Water Technology of Uncharted

Carlos Gonzalez Ochoa
Graphics Programmer – Naughty Dog

Doug Holder
Fx Artist – Naughty Dog

GAME DEVELOPERS CONFERENCE®
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9
2012

Contributors

Eben Cook
Michal Iwanicki
Ryan Broner
Vincent Marxen
Jacob Minkoff
Matt Morgan
Jerome Durand
Peter Field
Junki Saita
Marshall Robin

UNCHARTED DRAKE'S FORTUNE

UNCHARTED 2 AMONG THIEVES

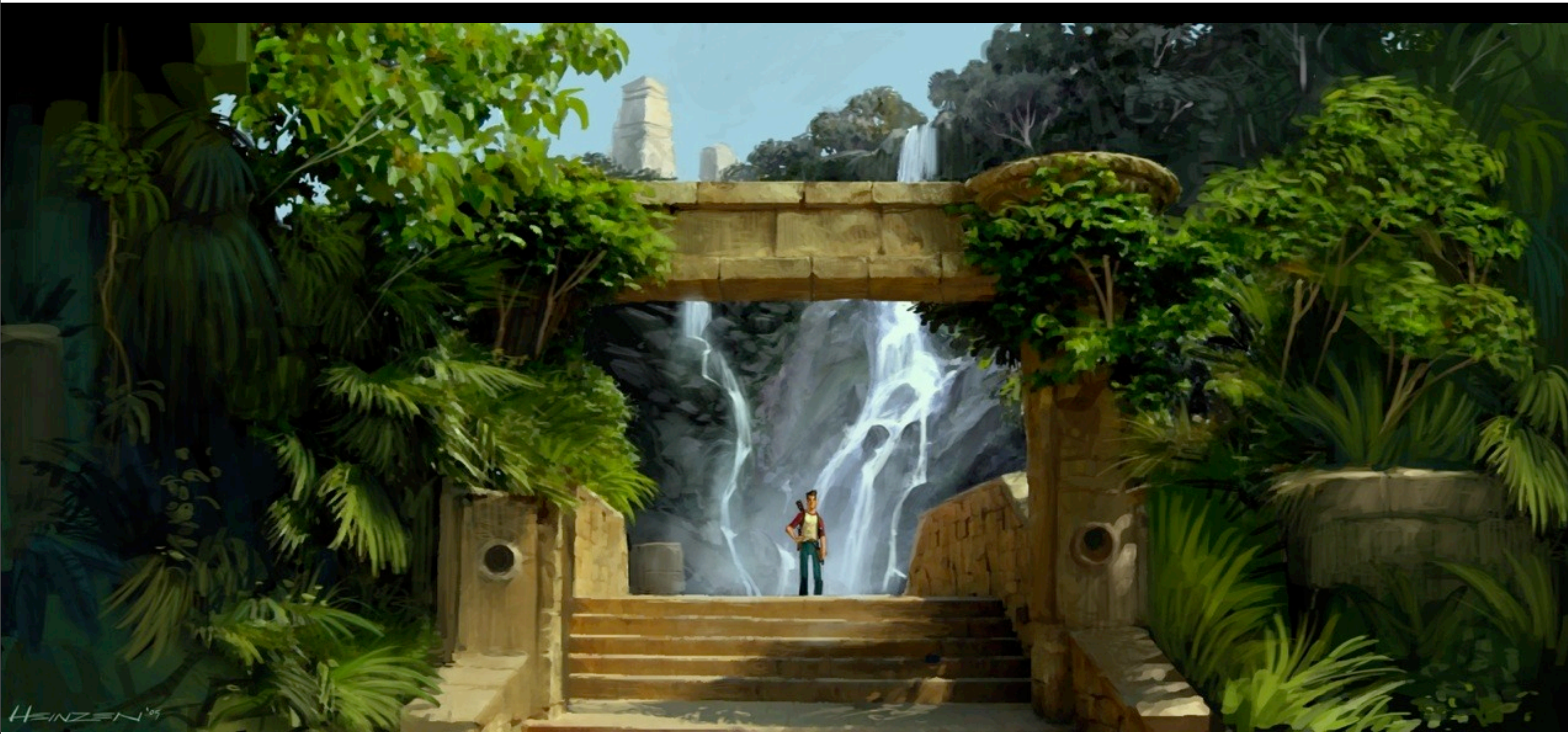
UNCHARTED 3 DRAKE'S DECEPTION

Action and Adventure

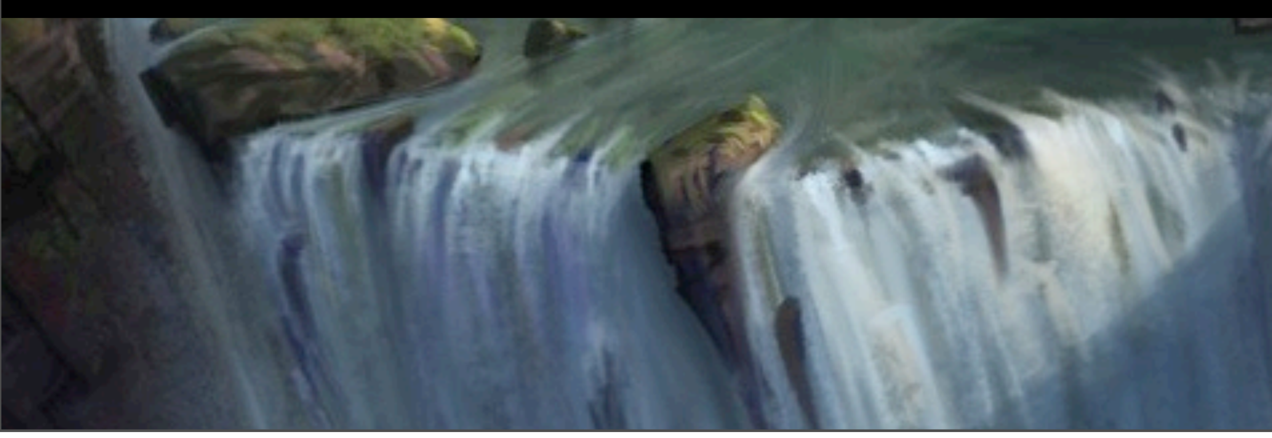
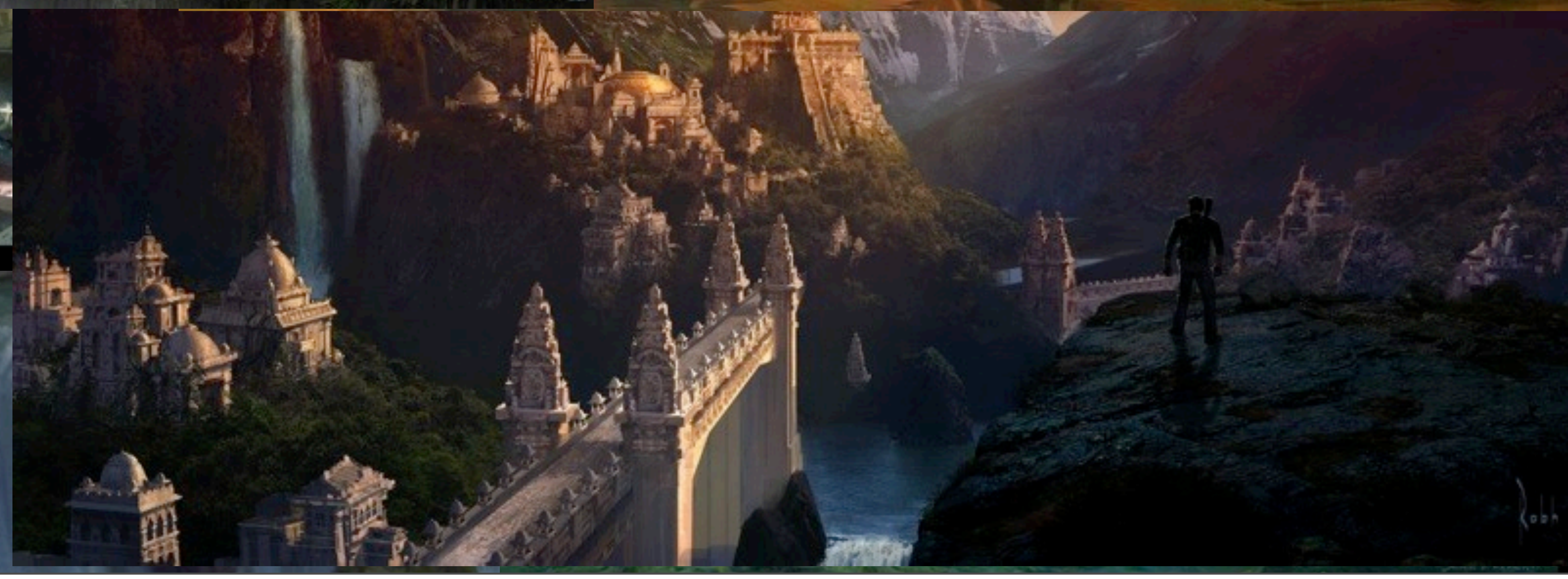
Not a “water” game

Uncharted is an action adventure game. Game play is combat and exploration. It’s not about water gameplay

There are tons of environments: Jungle, temples, cities, caverns, ruins, etc
In each we generally add some water element. puddles, streams, rivers, pools, lakes, and lastly the ocean
Water has been a major design element and we have improved it through the series





UNCHARTED

DRAKE'S FORTUNE

Water in many forms

From small to large bodies of water

Interaction with the water. Clothes get wet

Fast moving

Slow and hard to navigate




The game has a very particular art style and the water had to be able to match it

Water has to move
Has to look better

Previous work... lots of it

Crysis, Kameo, Resistance, Bioshock, Halo and many, many others

Perfect storm, Cast Away, Poseidon, Surf's Up,...

Tech demos

Scientific visualization

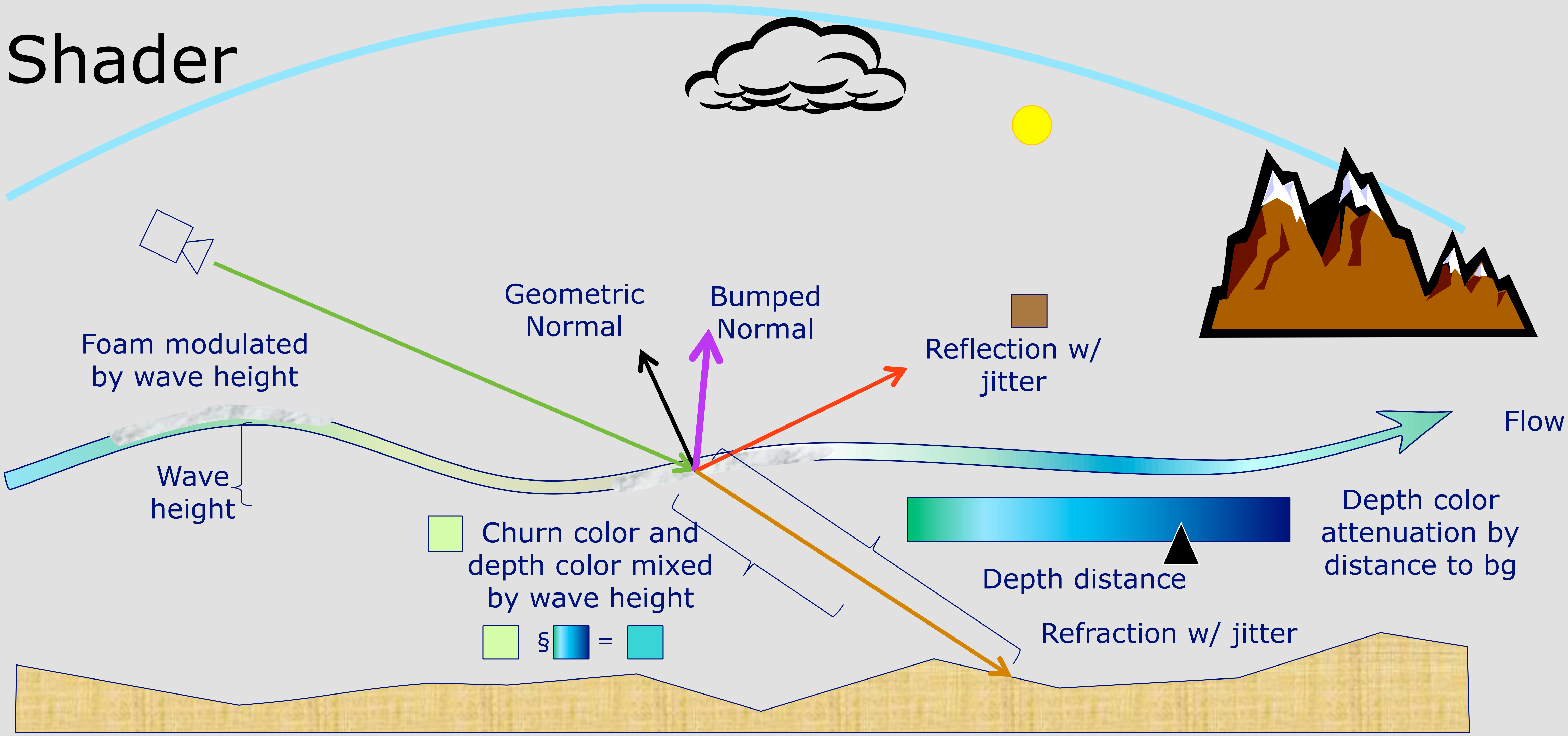
SIGGRAPH, IEEE, I3D papers

We saw lots of reference work in games, films and other scientific papers. The work in Scientific visualization was key for the work in flow

Complex refraction/reflection model
Flow: Scroll uvs (per pixel/vertex) of normal map by a vector field
Refraction: depth based jitter and coloring
Foam movement using a threshold operation on a gradient field
Churn: further depth based coloring.Use foam texture to modulate water depth, blend again. Pseudo-volumetric effect
All parameters are artist controlled!

Complex refraction/reflection model
Flow: Scroll uvs (per pixel/vertex) of normal map by a vector field
Refraction: depth based jitter and coloring
Foam movement using a threshold operation on a gradient field
Churn: further depth based coloring.Use foam texture to modulate water depth, blend again. Pseudo-volumetric effect
All parameters are artist controlled!

Shader



The water shading uses already used ideas:
Use a bump map to jitter the normal of the surface.
Use a fresnel term using the jittered normal to blend between a refraction and reflection contribution.

In addition, we use other ideas to modulate the coloring of each contribution
We add foam that get lit on top of the reflection and refraction. We can also choose to mix the foam w/ the refraction, this can create muck

We use churn to simulate a volumetric effect

Refraction. We color the refraction depending on the distance between the surface of the water and the background
The depth coloring can be a linear or exponential function
The depth coloring can be affected by churn. We take one channel of the foam texture to blend between the depth coloring and the "churn" coloring.
As the foam moves (which it also gets modulated by the waves), the coloring changes with the waves and the foam scrolling

The reflection can also be added on top the refraction instead of just blended.
All fresnel coefficients, start-end are artist controlled

Water shader by layers


Reflection - no bump

Reflection - with bump

Refraction

The reflection and refraction are blended using a fresnel coefficient

Depth based function is used to blend a base color and the refraction color. Here no jittering is applied.

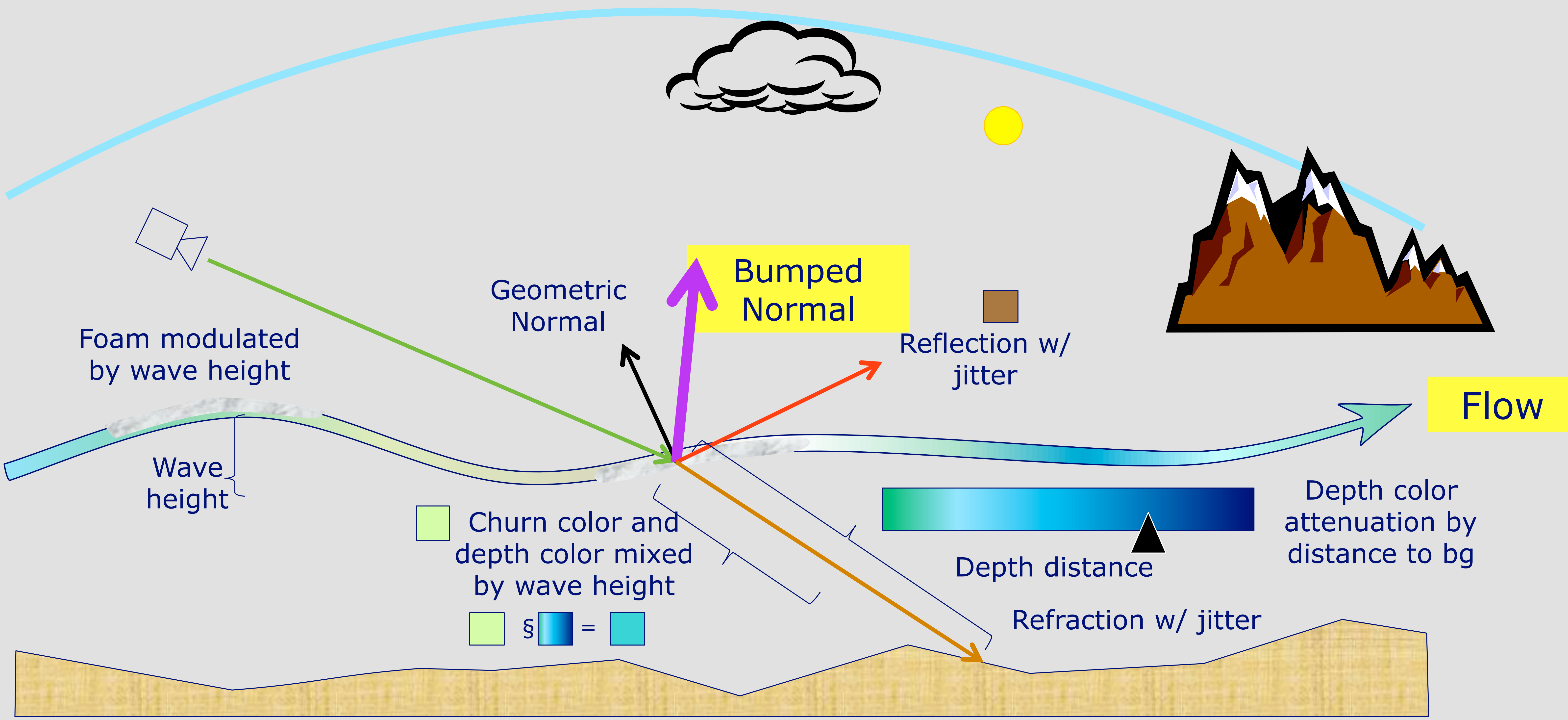
The jitter factor is also modulated by the depth

Soft shadows

Foam and churn which add a texture above and below
The foam is also moving by the flow

Specular lighting


Final



We will look into how we moved the normal maps, this is the key of the water shader

Flow shader

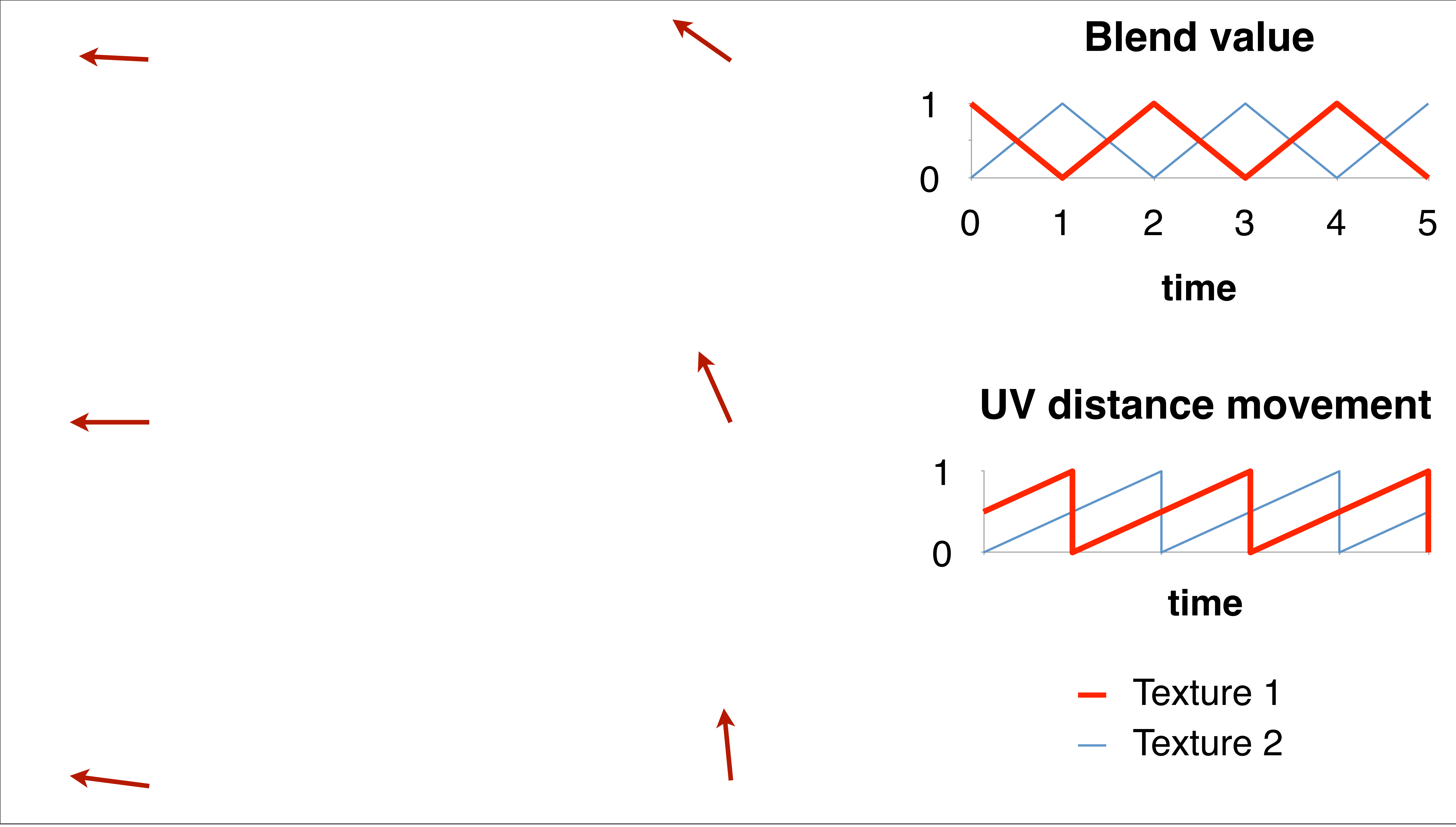
Vector field used to advect mesh properties

- UV scrolling
- Displacement

Shader based on scientific visualization work:

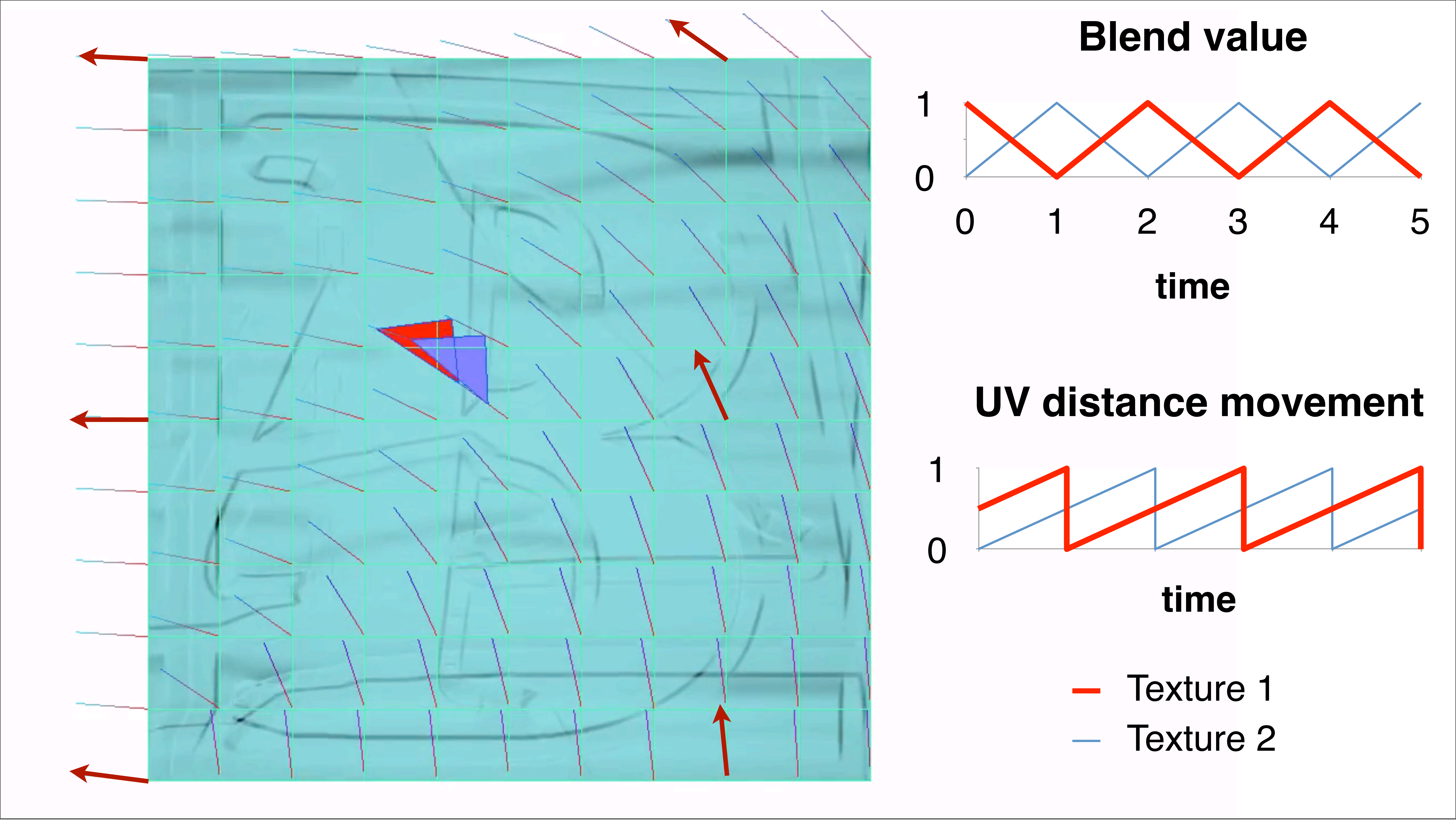
- N. Max, B. Becker, “Flow Visualization using Moving Textures”, 96
- F. Neyret. “Advected Textures”, 03

The flow shader was originally created by C. Gonzalez-Ochoa at ND for Uncharted 1.



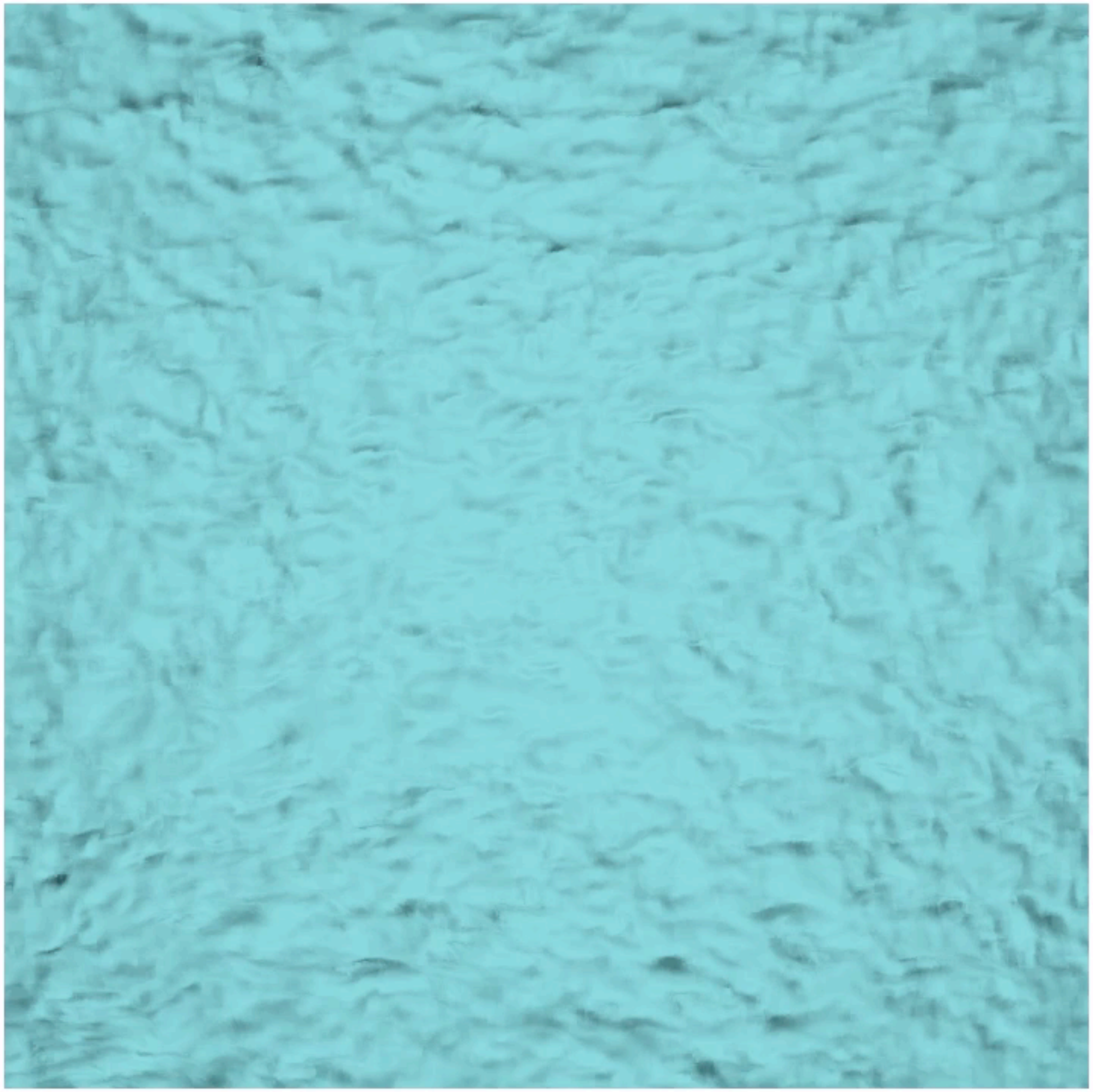
Here is an example of flow using a vertex shader to compute the blend values for triangles (instead of pixels).

Remember, to scroll in the right direction the uvs should move OPPOSITE to the intended direction



Here is an example of flow using a vertex shader to compute the blend values for triangles (instead of pixels).

Remember, to scroll in the right direction the uvs should move OPPOSITE to the intended direction



We can use several other effects besides using the basic flow. Here we use a second fetch (feedback) to give extra fluidity

Blend between two “flow” textures, one offset in phase by $\tau/2$

Ways to improve it:

Offset the placement after each cycle
Offset uv starting position to minimize distortion
Offset in space the phase
Use texture feedback (ping back) to get extra motion

There are many ways to improve flow.
Since the texture moves in cycles of Tau seconds, the improvements come from where the texture starts moving from (1 and 2 above)
The offset in space, is when using flow on a vertex shader and we want each pixel to blend at a different rate.
Be careful because one does not want to big discontinuities across the whole texture.

Flow shader

```
half3 result, tx1, tx2;
half  s = .1; // small value

float  timeInt = (g_time) / (interval * 2);
float2 fTime = frac(float2(timeInt, timeInt + .5));
float2 flowUV1 = uv - (flowDir/2) + fTime.x * flowDir.xy;
float2 flowUV2 = uv - (flowDir/2) + fTime.y * flowDir.xy;


tx1 = FetchTexture(flowUV1);
tx2 = FetchTexture(flowUV2);
tx1 = FetchTexture(flowUV1 + s*tx1.xy); //[optional] Fetch 2nd
tx2 = FetchTexture(flowUV2 + s*tx2.xy); //time for extra motion
result = lerp(tx1,tx2, abs((2*frac(timeInt))-1);
```


This is the most basic flow shader.

There are many ways to improve it. By adding an offset to where the texture starts:

float2 offset1 = float2(floor(timeInt) .1);
float2 offset2 = float2 (floor(timeInt + .5) * .1 + .5);

flowUV2 = uv + offset1 - (flowDir/2) + fTime.x * flowDir,xy;
flowUV2 = uv + offset2 - (flowDir/2) + fTime.y * flowDir.xy;

Offset the uv starting position to minimize distortion. One wants the uv coordinates to have less distortion (that mean their displacement is 0) when the blend is at its maximum.

```
float  timeInt = (g_time) / (interval * 2);
 float2 fTime = frac(float2(timeInt, timeInt + .5));
 float2 flowUV1 = uv - (flowDir/2) + fTime.x * flowDir.xy + .5 * flowDir.xy;
 float2 flowUV2 = uv - (flowDir/2) + fTime.y * flowDir.xy + .5 * flowDir.xy;
```

You can spread around where the phase changes, this is easy to do if the flow is done on a pixel shader (not on triangles)

```
float  timeInt = (g_time) / (interval * 2);
float offsetPhase = phaseScale * FetchPhaseTexture(u,v); // texture goes from 0..1
 float2 fTime = frac(offsetPhase + float2(timeInt, timeInt + .5));
```

You can think on many other ways to use flow.

Scientific visualization results can be handy to do cool shader effects.

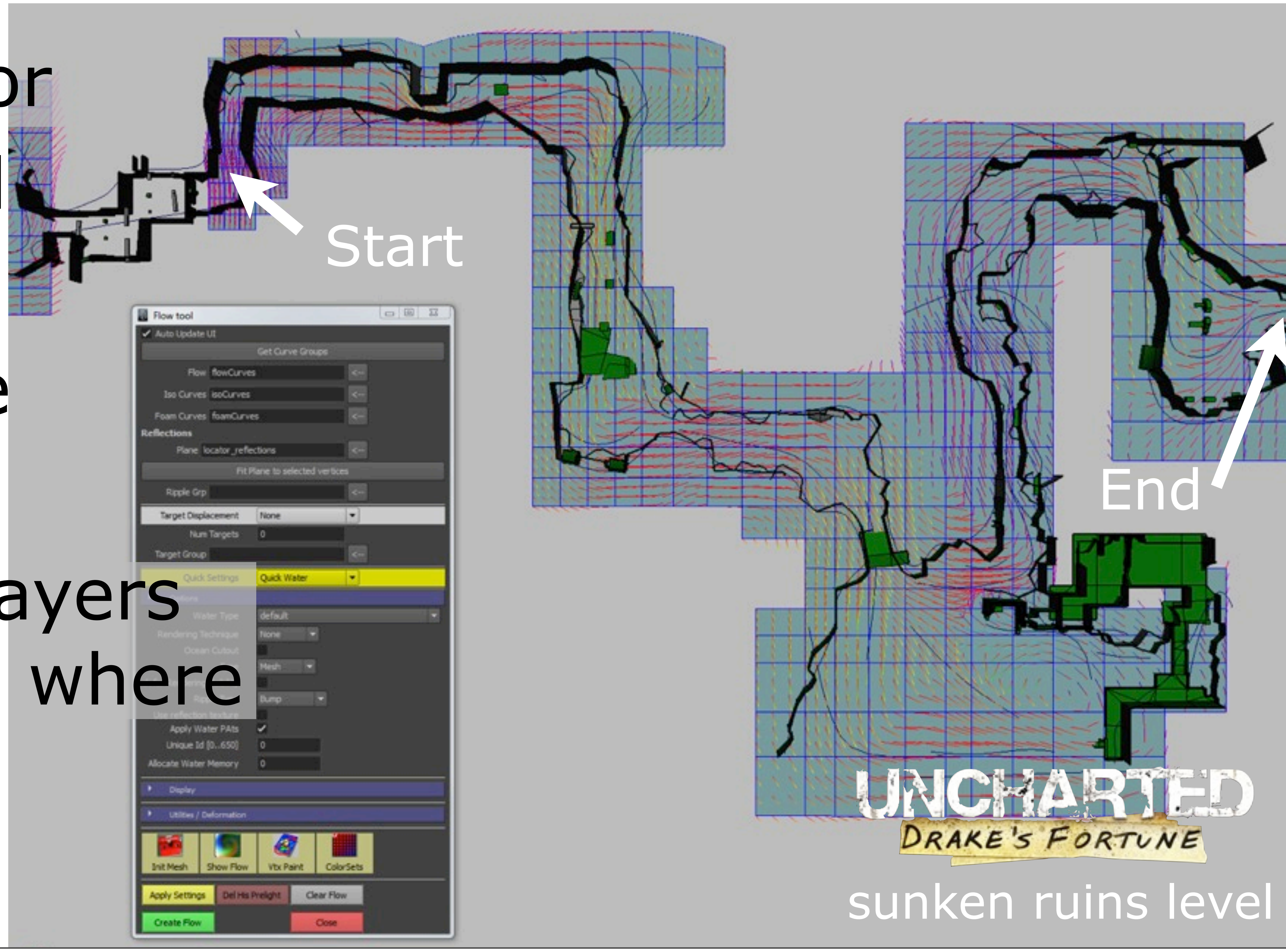
Easy to author

ND Maya tool

Dir, velocity

Foam, phase

Flow gives players a direction to where to go.

