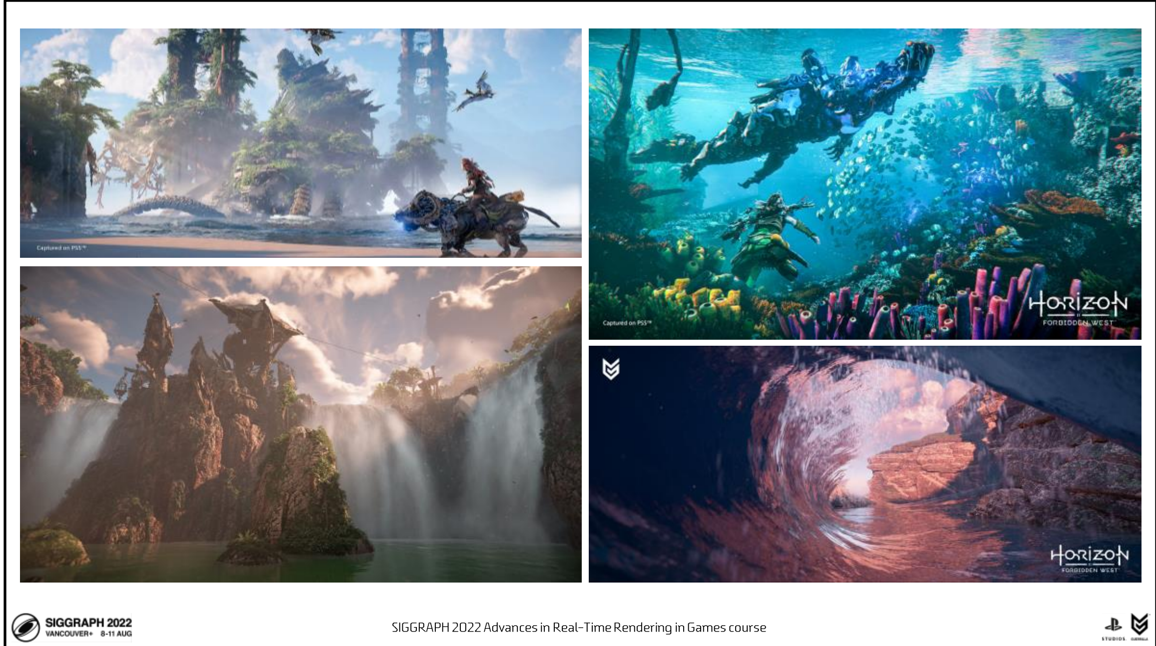
Rendering Water in Horizon Forbidden West

Hugh Malan / Guerrilla



Hi, my name is Hugh Malan and I'm a Senior Principal Tech programmer at Guerrilla.

I'm going to talk about how water is rendered in Horizon Forbidden West. The big new feature is breaking waves, as shown in the bottom right screenshot: that is, waves with an overhanging shape, so that's going to be the focus of the presentation.

There's a lot of other features in the water system such as the spectacular waterfalls and underwater effects, but unfortunately we aren't going to be able to cover them here.

Just in case you’re not familiar with the water in Horizon Forbidden West, here’s a trailer.

Things started back on December 20, 2017 with the art vision presentation by the art director Misja Baas. This was the image for the water.

Getting the render quality anywhere close to this in a PS4/PS5 game would be tough.

But on the sim side, this is not plausible for realtime. And it doesn't need to be. These waves should be driven by an baked offline sim.

Animating water with baked data isn't a new idea: for instance, the ocean waves in Killzone 3 used streamed mesh data. Likewise, the sea in COD WWII was driven by a looping Houdini sim.

We needed something more flexible though, because we needed waves to fit a long and complicated coastline. We also have rivers, lakes, streams and waterfalls with standing waves, eddies, whirlpools and other effects, and need solutions for them as well.

So we took a hybrid approach. We start with a Houdini sim of a localized area, bake it down for realtime use. At runtime we'll instance that data, it'll be played back and

blended to create the rendered water surface. The idea is that we get the realistic 3D movement of the offline Houdini sim, but can bend it to any shape we need.

The “waveparticles” paper by Cem Yuksel (http://www.cemyuksel.com/research/waveparticles/) was a good inspiration for this approach.

So how can we bake a breaking wave down to a small amount of data, that’s fast and flexible for what we need at runtime?

[Play video]

This wave illustrates the approximation we use. The curling and breaking part of this wave looks a lot like it's a constant shape translating along the wavefront. This means the cross section of any part of the wave at any time comes from a single animated wave cross section.

So the idea is that we need to bake out a single animated cross section of the breaking wave, and then provide good controls for how it's defined for each part of the wavefront and how it changes over time, and that will give us the base wave shape.

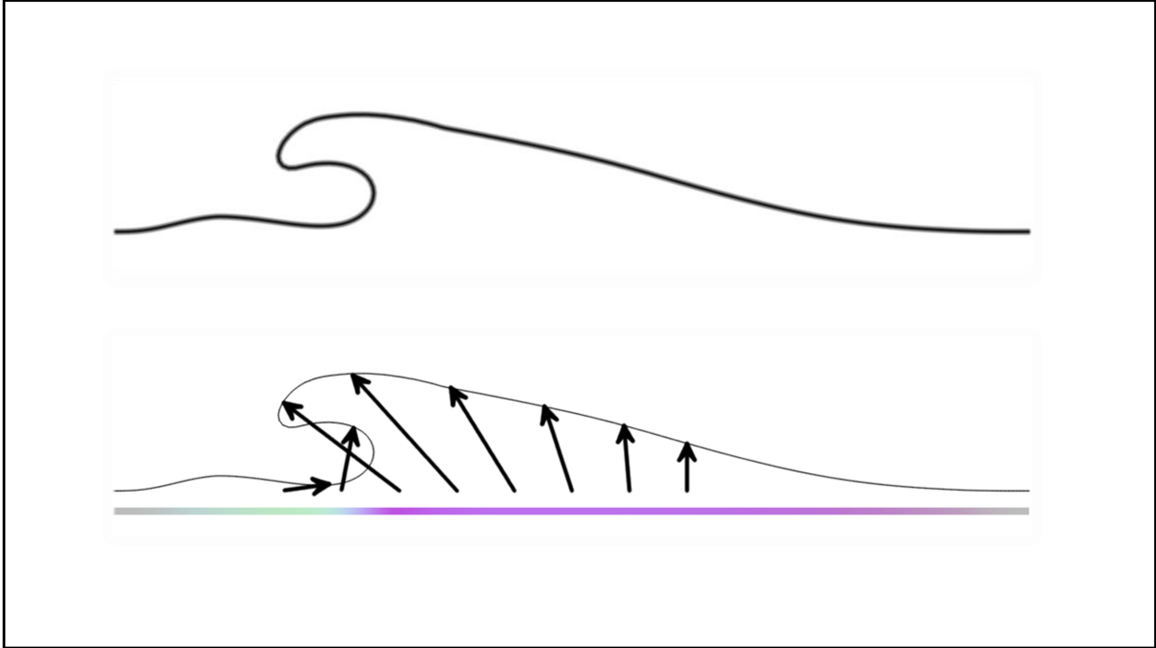
Building the deformation

Offline bake process

SIGGRAPH 2022 VANCOUVER+ 8-11 AUG

SIGGRAPH 2022 Advances in Real-Time Rendering in Games course

First of all, here's how the offline process to bake out the cross section worked. I'll talk about the runtime part later.

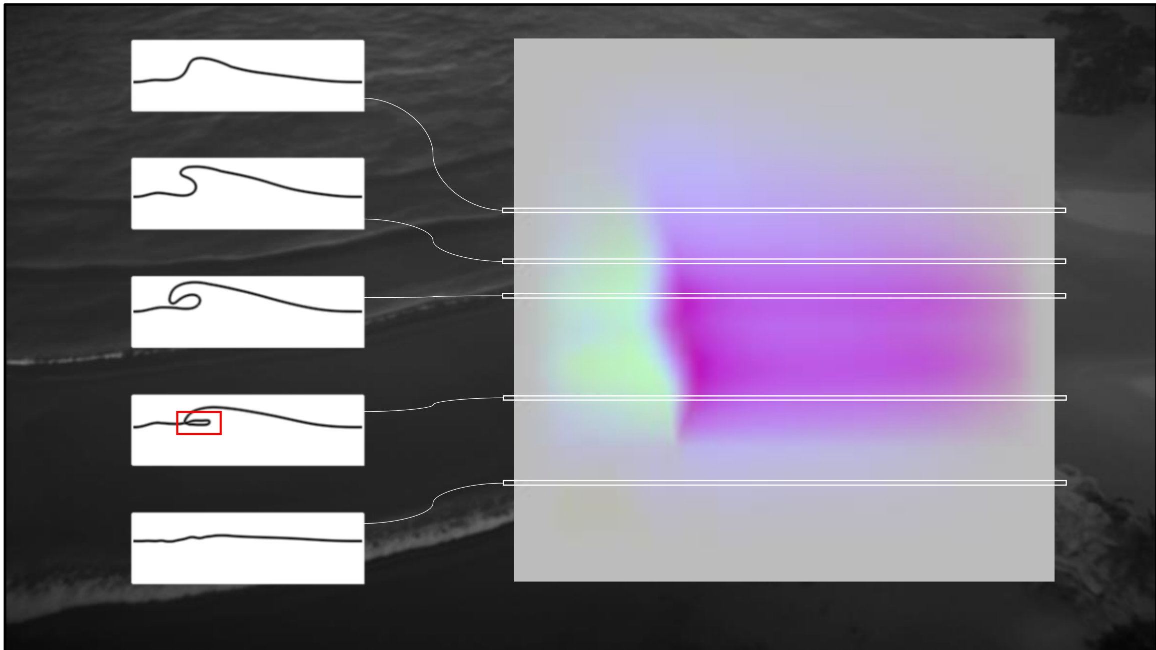


We began by doing a Houdini sim of a breaking wave. From that we extracted a cross-section for each frame. The upper image is an example.

