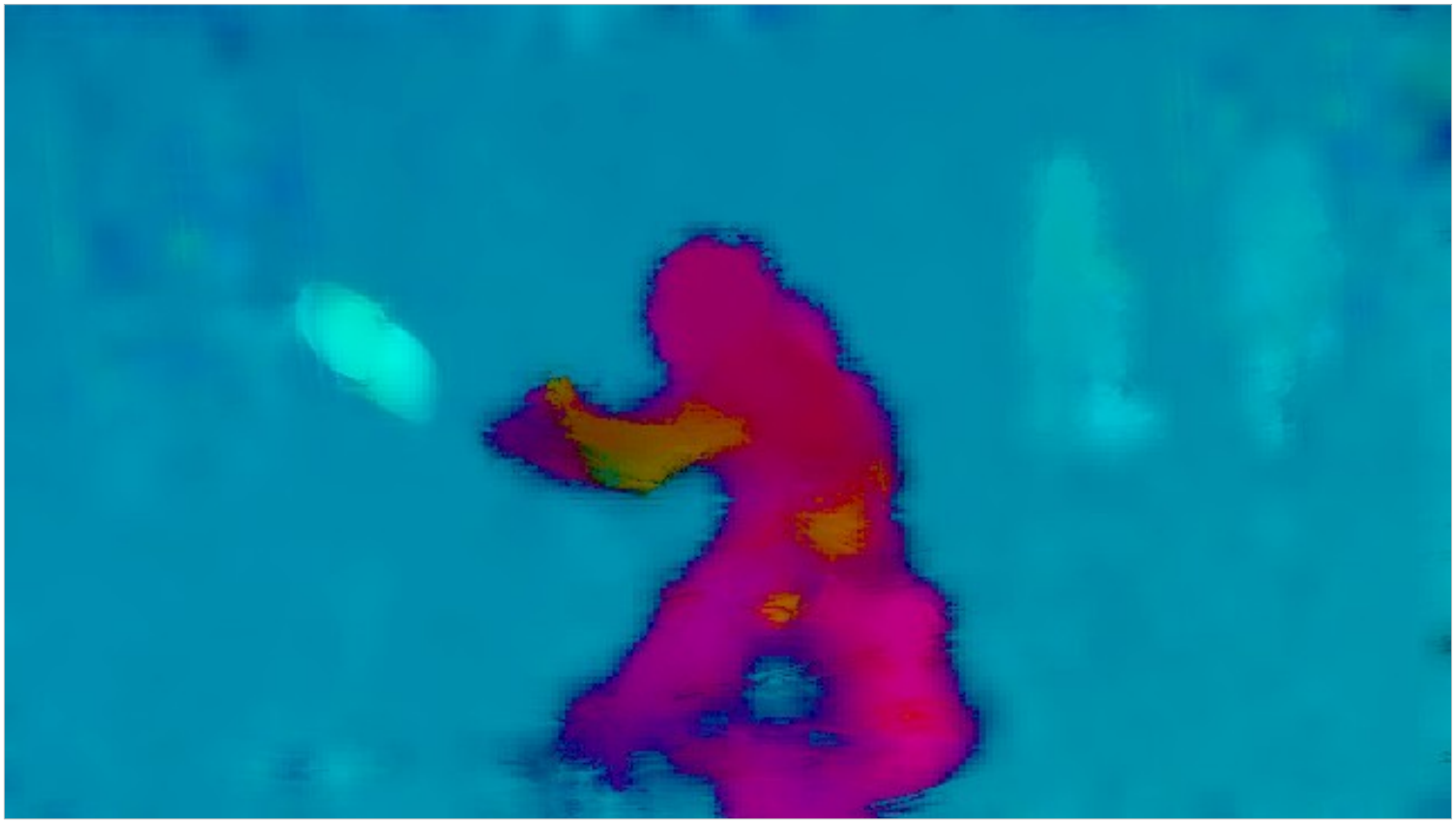
Densification again.