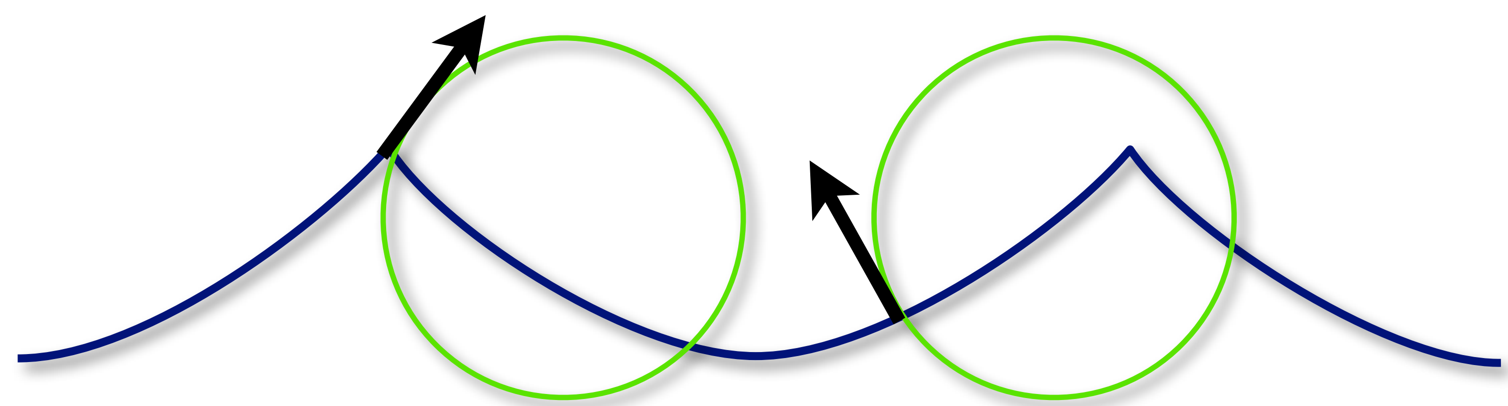


This is the Maya tool used in all the Uncharted games.
The tool uses splines to define the direction and color maps to control the magnitude of the velocities.
Also,we can generate foam maps, and phase (for displacement).
Later we will talk more about the basic flow displacement

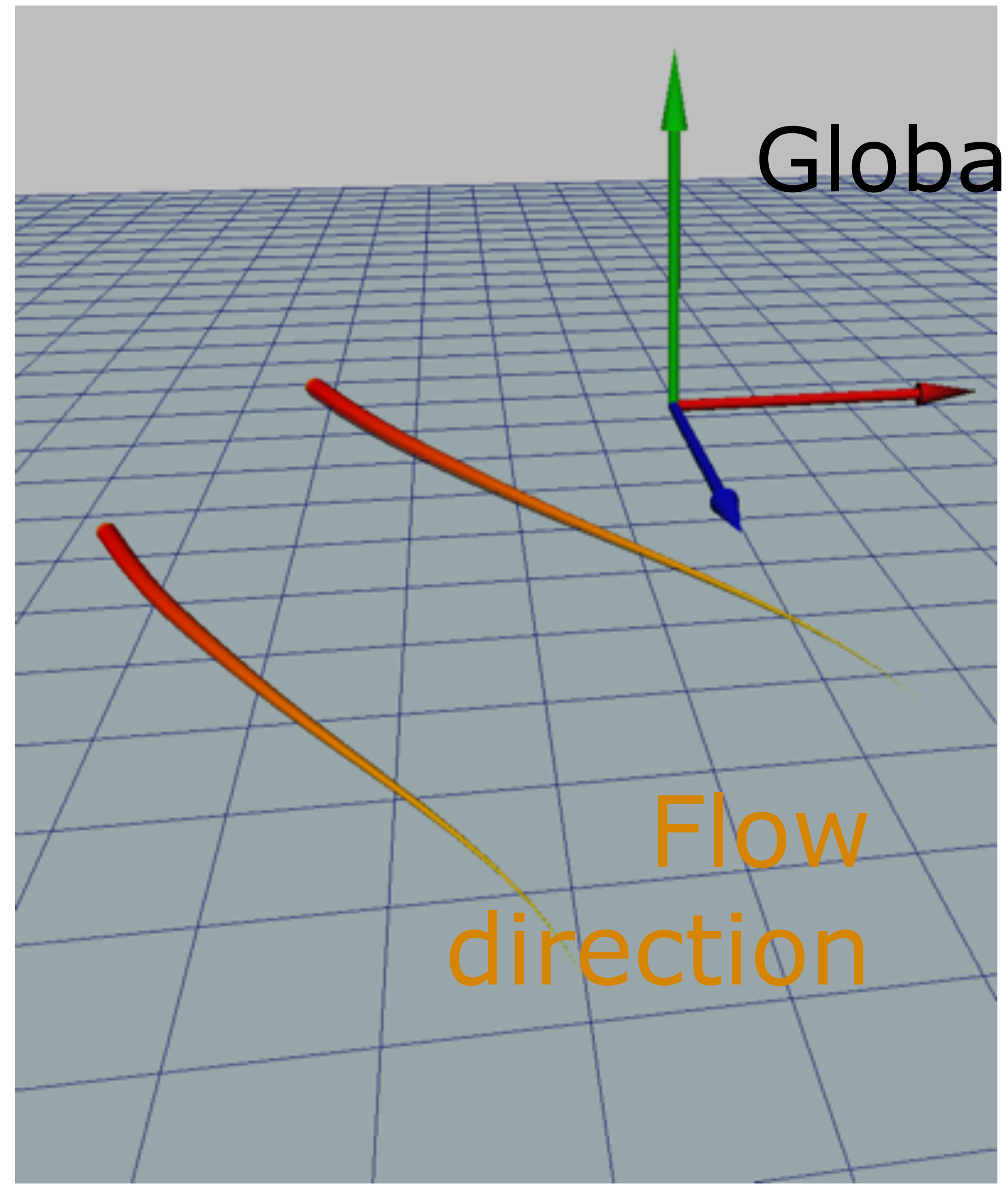

UNCHARTED
DRAKE'S FORTUNE
sunken ruins level

Flow based displacement

Each vertex moves on a circular pattern at a different phase ϕ



$$x_1' = x_1 + \cos(\alpha t + \phi_1) \qquad x_2' = x_2 + \cos(\alpha t + \phi_2)$$
$$y_1' = y_1 + \sin(\alpha t + \phi_1) \qquad y_2' = y_2 + \sin(\alpha t + \phi_2)$$

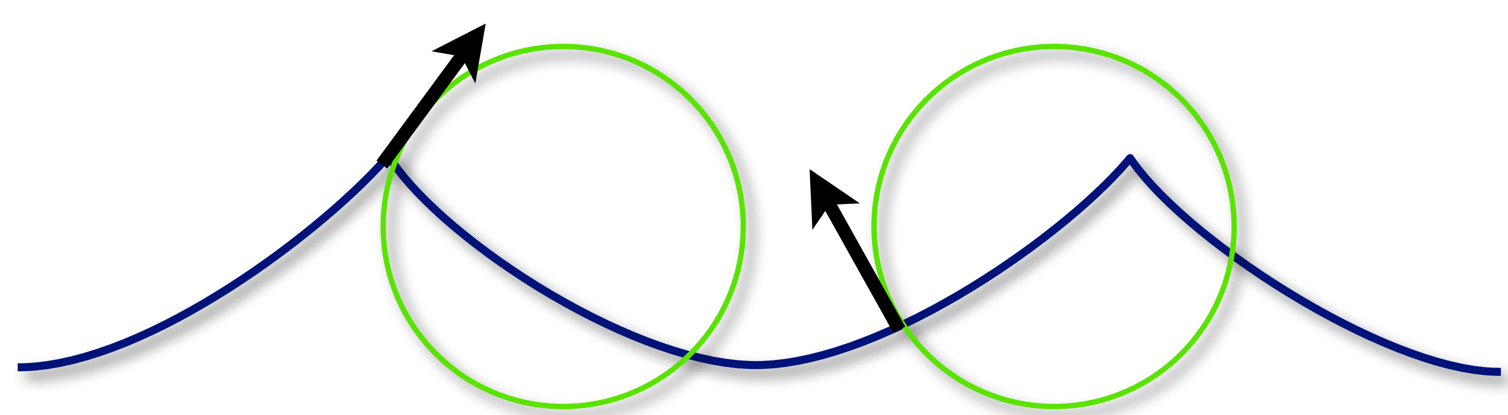


Flow can also be used for displacement.
In the case of U1, I encoded flow into the vertices. And we could do displacement in the SPU's.
In addition to a vector field, we encoded a a phase field over the mesh.
Every vertex moves in a anisotropic scaled circle.

So if we set the phase field nicely spaced not only in the direction of flow, but also perpendicular to it, we can get some interesting waves.
Of course these waves will not be as dramatic nor complex as the one in U3

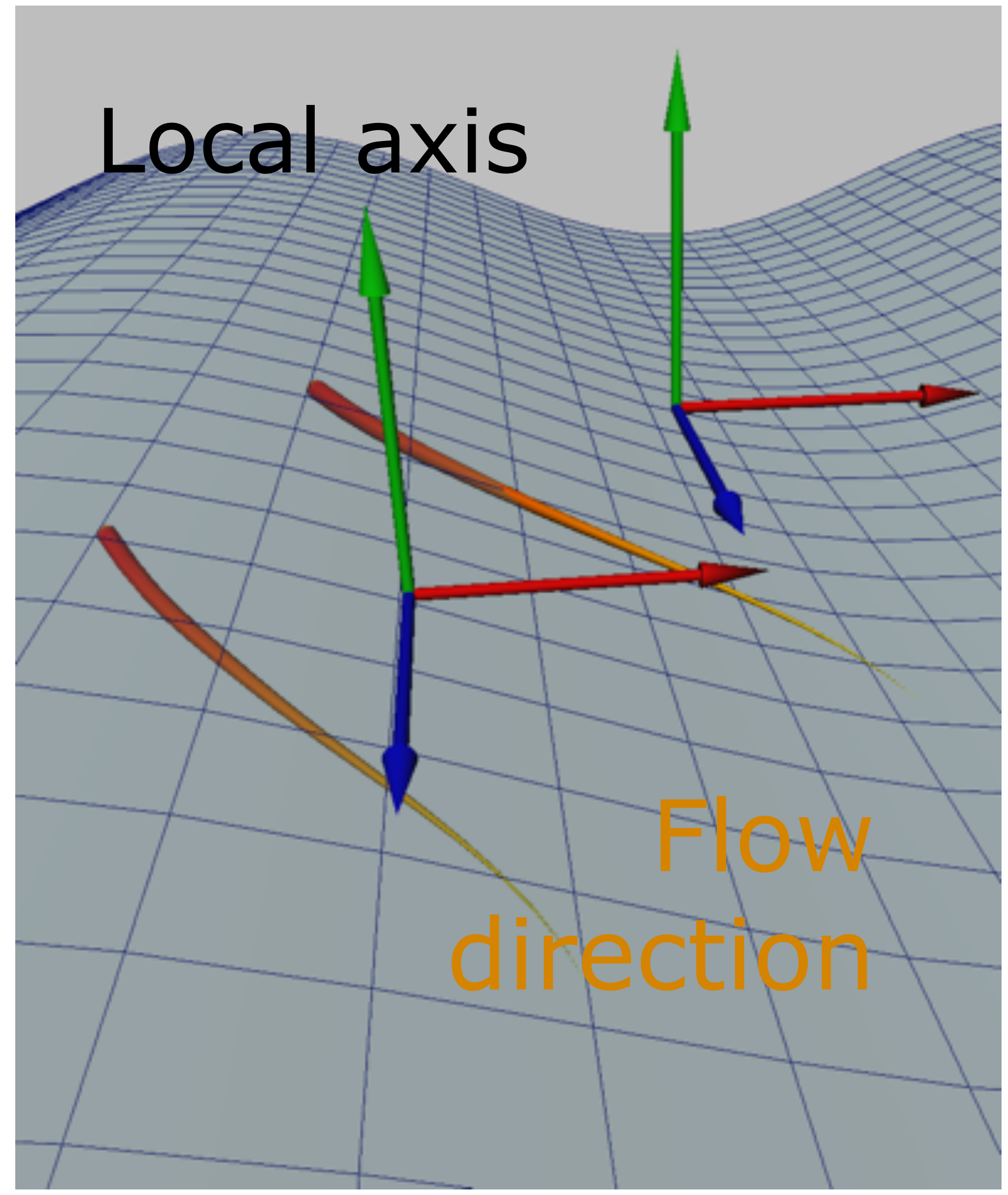
Flow based displacement

Each vertex moves on a circular pattern at a different phase ϕ



$$x_1' = x_1 + \cos(\alpha t + \phi_1)$$
$$y_1' = y_1 + \sin(\alpha t + \phi_1)$$

$$x_2' = x_2 + \cos(\alpha t + \phi_2)$$
$$y_2' = y_2 + \sin(\alpha t + \phi_2)$$

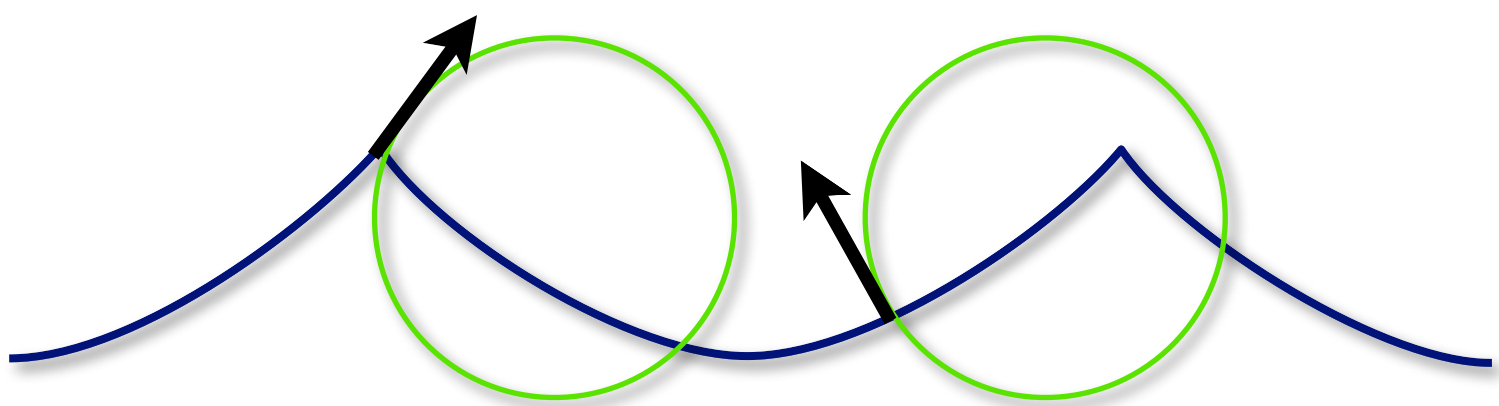


Flow can also be used for displacement.
In the case of U1, I encoded flow into the vertices. And we could do displacement in the SPU's.
In addition to a vector field, we encoded a a phase field over the mesh.
Every vertex moves in a anisotropic scaled circle.

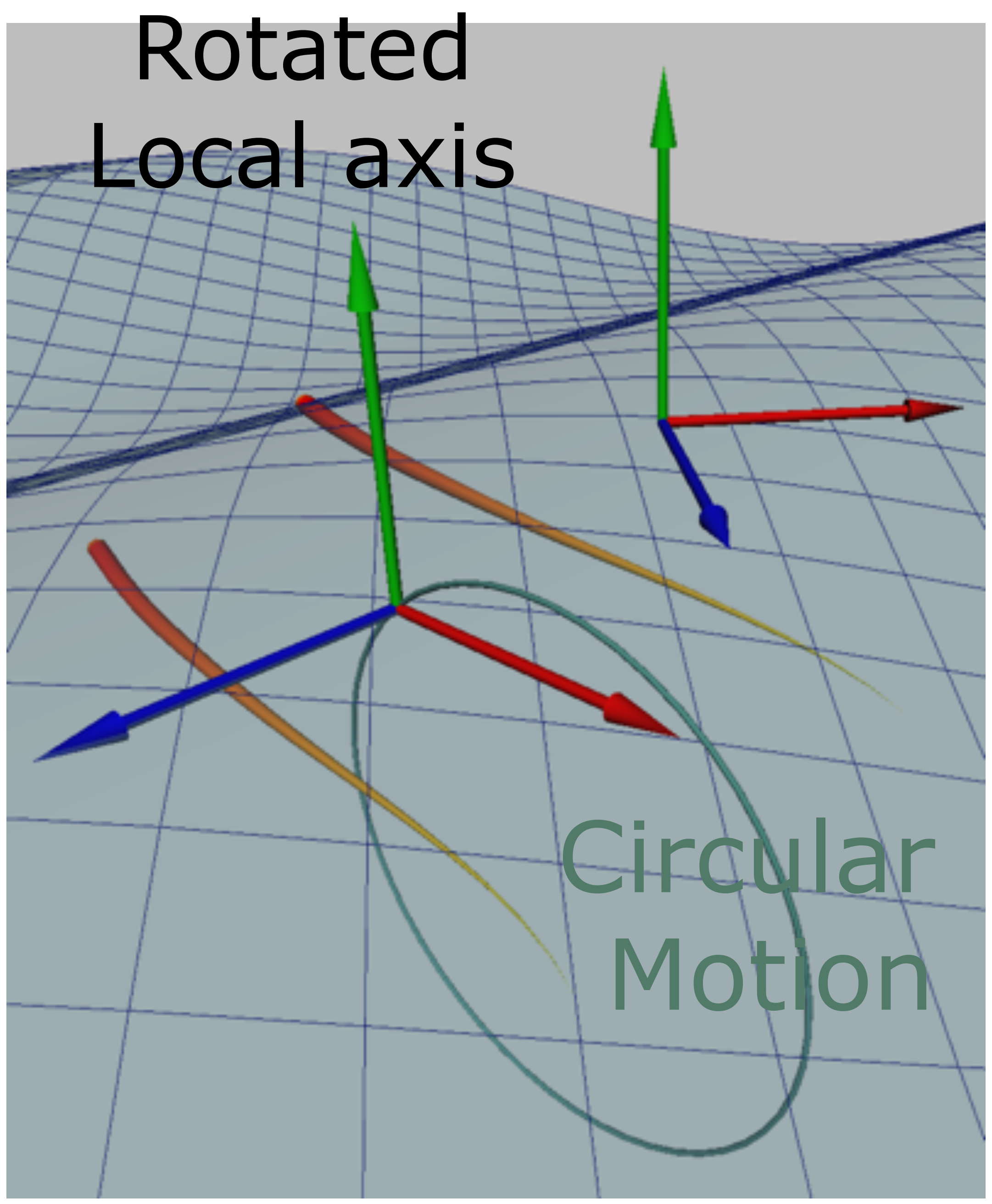
So if we set the phase field nicely spaced not only in the direction of flow, but also perpendicular to it, we can get some interesting waves.
Of course these waves will not be as dramatic nor complex as the one in U3

Flow based displacement

Each vertex moves on a circular pattern at a different phase ϕ



$$x_1' = x_1 + \cos(\alpha t + \phi_1) \qquad x_2' = x_2 + \cos(\alpha t + \phi_2)$$
$$y_1' = y_1 + \sin(\alpha t + \phi_1) \qquad y_2' = y_2 + \sin(\alpha t + \phi_2)$$



Flow can also be used for displacement.
In the case of U1, I encoded flow into the vertices. And we could do displacement in the SPU's.
In addition to a vector field, we encoded a a phase field over the mesh.
Every vertex moves in a anisotropic scaled circle.

So if we set the phase field nicely spaced not only in the direction of flow, but also perpendicular to it, we can get some interesting waves.
Of course these waves will not be as dramatic nor complex as the one in U3


UNCHARTED
DRAKE'S FORTUNE

We have used flow for sand moving on top of the dunes, the psychedelic effects in U3, cloud movement.
FX artists even use it for fire and trails

Water good enough for

UNCHARTED
DRAKE'S FORTUNE

UNCHARTED 2
AMONG THIEVES

What is next?

UNCHARTED 3
DRAKE'S DECEPTION

early 2009 idea

"What if we have a ship in a storm...
and we turn it 180 degrees and sink it?
Wouldn't that be awesome?"

Jacob Minkof

Lead Designer

Simple displacement mesh with a shader will not be enough for this idea

The cruise ship

Nor for this rough ocean

Ship graveyard

This one was crazy. There is so much detail that we needed to capture on the ocean. All the action on the ship graveyard would be close to the water.
We would smoothly transition from a calm ocean to a stormier and stormier ocean

*


FFFFFFFFFFFF
FFFFFFFFFFFFFFFUUU
UUUUUUUU

*

* Thanks, we truly appreciate the challenge

Ocean

Render challenges

Open ocean, big waves (100 meters+)

Waves drive boats and barges

Animation loops were considered but not used

Swimming

Initially designers thought about using a can animation for the cruise boat
(silly designers, we can do better)
One problem was that once we are on the boat is hard to read the scale of the waves.
We are high on the boat and the camera attached to the player smooths out the waves.To compensate we have to exaggerated the amplitude of the waves to read as a storm.

Ocean

- Wave System
  - Procedural
  - Parametric
  - Deterministic
- LOD

Procedural

~~Simulation~~ = too expensive

~~Perlin like noise~~ = not that good visually

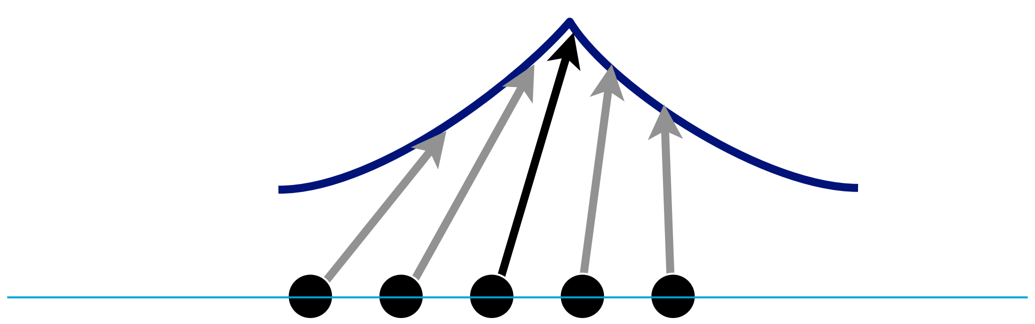
Procedural geometry and animation are good if we can find a good model.
Simulation would be too expensive to compute (even in SPUs) and hard to control by designers
Perlin noise results are not that great visually, tend to look very artificial
The FFT technique is great, but the parameters are hard to control and tweak by artists. Also is hard to get right.

Parametric

Evaluate at any point in $\mathbb{R}^2$ domain

$$F(<u,v>, t, parameters) \rightarrow <x,y,z>$$

Vector displacement, **not** a heightfield



Since we would have an ocean any point should be able to be computed.
The ocean was not restricted to a fixed grid and had to be compatible with other parametric equations.
This way we could do a compositing wave system

It's important to note we are generating a vector displacement.
One would need a super fine mesh to have by sharp wave peaks with a heightfield.
Its easier with a vector displacement, one does not need a fine mesh for it.

Deterministic

Evaluate at any point in $\mathbb{R}^2$ space and **time**

$$F(<u,v>, t, parameters) \rightarrow <x,y,z>$$

Needed for cutscenes and multiplayer

For FMA scenes we needed to be able to rewind the ocean, so we cannot "advance" the state. So our system is stateless
Our system is completely stateless

Waves

Gerstner waves

- Simple but not enough high frequency detail
- Can only use few [big swells] before it's too expensive

So all these requirements force us to certain techniques to use to evaluate waves.
So the easiest ones are the Gerstner waves. These are the workhorse of waves, any one uses them. However, they get expensive to evaluate to get a good number (20+) of them.

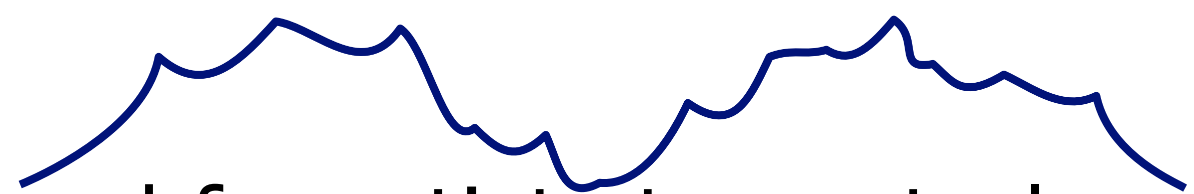
The FFT technique is the most realistic one. But is hard to get the right parameters, the artists have to spend lot of time searching for the "correct" values
We also found some tiling artifacts when one has small grid resolutions (64 side)

Waves

Gerstner waves

- Simple but not enough high frequency detail
- Can only use few [big swells] before it's too expensive

FFT Waves - Tessendorf "Simulating Ocean Water", 1999



\+ Realistic, more detail

\- Spectrum of frequencies - hard for artists to control

½ Tiling visual artifacts at low resolution grids

So all these requirements force us to certain techniques to use to evaluate waves.
So the easiest is Gerstner waves. The work horse of waves, any one uses them. However, they get expensive to evaluate to get a good number (20+) of them.
FFT again the most realistic ones. But as we said, hard to get right parameters. We found some tiling artifacts

Wave Particles

Yuksel, House, Keyser, SIGGRAPH 2007

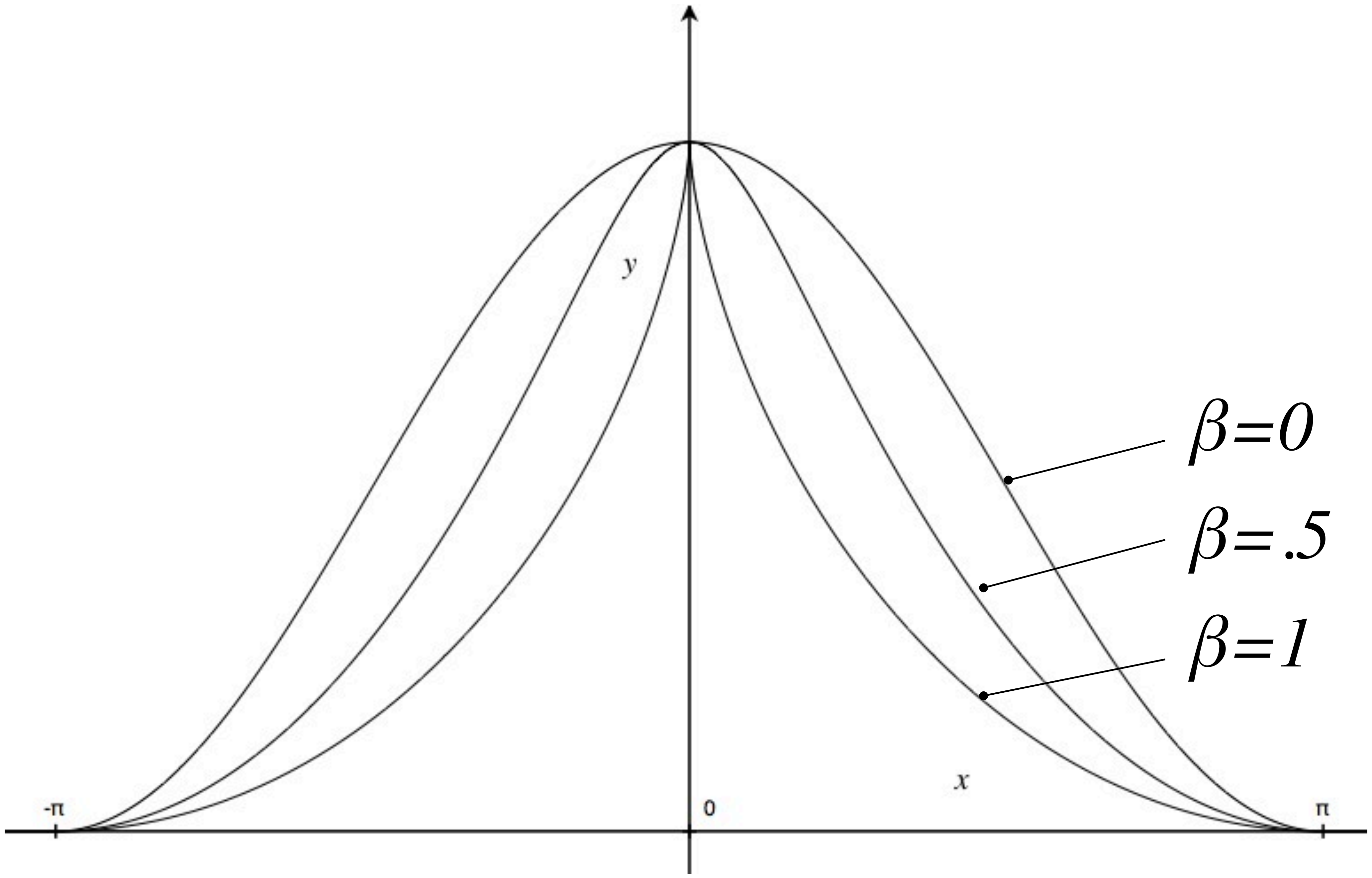
Waves from point sources

Ours -> Don’t use point sources
Instead, in a toroidal domain place a random distribution of particles to approximate the chaotic motion of open water

Random positions and velocities within a some speed bounds

$\longrightarrow$ Yields a tileable vector displacement field

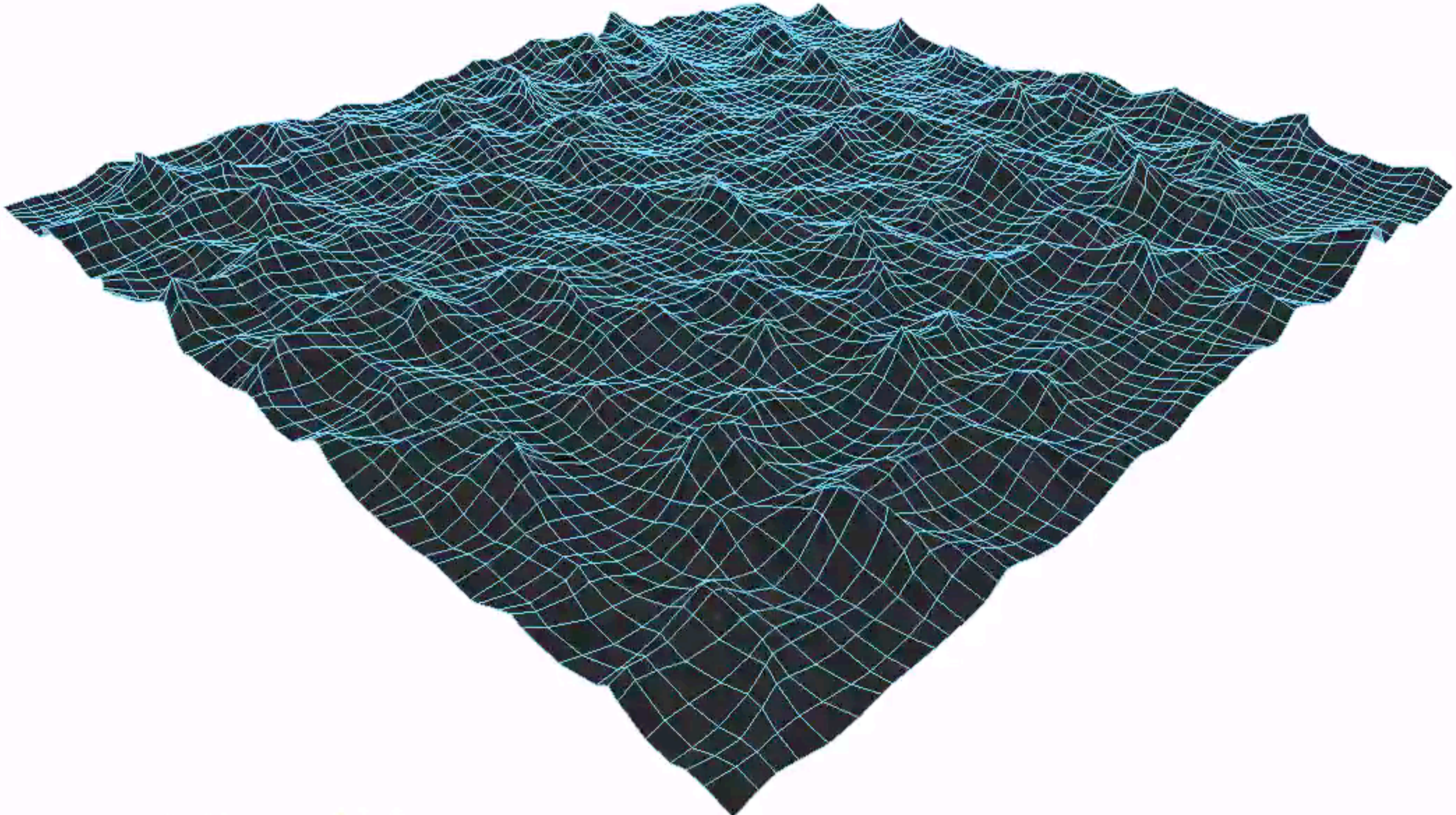
This is the main idea to get our open ocean waves. It's easy to implement and to optimize.
See the Wave Particle paper for more details



$$x = t - \beta \sin(t)$$

$$y = \frac{1}{2}(cos(t) - 1)$$

$$t \in [-\pi..\pi]$$

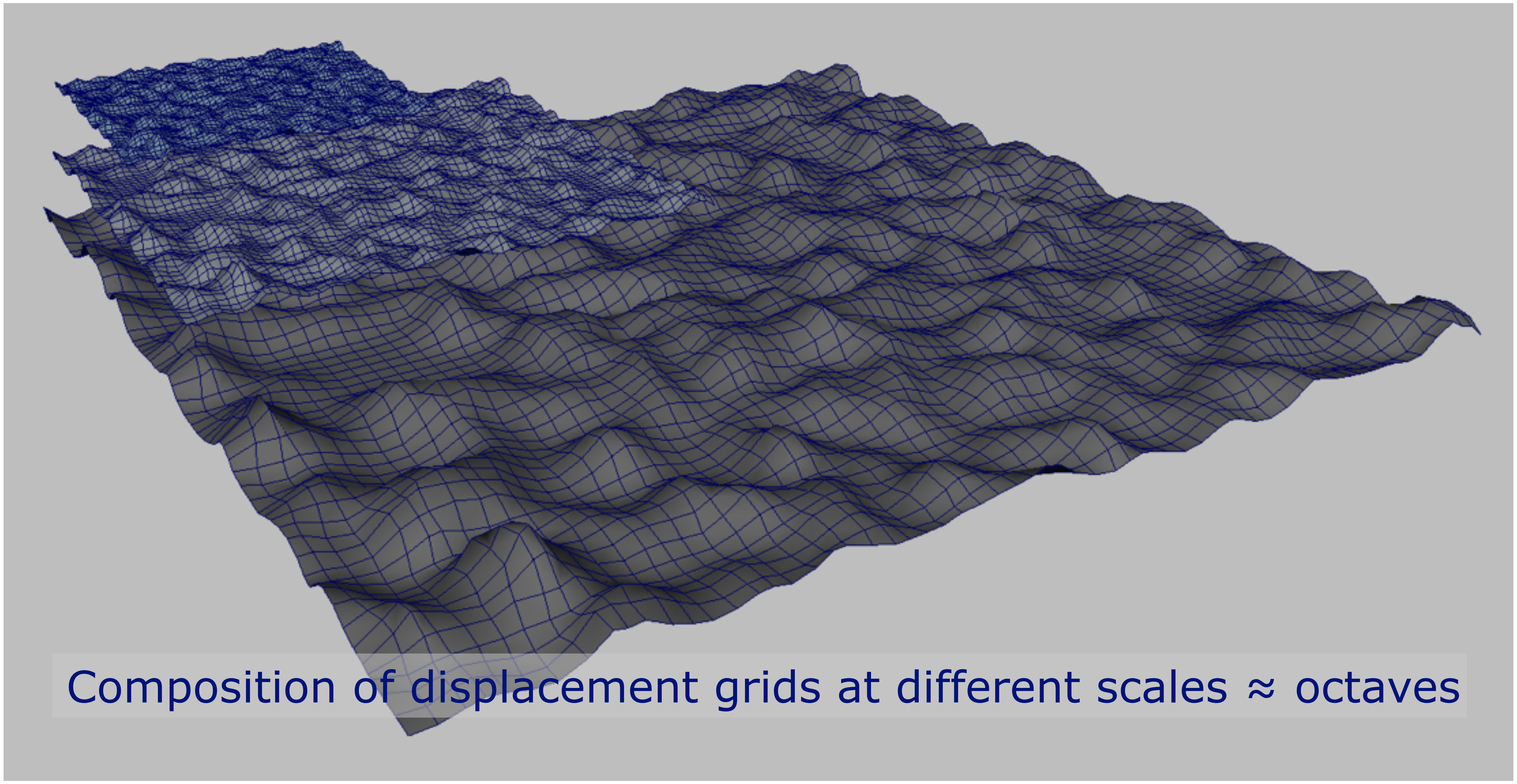


num particles: 600

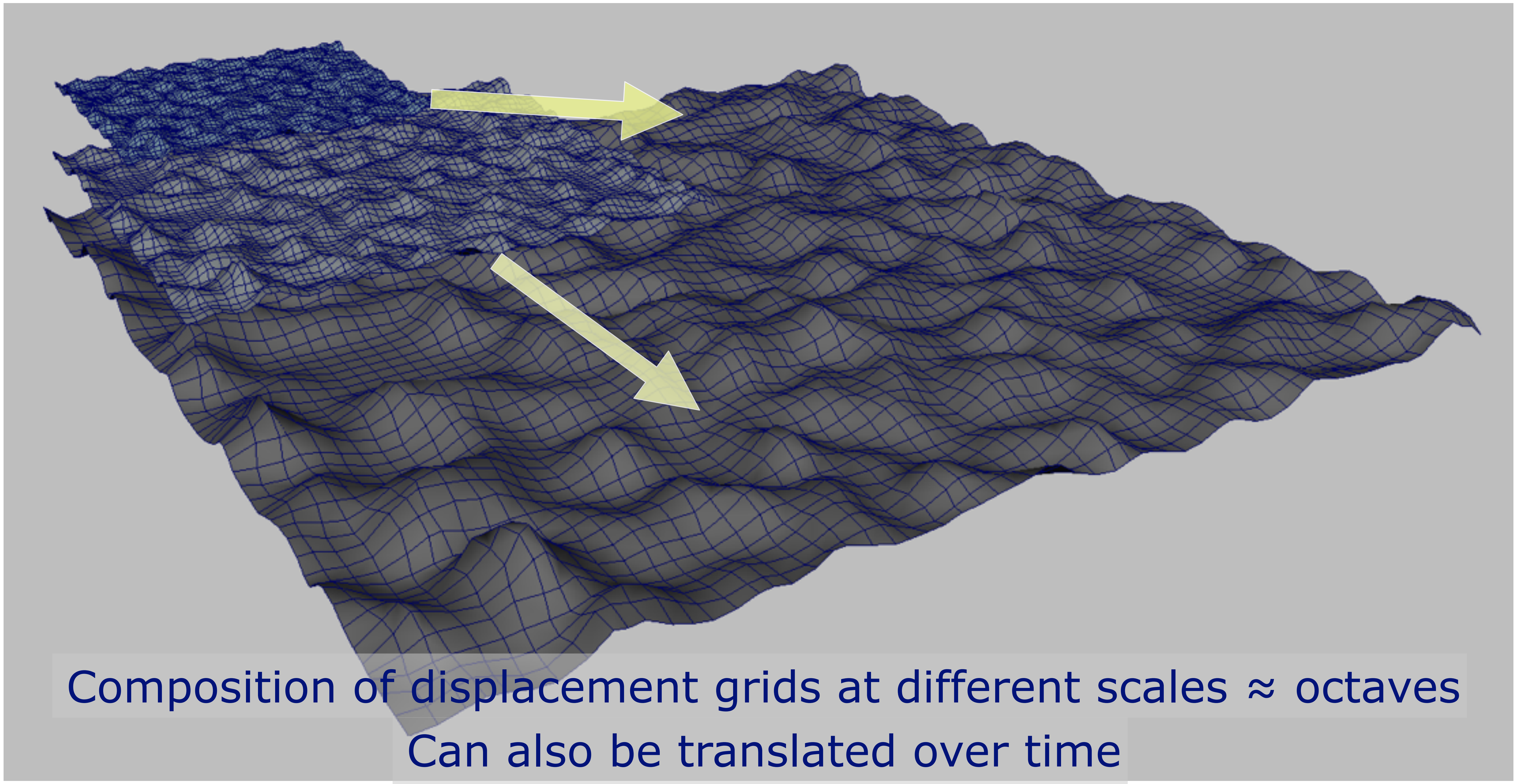
Wave Particles

+ Intuitive for artists to control
+ No tiling artifacts
+ Fast! Good for SPU vectorization

  [Optimization by Michal Iwanicki]
+ Deterministic in time. No need to move particles
  New position is derived from initial position, velocity and time