The cross section line is expressed as a deformation from a flat surface, as shown by the arrows in the lower image. The XYZ offset is converted into RGB, the thin green and purple strip underneath shows the result.

[Play video]

Here’s how the full wave animation looks.



We repeat the process for all the frames of the animation. Each frame gave us a horizontal strip of pixels. They're assembled into a vertical stack to form a single 2D texture for the whole animation, as shown here. A couple of example frames are shown here, with the corresponding row of the deformation texture outlined.

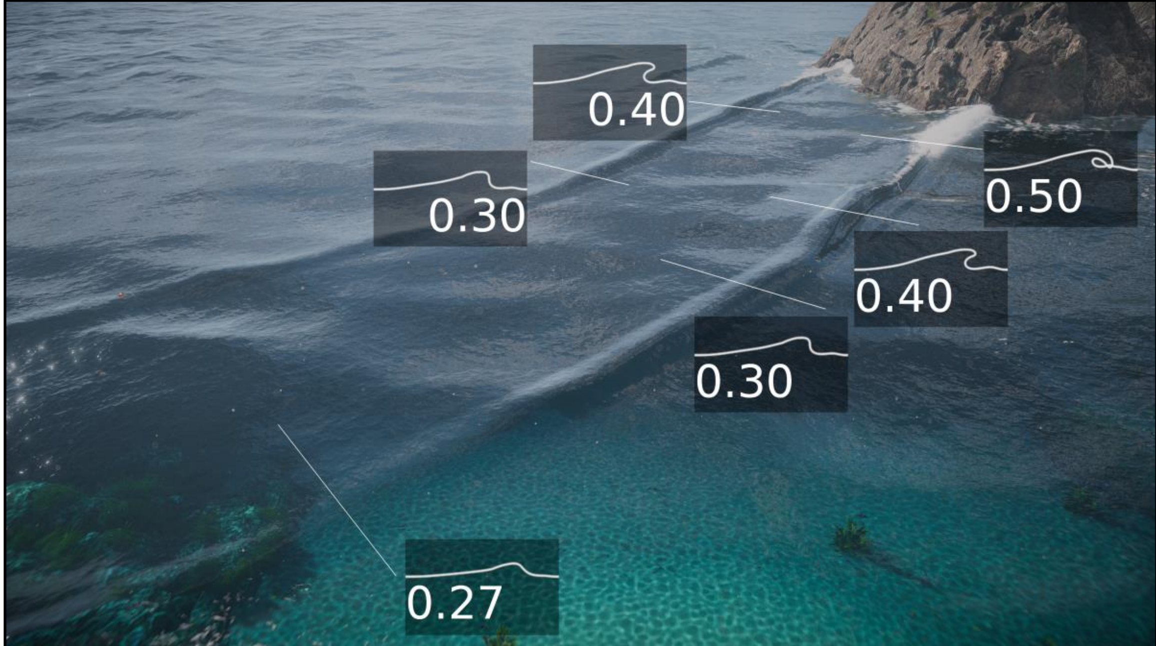
Setting up the animation of the cross section was challenging. We started with one extracted from a Houdini sim of a crashing wave in very thin wave tank, but it needed so much cleanup and tweaking that we ended up using a hand-animated curve. For example, we need the animation to end at a flat undeformed surface, so the deformation can be removed without artifacts.

[Click]
Note that when the wave arches over we need still need the inner loop, even though this is complicated and massively increases the length of the cross-section curve and magnifies the size of the triangles. It's needed because you can look into the tube of the breaking wave, and see the surface corresponding to that inner loop.

Applying the deformation

Runtime

Now let's look at how the deformation data is used at runtime to create the breaking wave.



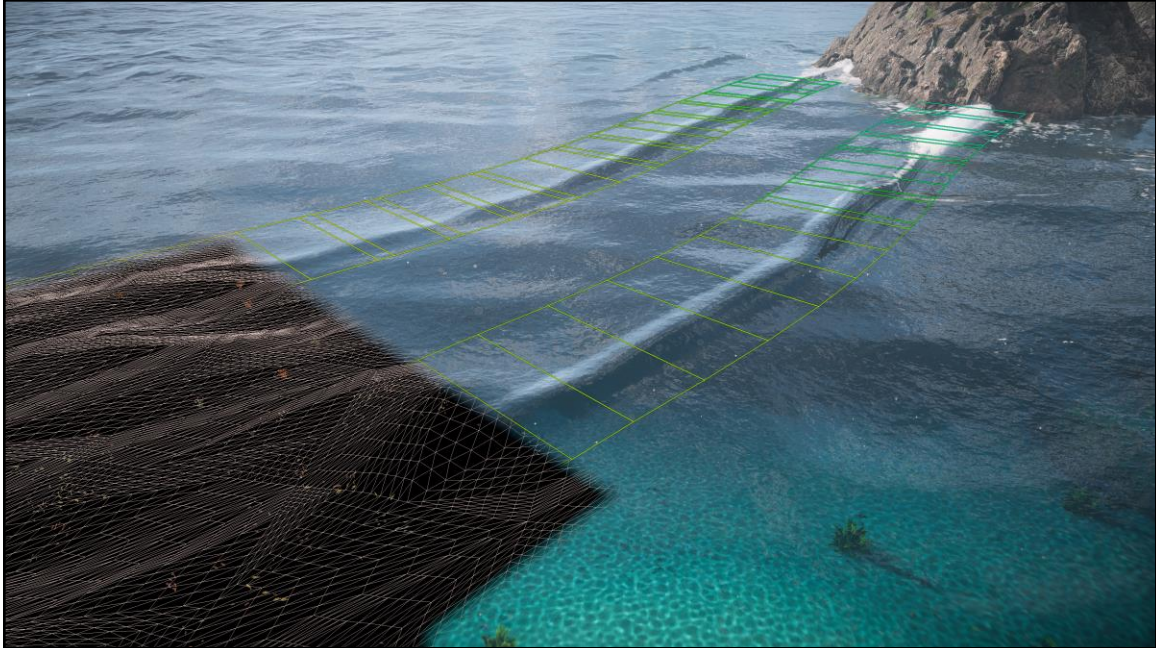
The two waves shown here are moving left right towards the shore. The lefthand wave is a younger version with has lower animation parameters, the righthand one is older and starting to break.

To create the shape we apply the cross-section deformation along the curve of the wavefront. The deformations take a flat water surface and bend it to match their cross section. The cross sections are shown at a few places, the number is the V coordinate of the deformation texture.

Looking at the righthand one, near the camera at 0.27 the water is just starting to rise, further along at 0.4 it starts to steepen, at 0.5 it's curling over and turning into foam.

As the wave moves towards the shore the animation parameter will keep rising, the point where the values hit 0.5 and the wave breaks will progress along the wavefront towards the camera.

How the wavefront is shaped and moved will be covered in the next section.