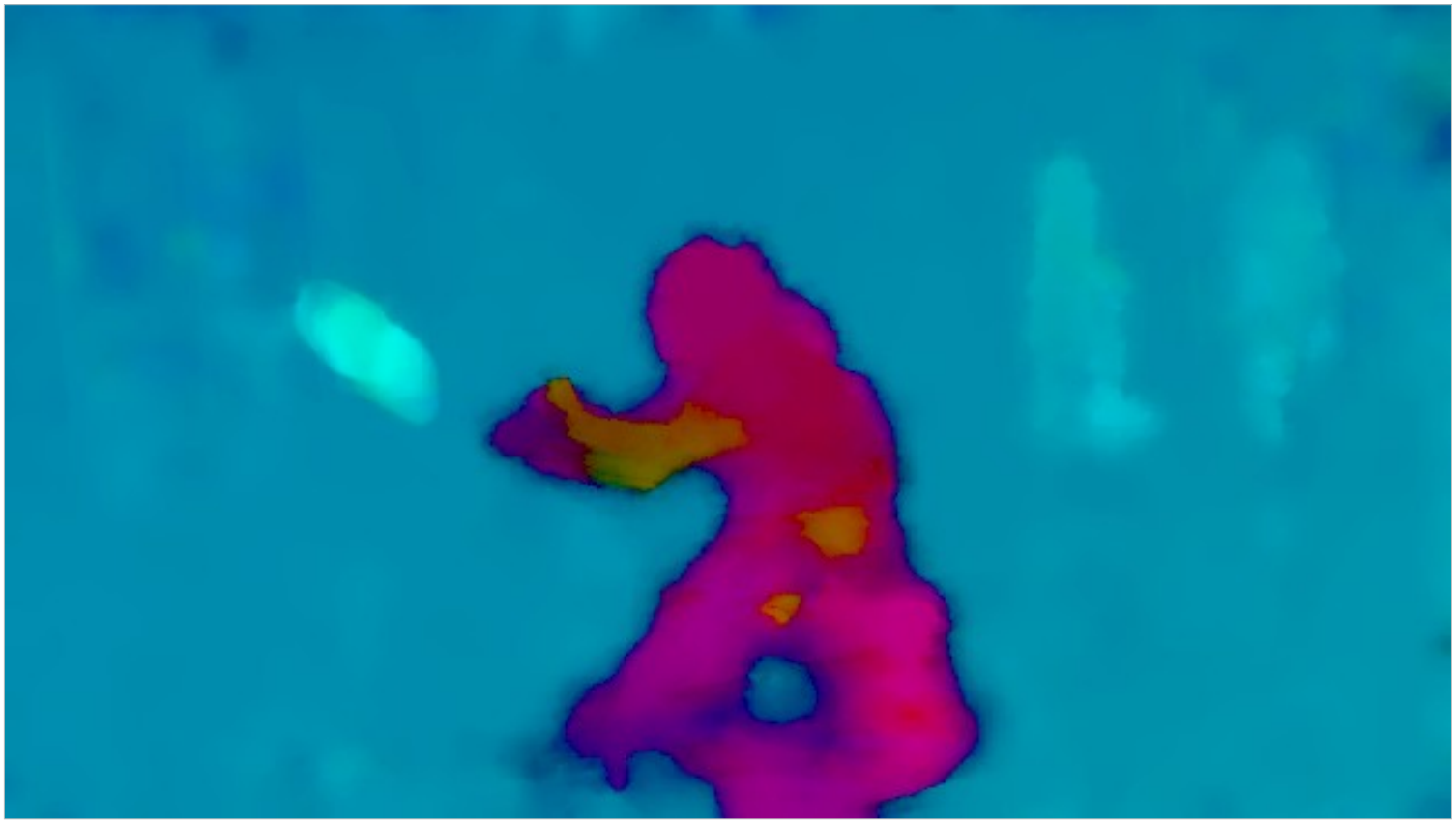
And variational refinement. Note that the disc is starting to show.