Composition of displacement grids at different scales ≈ octaves



Composition of displacement grids at different scales ≈ octaves

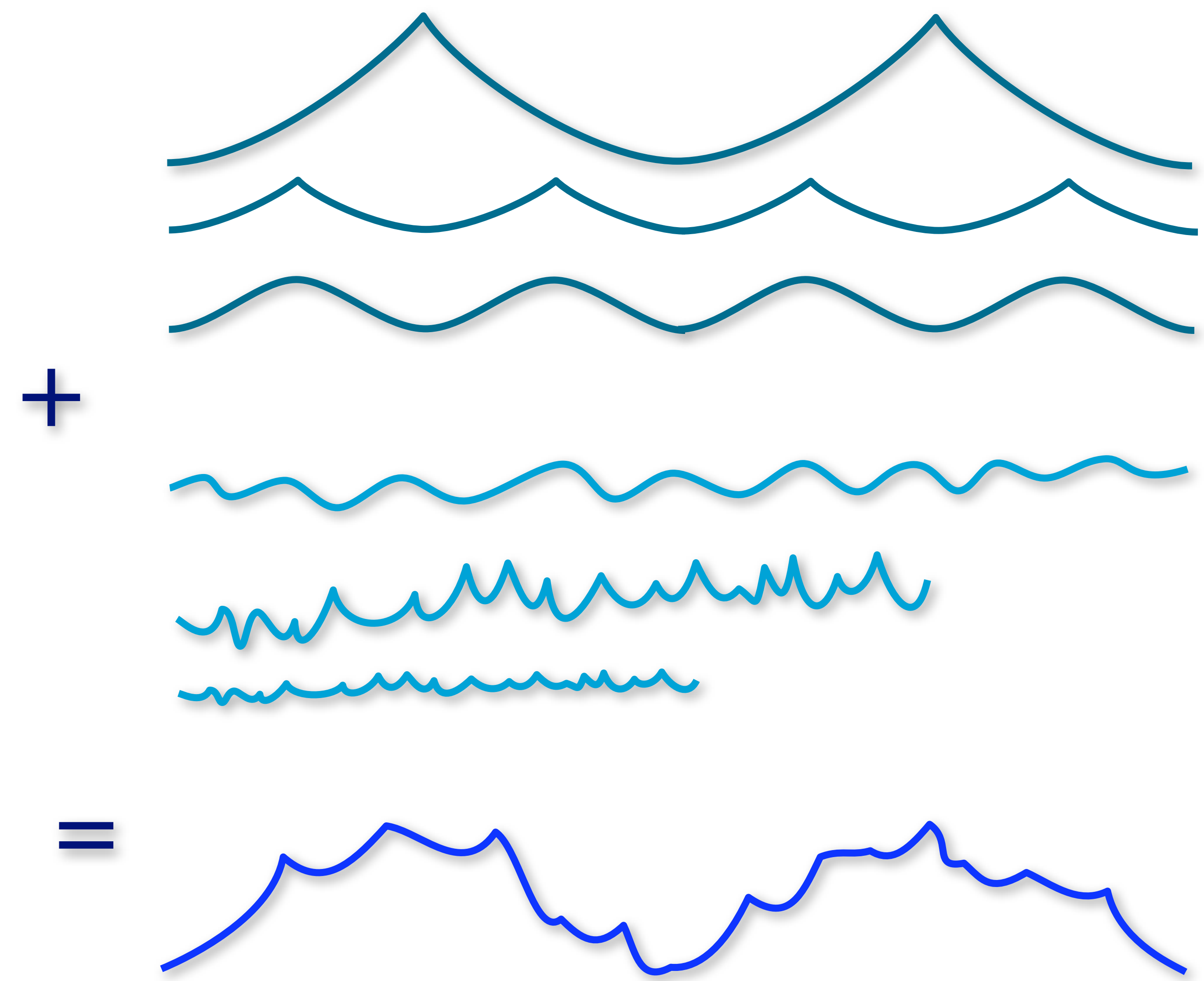
Can also be translated over time

Fade out by distance

Wave field

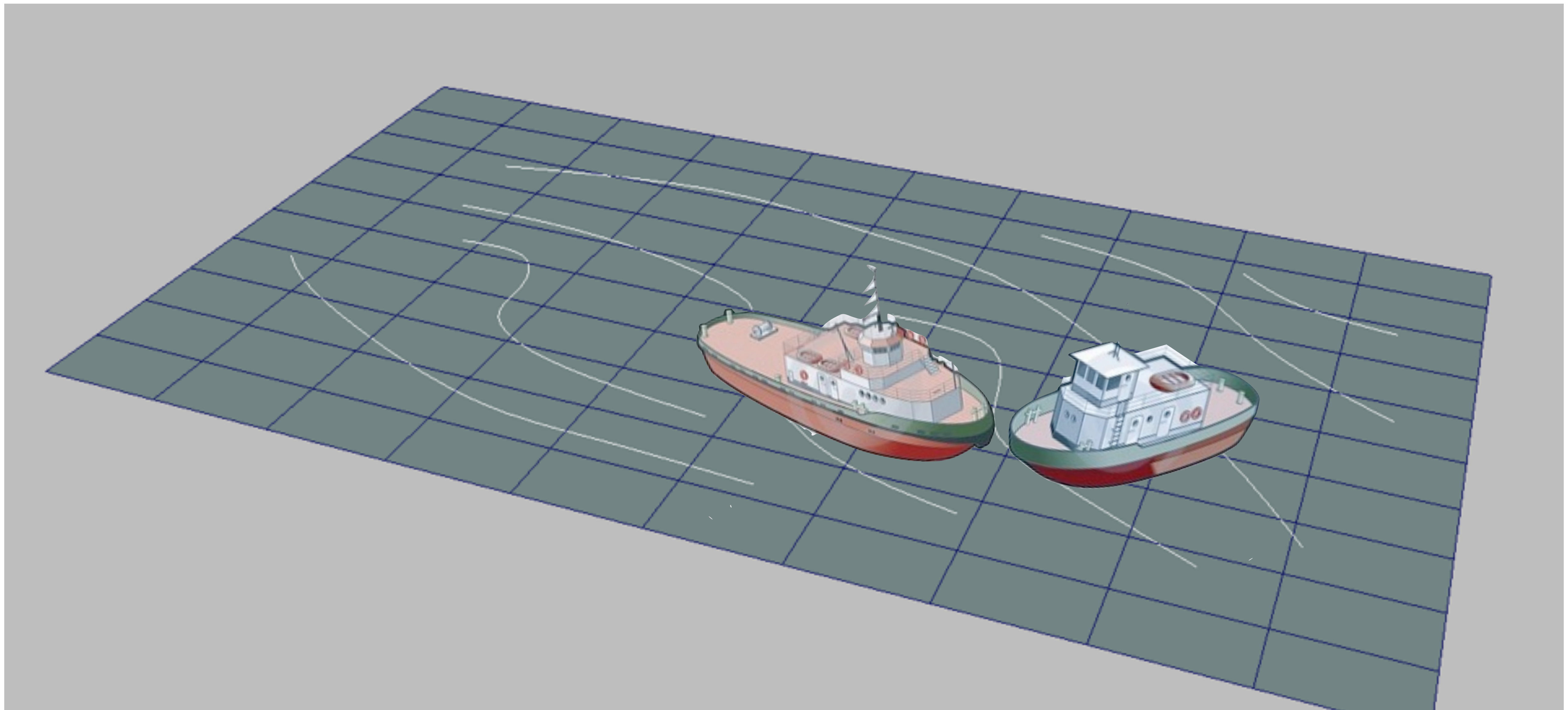
Gerstner waves (x4)

Wave particles
(1 grid used x4)



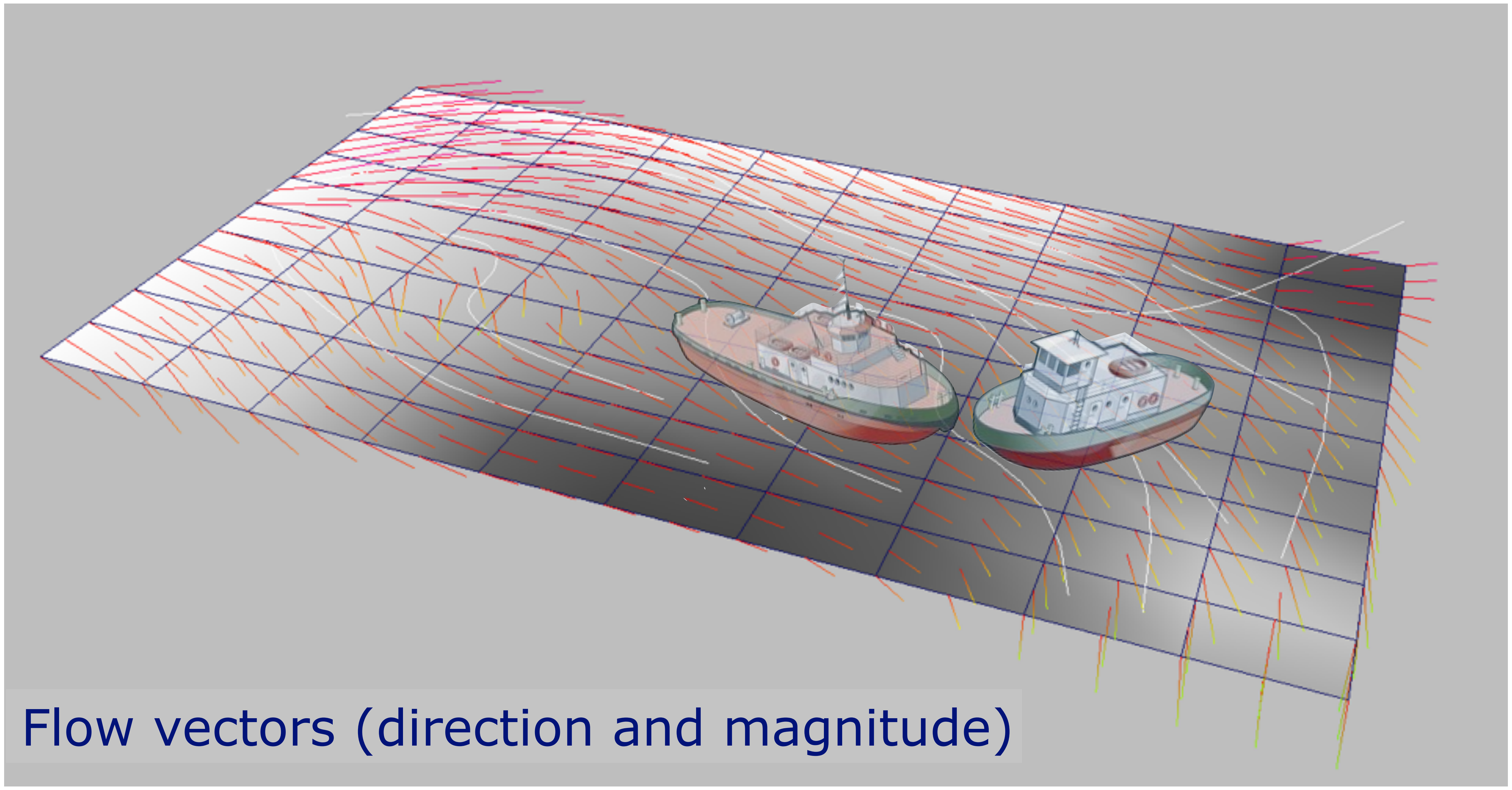
At this point , the displacement field, evaluates 4 gerstner waves plus 4 wave-particle grids per vertex

Flow grid: Encode flow, foam, amplitude multipliers in a grid

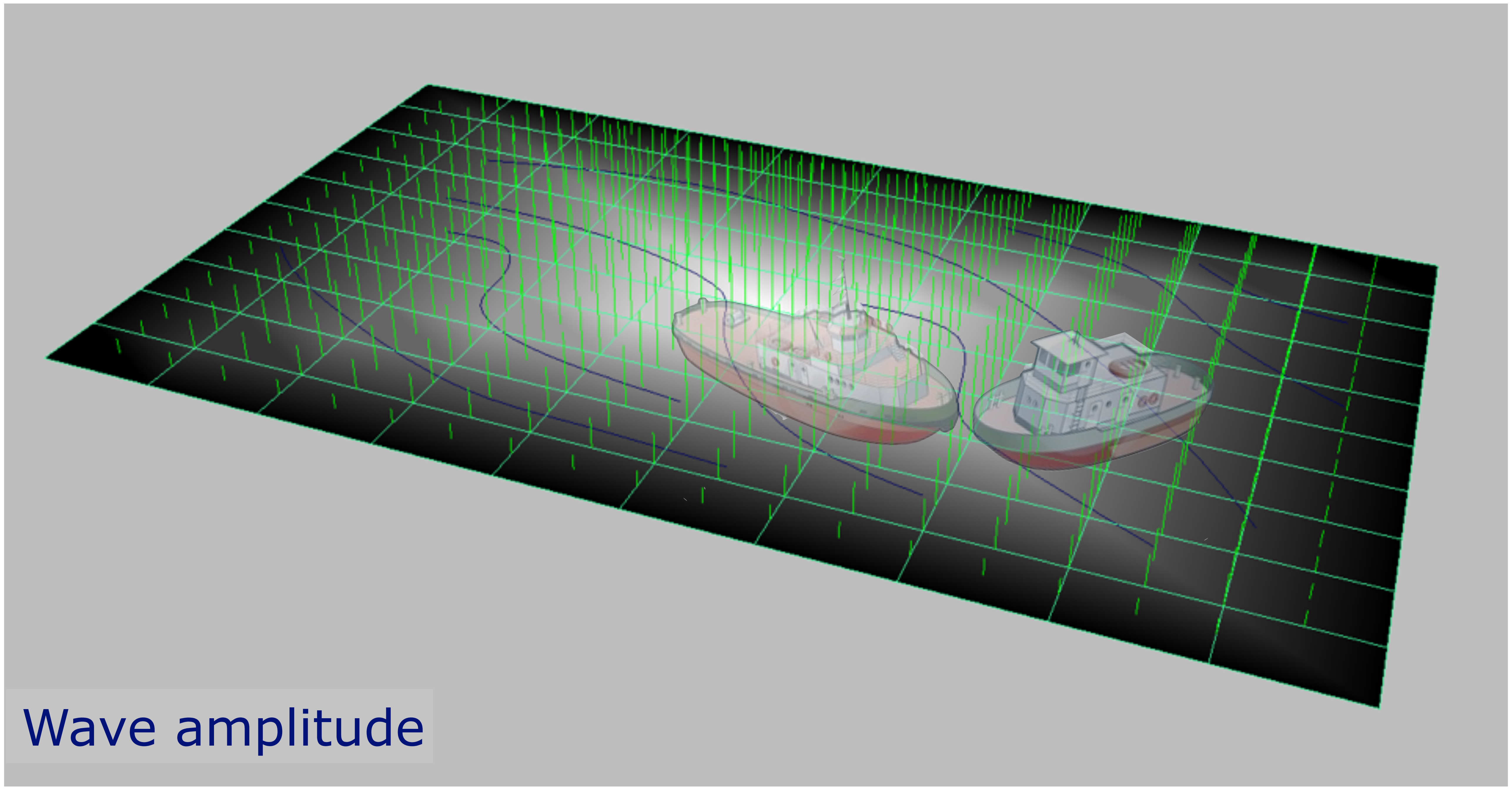


Base grid with flow curves

Flow grid: Encode flow, foam, amplitude multipliers in a grid



Flow grid: Encode flow, foam, amplitude multipliers in a grid



Wave amplitude

Wave's amplitude

modulates foam

The foam modulation is outputed as a vertex color when we generate the final mesh


Hey, we need this!
Robh

Addition of simpler waves

Gerstner waves (x4)

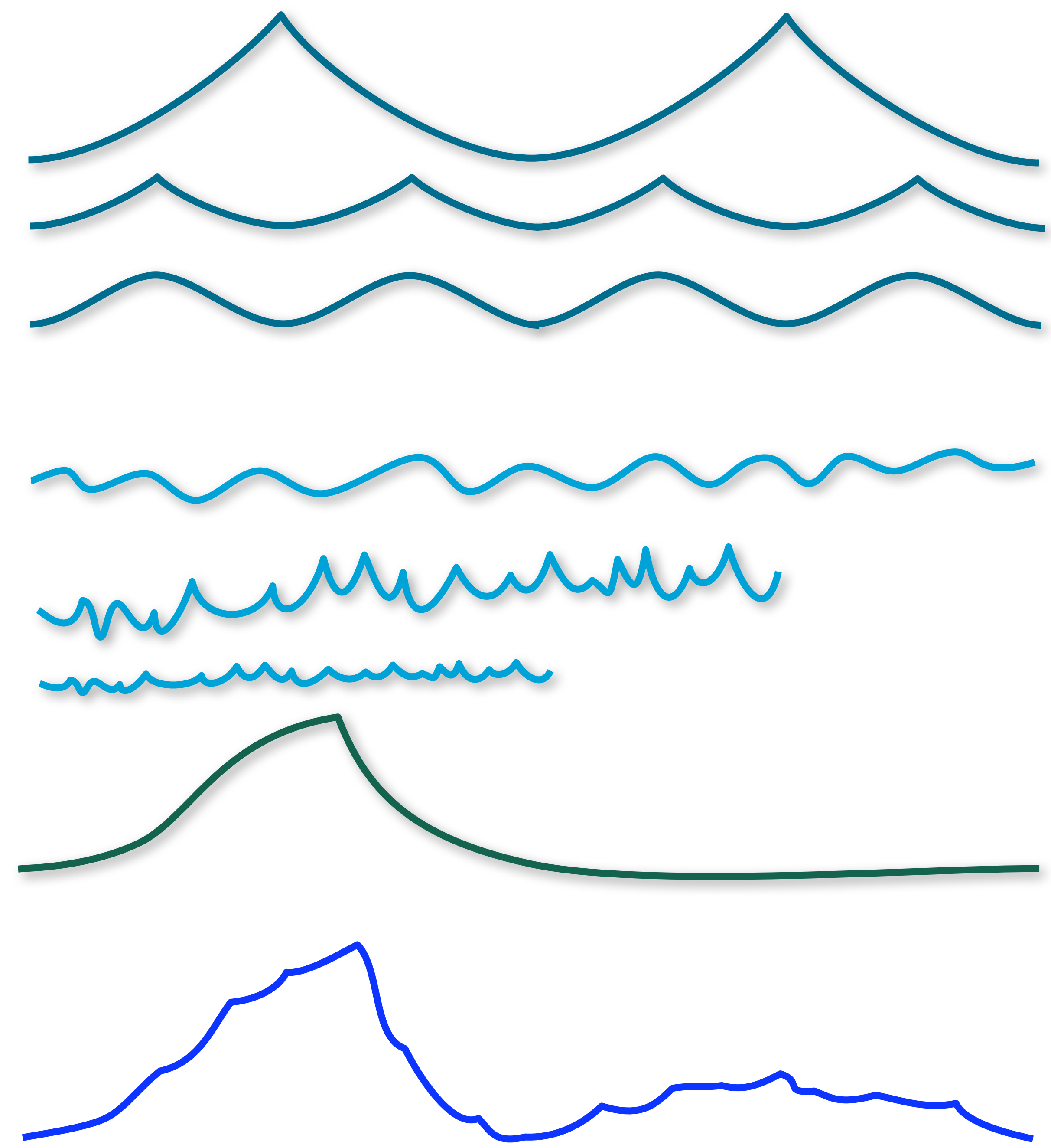
\+

Wave particles (x4)

Big wave

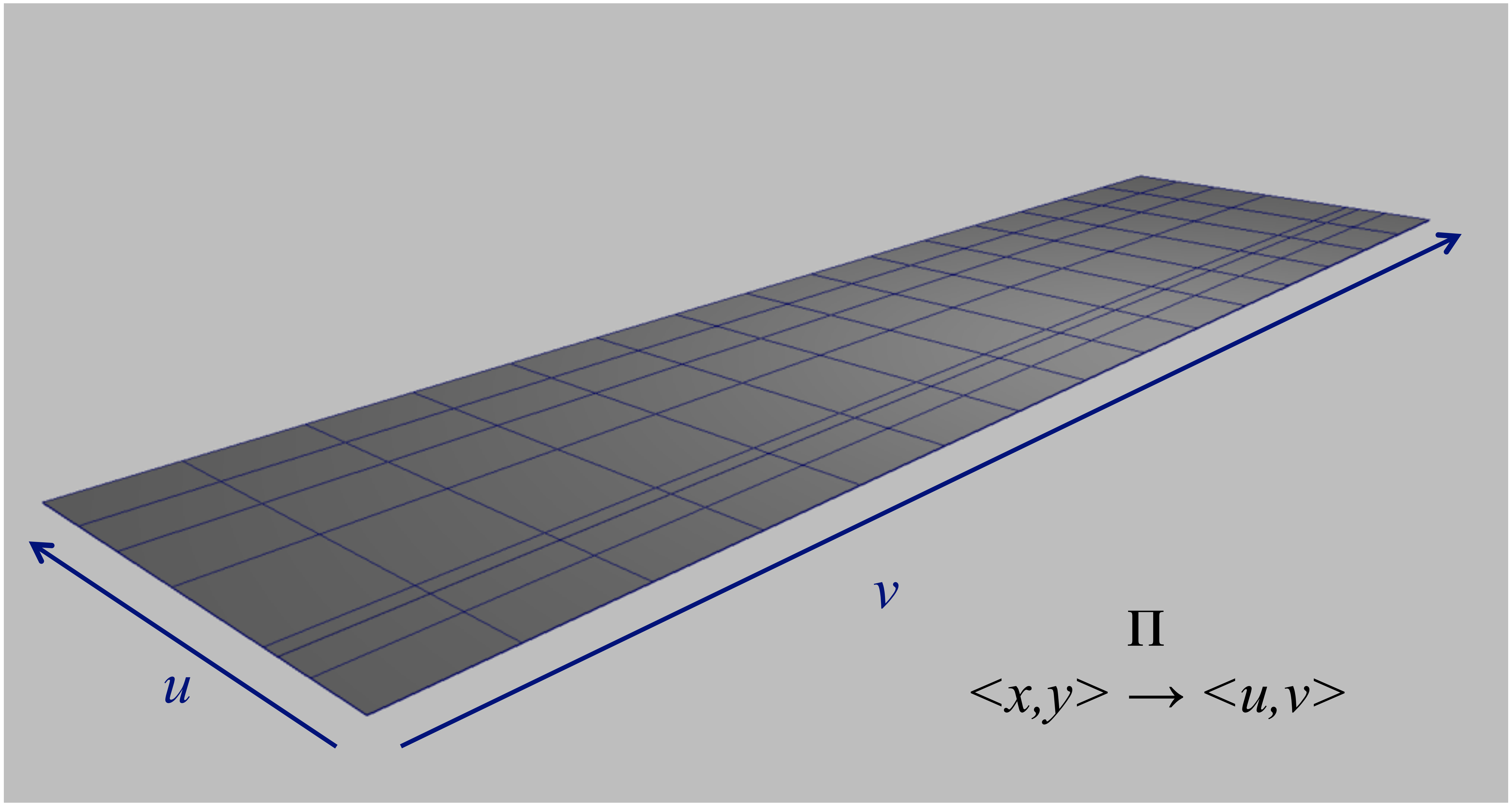
\+

=



In addition to the previous waves, we add a custom artist wave.

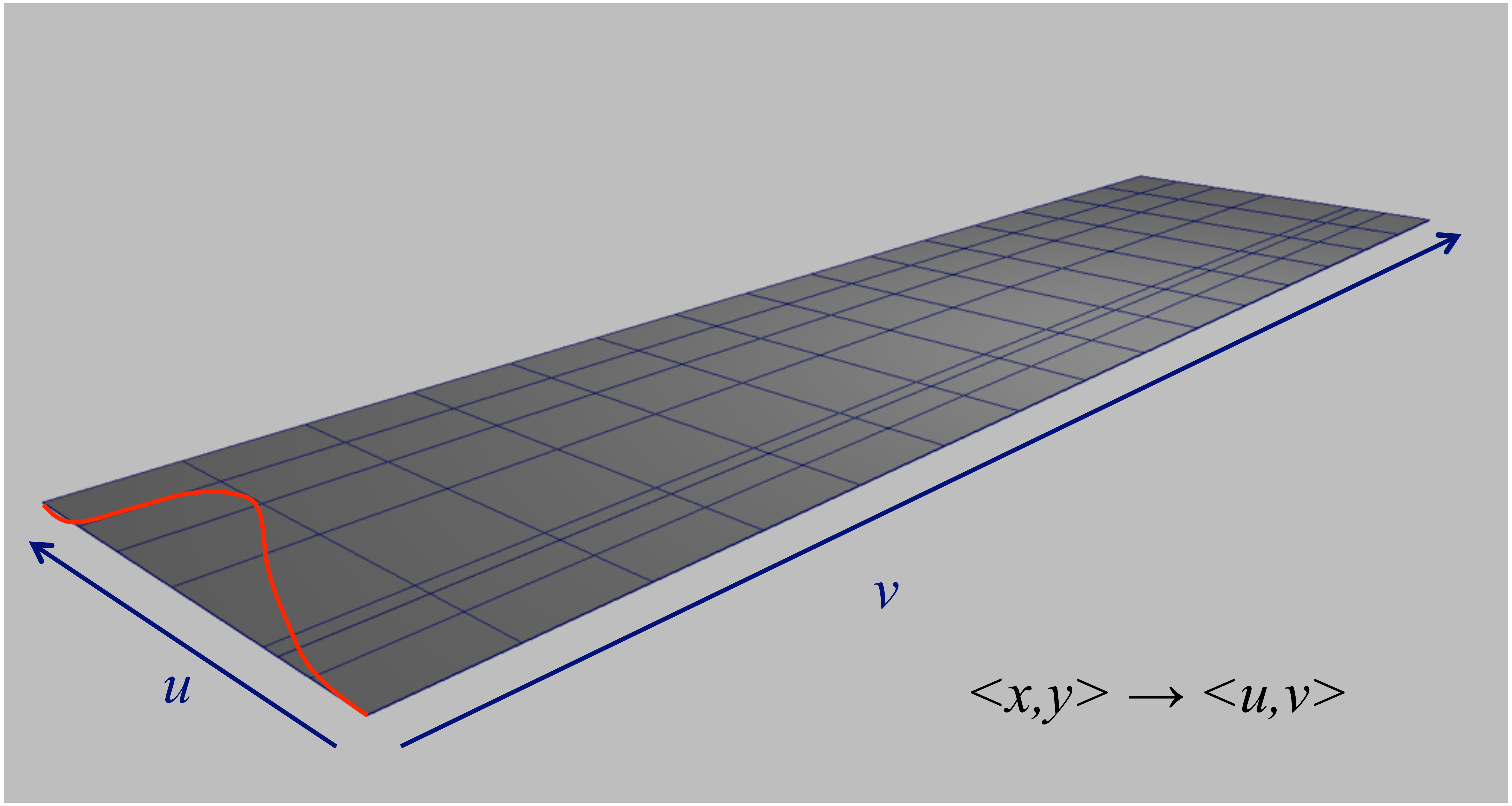
This type of wave will be used for the crash wave scene and to also keep the player from swimming away from the game play area



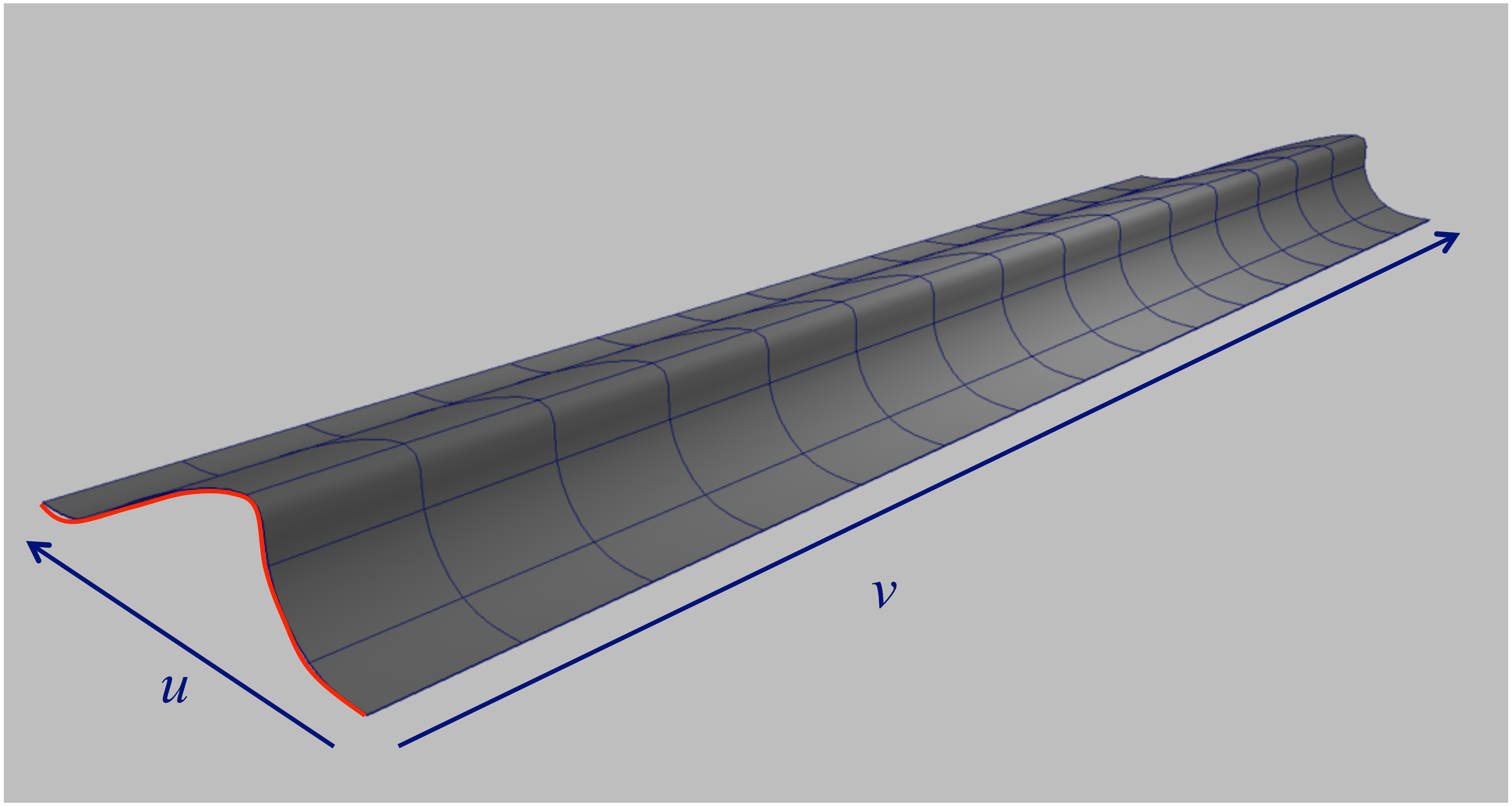
Big wave - Orient a square region Π over domain

The wave is quite easier to construct. We need to reparametrize a rectangular domain over the plane.
The rectangle can be though as the base of a NURBS patch.
We can use the (u,v)->(0..1, -1..1) domain or the normalized one(0..1, 0..1), depending where is the center.
The u is used to parameterize a Bspline curve. The v we extrude the curve and tapper it.

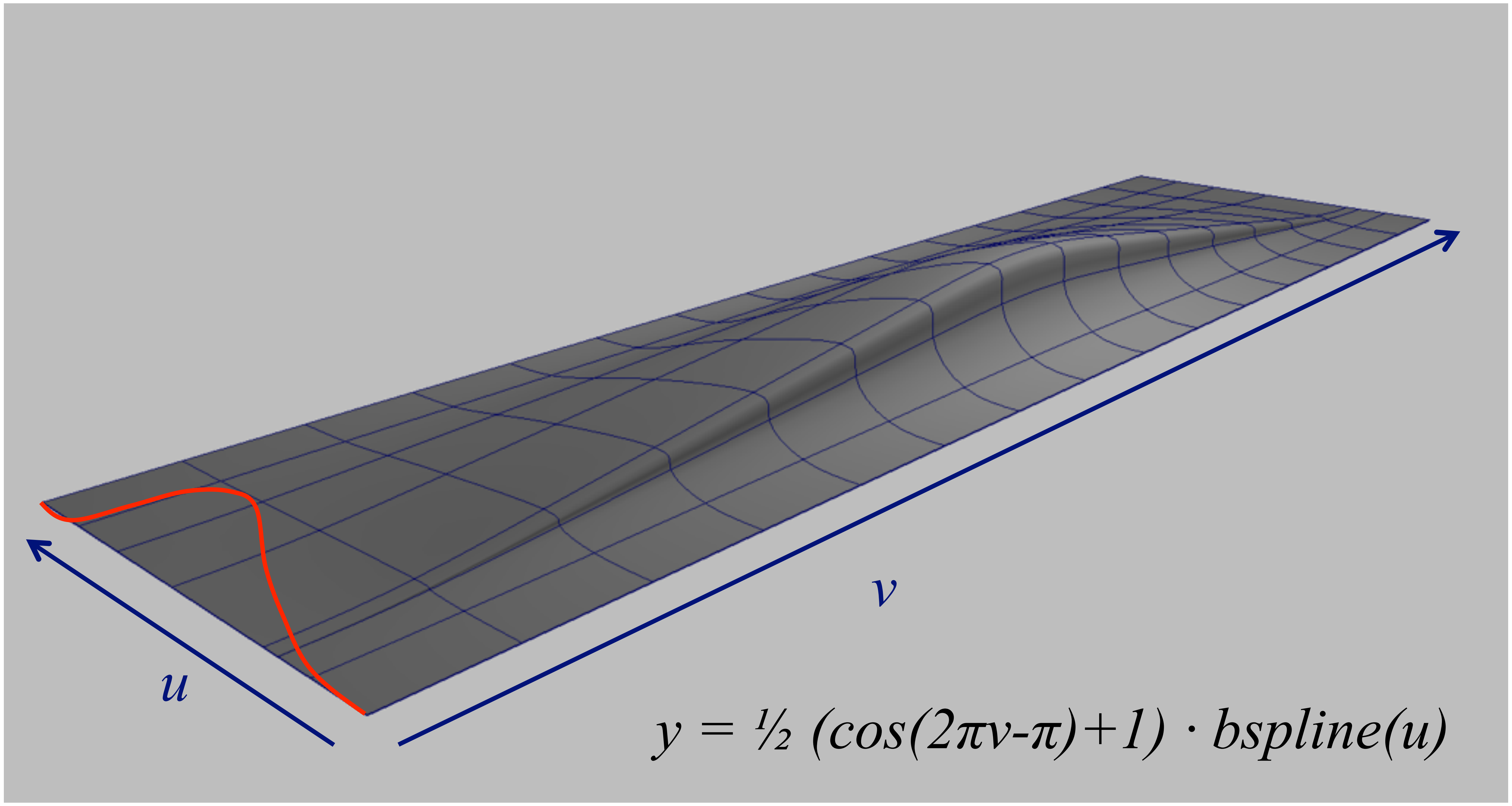
The subdomain can be scaled and translate over time to animate the wave



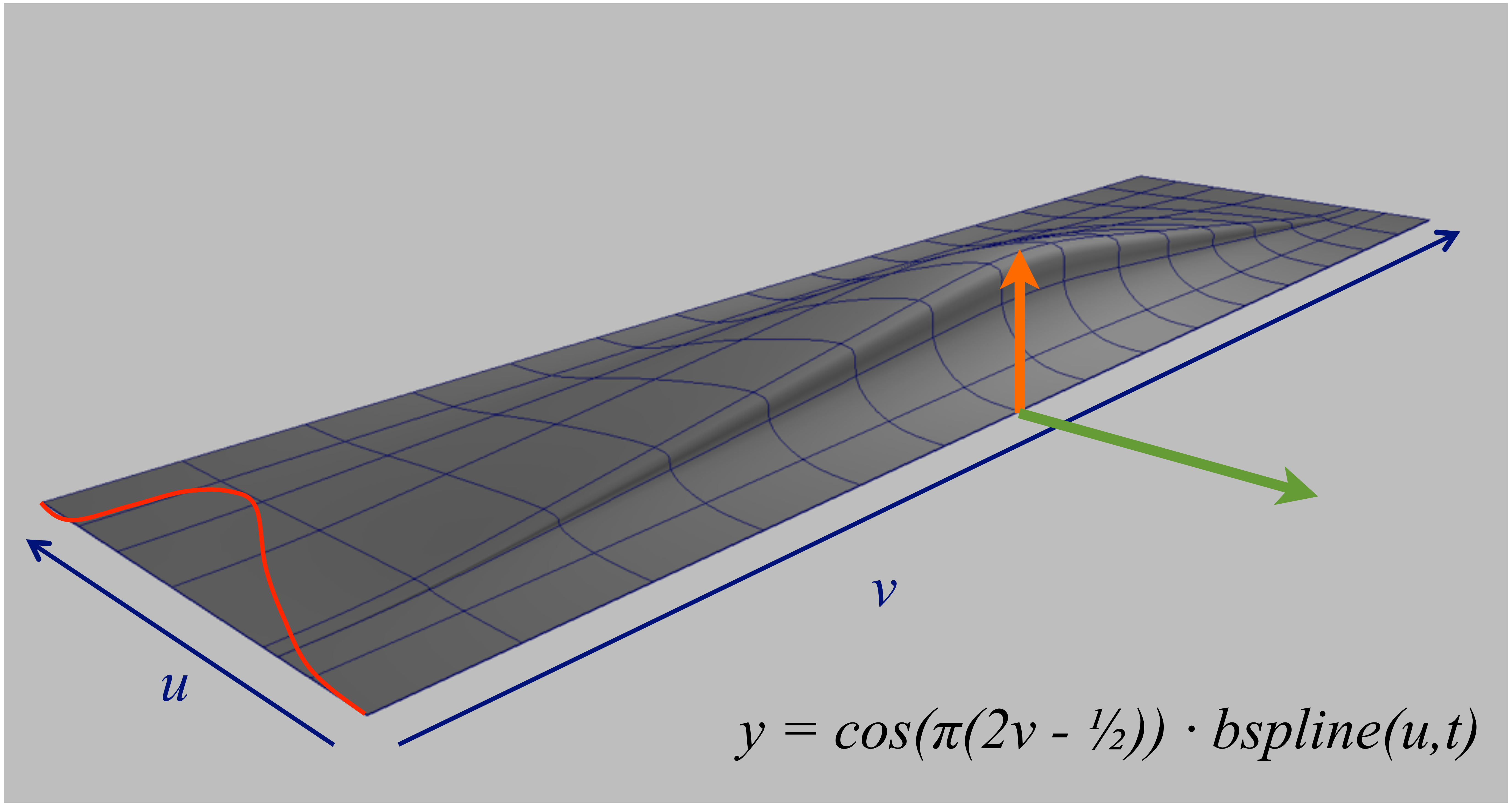
Big wave - Create a Bspline curve oriented in the ***u*** direction



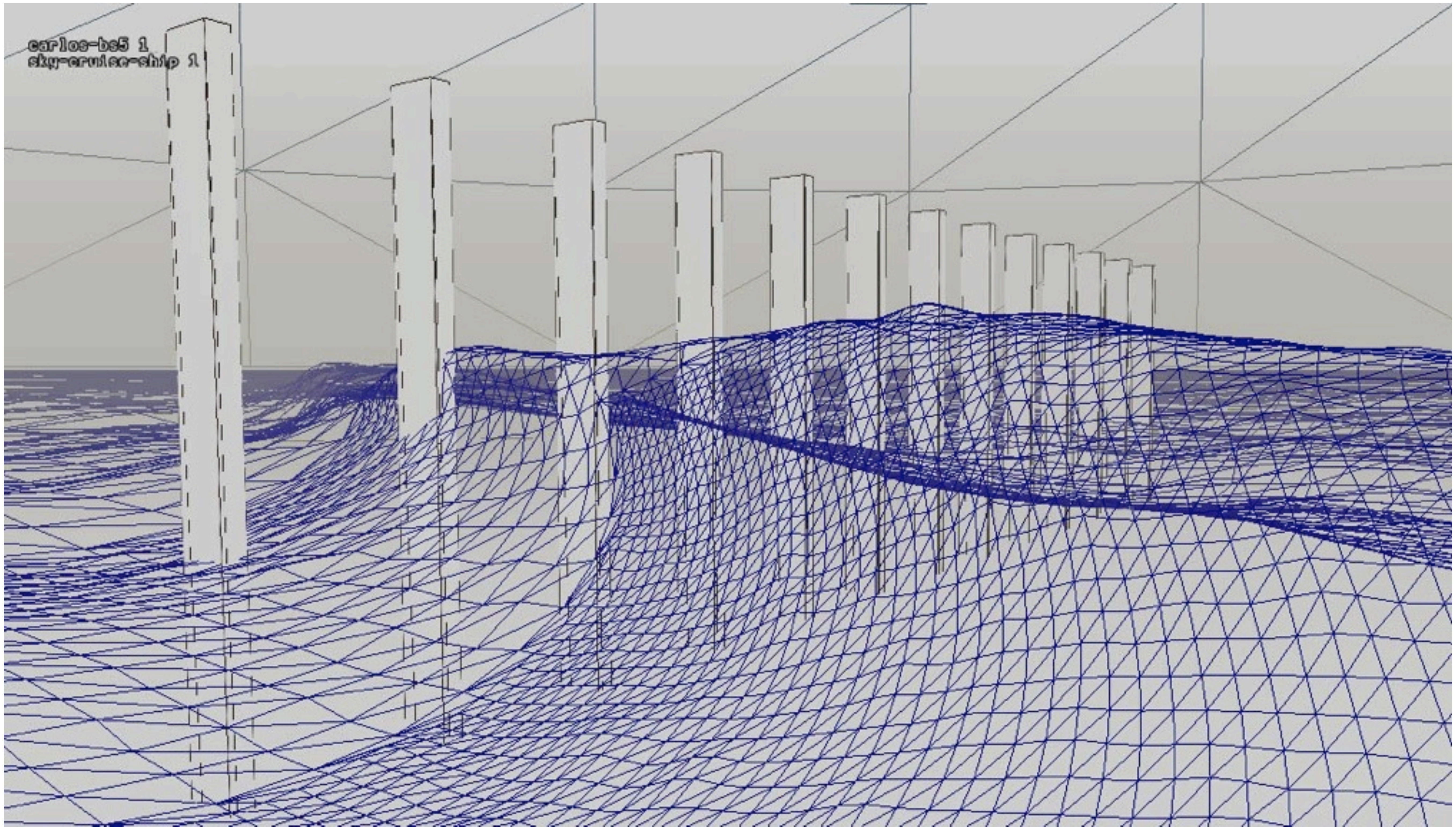
Big wave - Extrude the Bspline along v direction

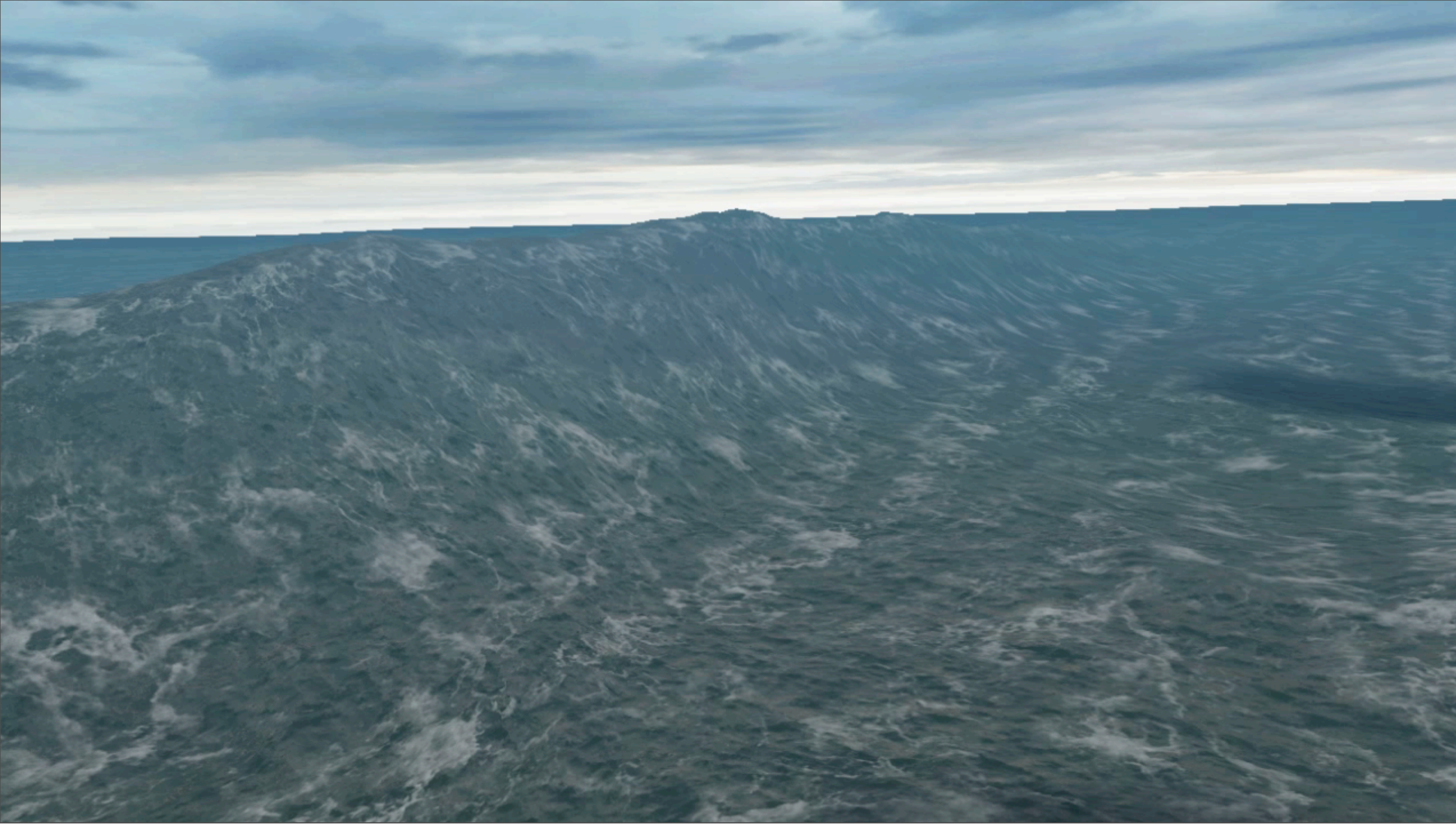


Big wave - Taper on the sides



Big wave - Translate, scale region or curve over time


carlos-bs5 1
sky-cruise-ship 1



$$f(u,v,t,\mathbf{p}) = \alpha \cdot \mathrm{grid}(u,v) \sum_i \mathrm{gerstner}(u,v,p_j)$$
$$+ \alpha \cdot \mathrm{grid}(u,v) \sum_j \mathrm{wp}(u,v,p_k)$$
$$+ \Pi(u',v'), \cos(\pi(2v' - 1/2) \cdot \mathrm{bspline}(u',t,p_b))$$

$$u' = u\cos(\theta)u + v\sin(\theta)$$
$$v' = u\sin(\theta)u - v\cos(\theta)$$

$$\mathbf{gerstner}(x_0, A, \lambda, t) = \begin{cases} x = x_0 - (\mathbf{k}/k)A\sin(\mathbf{k}\cdot x_0 - wt) \\ y = A\cos(k \cdot x_0 - wt) \end{cases}$$

$$\mathbf{k} = 2\pi/\lambda$$

$$\mathbf{bspline}(u,v,\mathbf{p}) = \sum_{i=0}^{m-n-2} \mathbf{p}_i b_{i,n}(u) \ , \ t \in [u_n, u_{m-n-1}]$$

$$\mathbf{p}_i \in \mathbb{R}^2$$
$$b_{j,n}(u) = b_n(u - u_j), \qquad j = 0, \ldots, m-n-2$$
$$b_n(u) := \frac{n+1}{n} \sum_{i=0}^{n+1} \omega_{i,n}(u-u_i)_+^n \quad \omega_{i,n} := \prod_{j=0, j\neq i}^{n+1} \frac{1}{u_j - u_i}$$
$$(u-u_i)_+^n := \begin{cases} (u-u_i)^n & \textbf{if } u \geq u_i \\ 0 & \textbf{if } u < u_i \end{cases}$$

This is a partial formula of the whole wave system.
The bspline is a uniform, non-rational bspline. We could have used a Bezier but it requires more code.