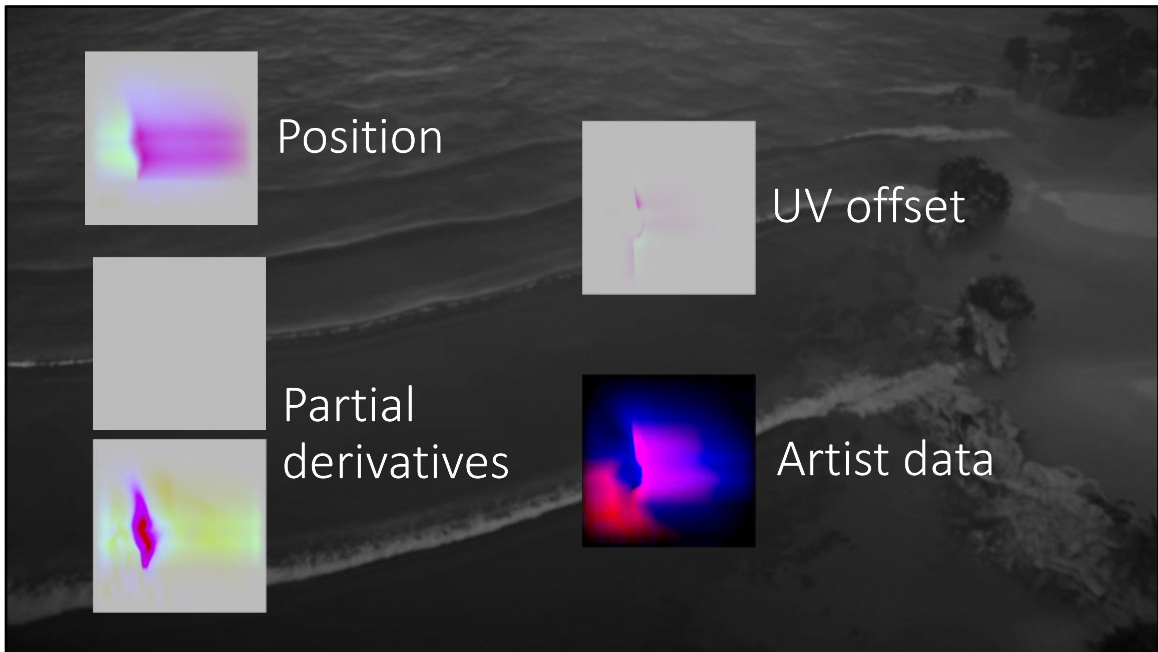


And to get down to the specifics. The vertex buffer for the water surface is generated by a compute shader. The input to the shader are quadrilateral shapes defining the breaking wave, each defines the four corner positions and the animation parameter on the left and right side. This is a screenshot with debug rendering showing them, they're the rows of green quads. The quads are set up by finding the wavefront shape, which is covered in the next section, and extending it outwards.

The compute shader processes each vertex, and finds the quads that contain it. The relative position within the quad is found. The back-to-front parameter is the U texture coordinate for sampling the deformation texture. The side-to-side parameter interpolates the animation parameter on each side, the result is the V part of the texture coordinate. With this UV the deformation texture is sampled, the displacement vector unpacked and transformed to match the orientation of the quad. The displacements are blended to find the final position and the vertex data is written out.

This can stretch the undeformed surface out by a factor of 2 or 3 or more, so it needs some really fine tessellation around the wavefronts. I reduced the tessellation by three levels when I took the wireframe screenshot, otherwise it was too dense to be

readable.



Position isn't enough though, we also need to write binormal and tangent so the water shader can do normalmapping. We did this by computing partial derivatives of position, this meant partial derivatives of the position offset texture had to be calculated and a corresponding 2D texture baked out as shown here.

The water shader also needs consistent UVs. Since the position offset is used to squash vertices together at sharp edges where the detail is needed, the UVs need to be adjusted to compensate. We likewise baked this out to a map, as shown.

The water shader also needed info about where to generate foam, the relative height, and deformation strength; these are passed through the vertex color.

We ended up only needing a single breaking wave deformation. So these textures are used for all the breaking waves seen in Horizon Forbidden West.

Wavefront shape

Offline artist tools

SIGGRAPH 2022 Advances in Real-Time Rendering in Games course

That's how the breaking wave is defined using deformations.

The next topic is the shape and movement and timing of the wavefronts. I'll explain how the tools work next, but before starting it's worth explaining that all these tools exist in the editor, and the resulting animation data is baked out, stored with the world tile data, and streamed in at runtime.

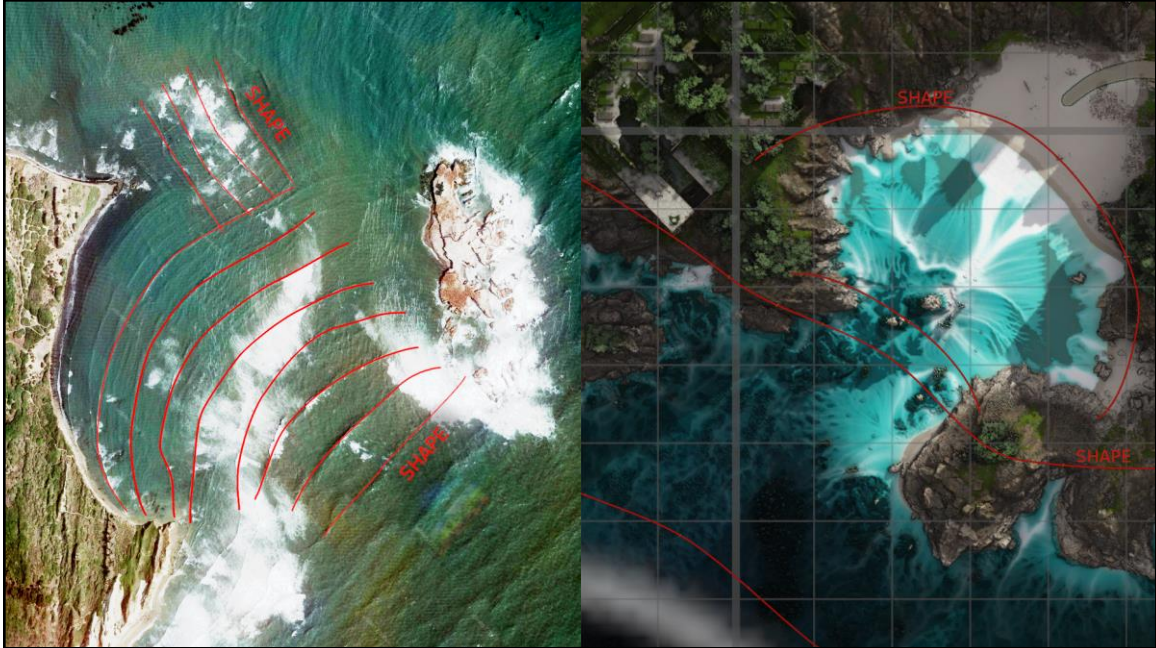
Also, we deliberately chose to let the artists define the wave shapes, rather than starting with a sim and extracting the wavefronts from that.

This is for two reasons. At Guerrilla art directability is a priority. It's also for gameplay reasons; we wanted to be able to respond to game design requests to change wave height or intensity or direction. Since these tools were required we built them first.

Although we started with the artist tools, we still had the option to use a sim to generate a first pass of the animation if needed. This wasn't ever needed, the tools were efficient enough to allow a single artist to lay out all the wave animations in the game.

To explain how they work I'm going to build up an example scene with a satellite image on the left for real-world reference, and an in-game region on the right with actual production data. The images on the right are screenshots from a regular editing session in CoreEd, our editor.

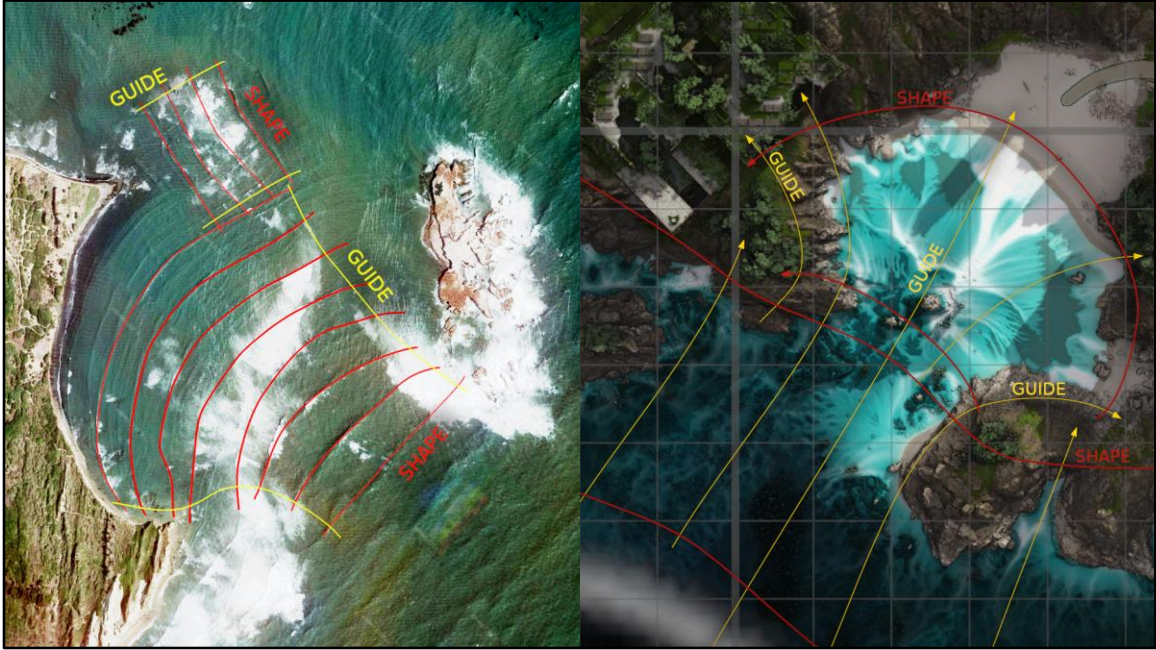
On the satellite image hopefully the wavefronts are clear. The waves approach from the bottom right. They enter the bay and spread out, and only start to break close to the shore. Some waves also bend around the rocky island and approach the shore from another direction.