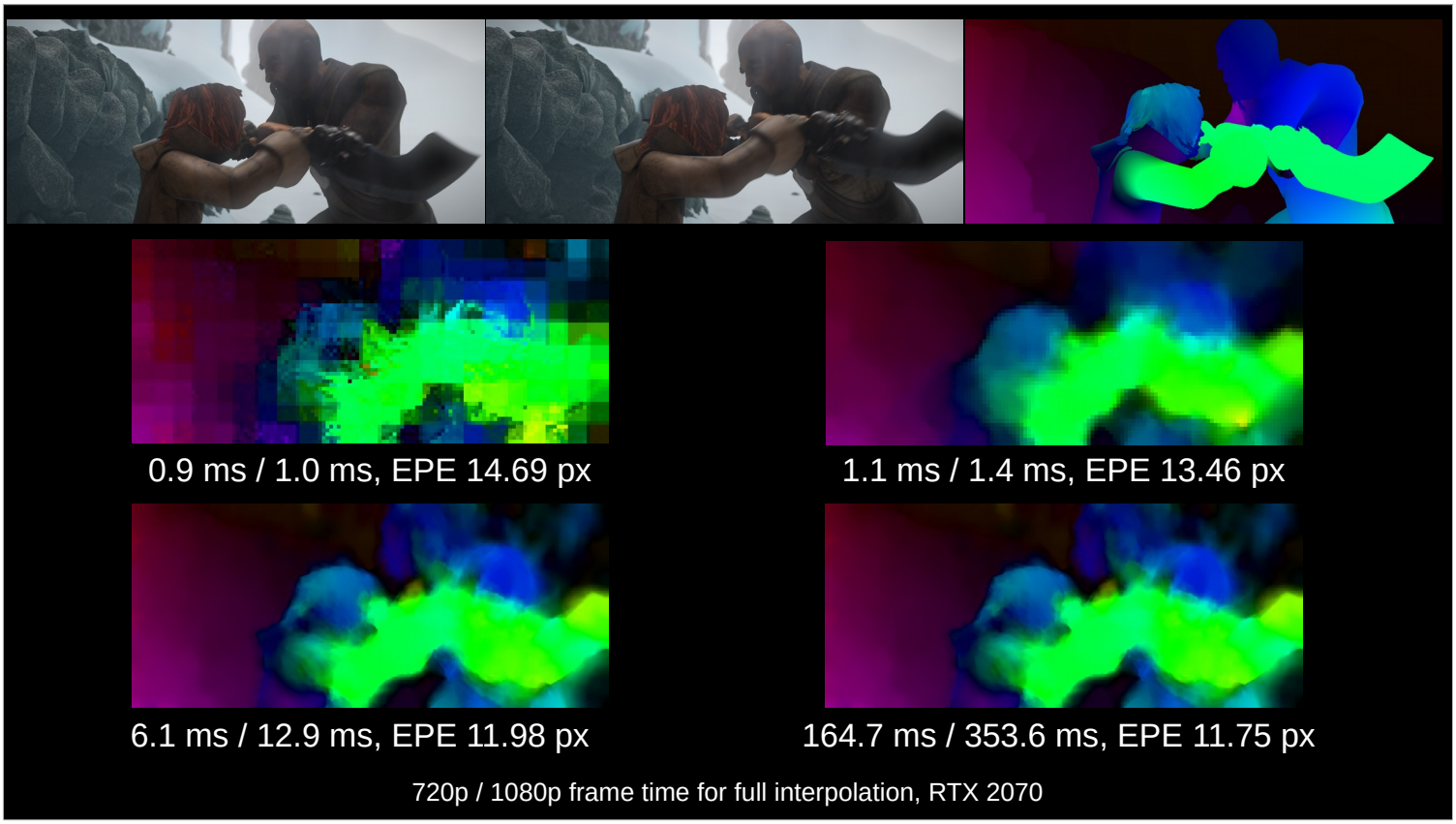
One more step of motion search.



Densification.



And finally, variational refinement.



So, how good is our estimated flow? This is is a test sequence from the so-called MPI-Sintel benchmark (using Sintel from the Blender Foundation). Since it's synthetically rendered, we have exact motion vectors for every pixel, and can compute the endpoint error (EPE) over the entire 50 frames.

These are the four quality presets recommended by the paper (1–4). We can see that even at 1080p, we are realtime at quality 3 at 60 fps; note that this is evaluating both forward and backward flow (in parallel), computing the intermediate flow, and doing the actual interpolation. We beat the reference code by more than an order of magnitude in performance, and a few percent in EPE.

Both visually and from the EPE numbers, we are clearly far from perfection. But is it good enough for reasonable interpolation?

This is a video of those two frames interpolated very, very slowly. You can see there are some issues around occlusion, but overall, it’s more than decent. Remember, at 2x, these artifacts will blur past really quickly.

(This slide purposefully left blank)

And here’s our test clip again, in 4x interpolation at quality 3. There are some errors if you look closely, but overall, this is something we can live with.

Demo!

It’s time to show the actual application, and what better way to do so than a demo. Do note that it’s really made for 1080p—I’ve moved a few controls and such around to fit the projector’s 1024x768.

We’re going to be running with input from file, since I don’t have four cameras and a Nageru machine with me. There’s twelve hours of four-camera test data readily available from the Futatabi home page, so you can download it and play with it yourself without setting up a large rig.

Thank you!

https://nageru.sesse.net/

That’s it. Futatabi is part of the Nageru distribution, since they share a fair amount of code, so that’s also where you want to go to download it.

While we’re doing Q&A, I’m going to keep playing a highlight reel in the background. This is a set of real, unedited replays made during the tournament and rendered out at our realtime quality and speed, with mistakes and all. So it’s very much a realistic example of what your finished video would look like. (I’m cutting it from the PDF in the interest of keeping the size down a bit; you can view it on YouTube).

Thanks for listening! Or reading the slide set, as appropriate.