The *grid(u,v)* function returns a scalar value given the *u,v,* coordinates. In this case, we have a multiplier for the wave scale

LOD

Many ways to create the water mesh

Screen projected grid → aliasing artifacts

Quasi projected grid → issues handling large displacements

Irregular Geometry Clipmaps

Based on:

Losasso, Hoppe, “Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids” SIGGRAPH 04

Modified for water rendering

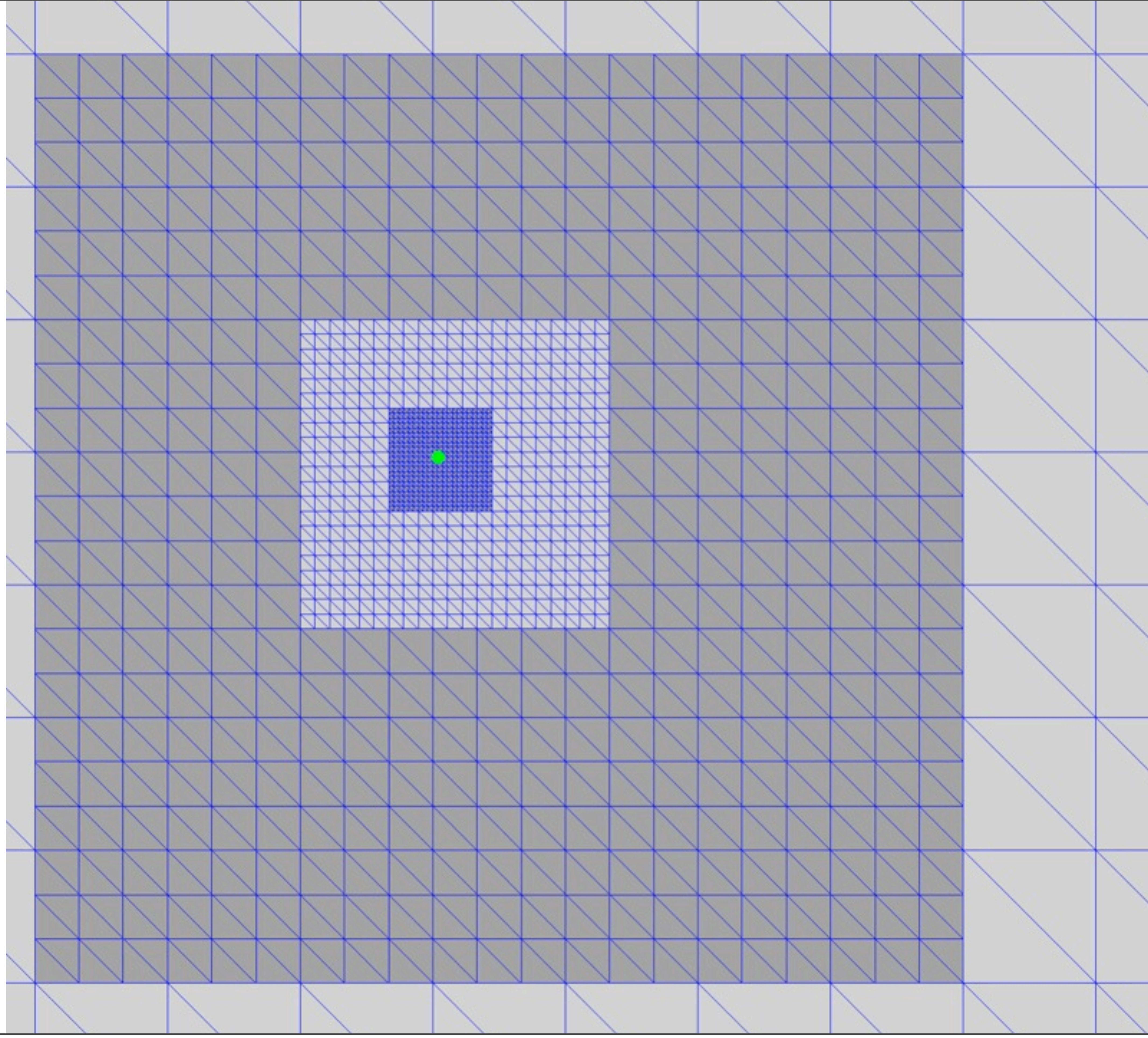
The irregular geometry clipmaps have different way to partition the rings and the blocks.

Irregular Geometry Clipmaps

Different splits to fix T-joints across ring levels

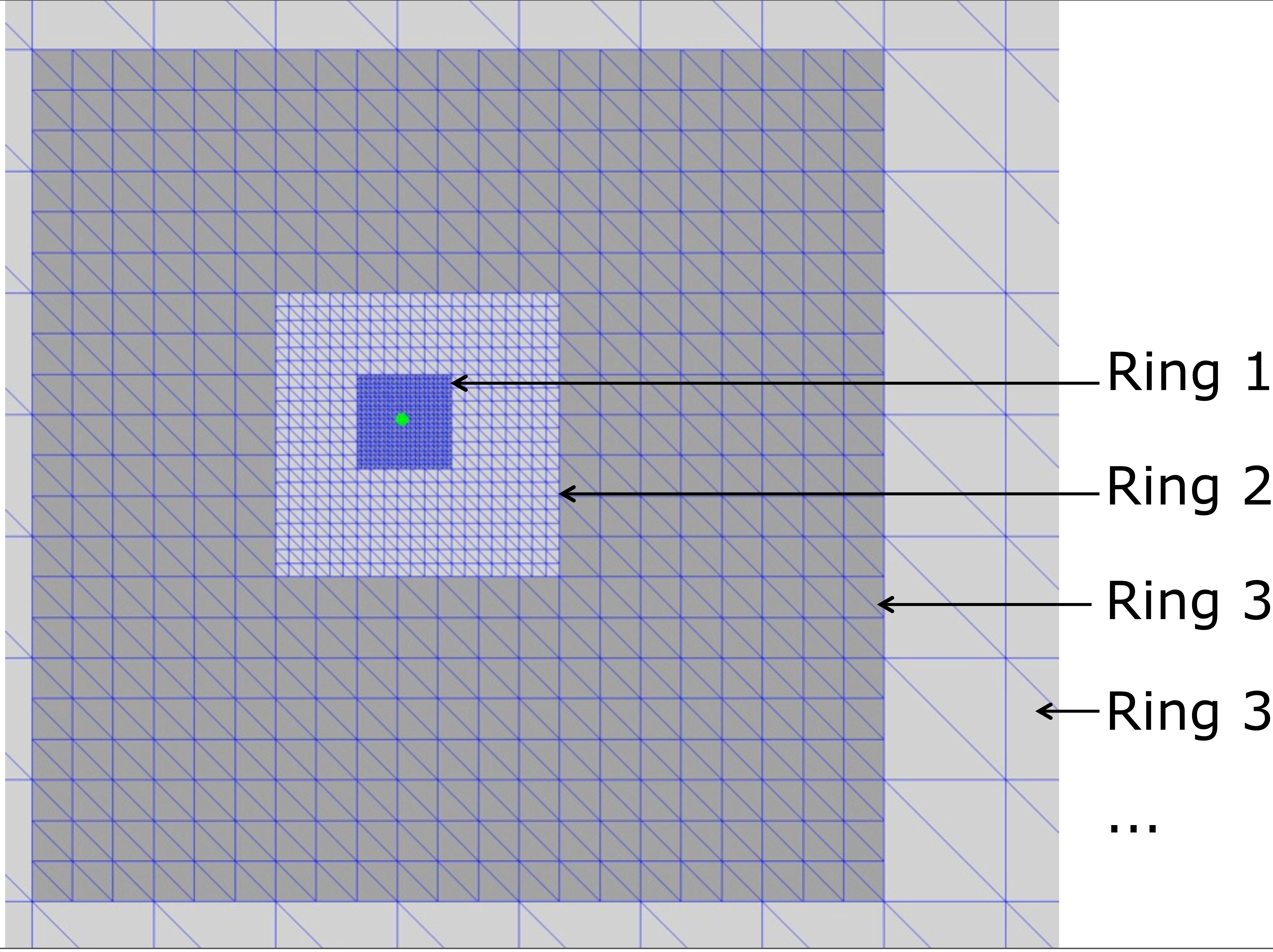
Dynamic blending between levels

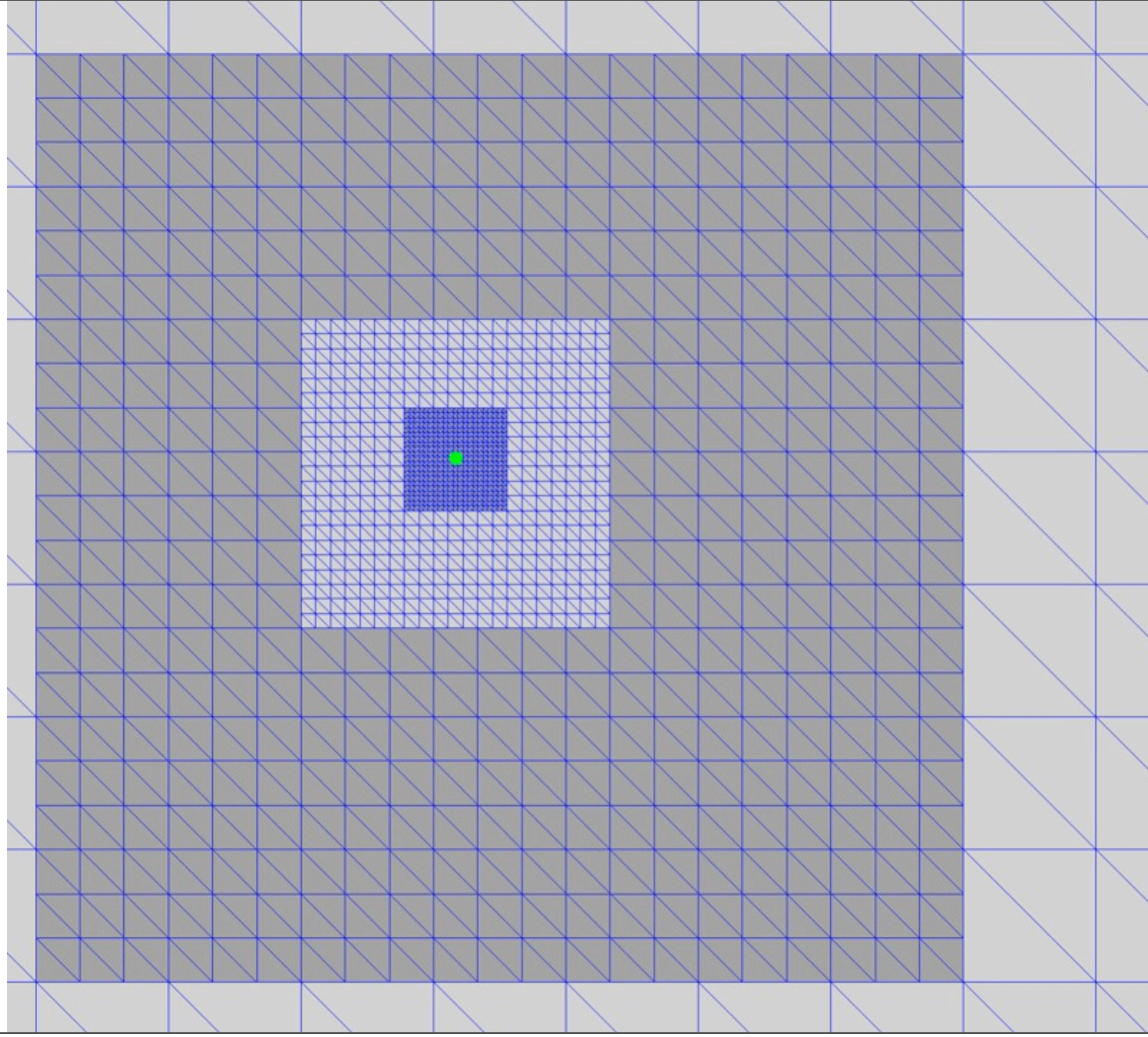
Patches lead to better SPU utilization

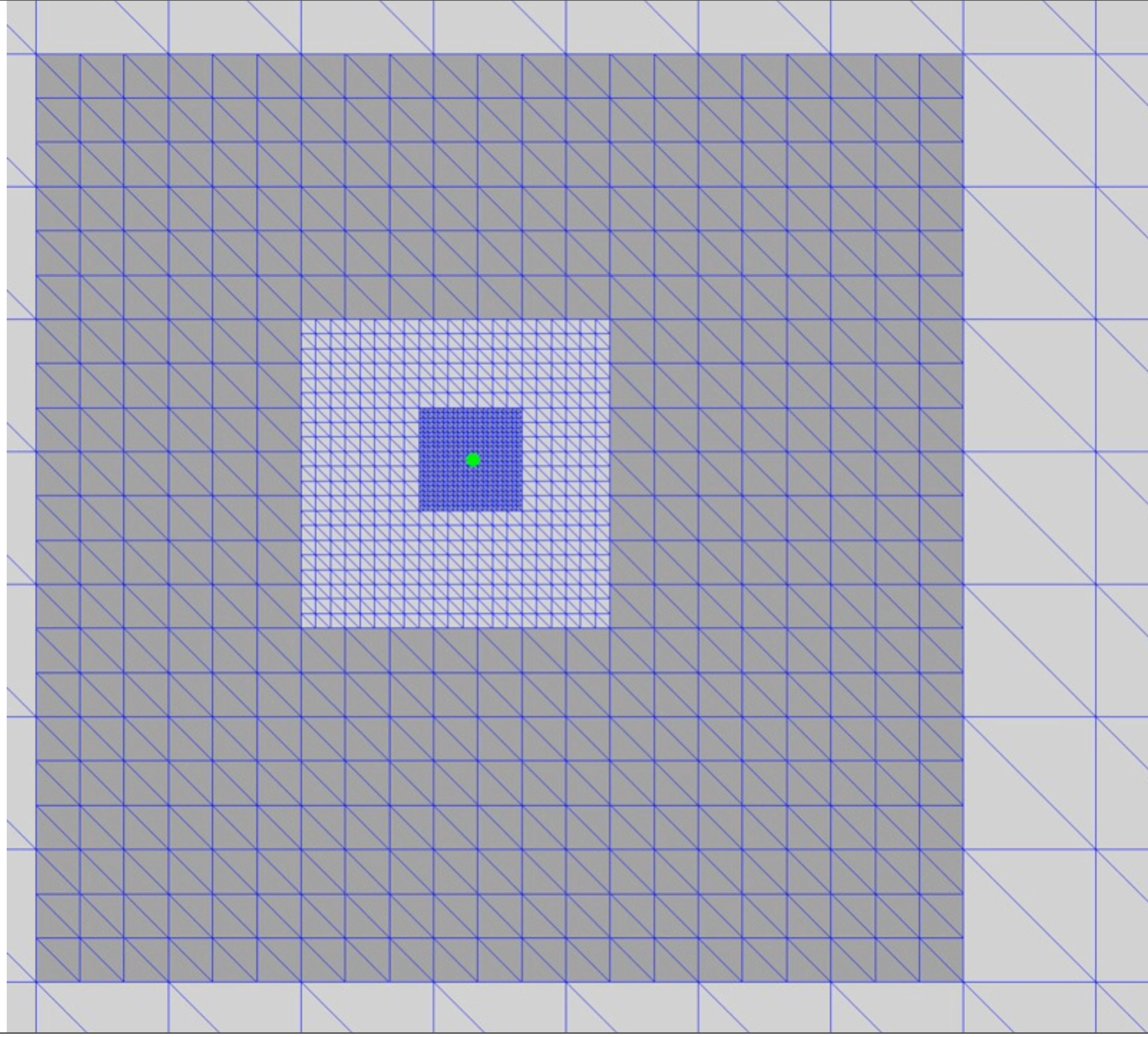


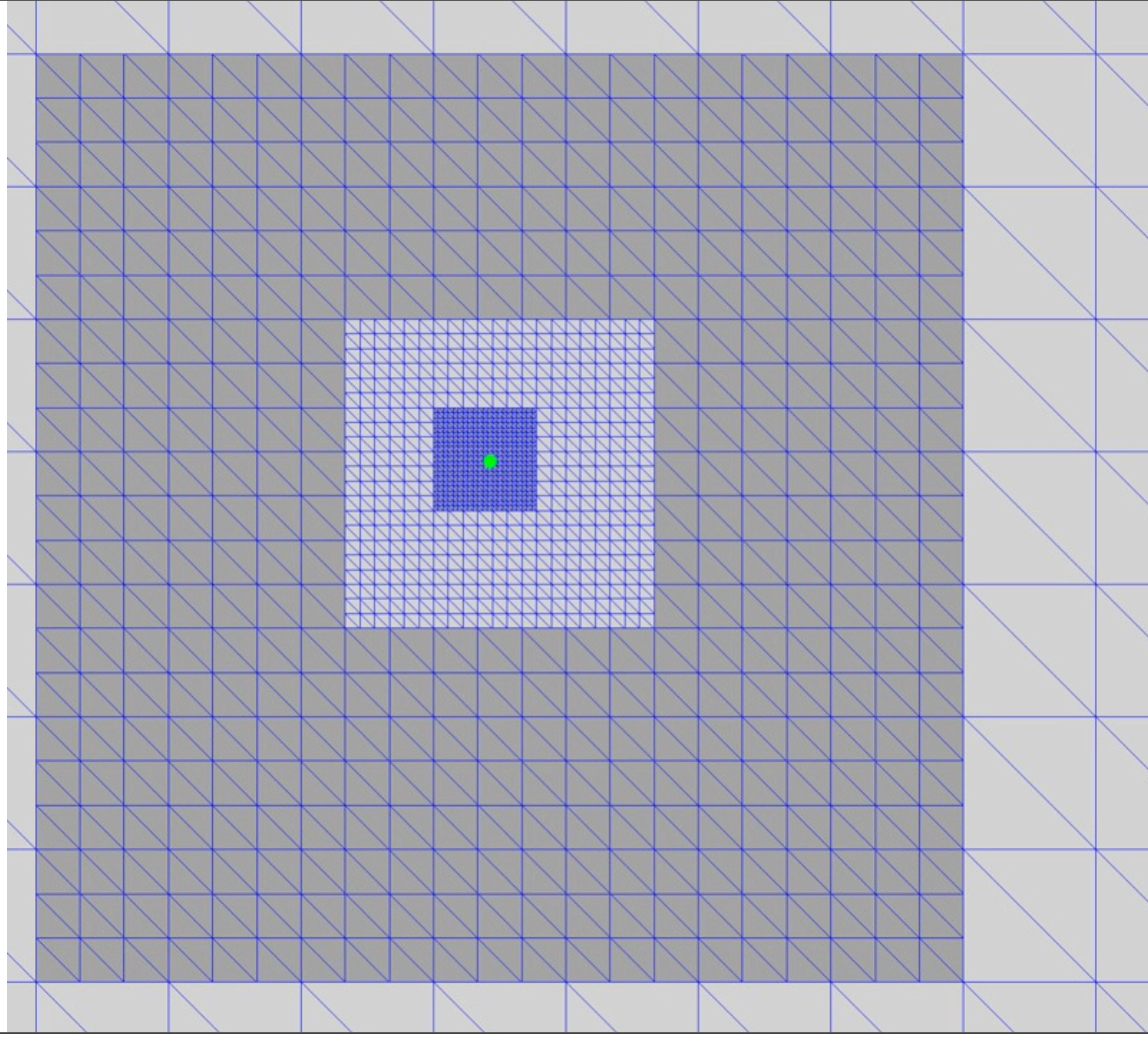
As the camera moves, the rings will move quads from one side to the other.

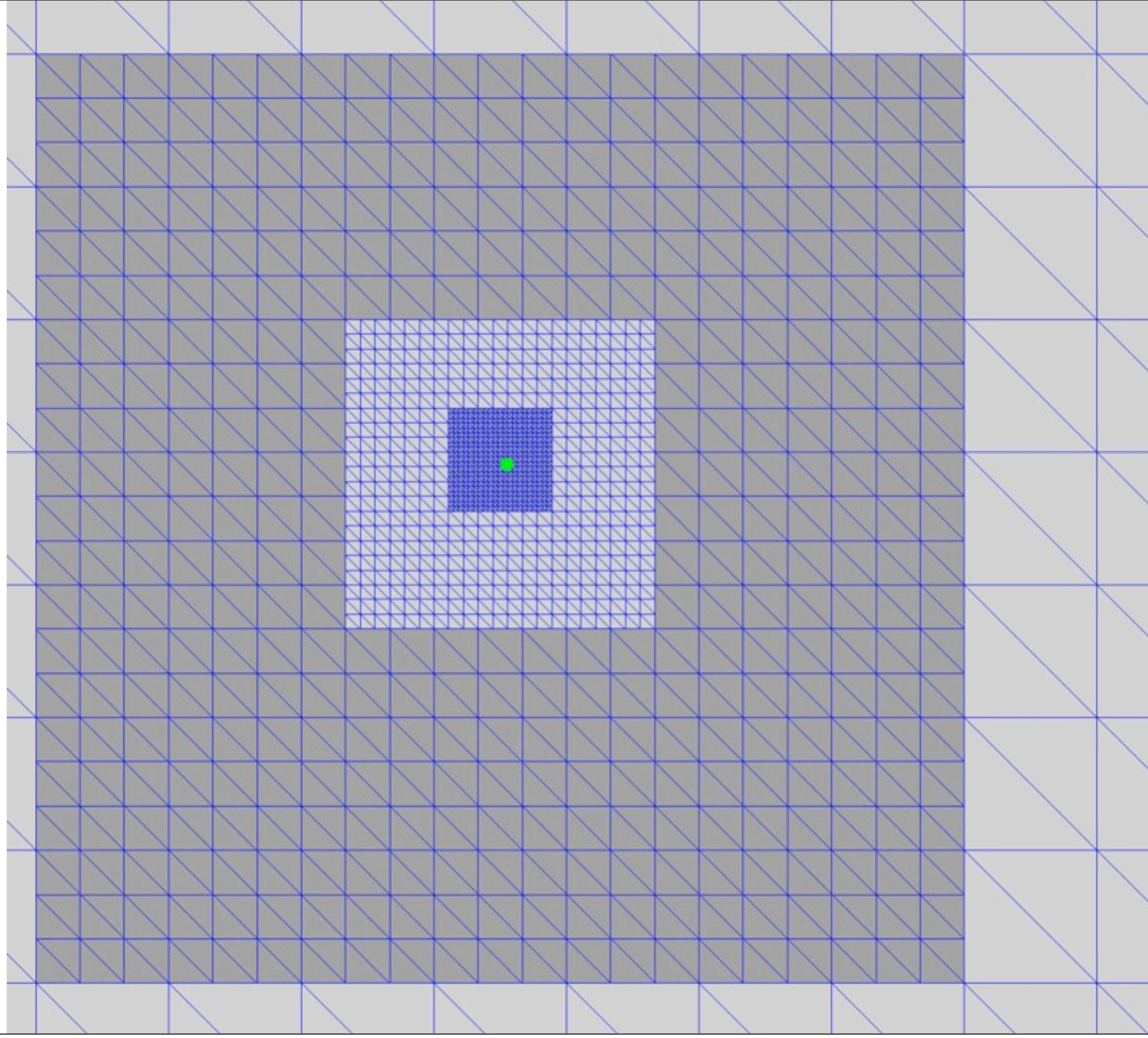
Any point in a ring will always be sampled from the same place. This way, we don’t have any jittering and aliasing problems

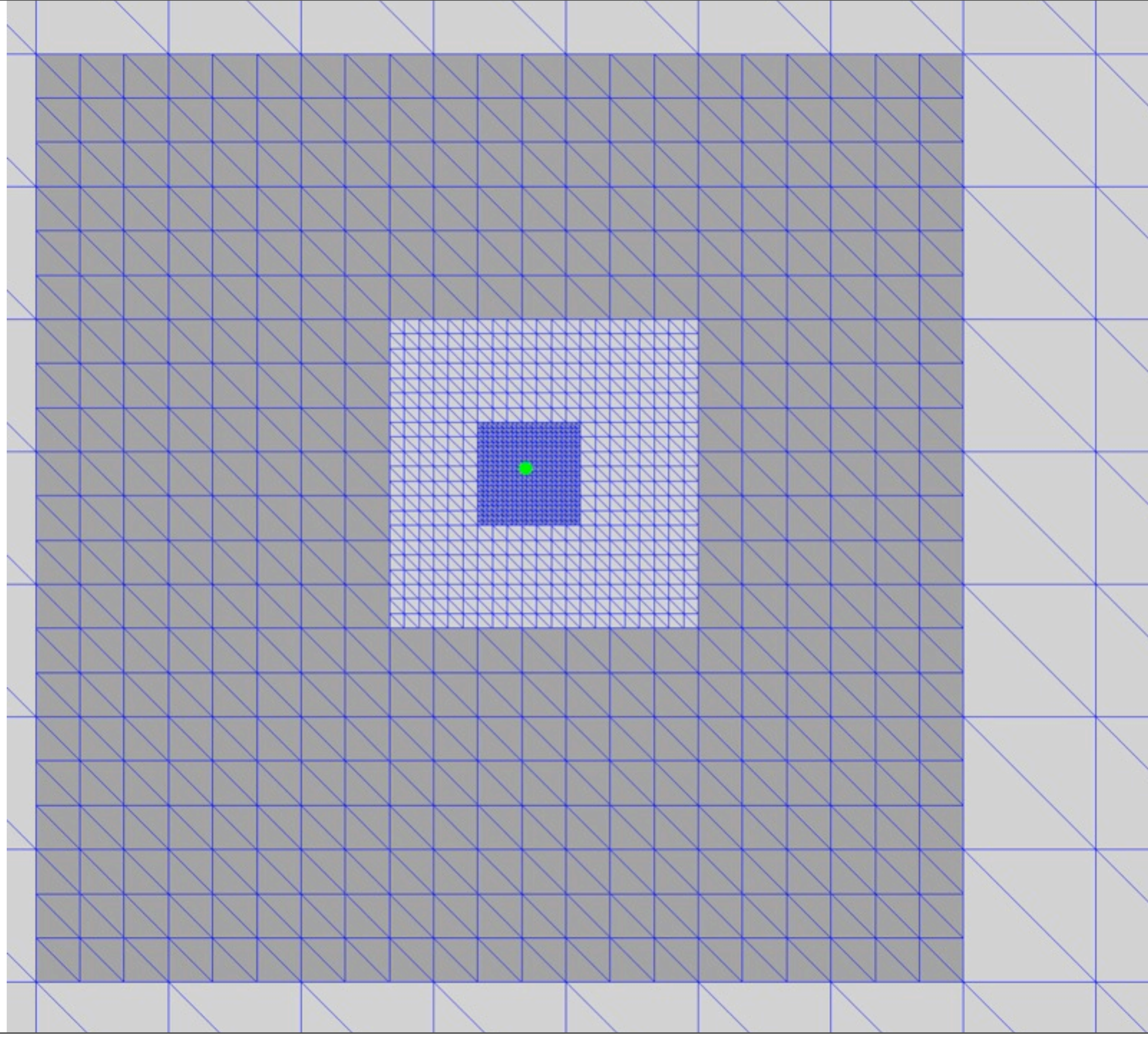

Ring 1
Ring 2
Ring 3
Ring 3
...