The first type of curve in the toolset is “shape curves”, because they define the shape of the wavefronts at a particular point in time. They’re the red curves.

On the satellite image, the curves trace out the wavefronts. The wavefront movement interpolates between them so we wouldn’t normally use this many.

In game the shape curves mean the waves are roughly parallel when they enter the bay and then they fan out.

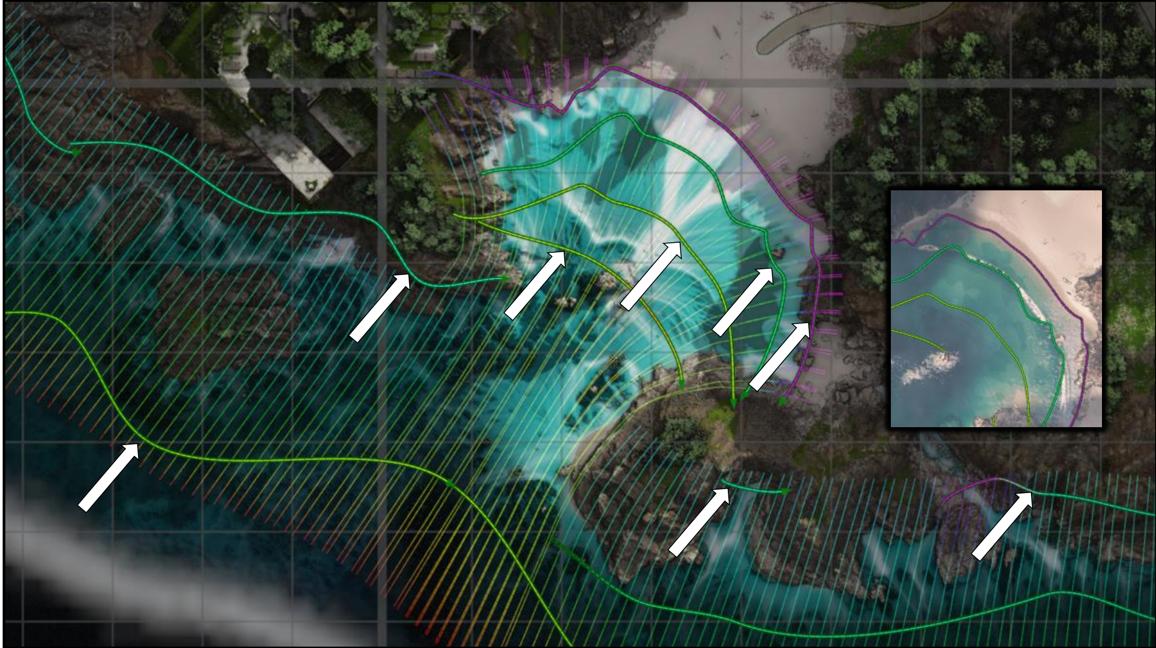


Next are "guide curves". These don't have as much effect on the wavefronts. They define the ends of the wavefronts, and implicitly provide connectivity info between the shape curves.

The blue paths show the resulting animation data, which is baked out and streamed in at runtime.

These paths are perpendicular to the wavefronts, which is probably a bit unexpected. Wavefronts are found by sampling all the paths with the same time parameter, and we can't see that in this visualization.

The reason for this is that that it serves the animation setup, which I'll explain next.



With the tools described so far, the artists can set up wavefronts that match the contour of the beach they wash up onto.

But it doesn't control where the waves break. To provide this we added "animation curves"; these are a curve in the world where the animation parameter is specified. They're marked by arrows here, I've hidden the guide and shape curves.

They use a rainbow coloring to make it easier to see the animation parameter. Notice that the coloring of the path matches the animation curve where they intersect.

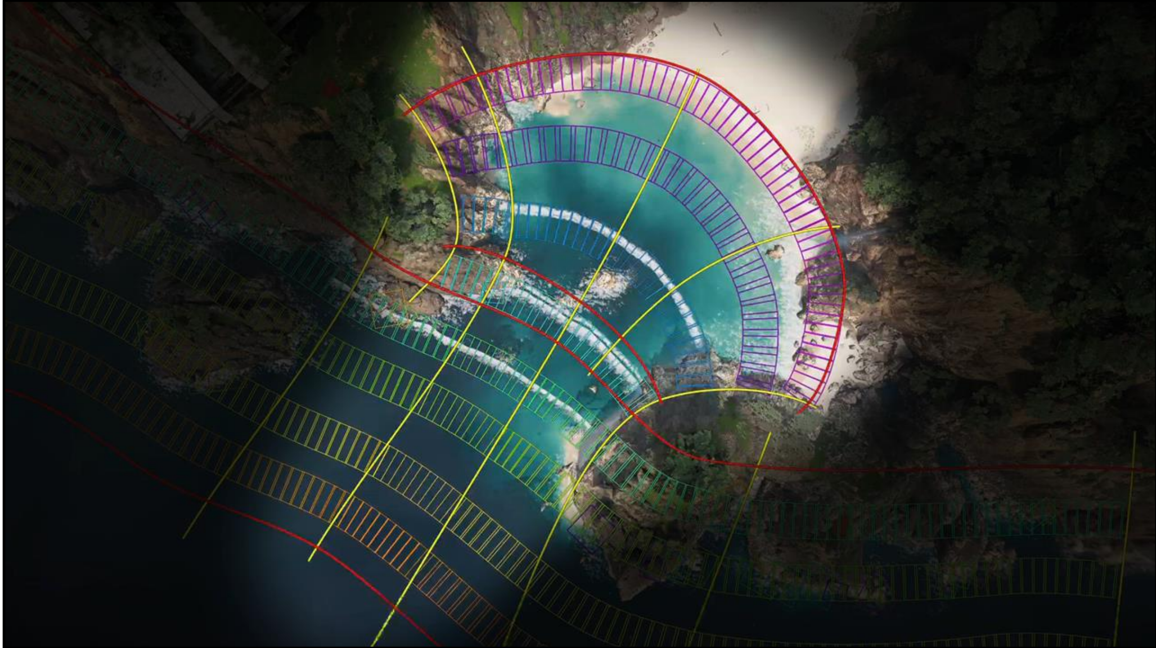
From the artist point of view, they define the animation parameter at that location The bright green lines indicate when the wave starts breaking.

On the right you can see an inset image with an in-game view of the bay, with the curves overlaid. Notice how the foam of the breaking wave appears just after the green animation curve in the middle.

The way they're implemented is paths are traced and the intersections with animation curves found. The animation parameter blends between the values at

those intersections.

This is the reason why all the images show paths rather than wavefronts.



Here’s a capture of a live editing session on PS5, showing the waves created by the curves set up in the previous slides.

The moving rows of quadrilaterals represent the wavefronts, you can see how they bend to match the shape curves.

[Cut]

Here's how the shape curves would look if we wanted the waves to stay parallel as they wash onto the beach.

[Cut]

And if we wanted them to wash diagonally along the beach.

[Cut]

I've hidden the guide and shape curves, to focus on animation.

There's no animation hint curves yet, so the wave animation is the default.

[Cut]

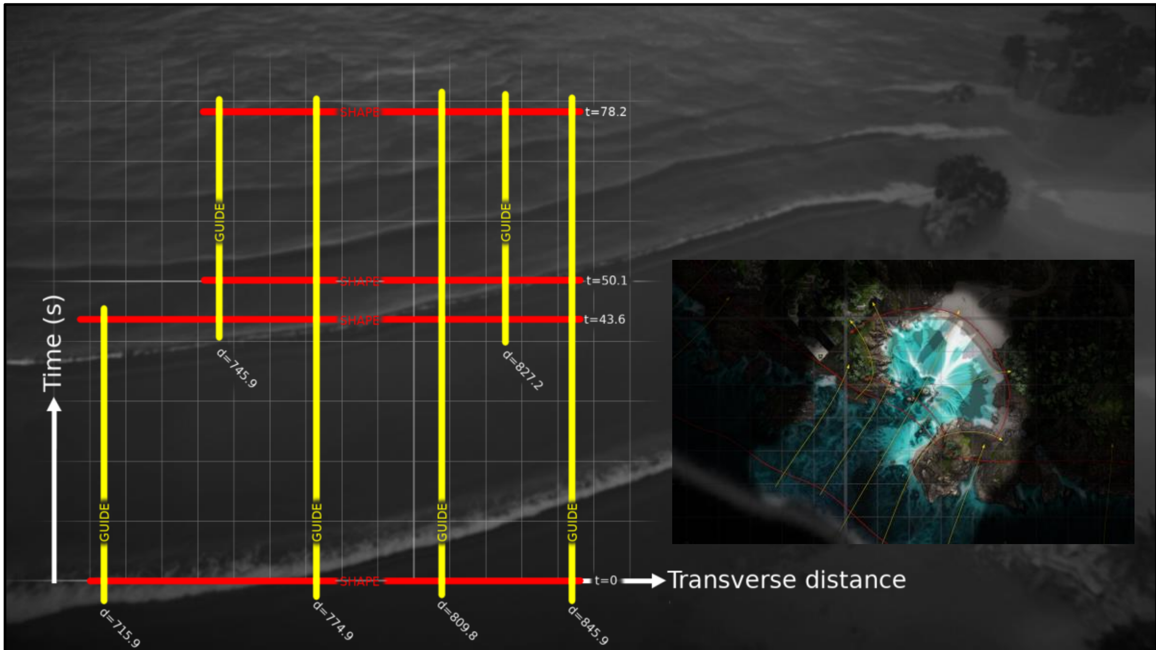
If we create an animation hint curve that sets the animation state to just before breaking, the wave will break just after it passes the curve.

To make it start to break at one end and progress along, we set up a hint curve that sweeps diagonally along it, as shown here.

Notice how foam appears just afterwards.

[Cut]

But we just want a naturalistic effect, where the wave breaks in a few different places. The curve shown here gives that.



Let’s talk about how the shape and movement of the wave is built from the guide and shape curves.

With these curves we set up a function that maps from time and transverse distance - ie distance along the wavefront- to worldspace position.

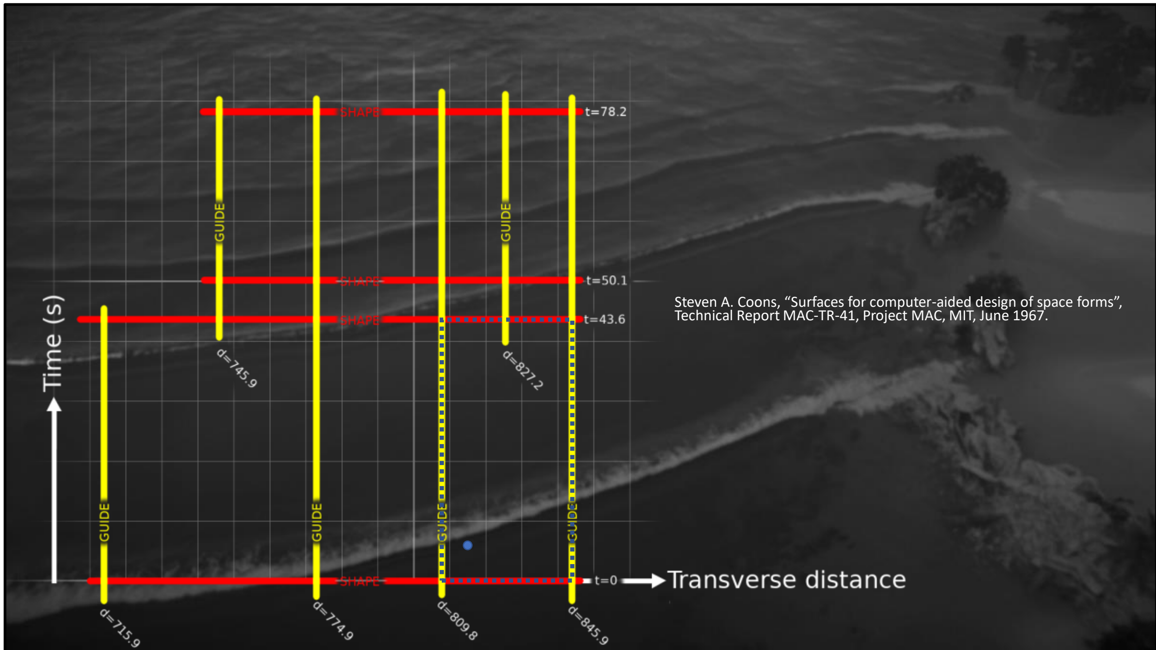
The shape of the wavefront at a particular moment is found by evaluating the function with a fixed time and letting the transverse distance vary along the wavefront. A path is found by holding the distance constant and varying time.

Shape curves have a constant time, guide curves have a constant transverse distance. So they correspond to axis-aligned lines on the domain - as shown in the image.

Given an unordered collection of guide and shape curves, the code uses numeric relax to find the time for each shape curve, and the transverse distance for each guide curve.

The guide and shape curves for the bay unwrap to the collection shown here. For

instance the red shape curve at the top corresponds to the semicircular one on the beach, with five yellow guide curves intersecting it.



We want to evaluate the position for a given time and transverse distance. This is done by finding the four enclosing splines and evaluating the position by interpolating them.

[Click]
For example, the sampling at the blue dot would use …

[Click]
…the four splines indicated by the dotted lines.

In this case it's trivial to find the four surrounding curves for any point. This isn't always the case, there's a bit more info in the bonus slides.

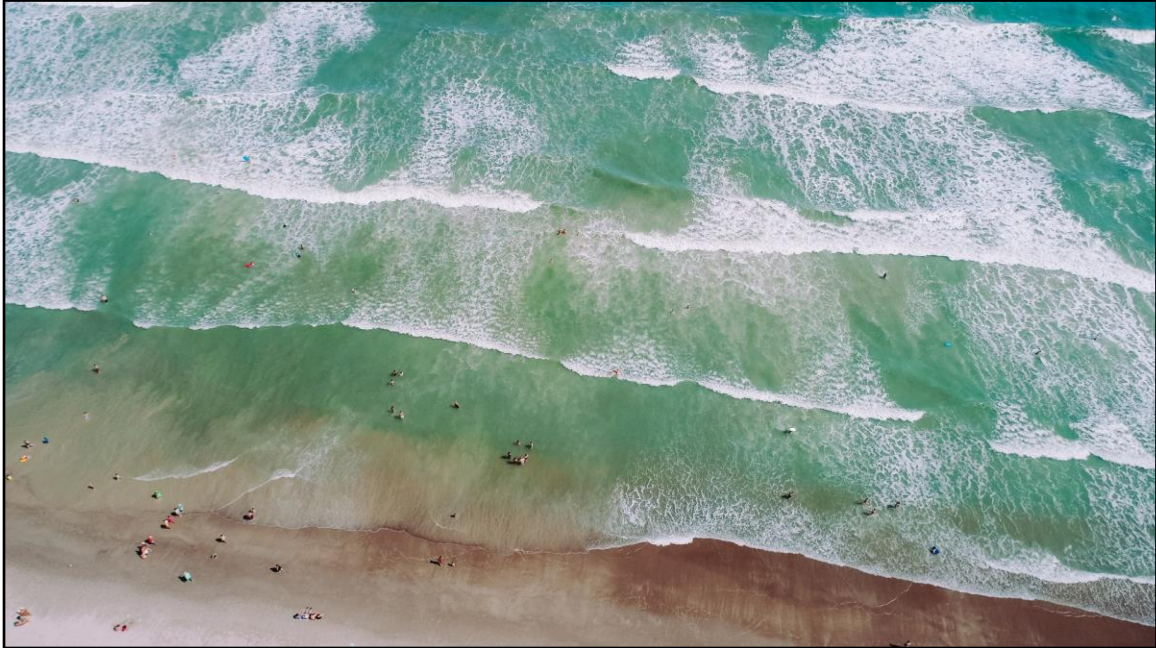
We want the interpolated value to precisely match the four curves on the border. The interpolation formula is in the bonus slides, it was first published by Steven Coons in 1967.

So to summarize the process.

- We're given a point on the 2D domain, the horizontal coordinate

corresponds to transverse distance along the wavefront, and the vertical coordinate is time.
- We find the two guide and two shape curves enclosing that point.
- They're evaluated and interpolated using the formula here to find the worldspace position.

This function, plus then finding the animation parameter by tracing the path is too expensive to evaluate at runtime. So we bake out the position and animation parameter for a grid of time and transverse distance samples. This is the animation data that's stored on disk and used at runtime.