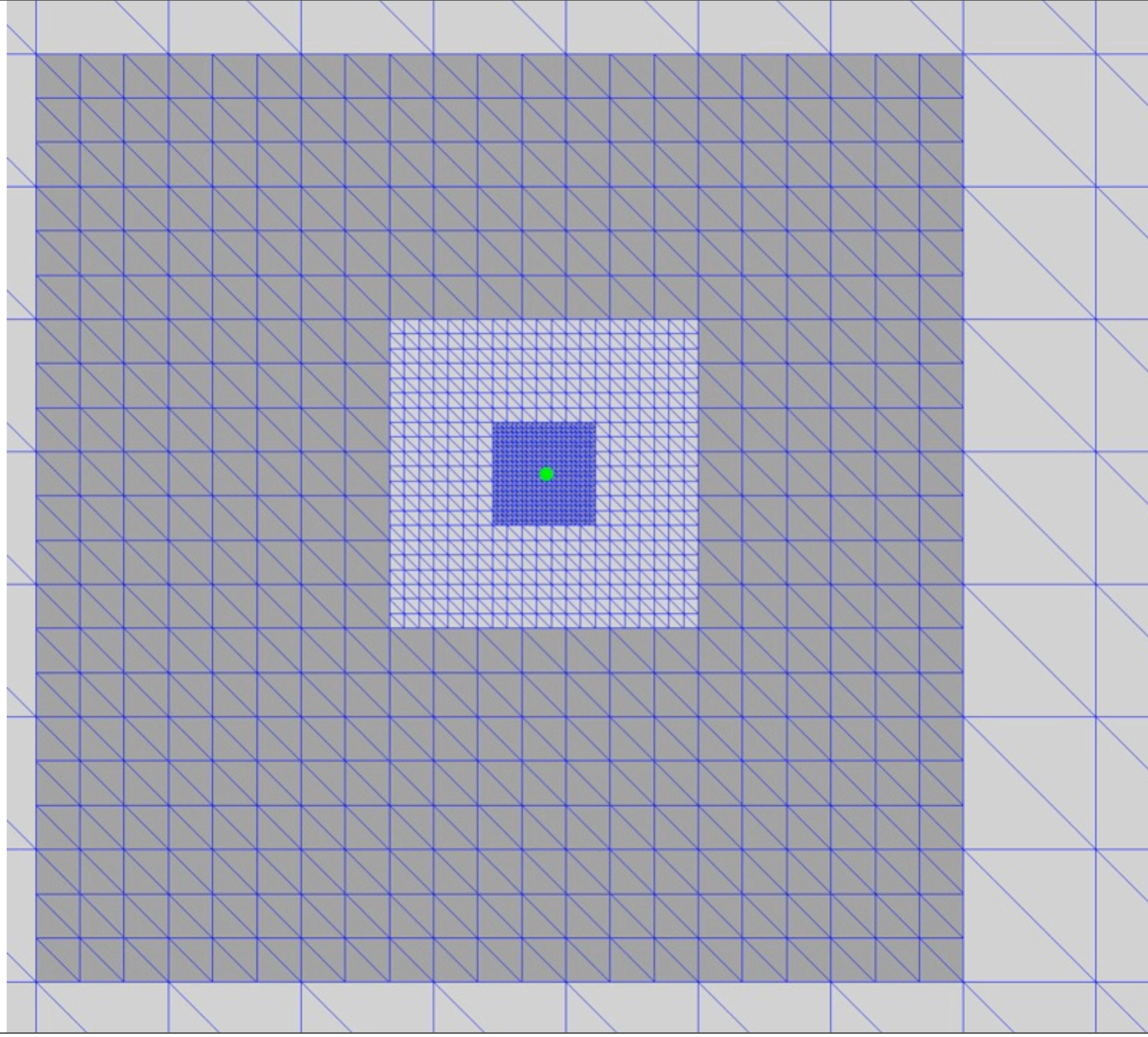


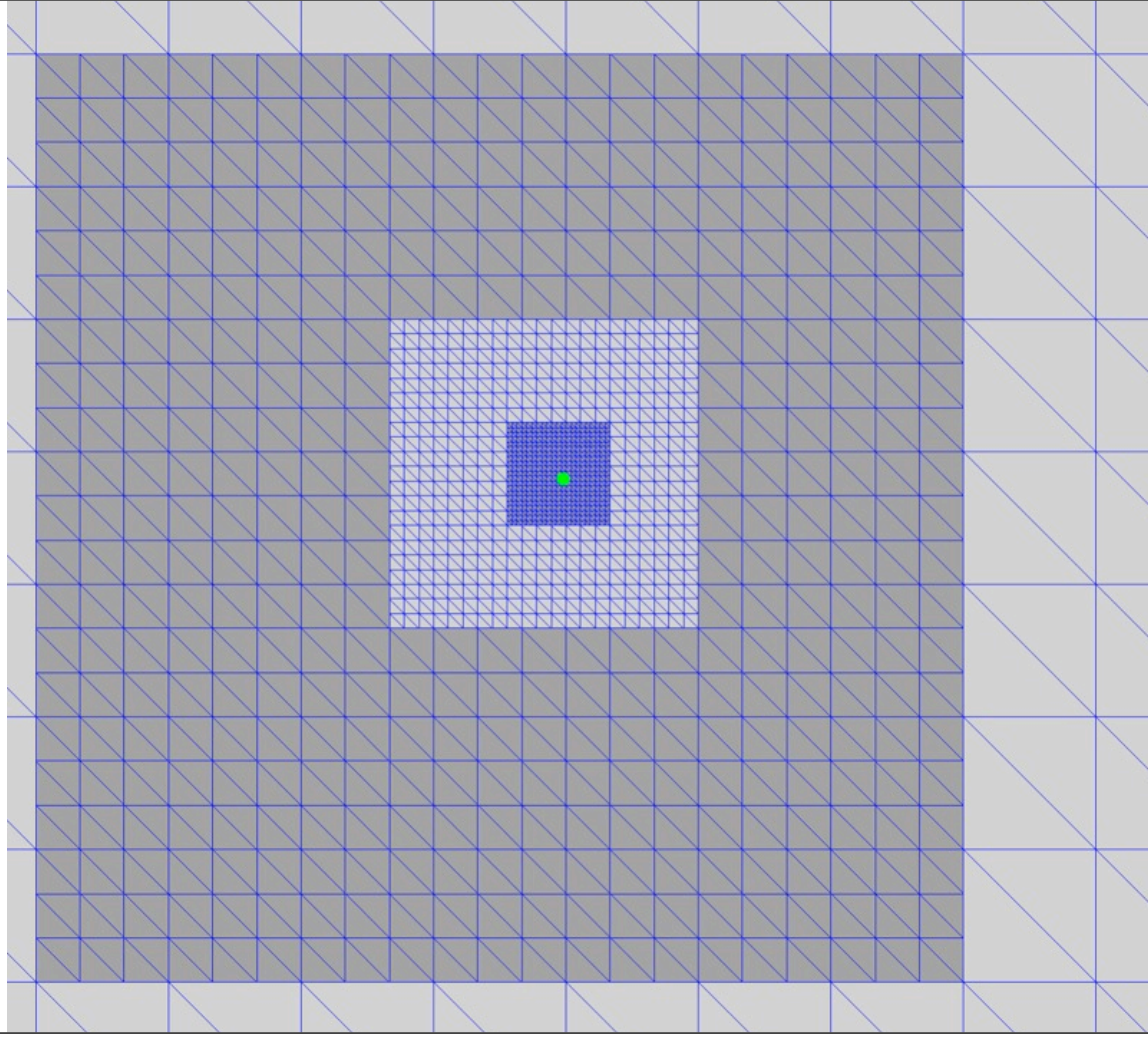


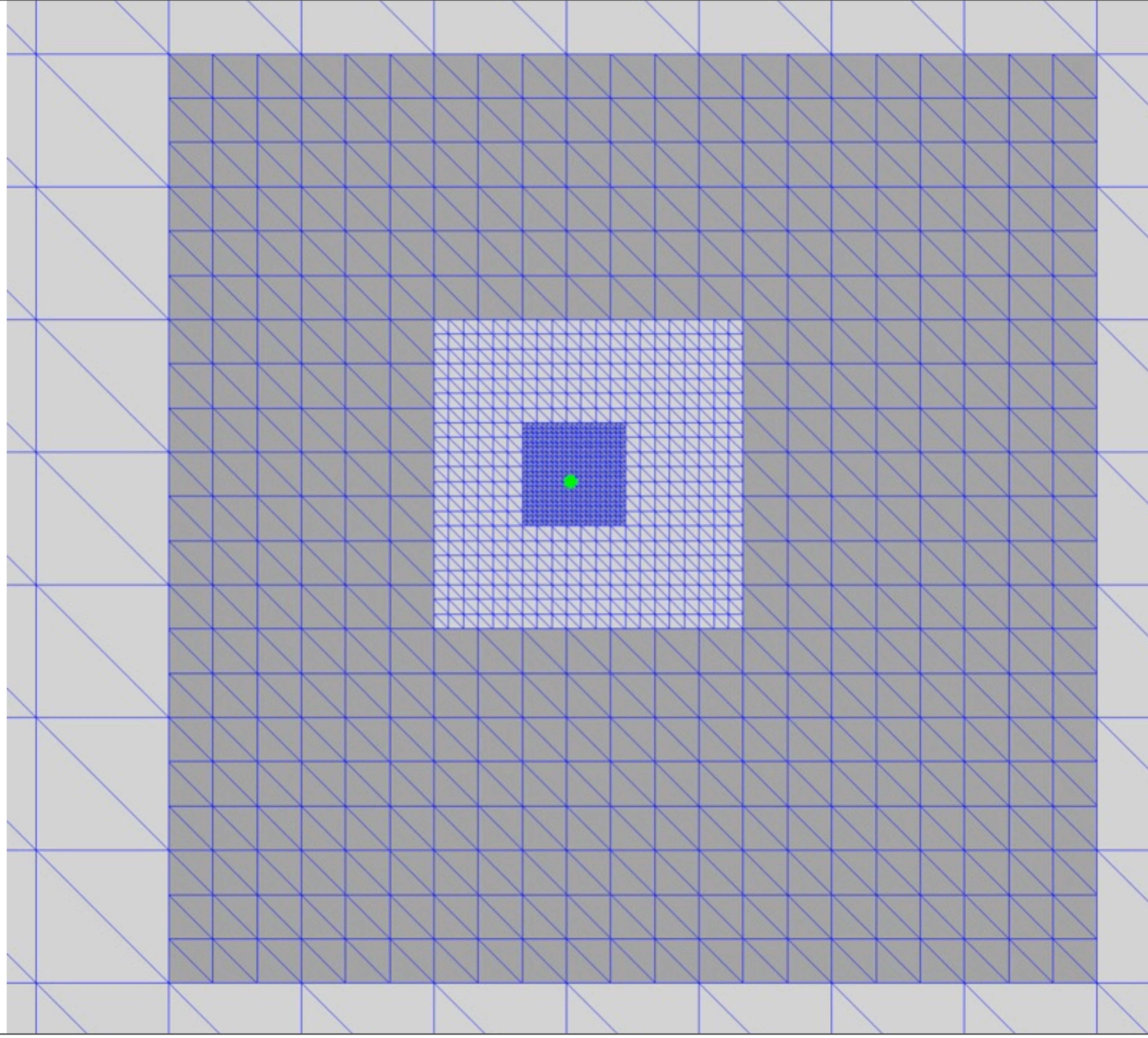




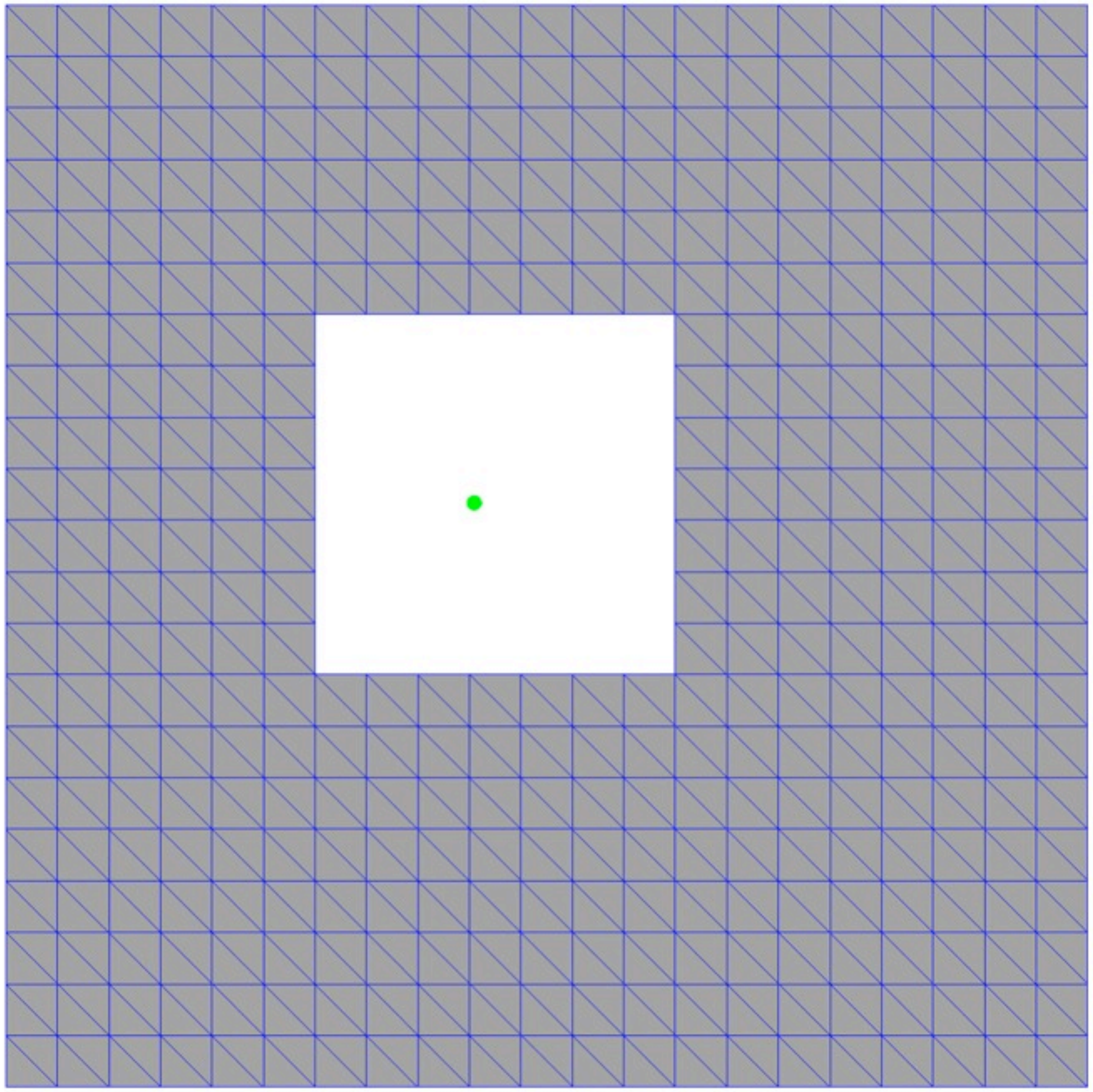






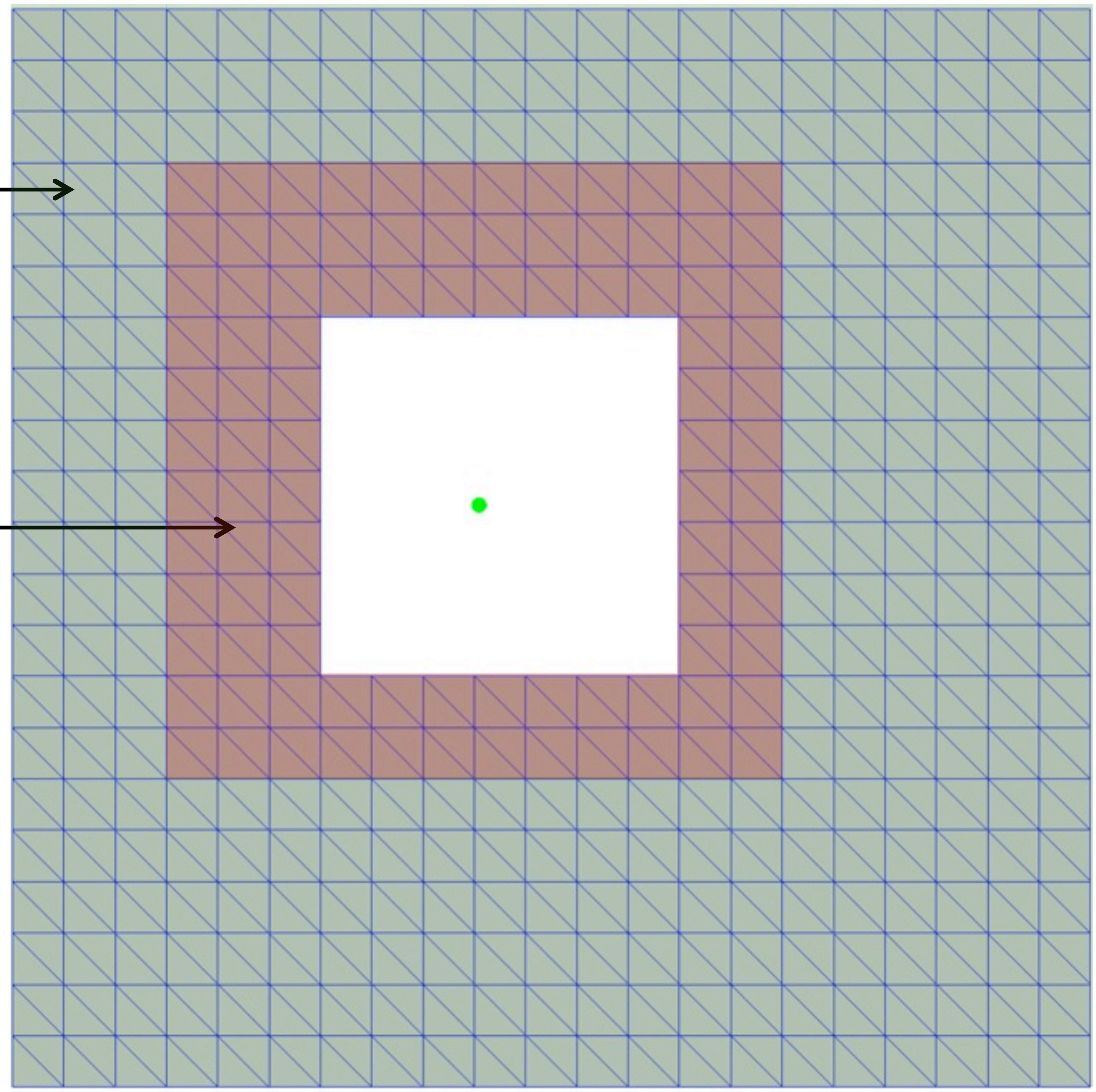


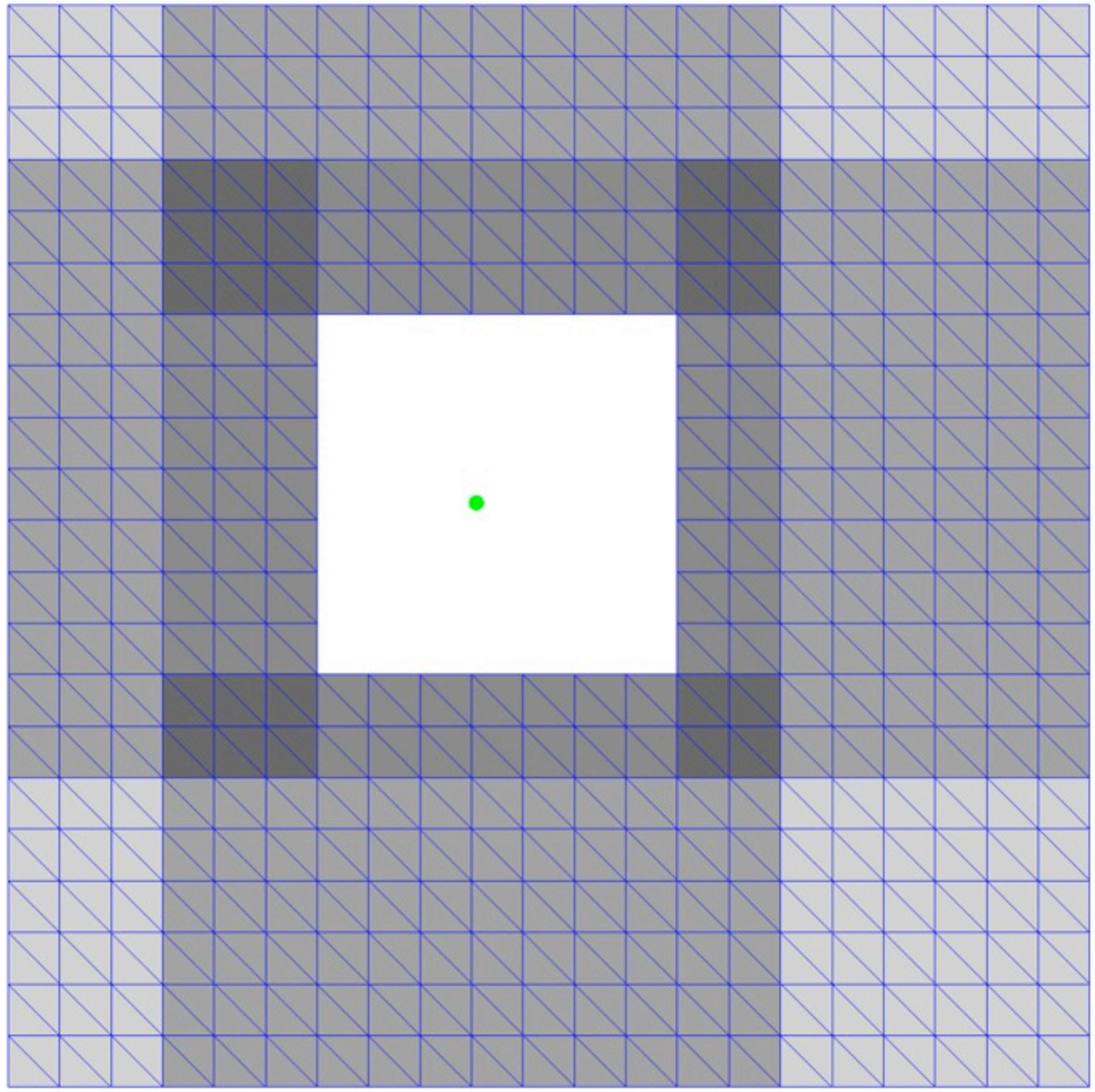
Single
Ring



Normal Patches

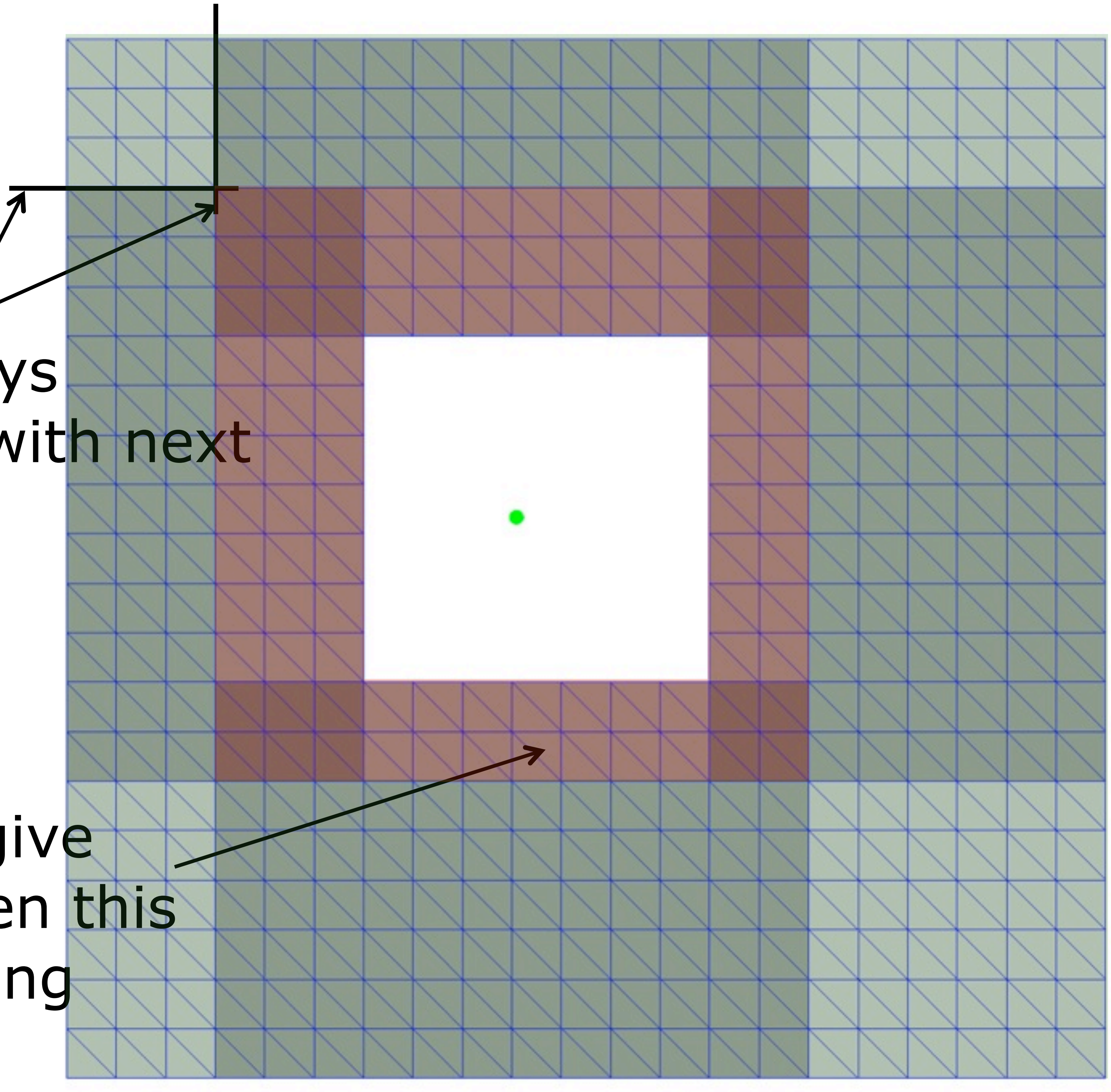
Fixer Patches





84

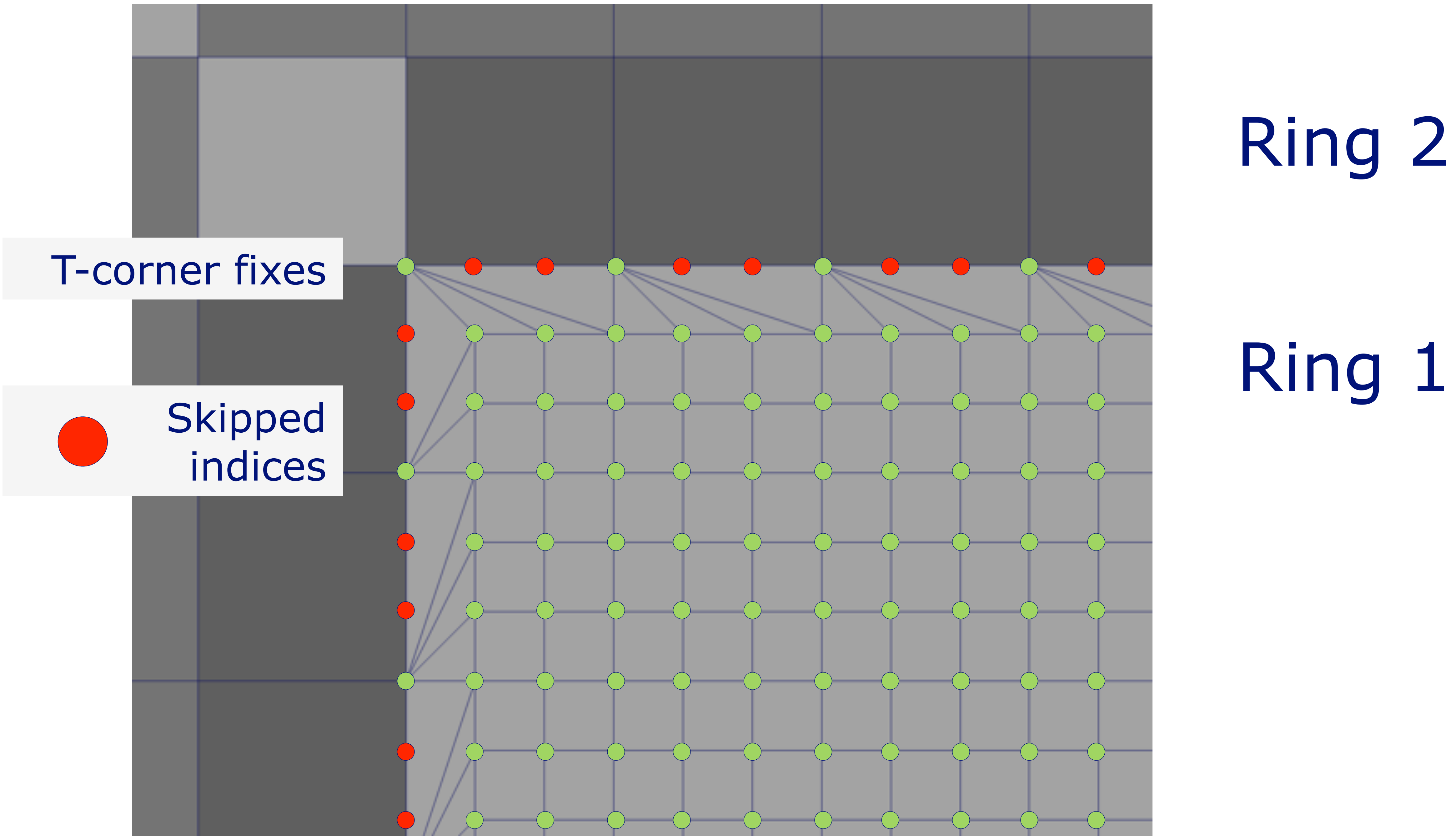
This is the general patch divisions. On a ring there are 16 patches.
The lowest level ring would have an additional patch in the center



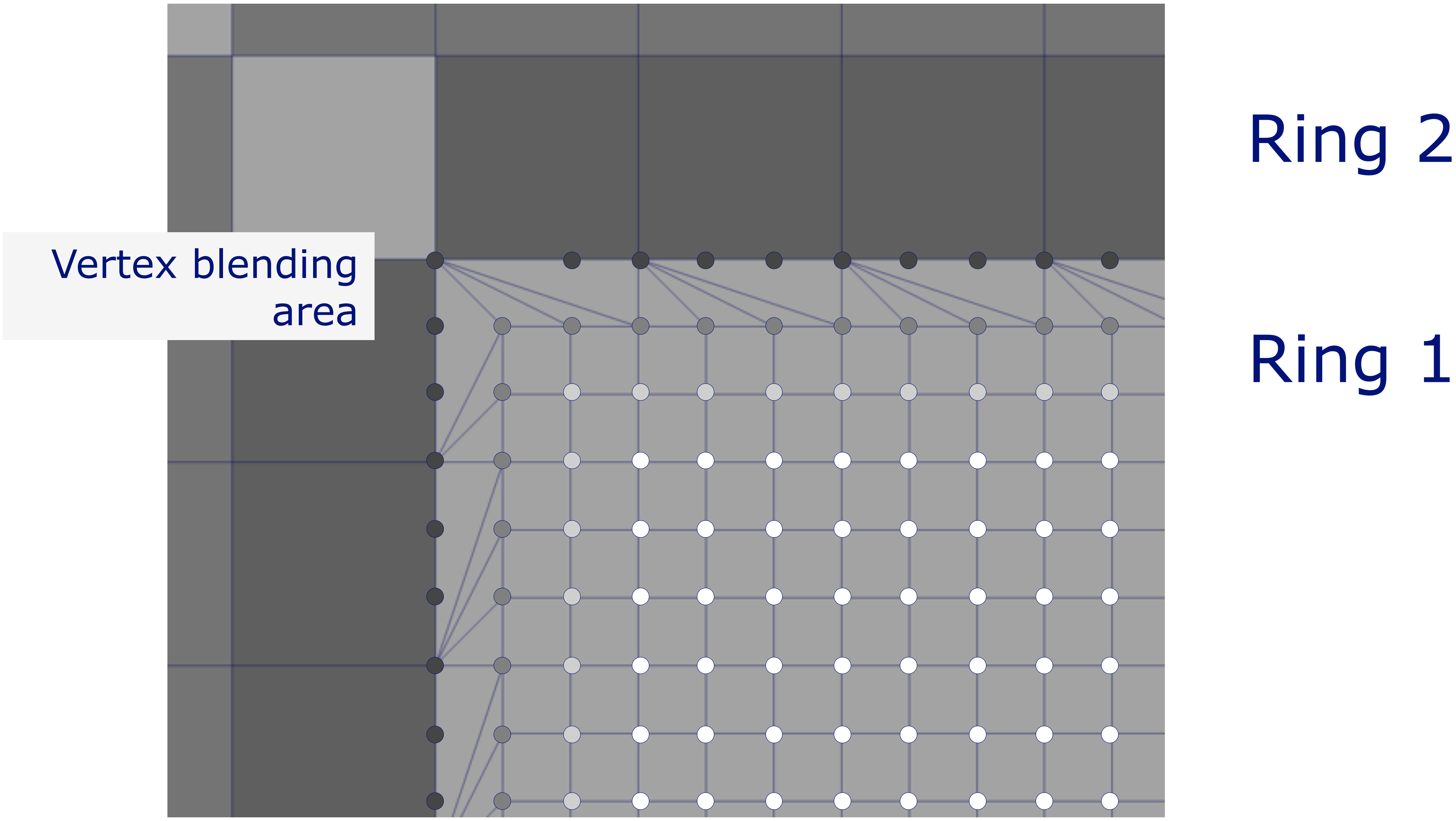
Splits will always avoid T-joints with next ring

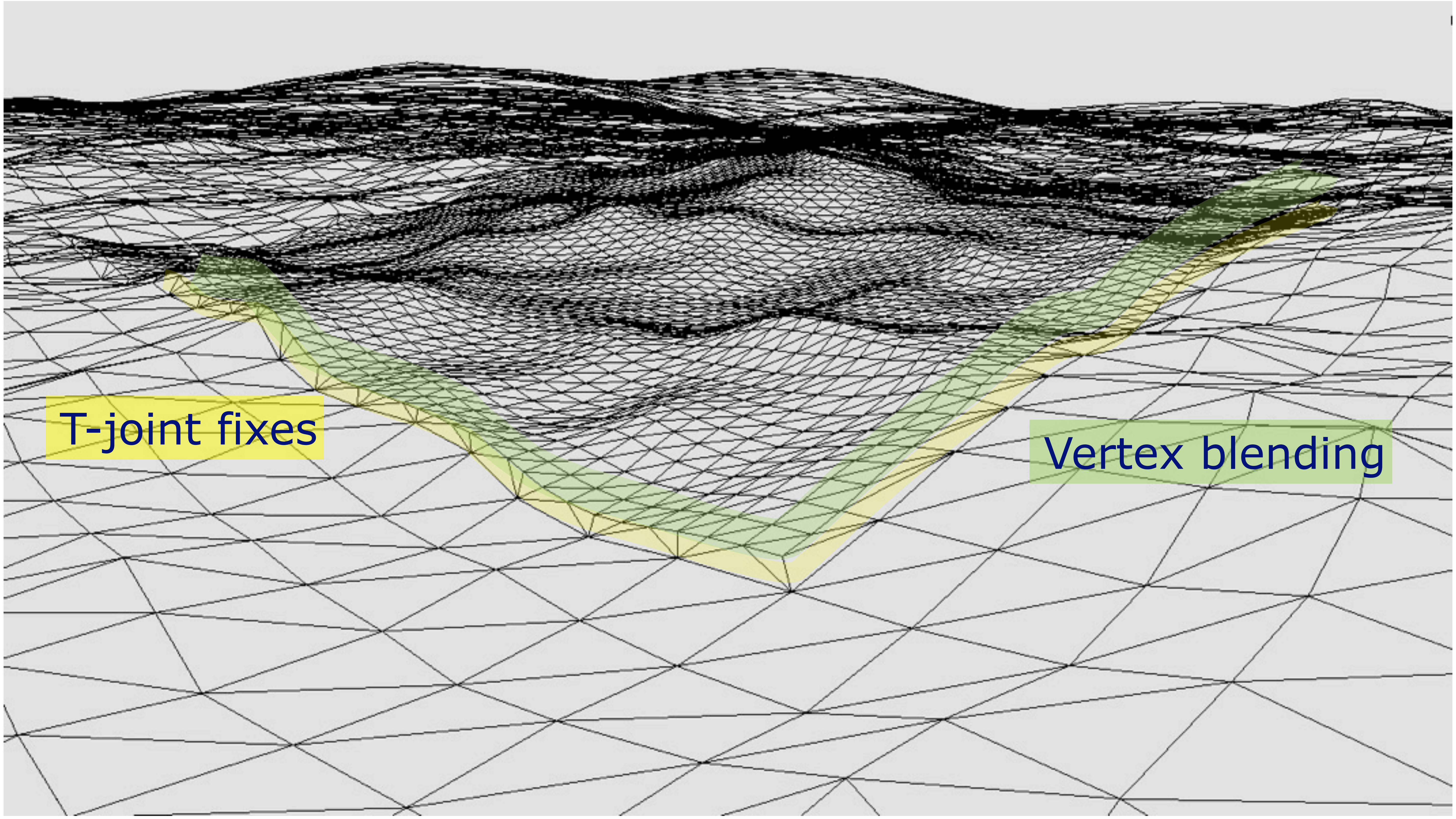
Fixer patches give "space" between this and previous ring

Border triangulation fixing



Border blending

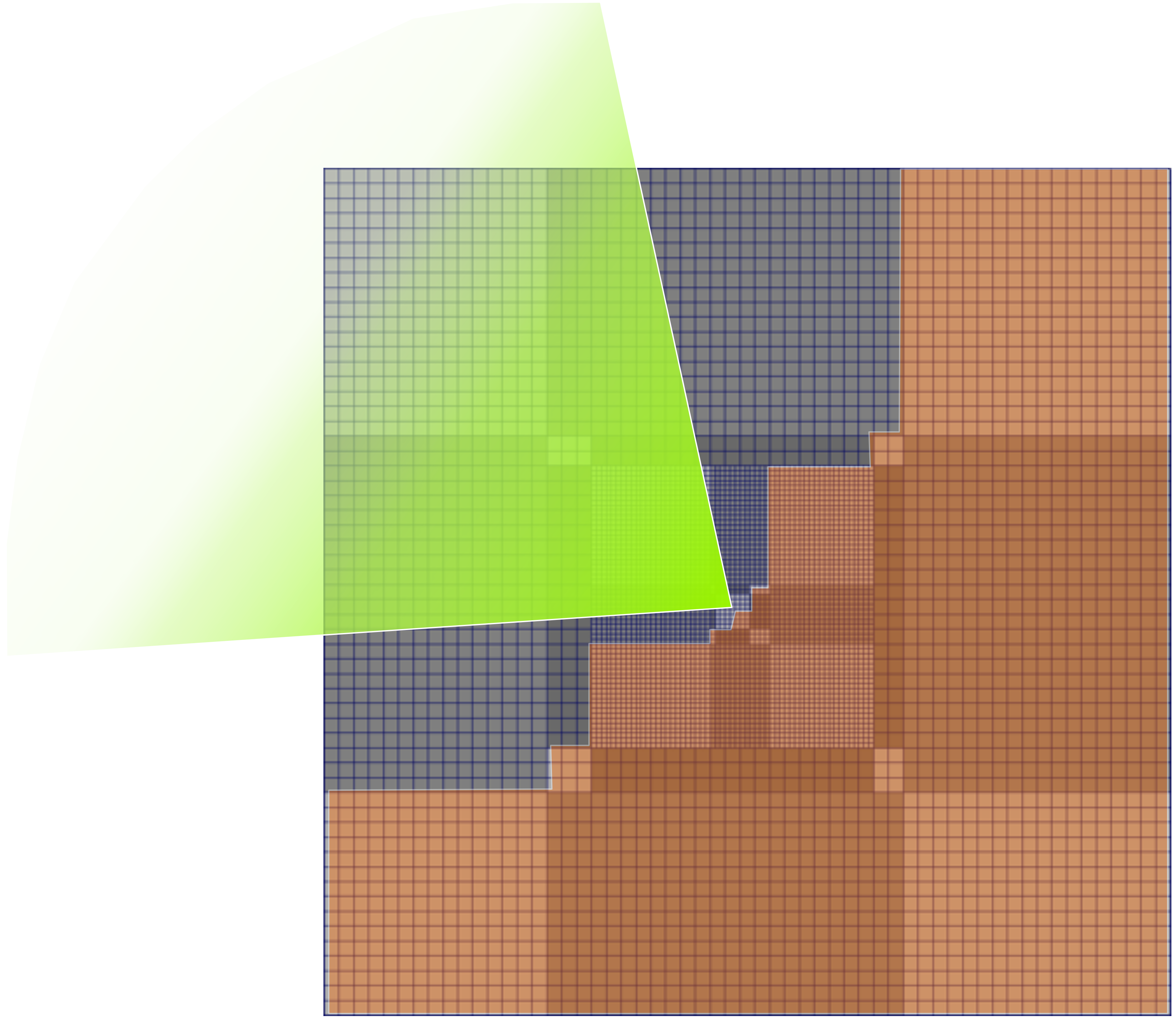



T-joint fixes
Vertex blending

Culling

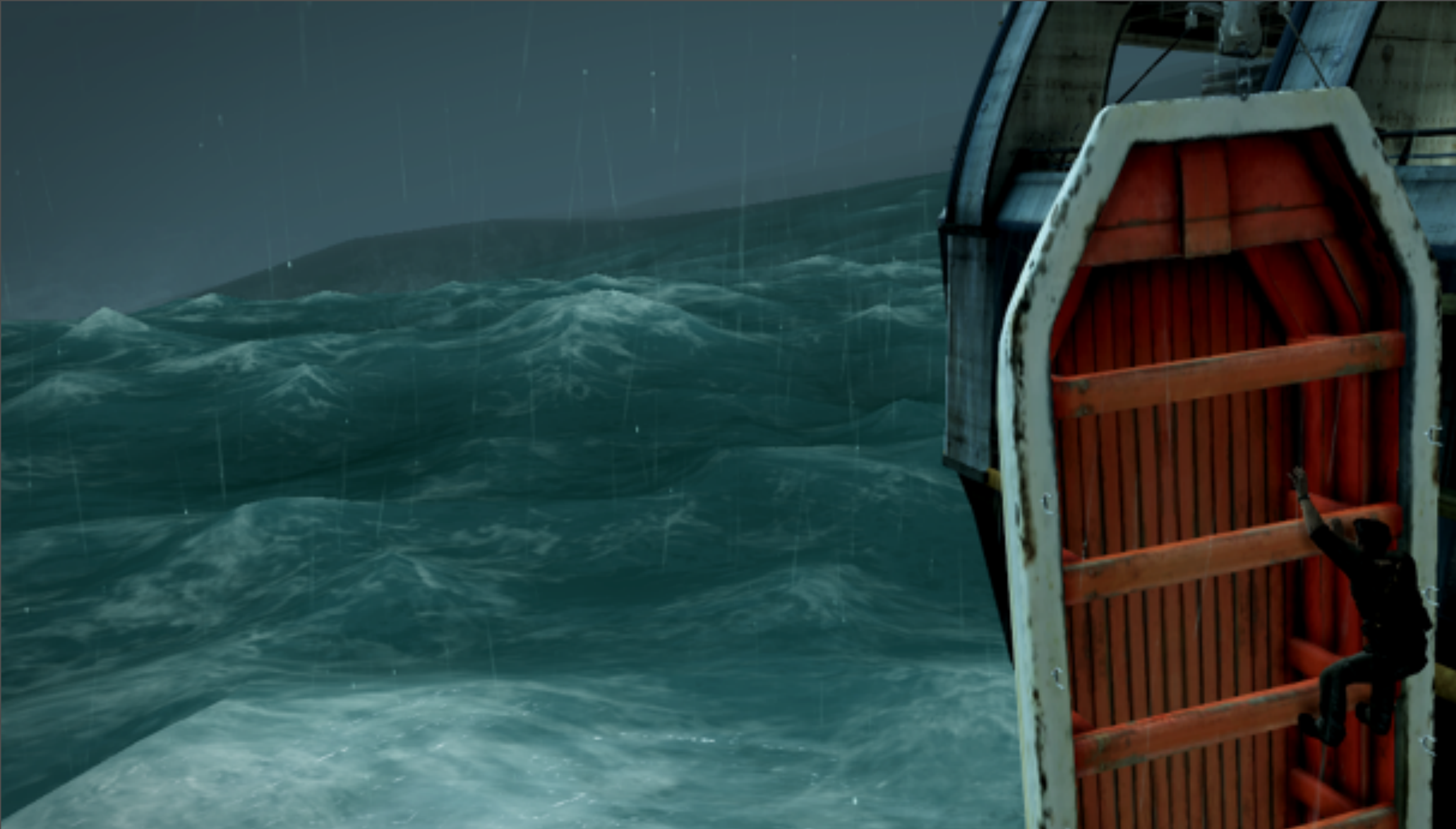
Cull out patches that are outside frustum

Frustum-bbox test



2 pass culling. First we generate bounding boxes with some extra slack from 4 corners and perform a quick intersection test.

After we generate the displaced surface, we have a tighter bbox, and can perform another intersection test.




Video of the ocean clipmap rendering. In wireframe is easier to see how the clipmap levels are rendered

Although we are inside the ballroom, we can see out the ocean. Furthermore the waves are moving the boat, so the chandeliers are being driven indirectly by the waves.