With the tools described so far, we can create wavefronts that roll up to the beach, with a nice progressively breaking wave. In real life, we have a constant series of waves, like the ones shown here. That’s easily done, we have several simultaneously active wavefronts all sampling the same animation data with different times. It does mean they’d all be identical – we need to add variation.

The most obvious variation is that there are gaps along the wave where it doesn’t break. This photo shows the effect well.

We can’t implement those with the animation parameter: if the animation parameter of the water in a gap stays at 0 and the animation parameter of the breaking wave nearby ends up at 1, then between them we’ll see all the intermediate shapes. We need an additional parameter to modulate the strength of the deformation, so we can fade it to zero to create the gaps along the waves.

[Play video]

The wave variation is defined with a texture. A fragment of it is shown here.

Each horizontal line controls a different wavefront. The sampled value scales deformation strength, so black creates gaps in the wavefronts.

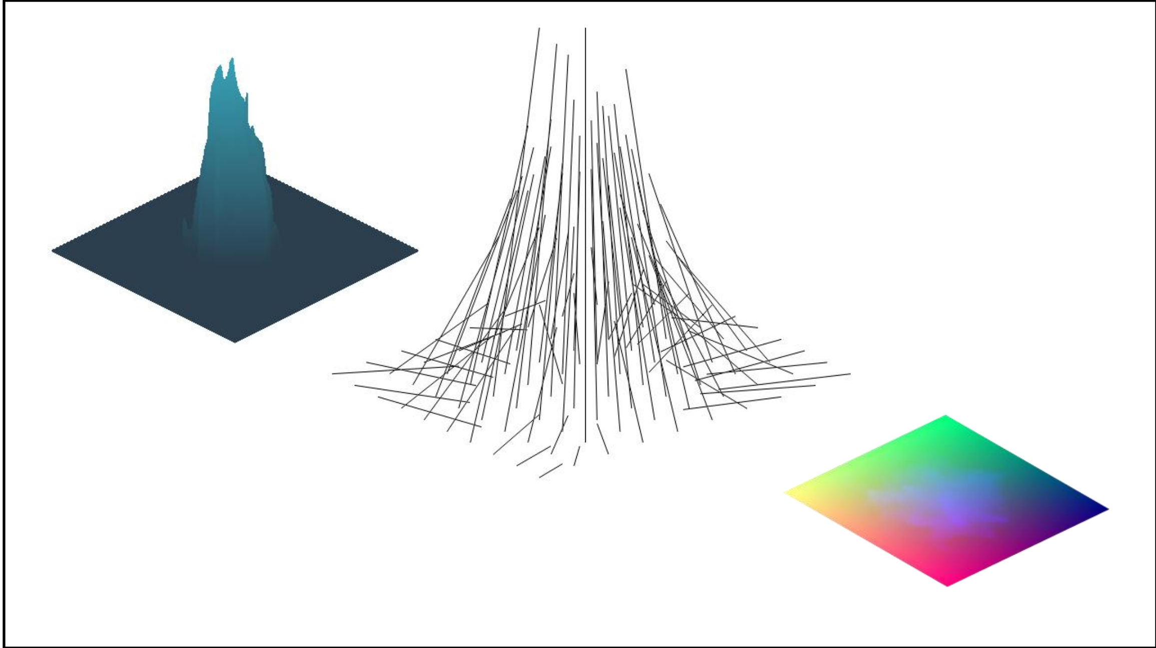
The video shows the change variation makes.

Animated 2d regions

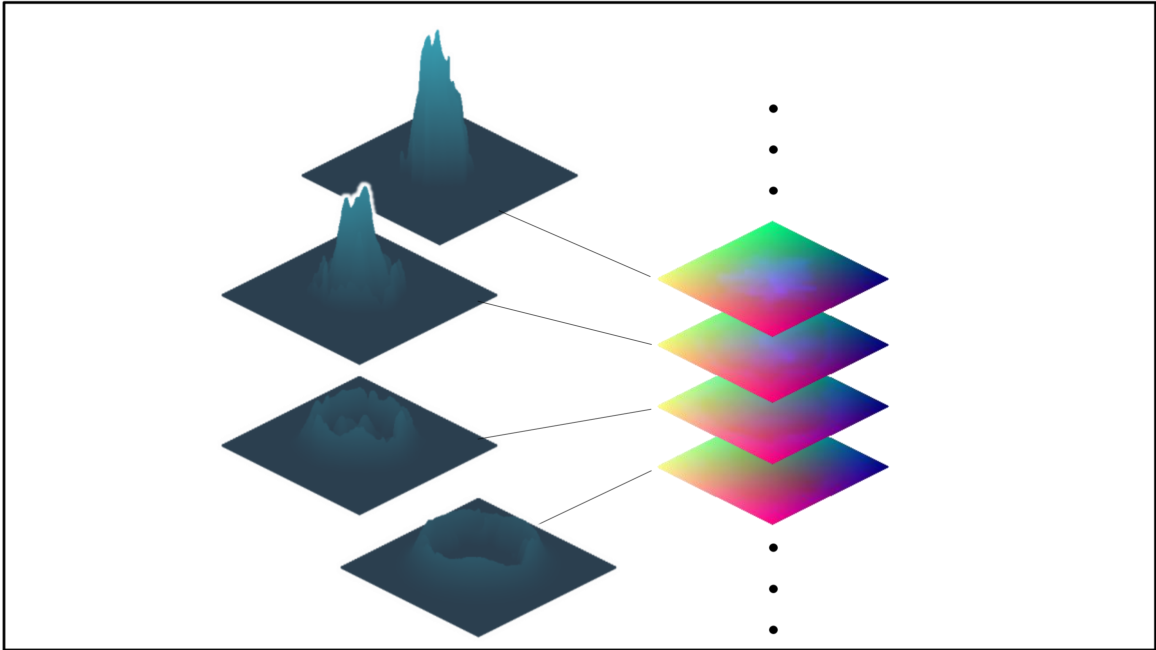
Extends offline tools and runtime code

That covers the technology for the breaking waves, but we also needed to support rivers and lakes. Among other things we needed to support standing waves around rocks, eddies and ripples like the ones shown in the photo.

Like the breaking waves we wanted to base the animation on a Houdini sim. But these effects needed an animated 2D region rather than an animated cross-section so we needed a 3D volume rather than a 2D texture.

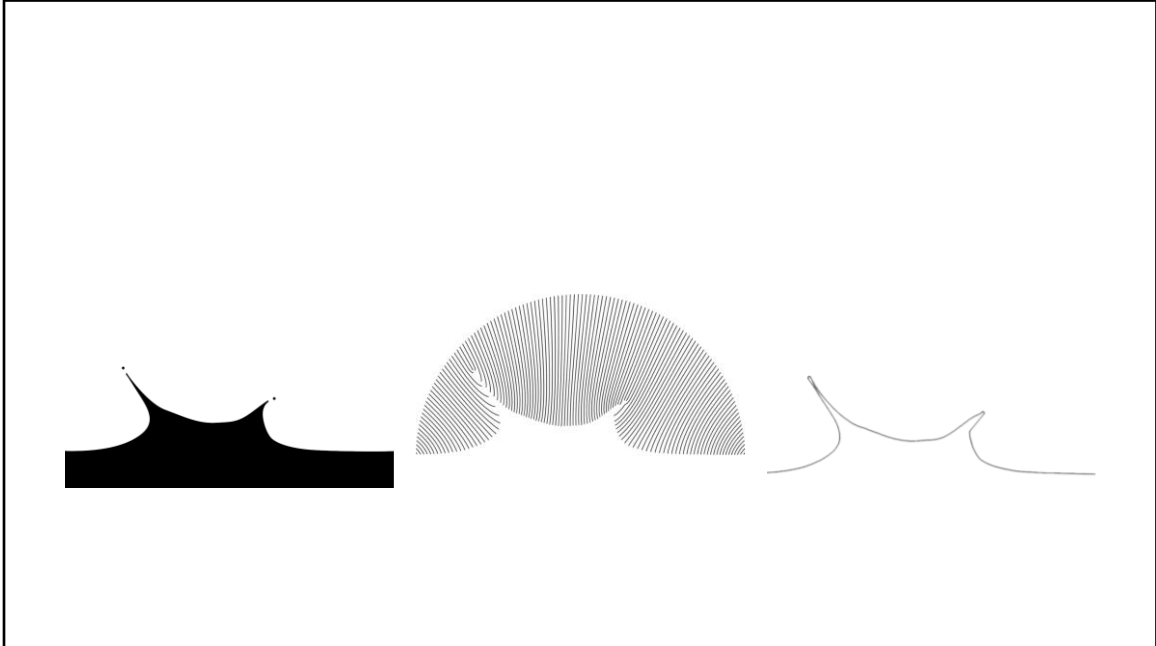


The bake process is similar to that for the breaking wave: we started with a Houdini sim, and extracted the water surface. It's expressed it as a deformation – the lines show some of the offset vectors. It's then baked out to a texture, the colored square shows the result.



Here's more frames from the sim, showing how each frame corresponds to a layer in the stack. This is just an extract from an actual deformation volume, which had 128 layers, not 4, but hopefully this makes the encoding clear.

One complication is that the input mesh can have arbitrary topology, including separate pieces. So setting up the deformation poses some problems.

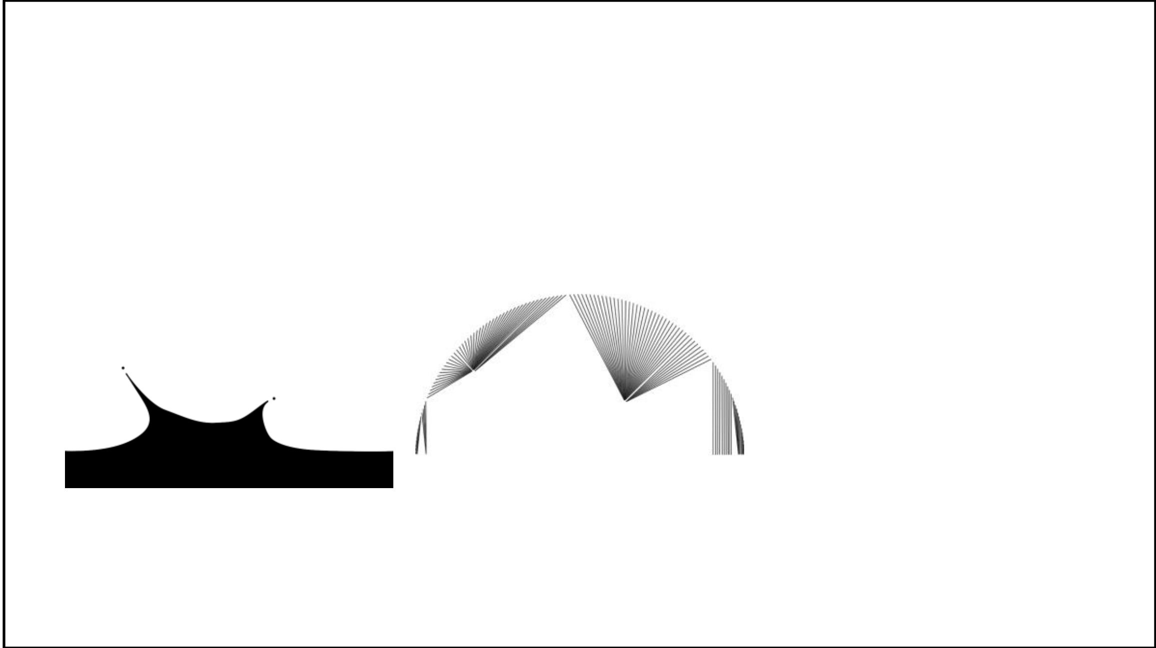


The approach we used to support arbitrary meshes is like a shrinkwrap. The idea was introduced by Overveld and Wyvill in 2004.

The slide shows a cross-section of a water blob with two small separated droplets.

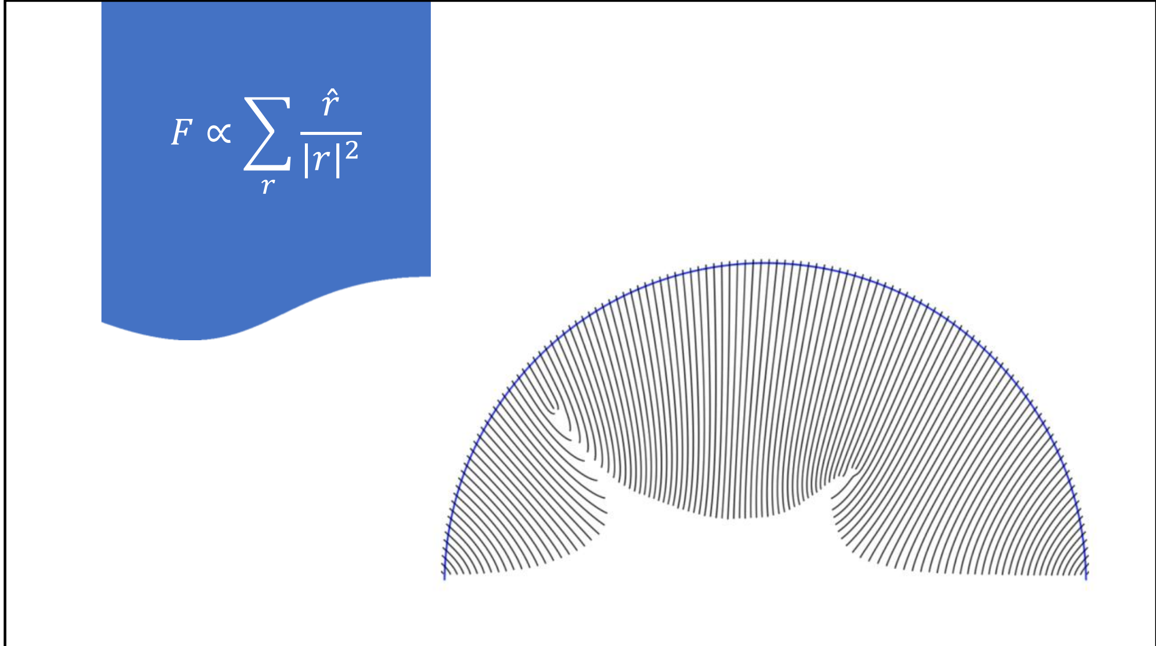
We convert the water mesh to an implicit surface plus potential function. We surround it with a polygonal hemisphere and then moved the vertices in along the field lines, shown in the middle. You can think of this as morphing the vertices of a surrounding hemisphere to the water surface. The end result is the polygonal approximation to the water shape shown on the right. The approximation doesn't have separated blobs. But it captures the overall shape, which is usually what we want.

The choice of potential function is important, we want the field lines to reach every point on the surface.



One choice which doesn’t work is to use the closest point. For surfaces with concavities some parts of the surface can be unreachable. This image shows the result, the resulting mesh is a horribly bad approximation.

We want something like the electrical field lines around a charged object, which are guaranteed to never meet.



[Play video]
So we took the obvious solution and used Columb's law to trace the lines for the vertex morph. This worked well, the resulting field lines are shown in the middle.

But evaluating the electrical attraction of a triangle mesh had to be done numerically and needed a few rounds of optimization to make it run fast enough.

This distributed the points fairly evenly across the surface, and has reasonable frame-to-frame coherence, so it was good enough for our needs.

With this feature we created a library of localized 2D water effects. Houdini was used to procedurally place instances throughout the world, which were then reviewed and refined by environment artists.

They provide all the waterfall impacts shown here.

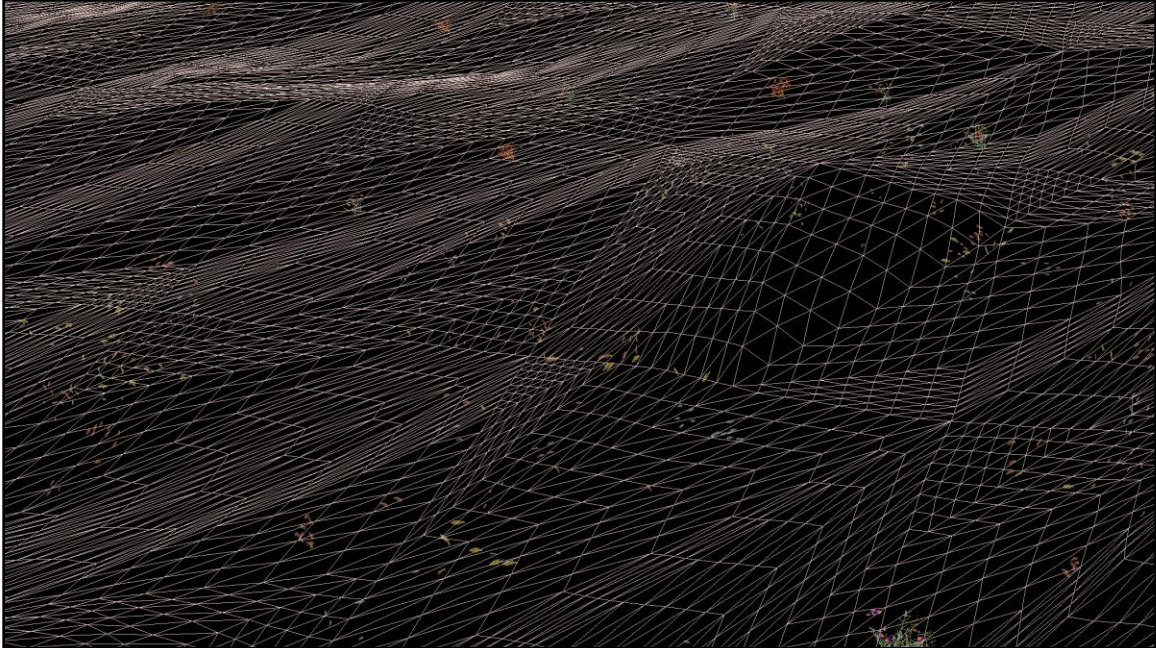
Here’s how it looks without them.
(Flip back and forth)

Future work

SIGGRAPH 2022 VANCOUVER+ 8-11 AUG

SIGGRAPH 2022 Advances in Real-Time Rendering in Games course

Just like in every other production in history, we had to make compromises to meet our ship dates. Here's the things we'd like to improve in future.