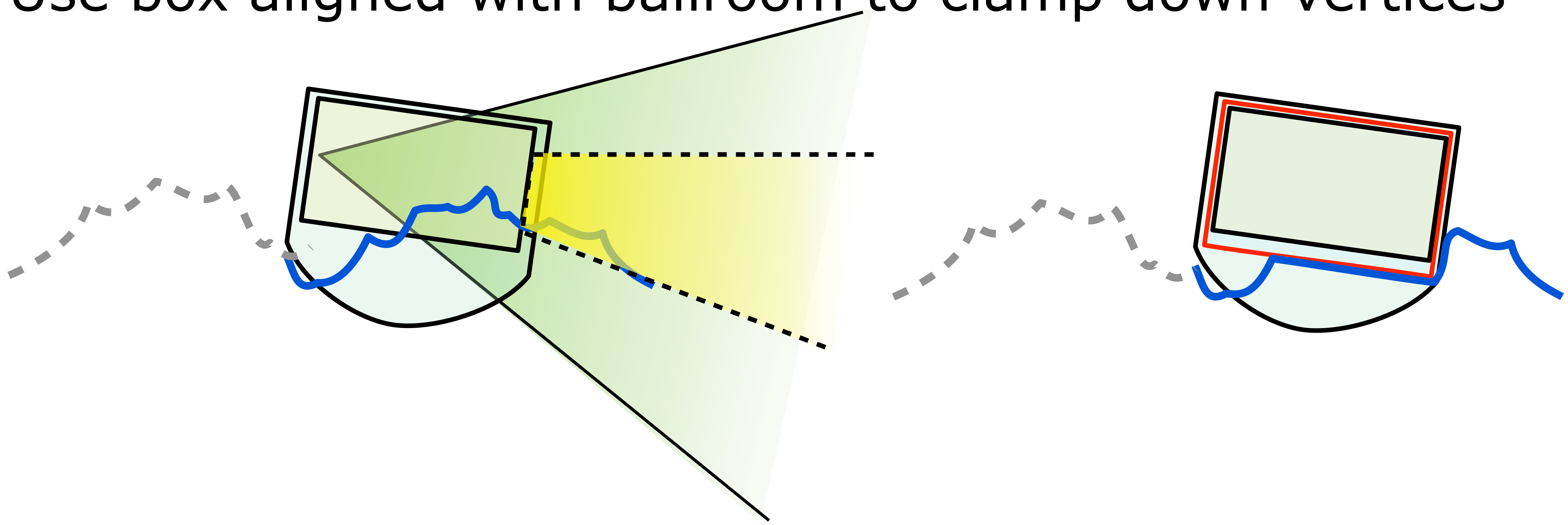
To heighten the drama we are closer to the water level

Culling

Use portals to show patches seen from windows

For waves intersecting ballroom:

- Use box aligned with ballroom to clamp down vertices

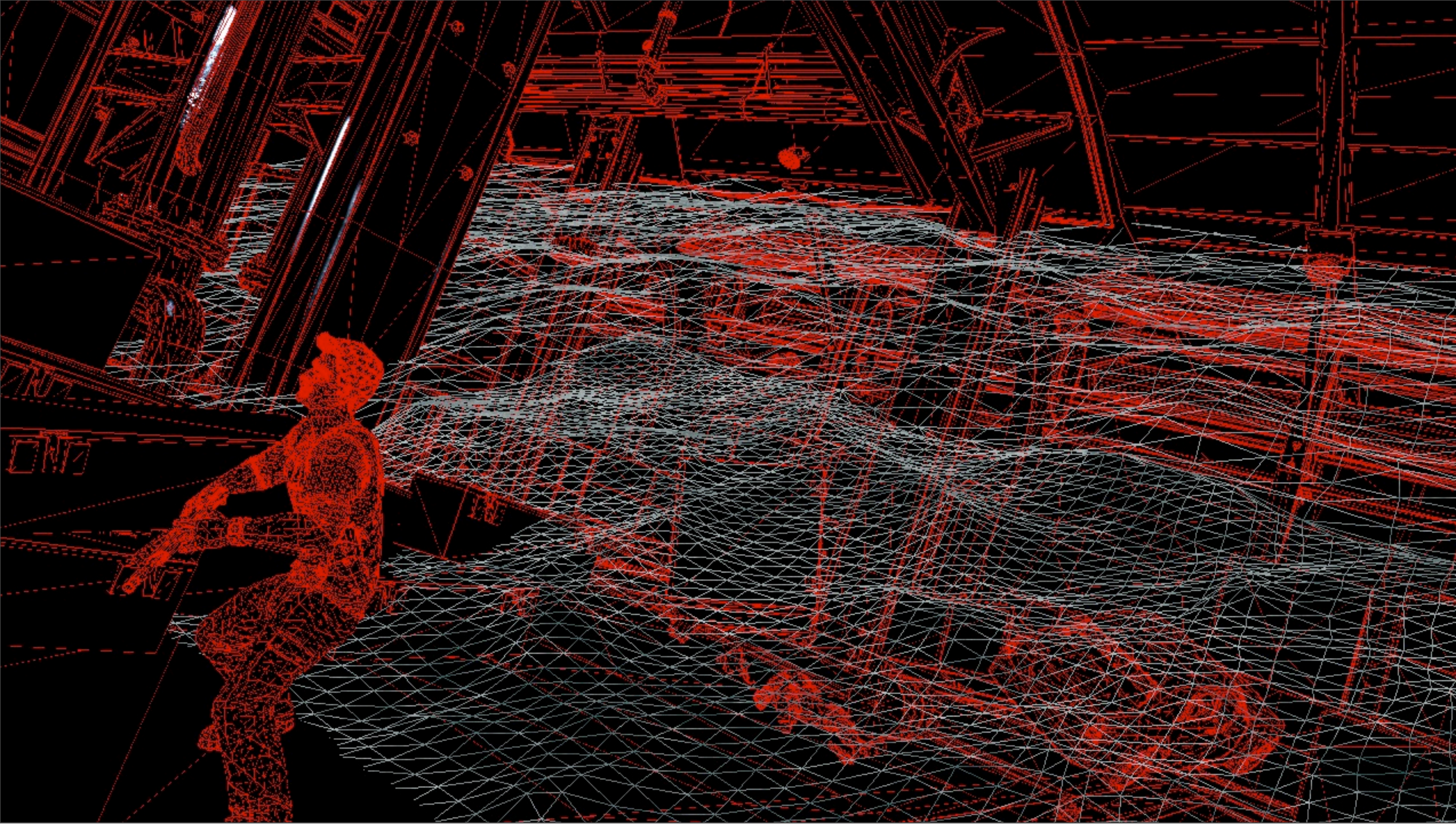


Another problem is that the ballroom is too low to the water line and we had clipping issues with big waves.
The solution is simple, test the points to a box, oriented to the ballroom, and push the vertices down

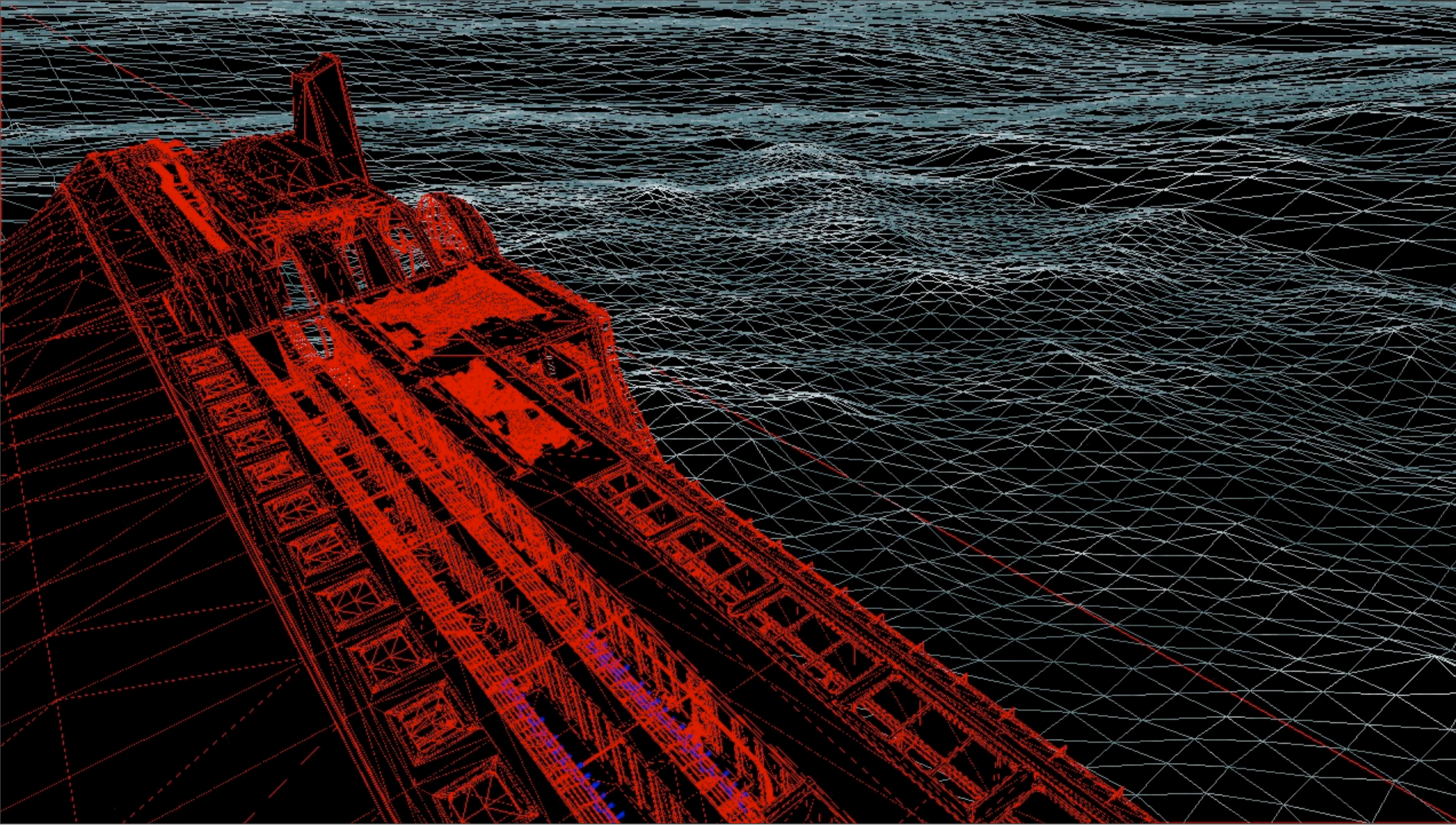
So something happens to the ship...

There is a sequence in U3, the Drakes goes from the top of the cruise-ship to the cargo hold.
At the cargo hold, Drake does what he does best: destroy everything

The ocean system is used in all the water in this sequence, we just change the shader and parameters.



The ocean system is used in all the water in this sequence, we just change the shader and parameters.



and we come back to the ballroom


A.KIM '10

We could only render one ocean at a time, so we constantly had to change between all the different water elements in the cruiseship.

We would switch parameters and shader of the water. At some points between 2 frames.

The skylight section (the ballroom tilted 90 degrees) is technically the most difficult section of the whole game.
We needed to render water outside, and could be seen from above and below. Also there would be a flooding stage.

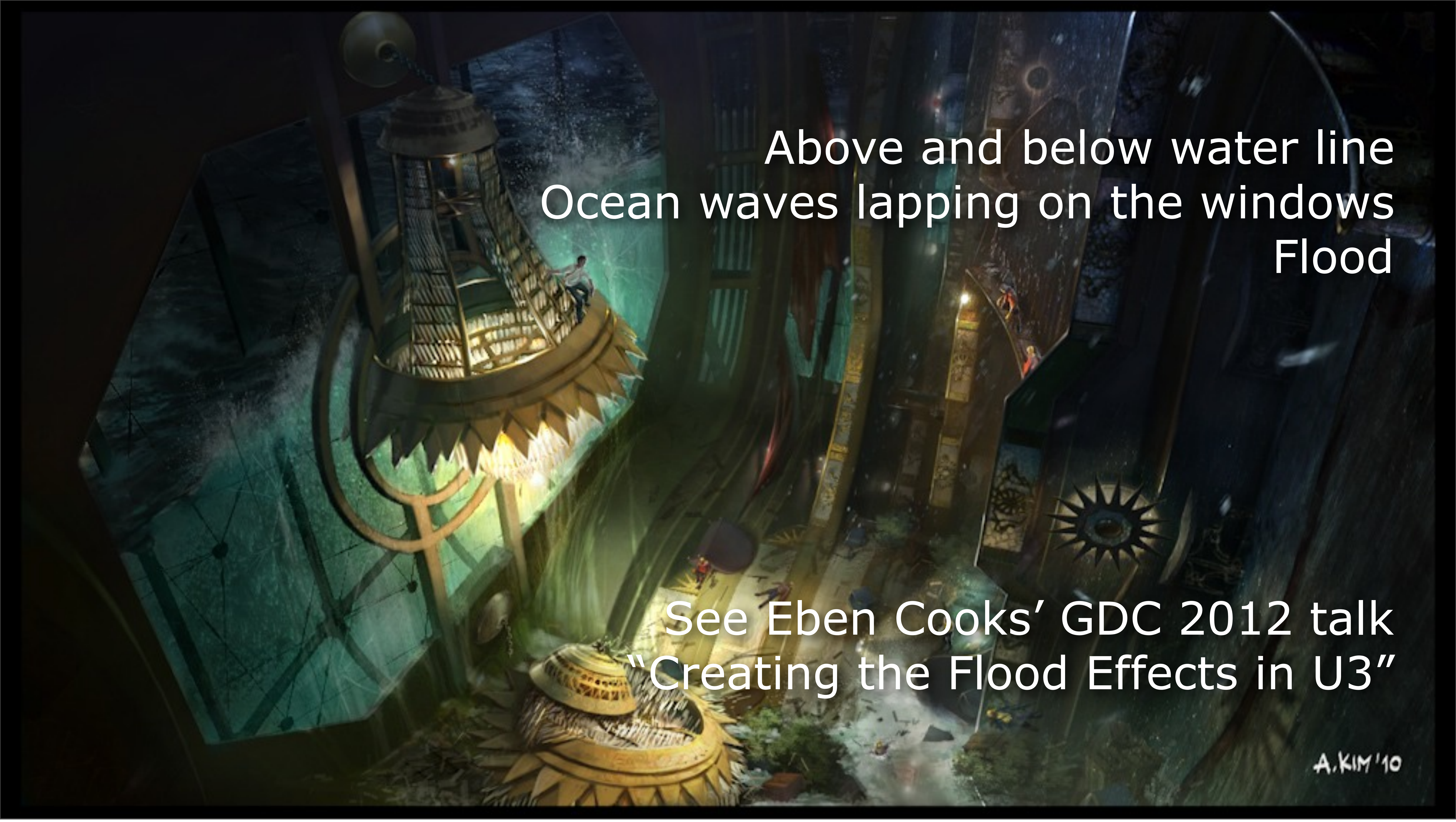
We had to limit the movement of the boat to one plane, so we could clip the water using a simple plane.
At some point we thought of using a curved glass, but we ran out of time.

We could only render one ocean at a time, so we constantly had to change between all the different water elements in the cruiseship.

We would switch parameters and shader of the water. At some points between 2 frames.

The skylight section (the ballroom tilted 90 degrees) is technically the most difficult section of the whole game.
We needed to render water outside, and could be seen from above and below. Also there would be a flooding stage.

We had to limit the movement of the boat to one plane, so we could clip the water using a simple plane.
At some point we thought of using a curved glass, but we ran out of time.



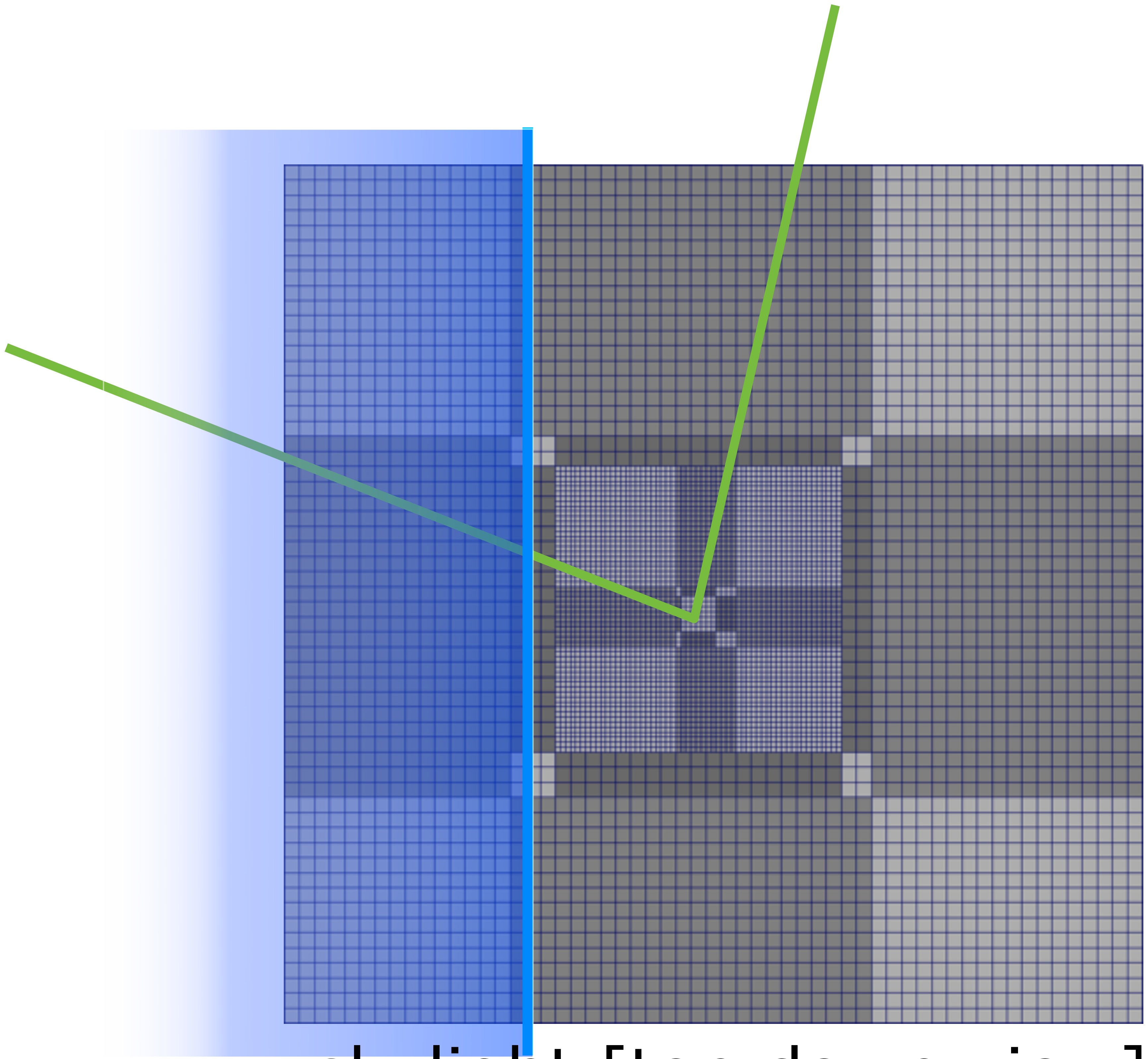
We could only render one ocean at a time, so we constantly had to change between all the different water elements in the cruiseship.

We would switch parameters and shader of the water. At some points between 2 frames.

The skylight section (the ballroom tilted 90 degrees) is technically the most difficult section of the whole game.
We needed to render water outside, and could be seen from above and below. Also there would be a flooding stage.

We had to limit the movement of the boat to one plane, so we could clip the water using a simple plane.
At some point we thought of using a curved glass, but we ran out of time.

Skylight scene

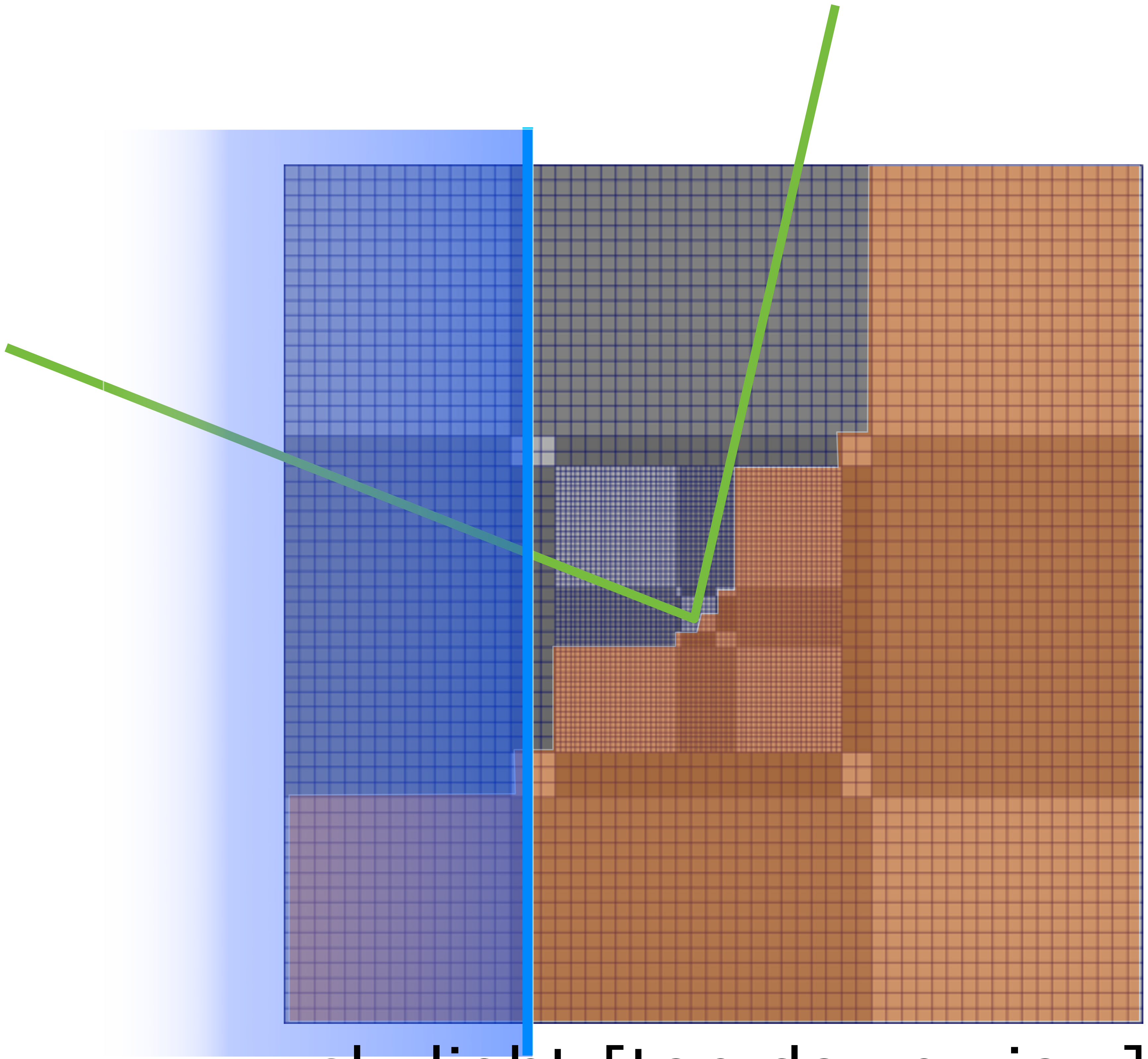


skylight [top down view]

Skylight scene

- Cull out patches that are outside frustum

Frustum-bbox test



skylight [top down view]

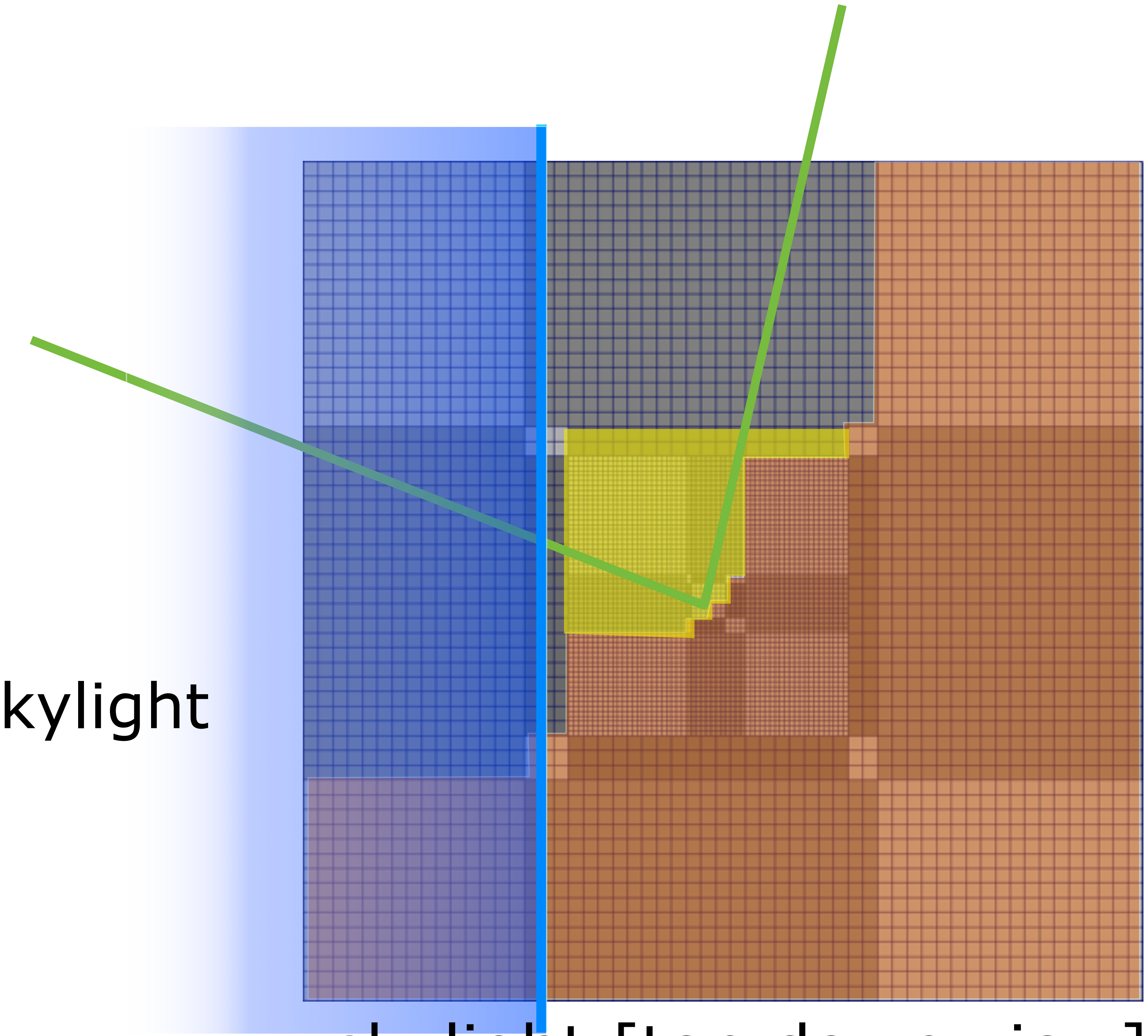
Skylight scene

- Cull out patches that are outside frustum

  Frustum-bbox test

- Cull out patches outside skylight

  Plane-bbox test



skylight [top down view]

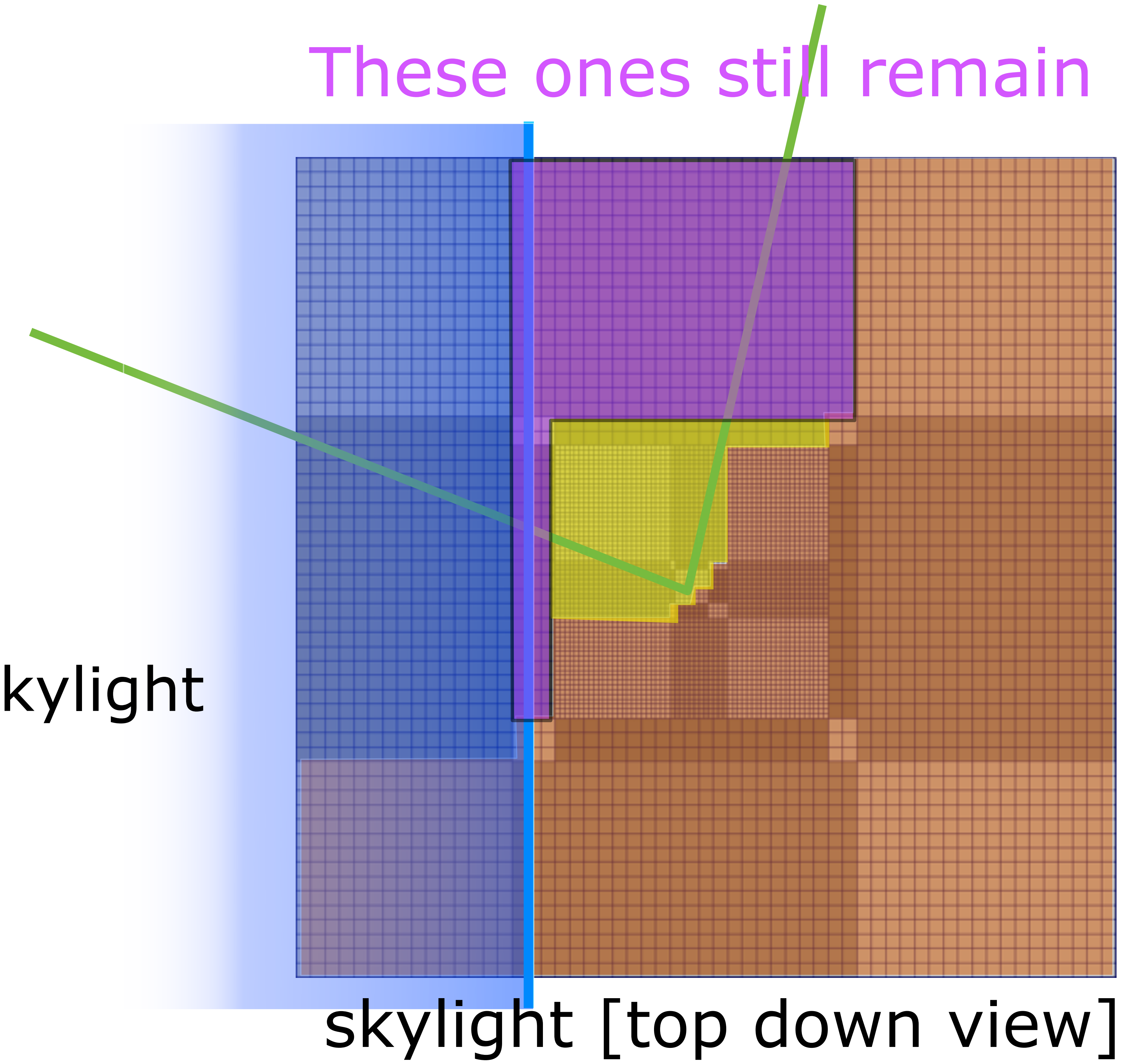
Skylight scene

- Cull out patches that are outside frustum

  Frustum-bbox test

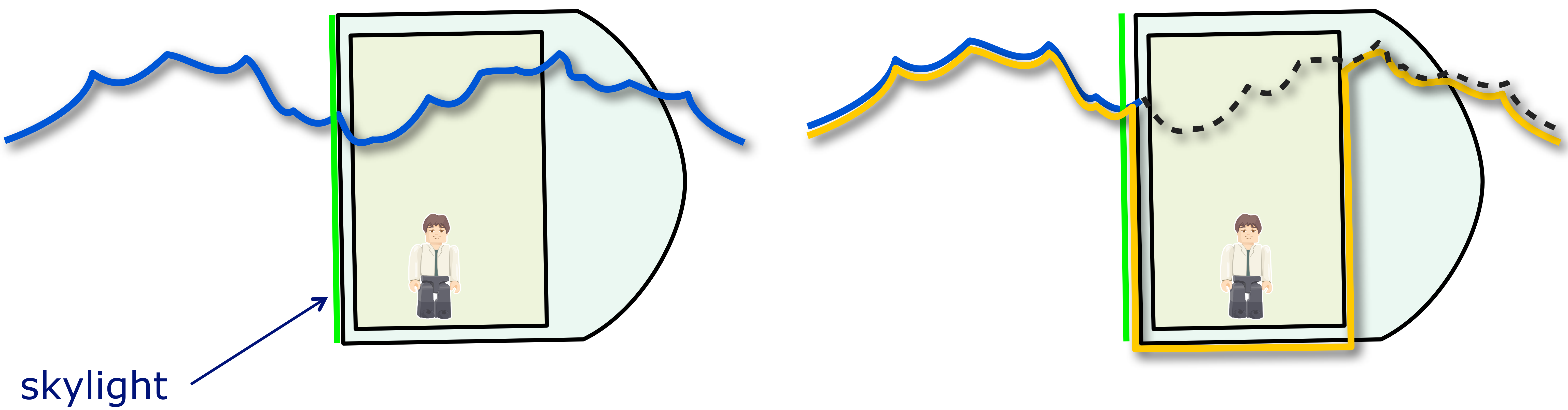
- Cull out patches outside skylight

  Plane-bbox test



Skylight scene

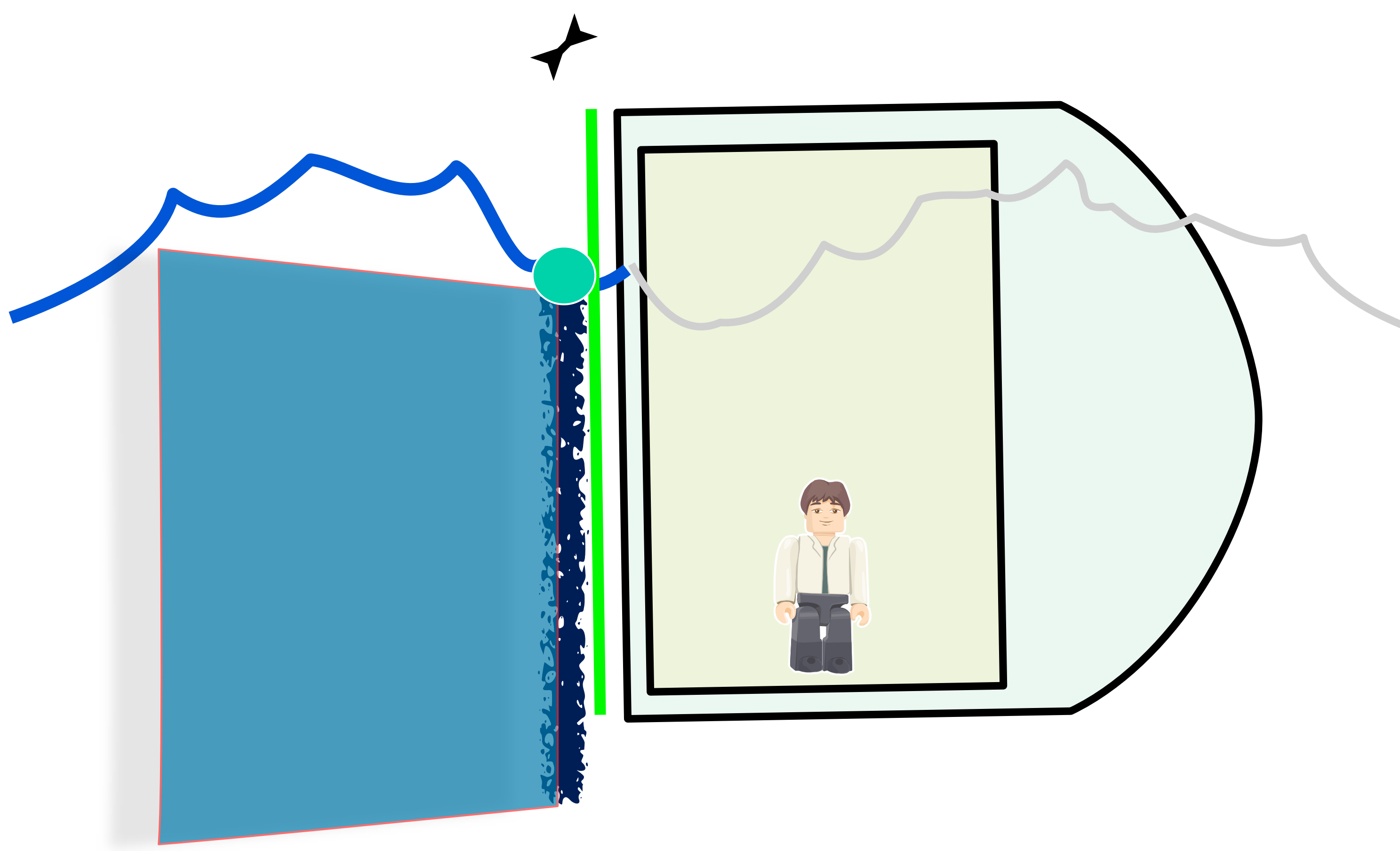
- For rendering use shader discard operation to do plane clipping
- For evaluation clamp down points inside ballroom bbox



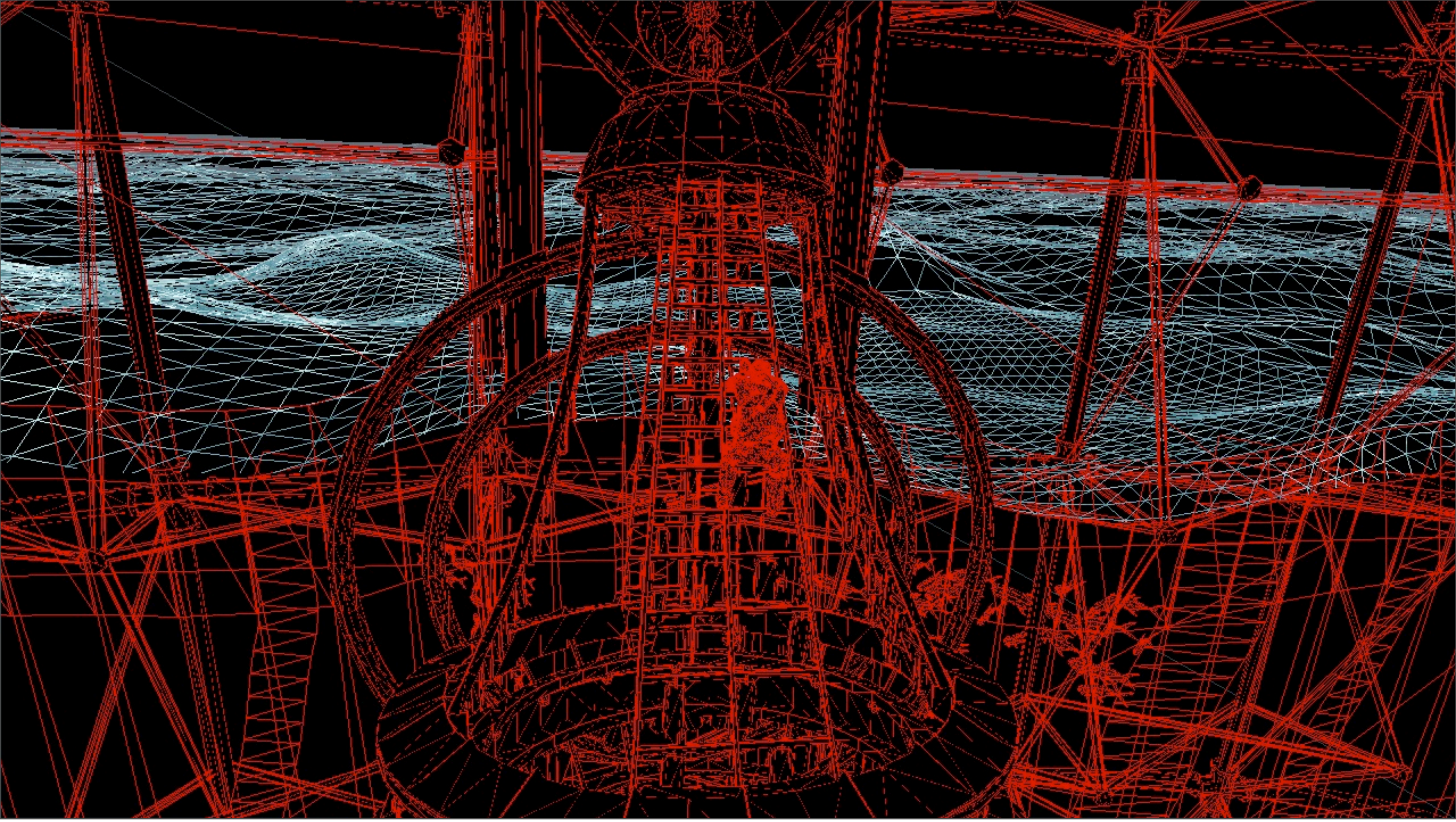
The culling using the shader was simple a plane culling

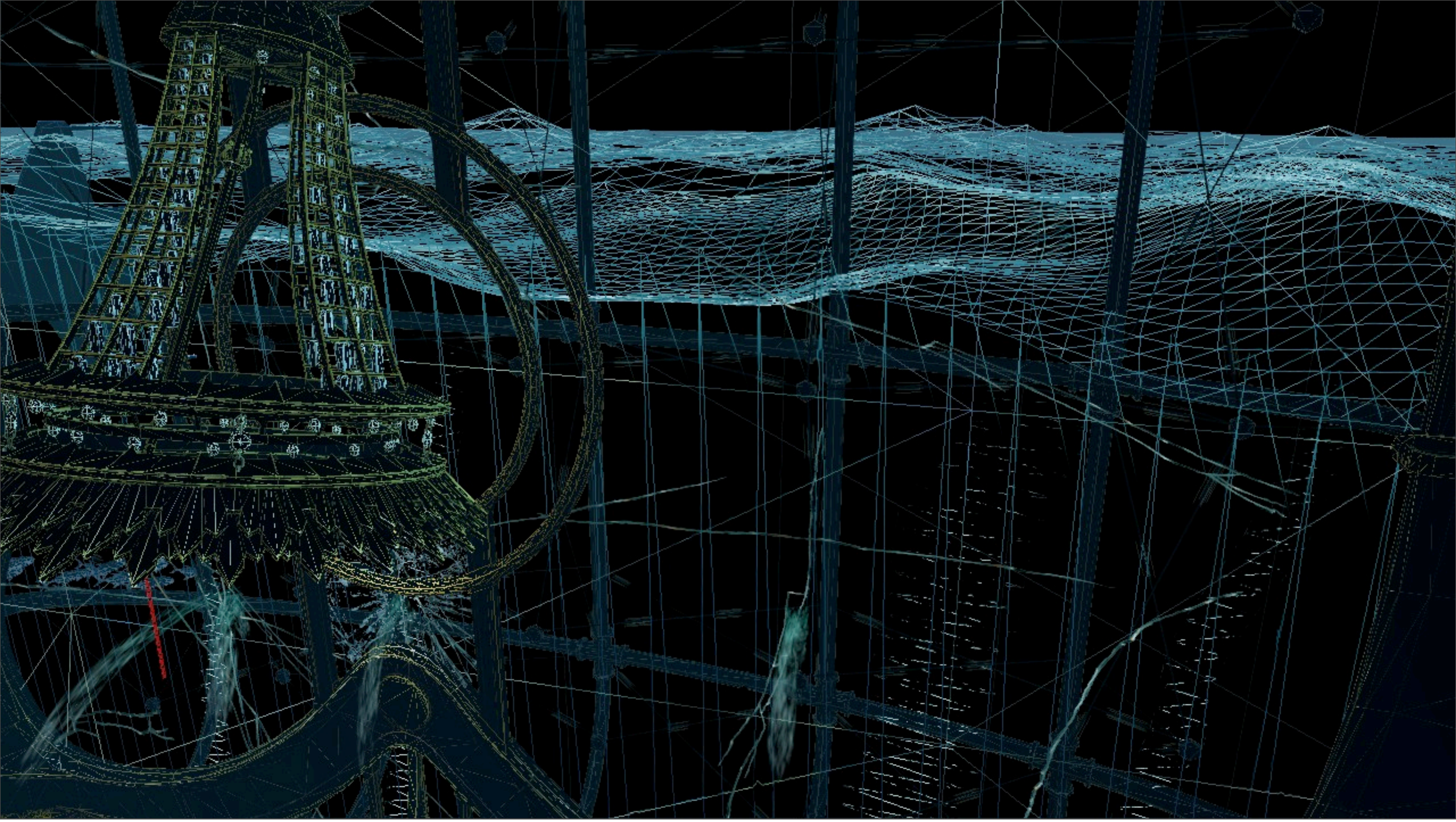
Skylight scene (90 degree cruise-ship)

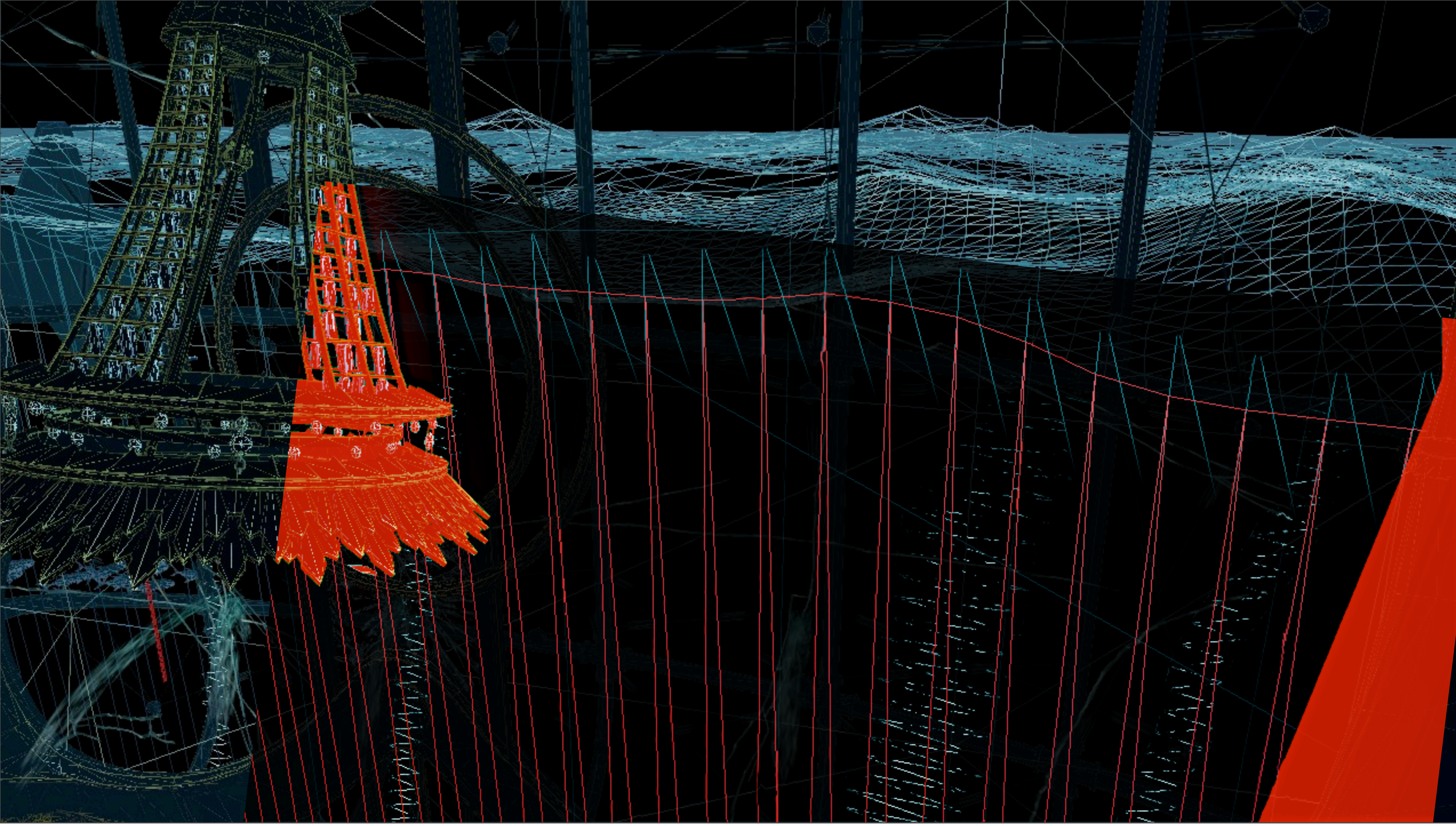
- Fake underwater fog with a polygon “curtain” driven by the water movement (Eben’s Cook hack)



Faking the underwater fog was more complicated.
Eben Cook did an awesome job figuring out a way to realize the effect without multipasses.
He used a polygon skirt that is driven by the waves and it covers the back of the glass. The skirt shader simulates the fog effect of water.