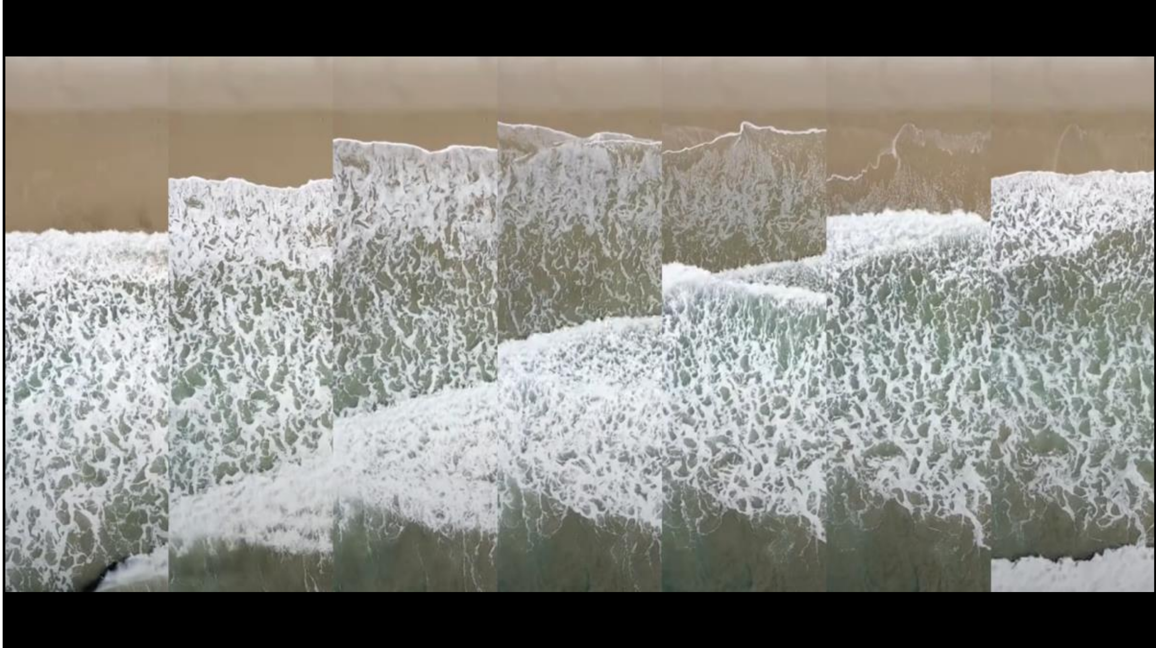


Through the project, the triangle density wasn't high enough to support what we wanted. For example, we had sims with high splashes, which had to get toned down to remove obvious polygonal edges. That's why the deformations on rivers are so flat, and why we aren't using deformations for water impact effects from machines. Also, since vertex color is used for properties like foam it can't create features smaller than triangles. This causes undersampling problems, like foam popping on the front side of the breaking wave.

Several times during the project we dug into the system to see if we could squeeze more triangles out of it. At the very end we added a system to raise the tessellation level around the waves, which improved the detail a lot. We reduced the tessellation level elsewhere slightly to compensate. We can probably push the deformations a bit further with this tech

Ideally, we'd like one vertex per pixel. But of course rendering a trimesh like that is really inefficient for the GPU. So maybe a splat-based solution is better fit.

One triangle per pixel is of course the same goal as Nanite, it's a common theme these days…



Surface foam is another area we'd like to improve. It's a really important part of beach interaction – as shown in the image here – but also entity interactions, flow around rocks, and the breaking waves.

There was a really inspiring paper from Weta, "Guided Bubbles and Wet Foam for Realistic Whitewater simulation" (Wretborn et al), which is of course intended for high quality offline sims, but the core of the water system is baking offline sims for runtime use so maybe that approach can be applied here too.

Thanks to…

Anton Woldhek
Ben Schrijvers
Ben Jaramillo
David Reyes Fornies
Elco Vossers
Inês Almeida
James McLaren
Jeroen Krebbers
Maarten van der Gaag
Marijn Giesbertz
Natalie Burke
Norris Chiu
Roderick van der Steen
Vladimir Lopatin

I've only been able to touch on a small part of the water tech – for example I haven't talked about audio or physics or interaction at all.

It was a big feature, and a lot of people helped build it. Here are the main contributors, everyone here owns some part of the system.

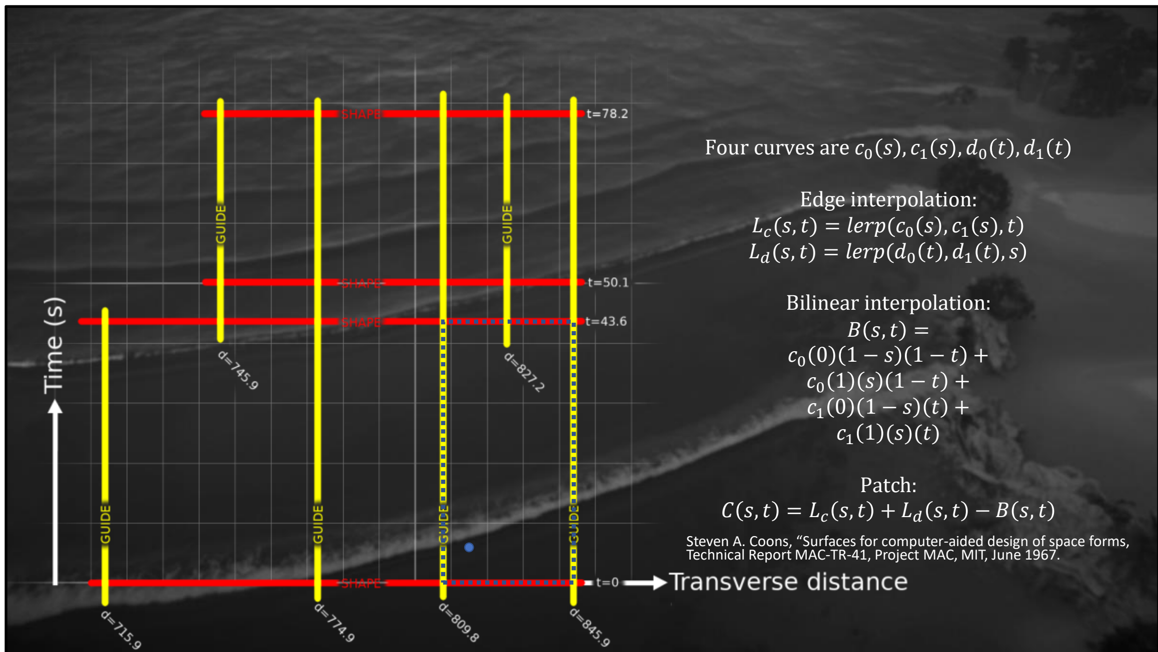
The end

We're hiring!

- References:

- Steven A. Coons, “Surfaces for computer-aided design of space forms, Technical Report MAC-TR-41, Project MAC, MIT, June 1967.
- van Overveld, K., Wyvill, B. Shrinkwrap: An efficient adaptive algorithm for triangulating an iso-surface. Vis Comput 20, 362–379 (2004). https://doi.org/10.1007/s00371-002-0197-4

- Breaking wave video extract from “Mirage: The ever-changing story of Skeleton Bay“, 28 Oct 2020 by “Now Now”.



We want to evaluate the position for a given time and transverse distance. This is done by finding the four enclosing splines and evaluating the position by interpolating them. For example, sampling at the blue dot would use the four splines indicated by the dotted lines.

It isn't always this straightforward; creative arrangements of guide+shape curves can create regions where a point isn't enclosed by a simple connected rectangle made up of two guide and two shape curve segments. When this happens, the nonconvex enclosing boundary can be updated by creating dummy edges, to cut off the nonconvex parts and reduce it to a rectangle.