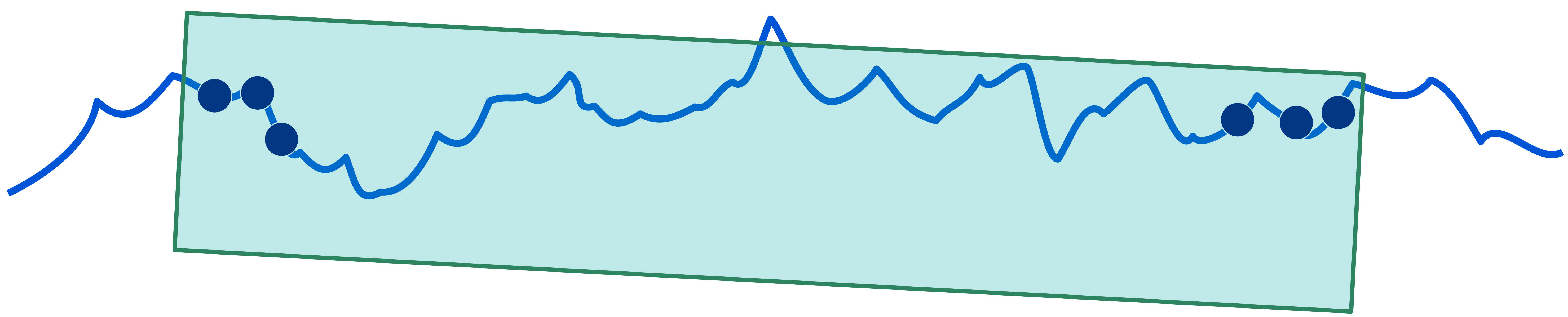




Floating objects

Sample points and best fit a plane for orientation

On intersection areas, multiply down amplitude

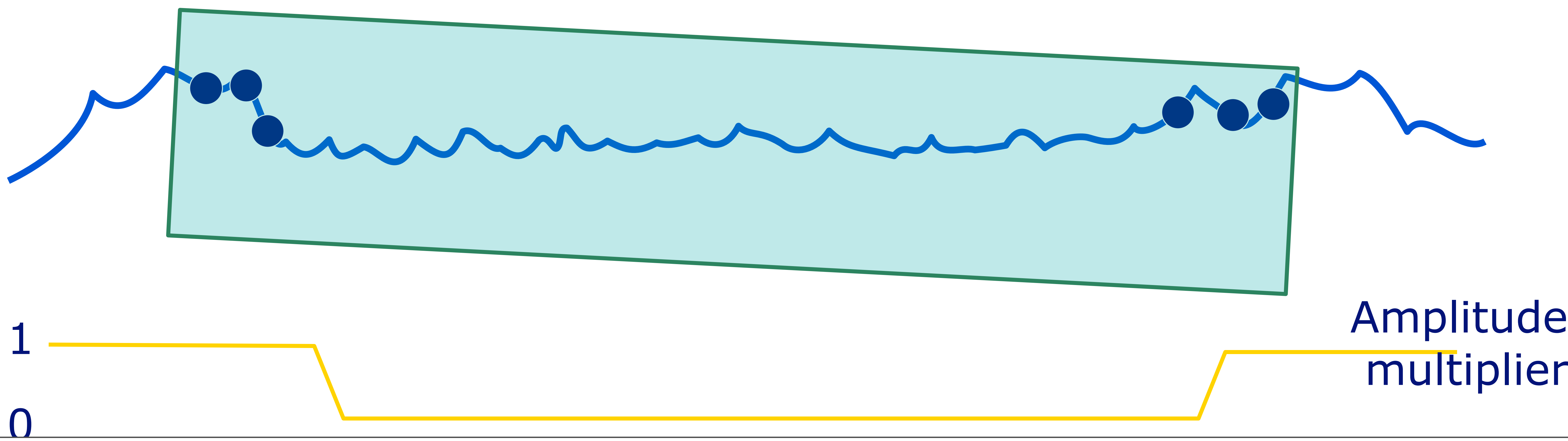


Depending on object, we can sample some points and best fit a plane to orient the object

Attaching objects

Sample points and best fit a plane for orientation

On intersection areas, multiply down amplitude

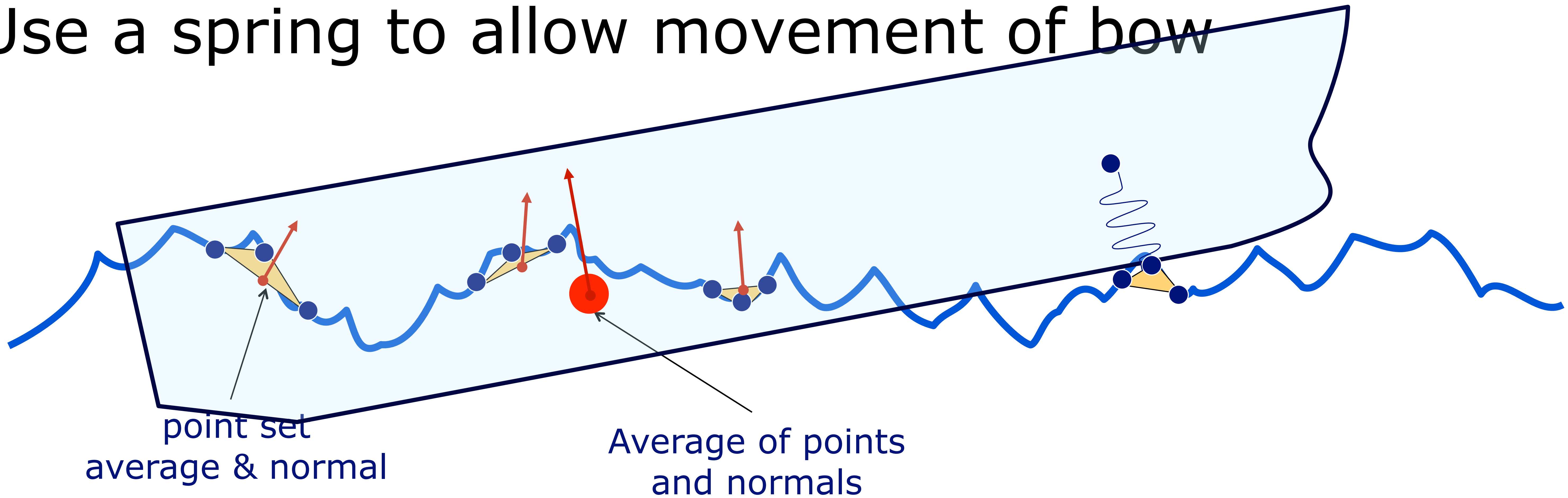


Cruise ship

Sample points along length of the boat

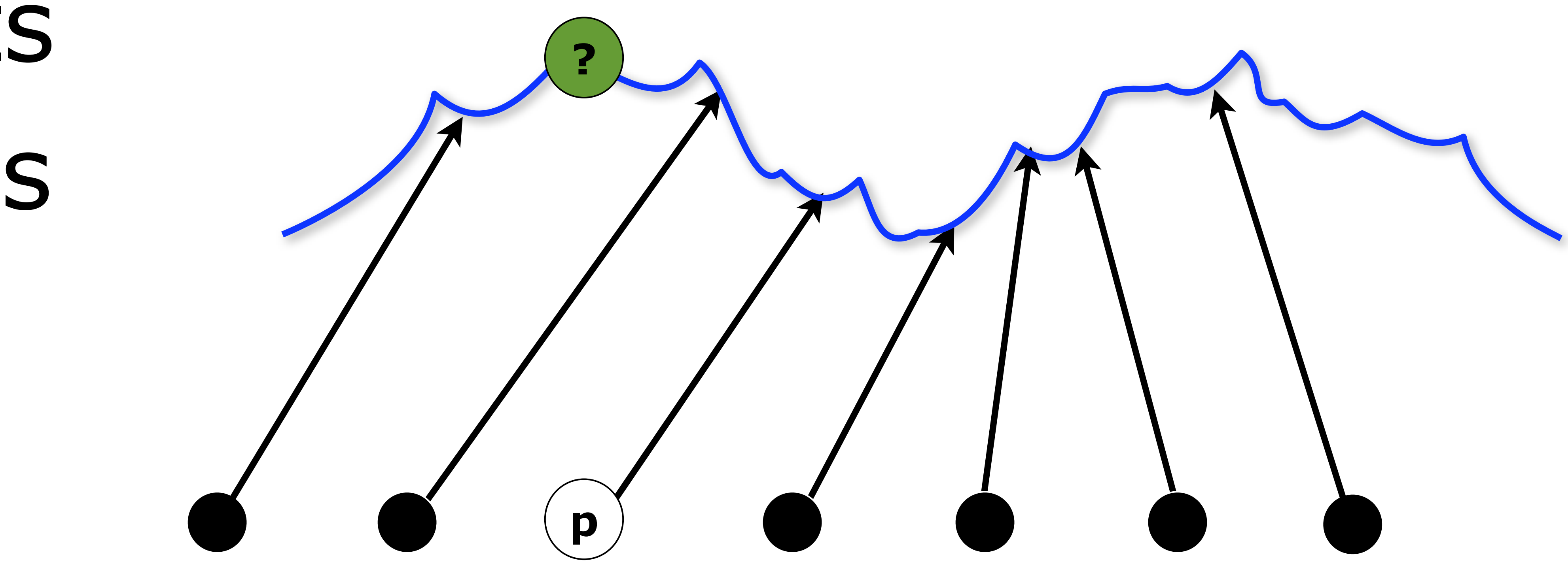
Don’t sample all waves (filter out high frequencies)

Use a spring to allow movement of bow



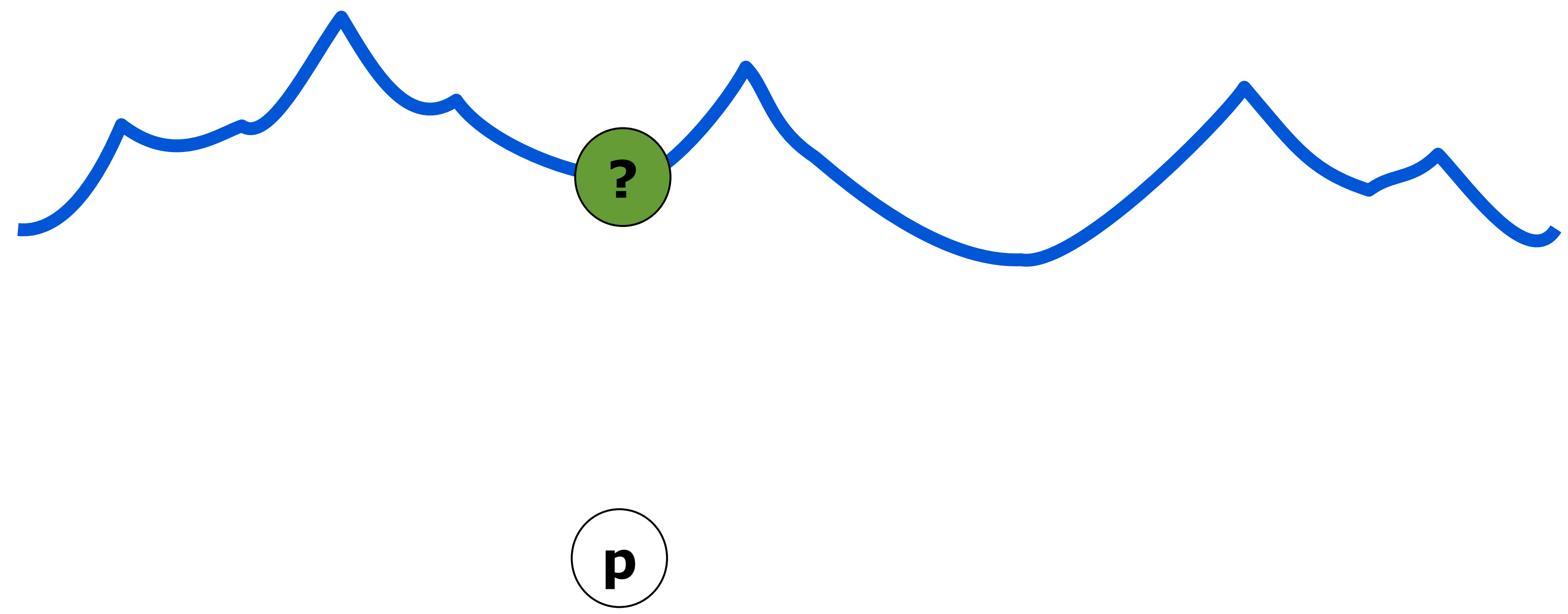
Point queries

- Player swimming
- Cameras
- Floating objects
- Collision probes



Sampling a vector displacement field is not as trivial as a heightfield.

Point queries



Cameras, player query:

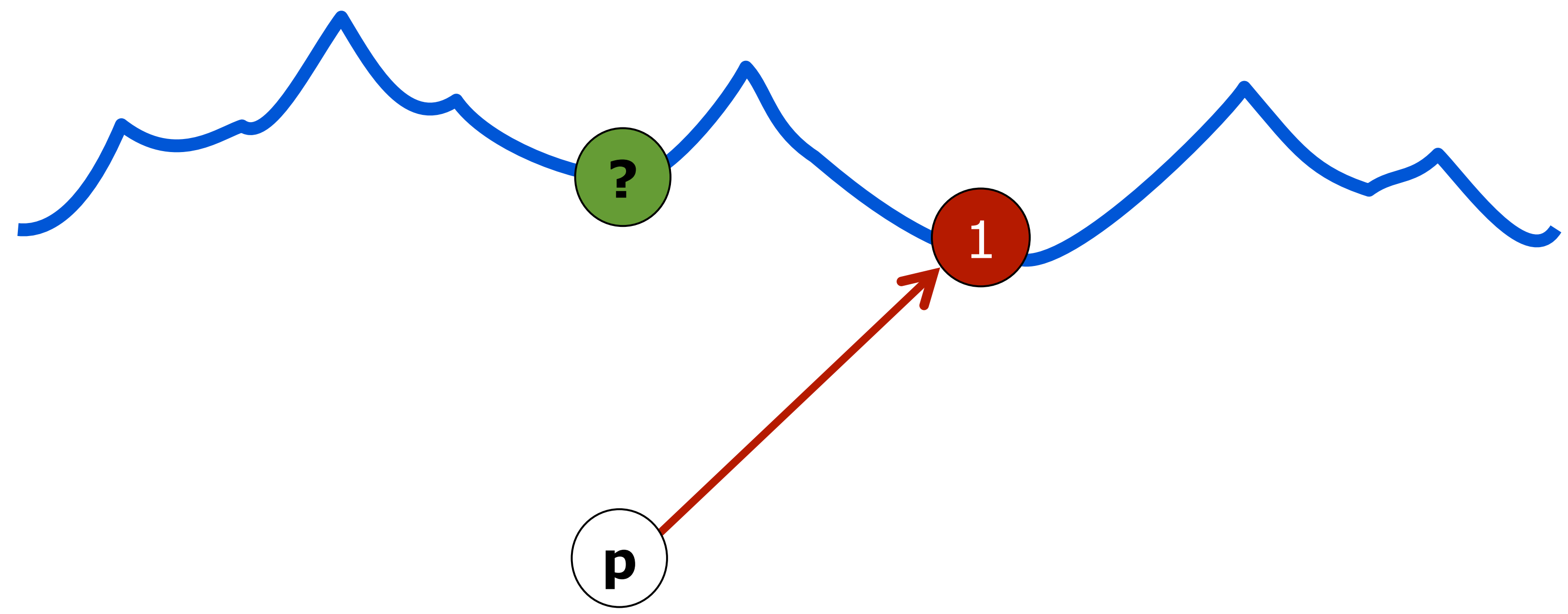
given point **p** $\langle u,v \rangle$ find water position **r** $\langle u, y, v \rangle$

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

- p query $\langle u,v \rangle$
- q result $\langle u,v \rangle + \langle x,y,z \rangle$
- s projection $\langle u,v \rangle + \langle 0,y,0 \rangle$
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

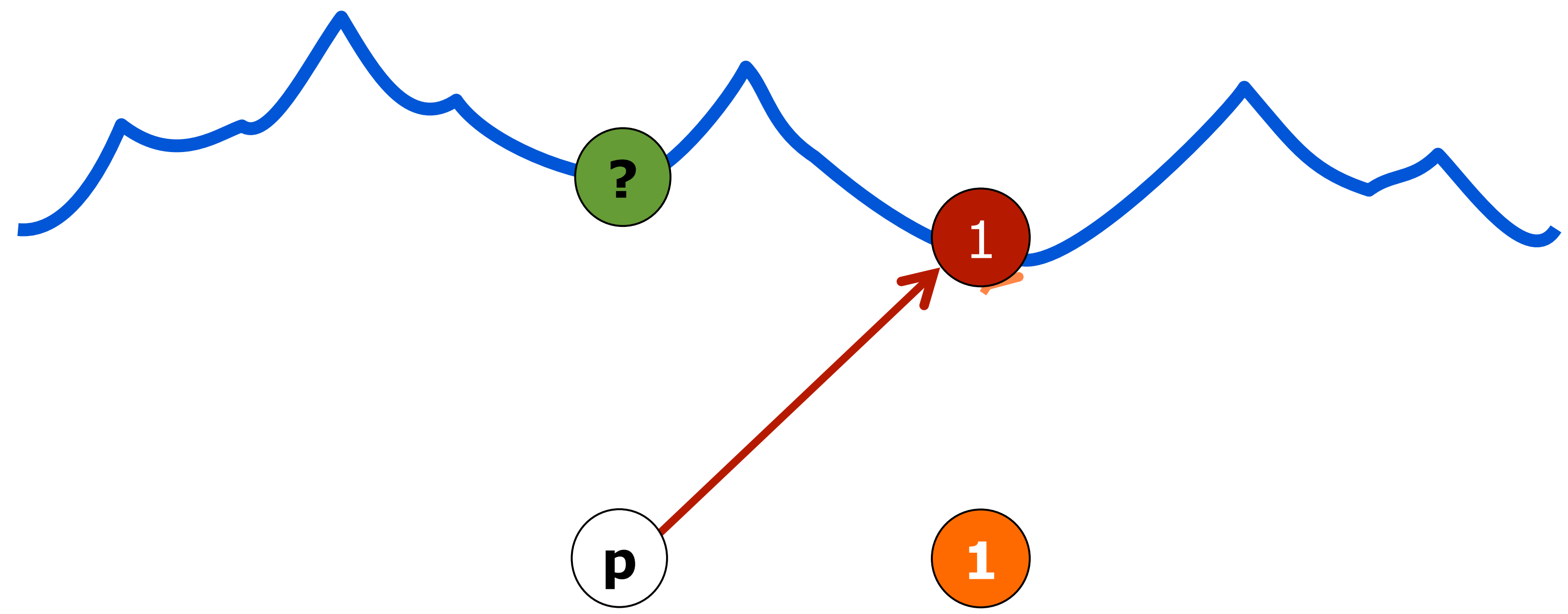
given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

- p query $<u,v>$
- q result $<u,v>+<x,y,z>$
- s projection $<u,v>+<0,y,0>$
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

p query *<u,v>*

q result *<u,v>+<x,y,z>*

s projection *<u,v>+<0,y,0>*

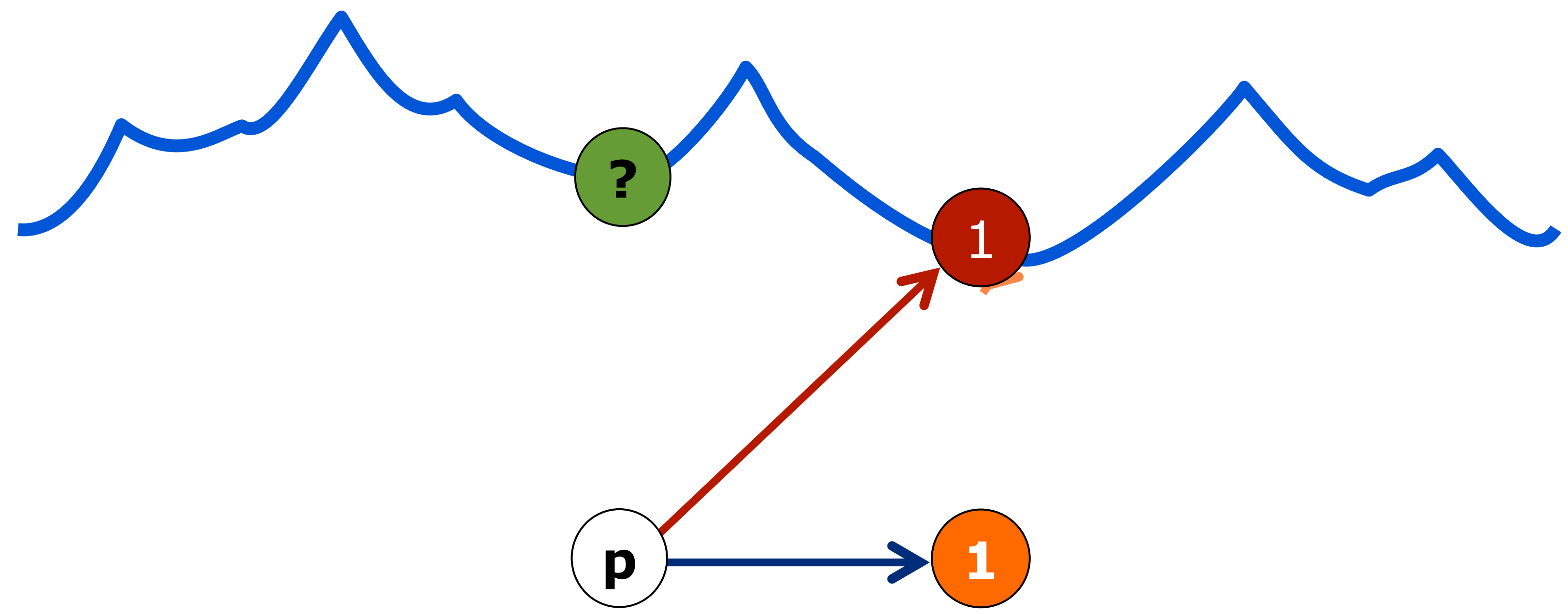
r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries

Cameras, player query:

given point **p** $<u,v>$ find water position **r** $<u, y, v>$



Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

p query $<u,v>$

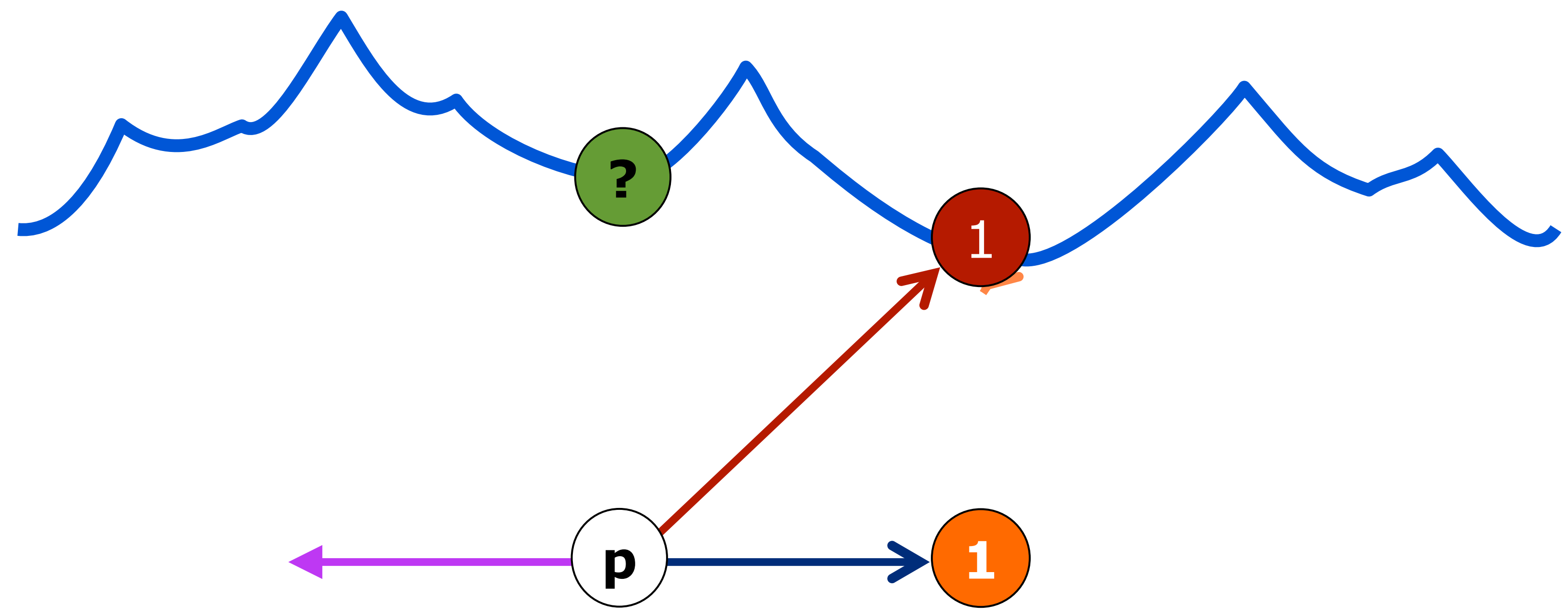
q result $<u,v>+<x,y,z>$

s projection $<u,v>+<0,y,0>$

r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

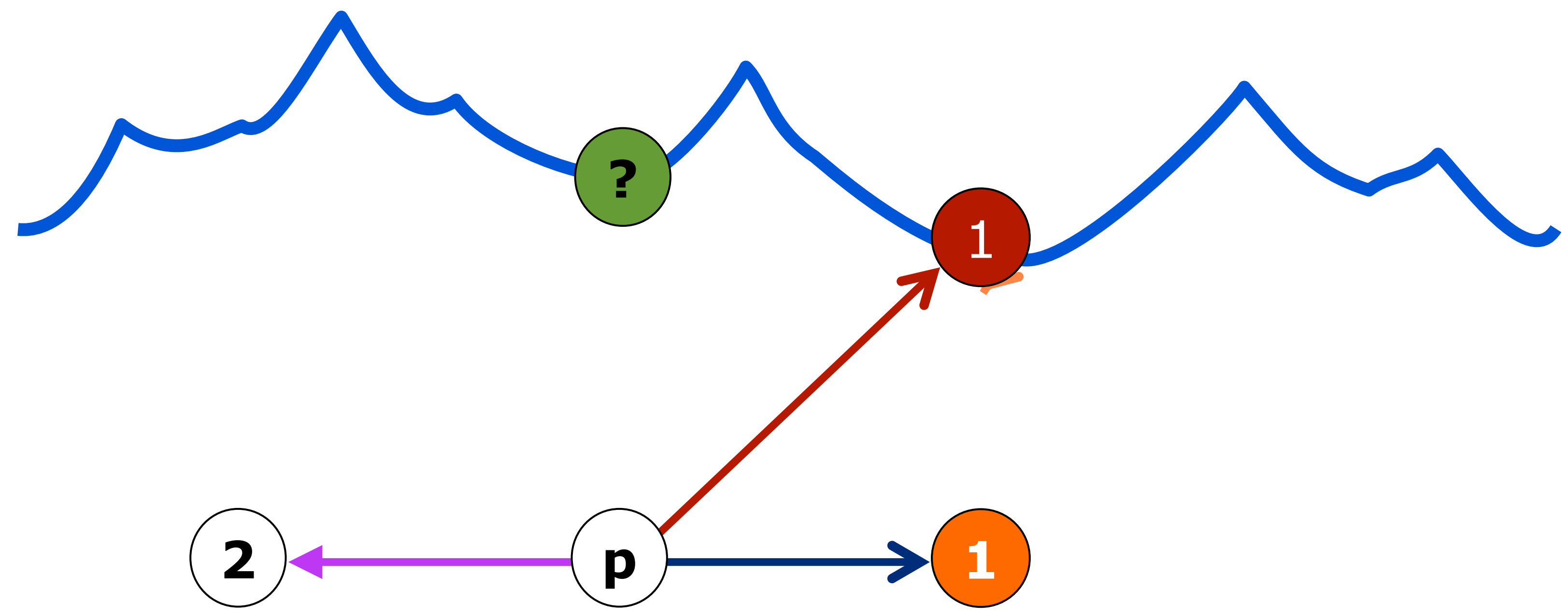
given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

- p query *<u,v>*
- q result *<u,v>+<x,y,z>*
- s projection *<u,v>+<0,y,0>*
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

p query $<u,v>$

q result $<u,v>+<x,y,z>$

s projection $<u,v>+<0,y,0>$

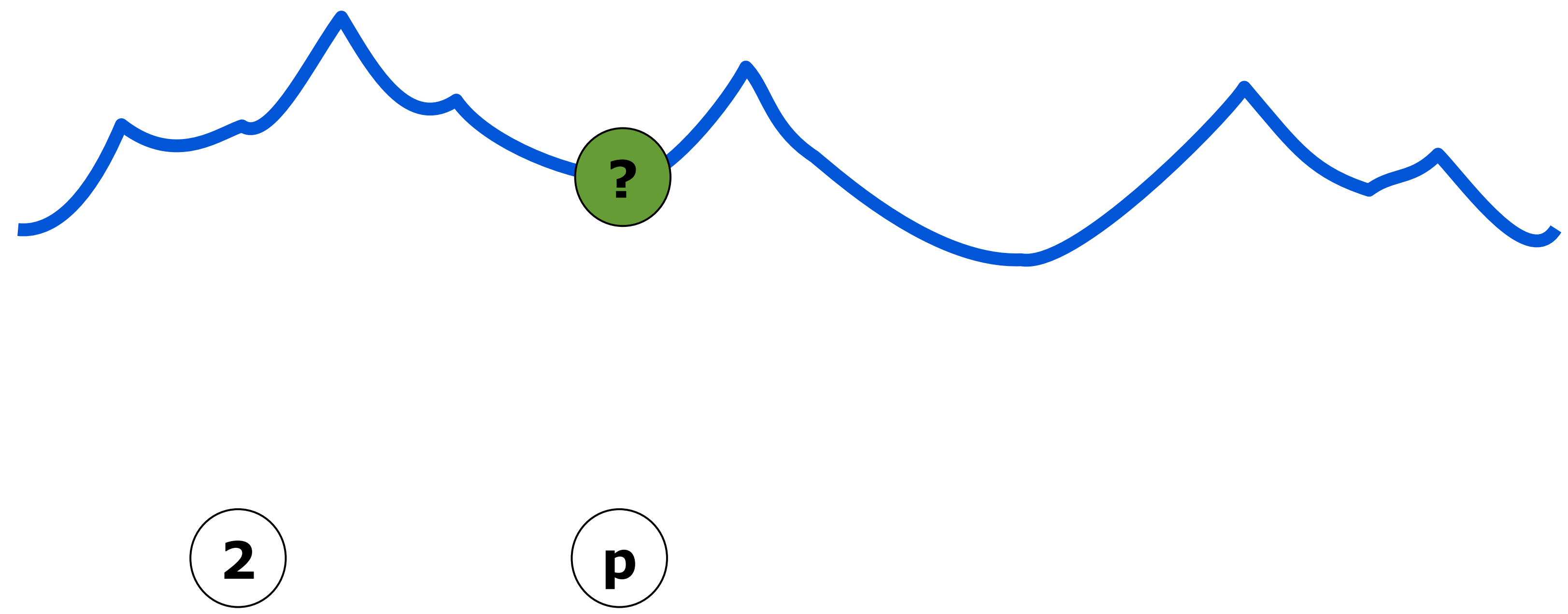
r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries

Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>



Ryan Broner’s search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

p query *<u,v>*

q result *<u,v>+<x,y,z>*

s projection *<u,v>+<0,y,0>*

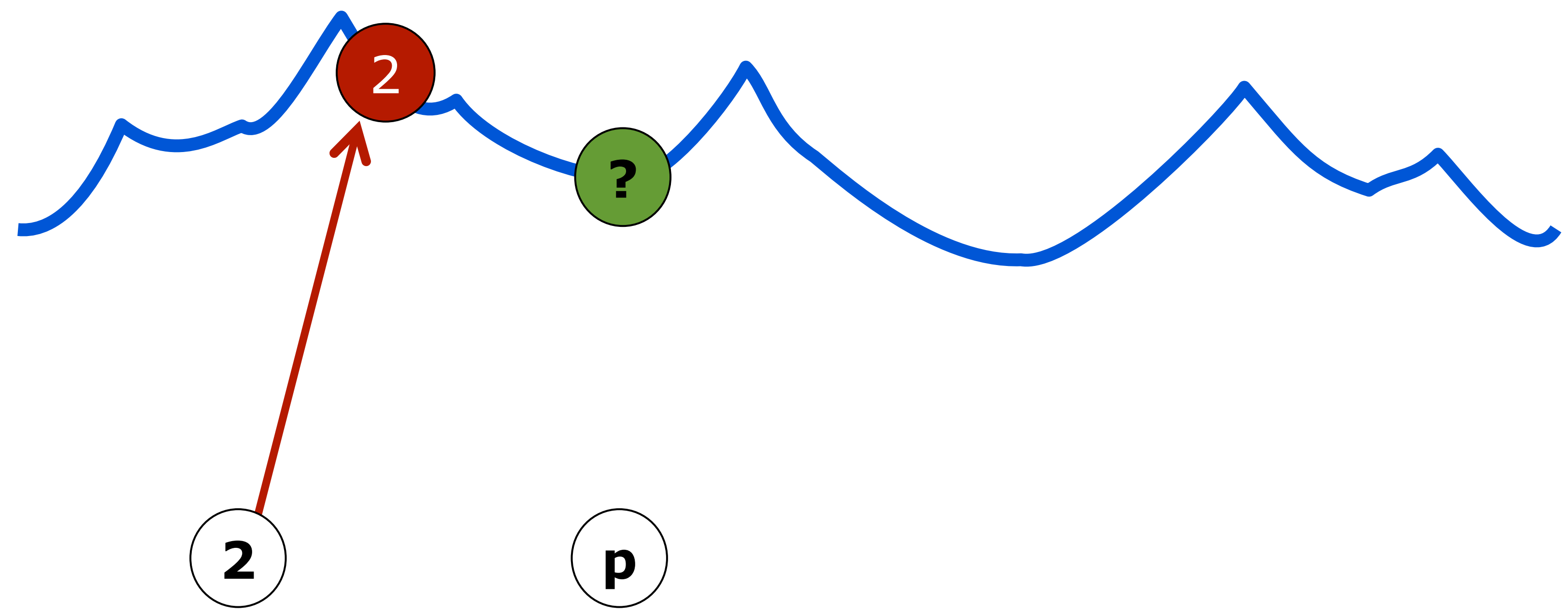
r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries

Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

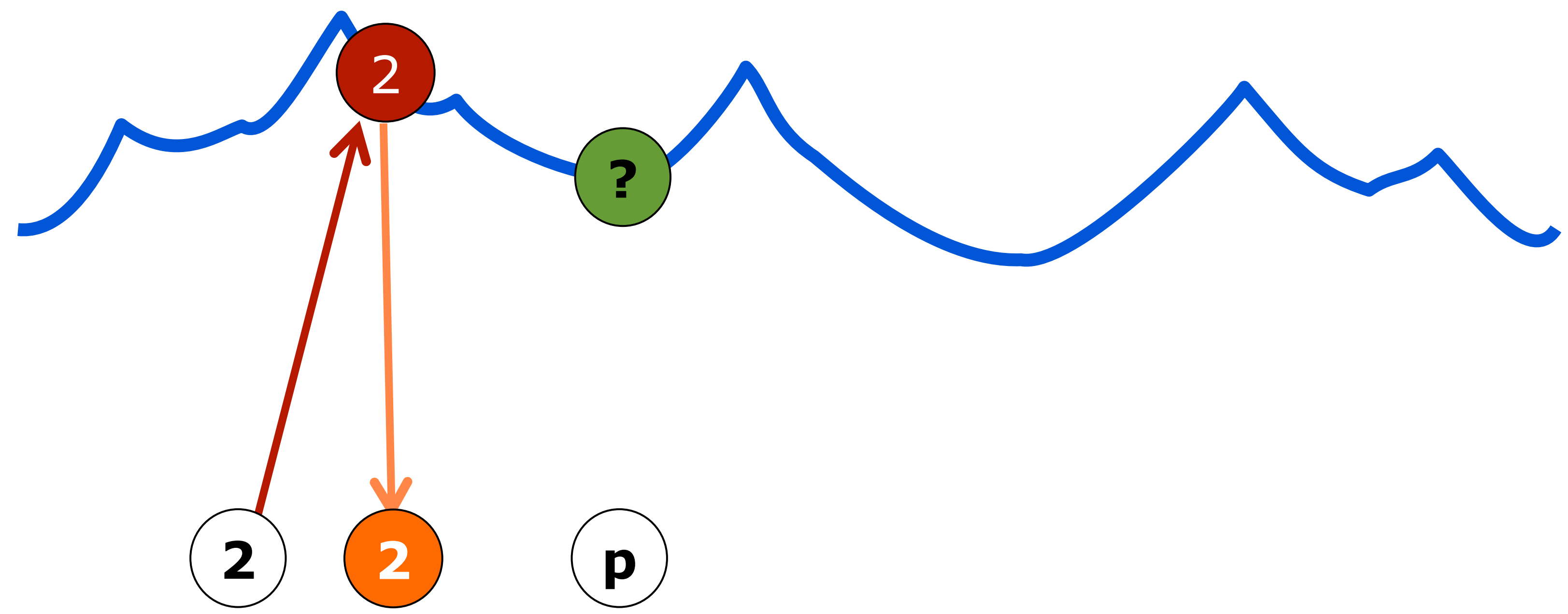


Ryan Broner’s search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

- p query $<u,v>$
- q result $<u,v>+<x,y,z>$
- s projection $<u,v>+<0,y,0>$
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

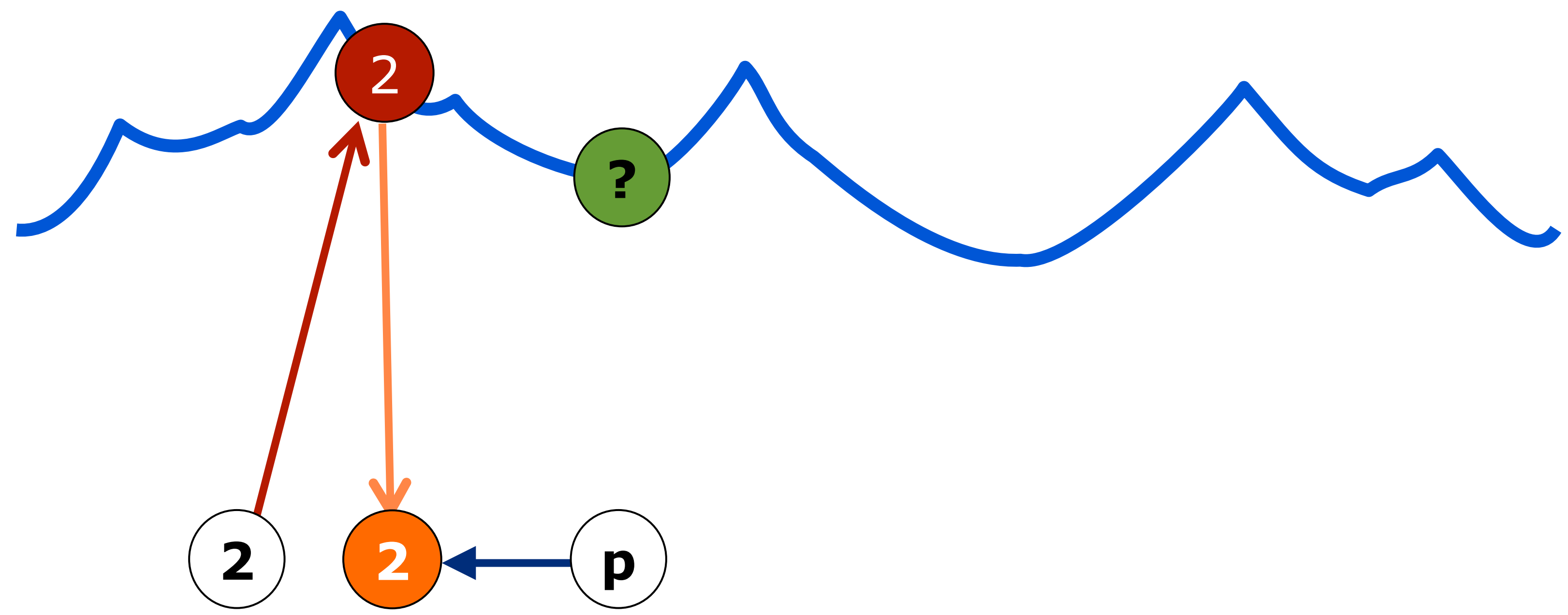
given point **p** $\langle u,v \rangle$ find water position **r** $\langle u, y, v \rangle$

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

- **p** query $\langle u,v \rangle$
- **q** result $\langle u,v \rangle + \langle x,y,z \rangle$
- **s** projection $\langle u,v \rangle + \langle 0,y,0 \rangle$
- **r** final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

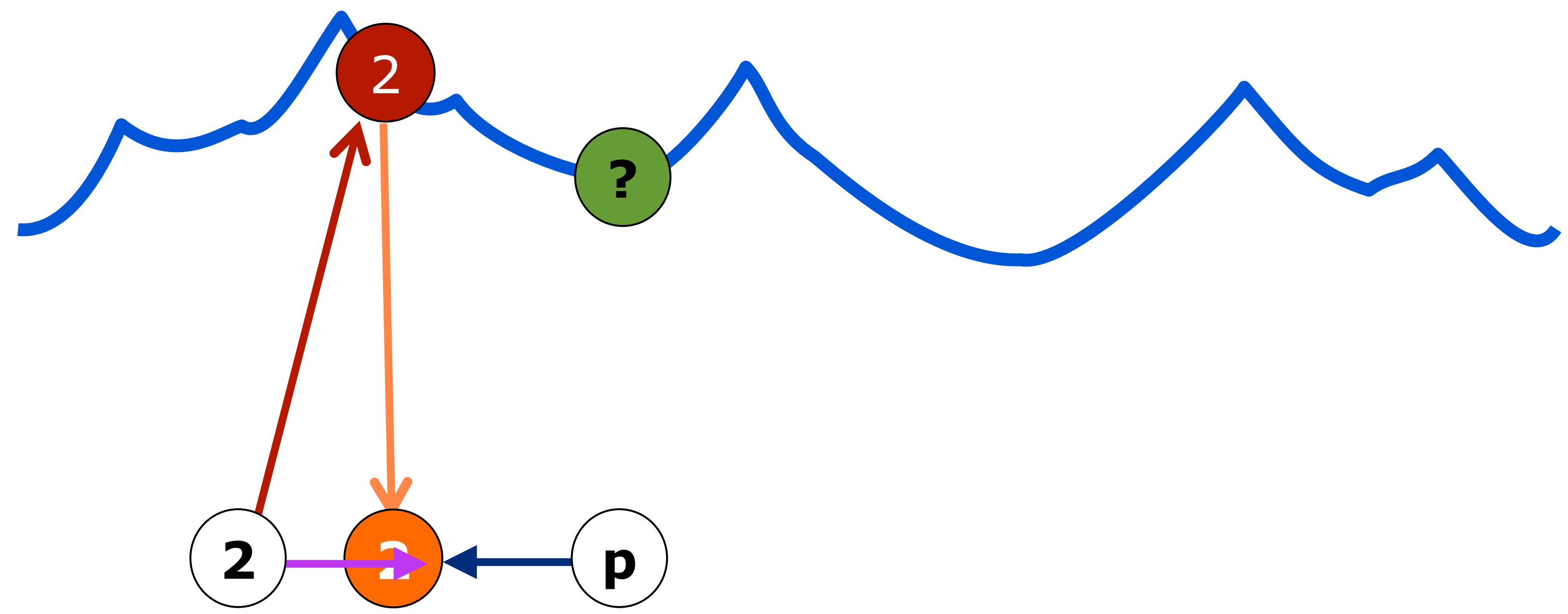
given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

- p query <*u*,*v*>
- q result <*u*,*v*>+<*x*,*y*,*z*>
- s projection <*u*,*v*>+<*0*,*y*,*0*>
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner’s search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

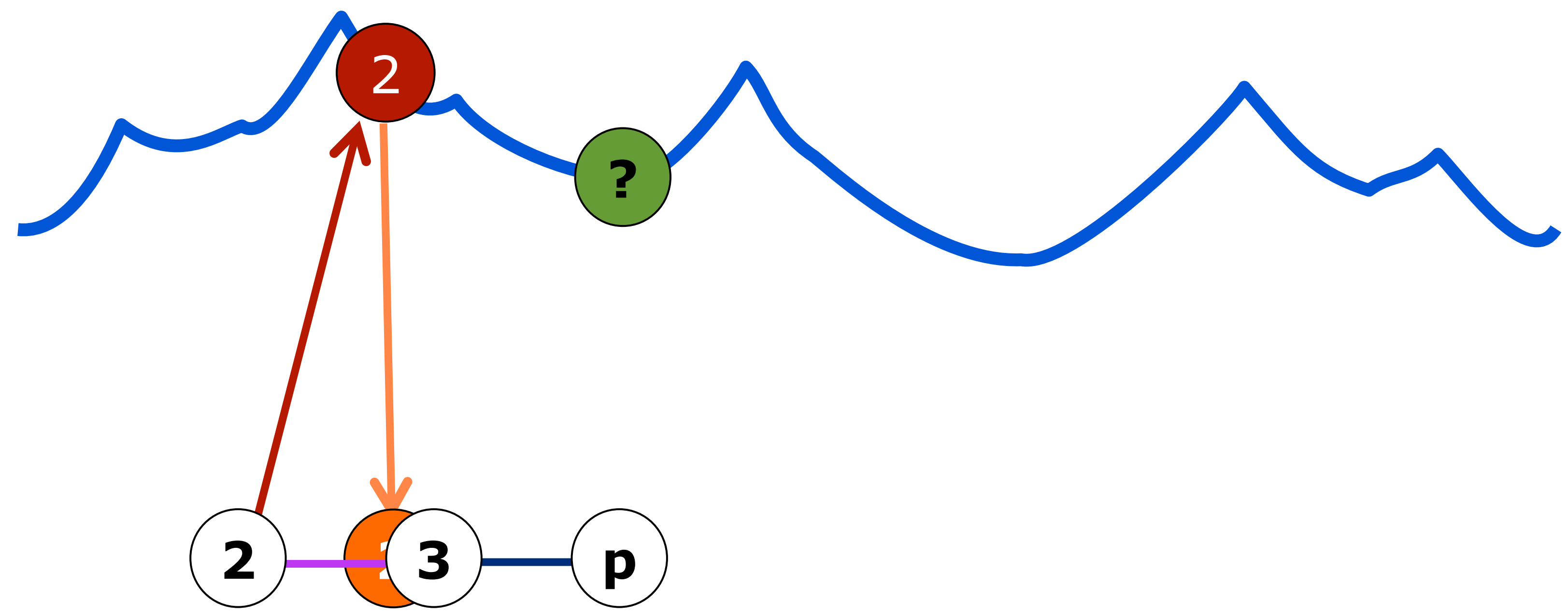
- p query *<u,v>*
- q result *<u,v>+<x,y,z>*
- s projection *<u,v>+<0,y,0>*
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries

Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>



Ryan Broner’s search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

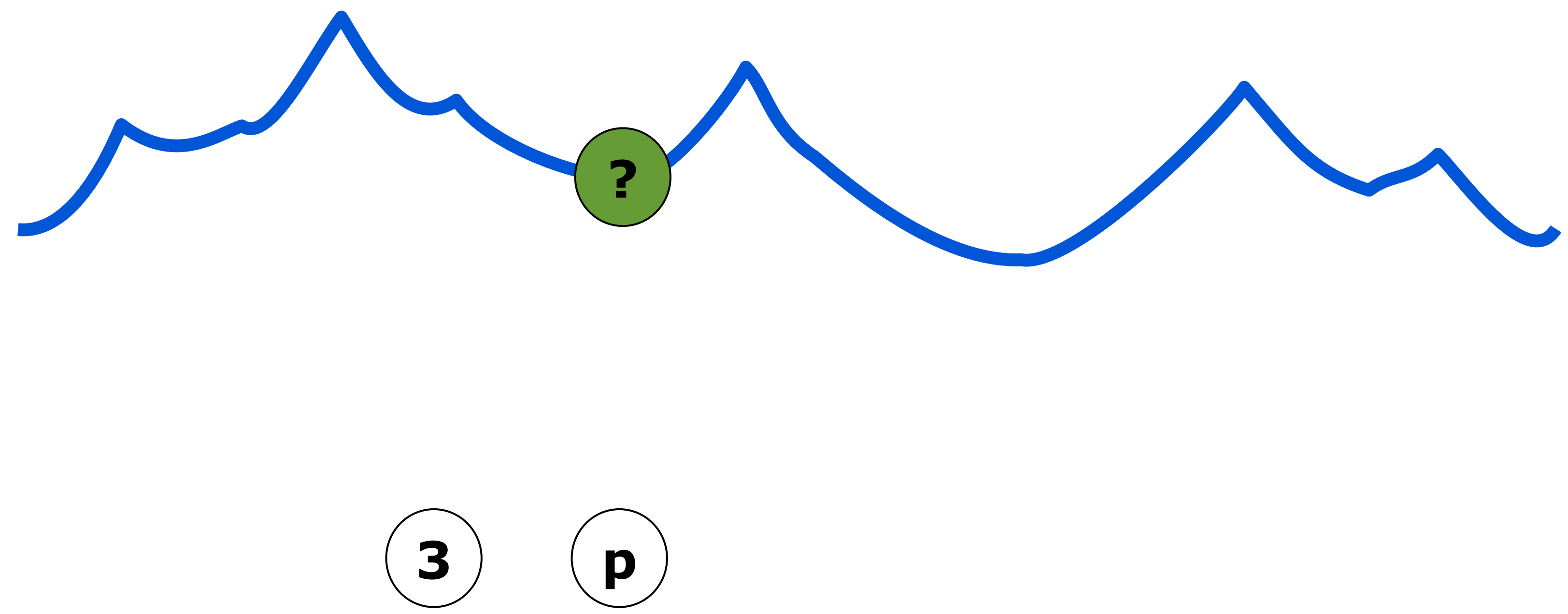
- p query $<u,v>$
- q result $<u,v>+<x,y,z>$
- s projection $<u,v>+<0,y,0>$
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries

Cameras, player query:

given point **p** <u,v> find
water position **r** <u, y, v>



Ryan Broner’s search method
Similar to the Secant method,
but use displacement in a
$R^2 \rightarrow R^3$ map

p query $<u,v>$

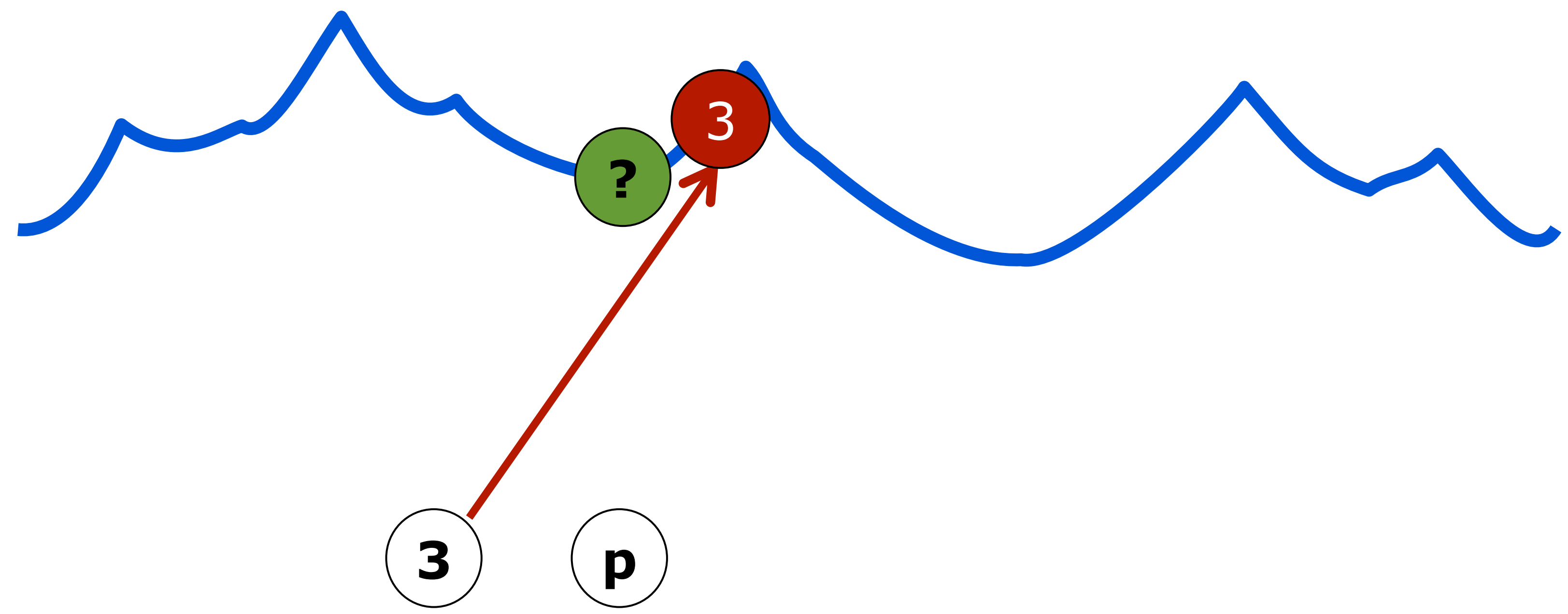
q result $<u,v>+<x,y,z>$

s projection $<u,v>+<0,y,0>$

r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

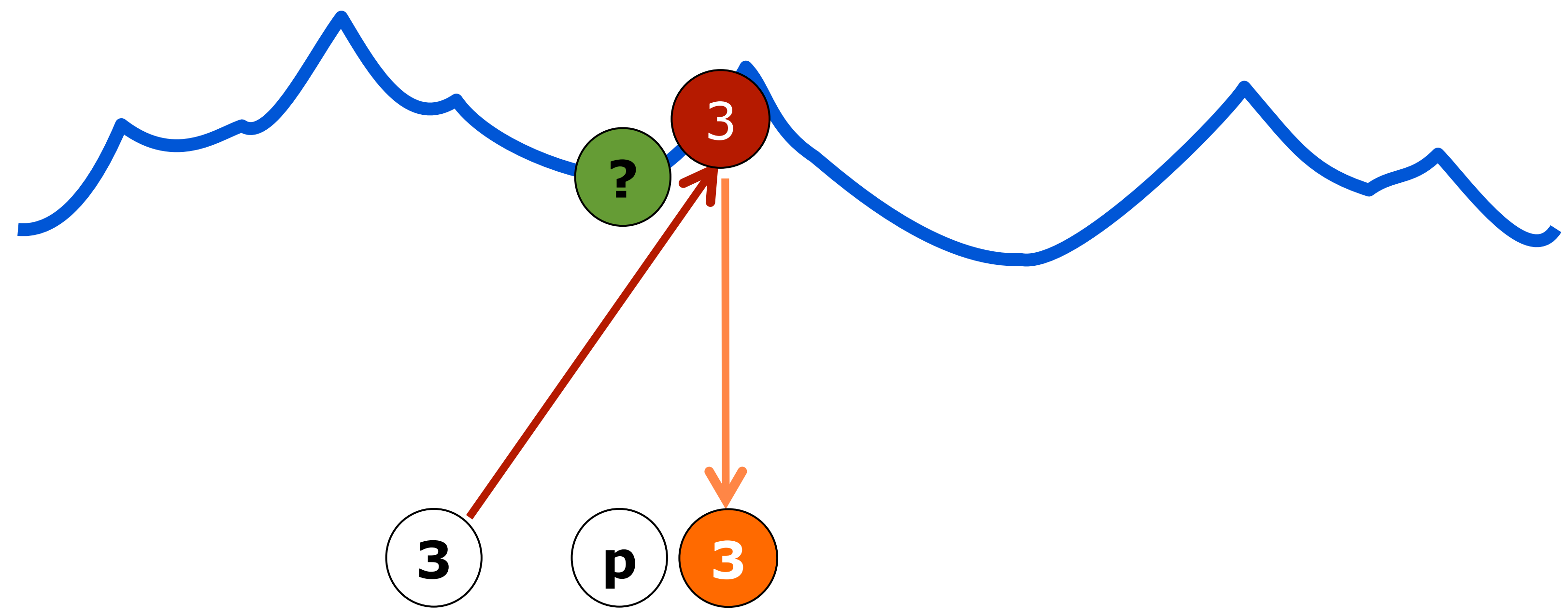
given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

- p query $<u,v>$
- q result $<u,v>+<x,y,z>$
- s projection $<u,v>+<0,y,0>$
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

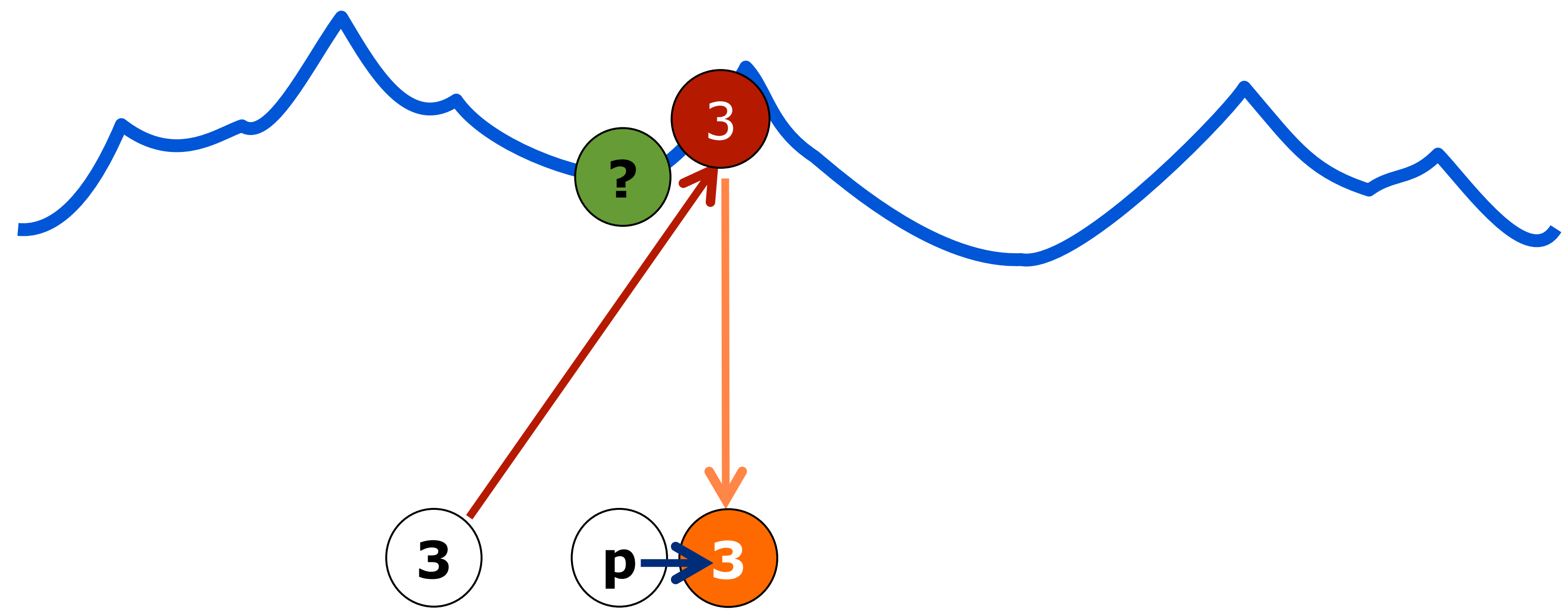
- **p** query $<u,v>$
- **q** result $<u,v>+<x,y,z>$
- **s** projection $<u,v>+<0,y,0>$
- **r** final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries

Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>



Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

p query $<u,v>$

q result $<u,v>+<x,y,z>$

s projection $<u,v>+<0,y,0>$

r final result position

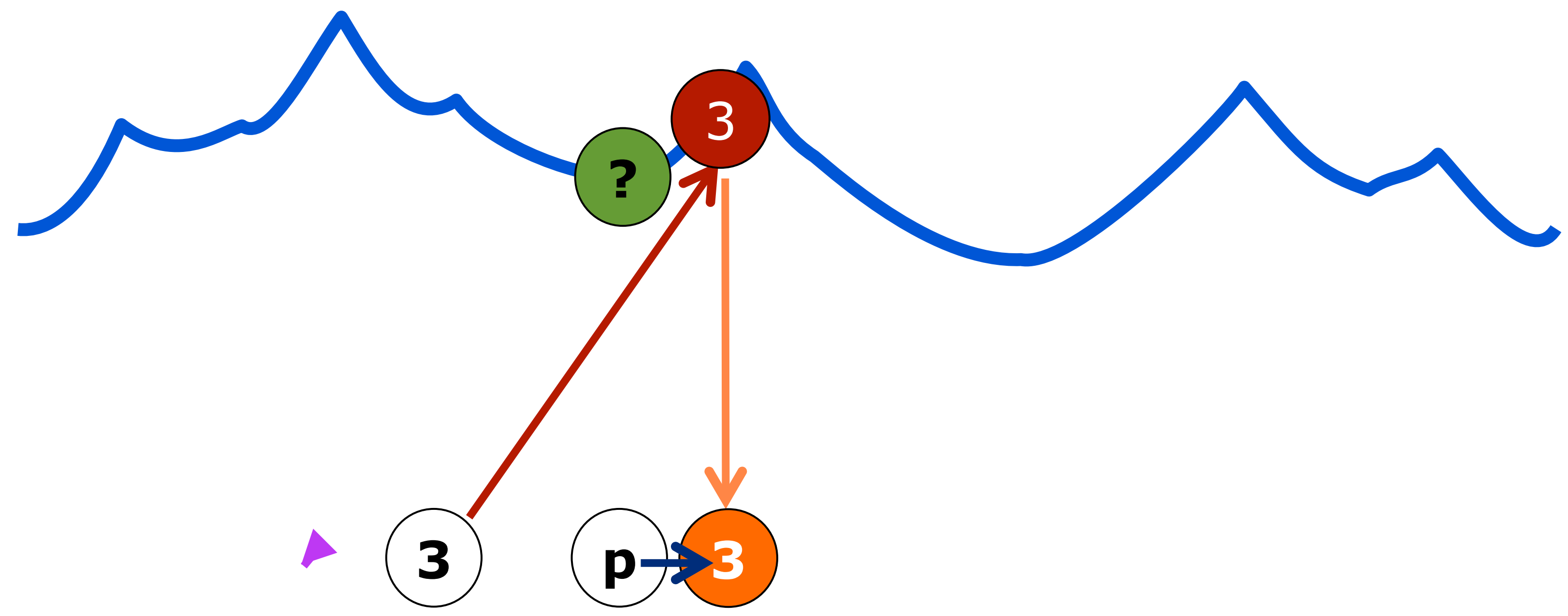
We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries

Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map



p query $<u,v>$

q result $<u,v>+<x,y,z>$

s projection $<u,v>+<0,y,0>$

r final result position

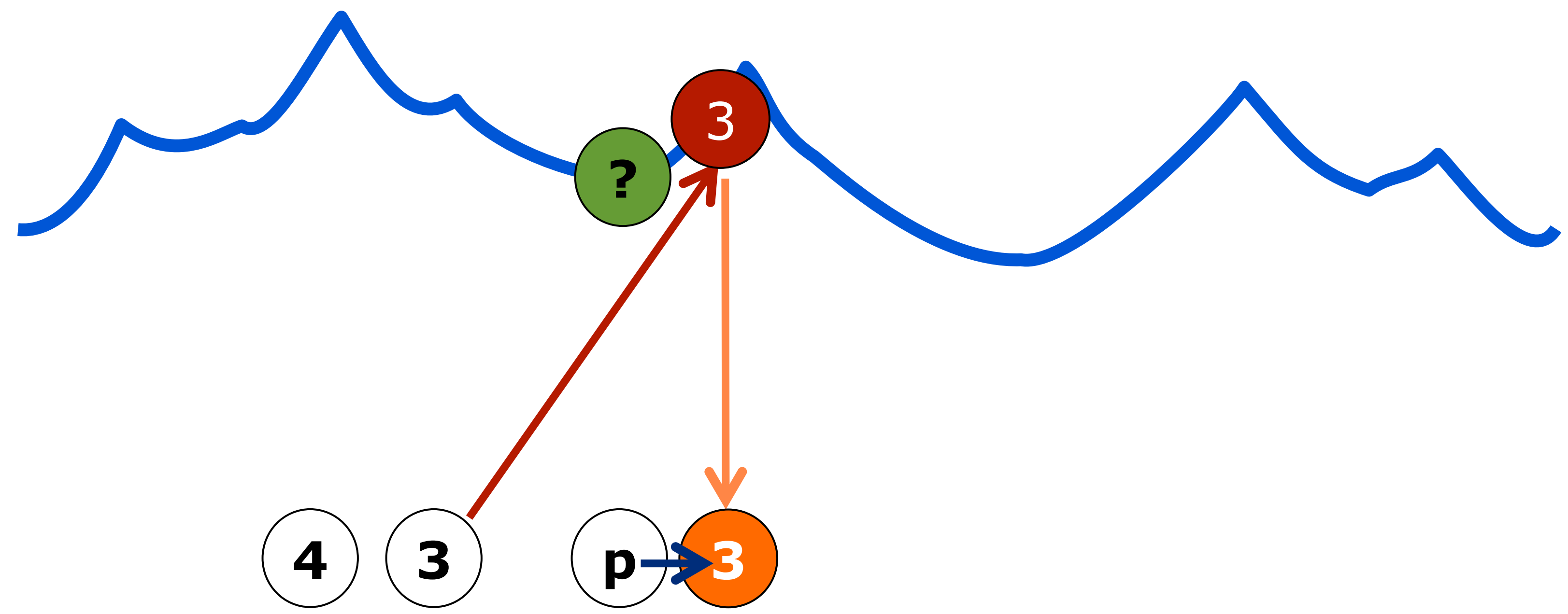
We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries

Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map



p query $<u,v>$

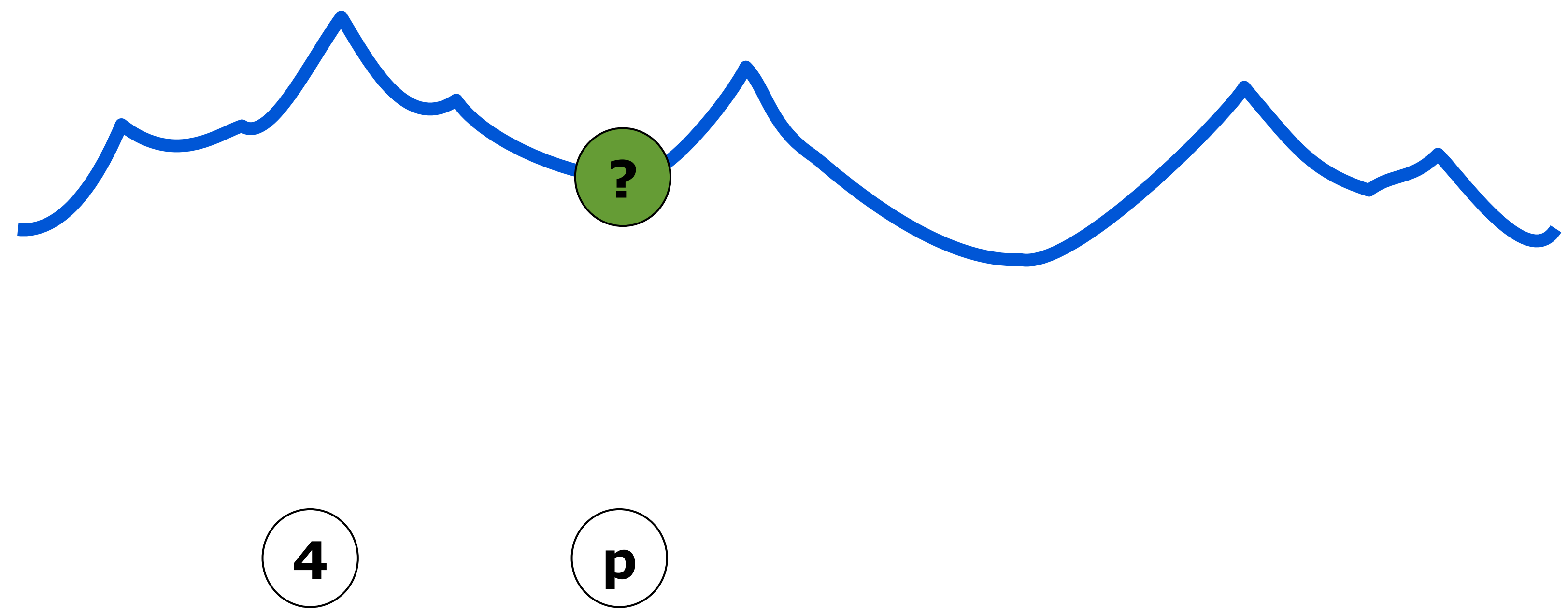
q result $<u,v>+<x,y,z>$

s projection $<u,v>+<0,y,0>$

r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

given point **p** $\langle u,v \rangle$ find water position **r** $\langle u, y, v \rangle$

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

p query $\langle u,v \rangle$

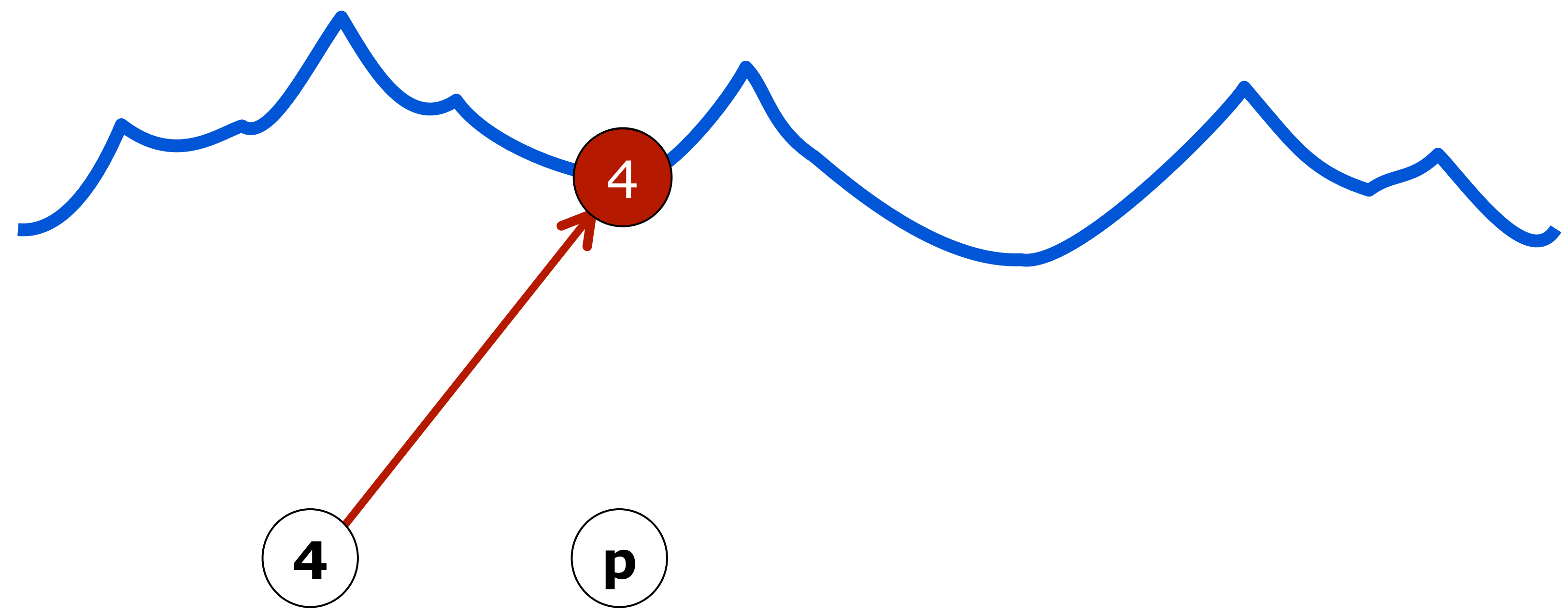
q result $\langle u,v \rangle + \langle x,y,z \rangle$

s projection $\langle u,v \rangle + \langle 0,y,0 \rangle$

r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

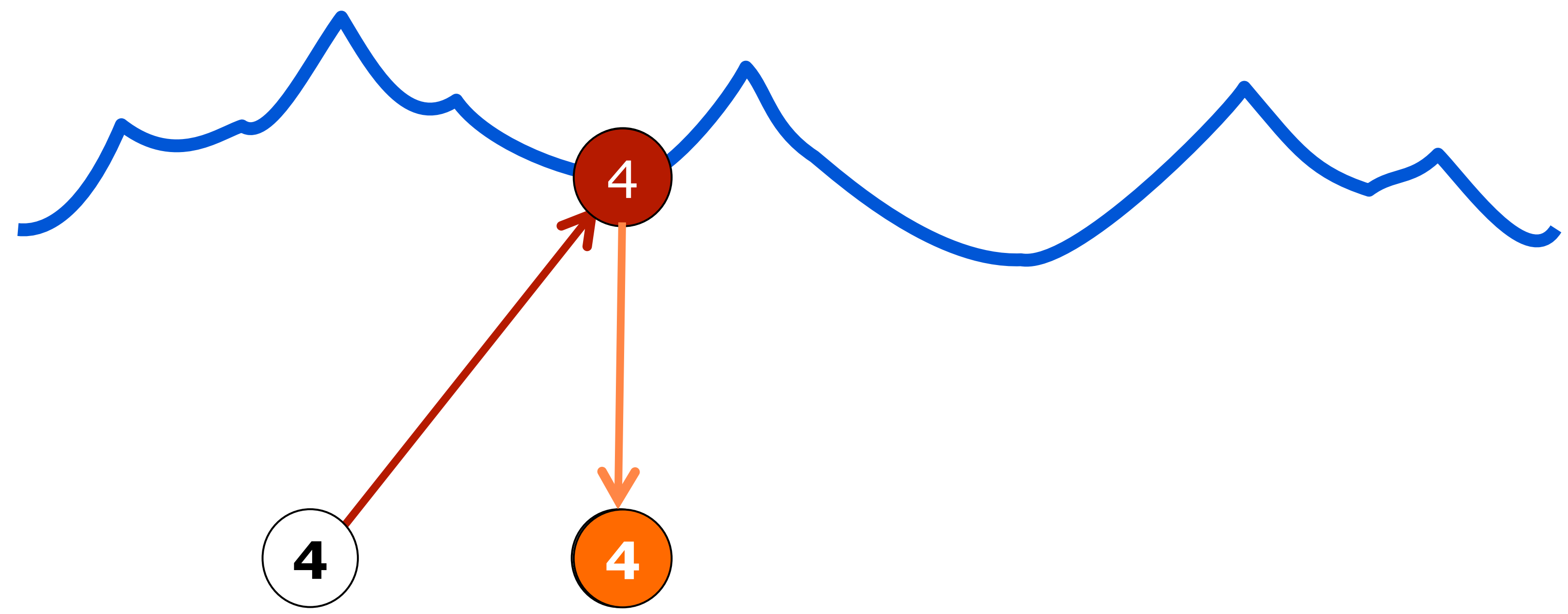
given point **p** $\langle u,v \rangle$ find water position **r** $\langle u, y, v \rangle$

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ map

- p query $\langle u,v \rangle$
- q result $\langle u,v \rangle + \langle x,y,z \rangle$
- s projection $\langle u,v \rangle + \langle 0,y,0 \rangle$
- r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

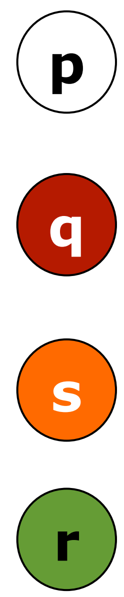
Point queries



Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map



query $<u,v>$

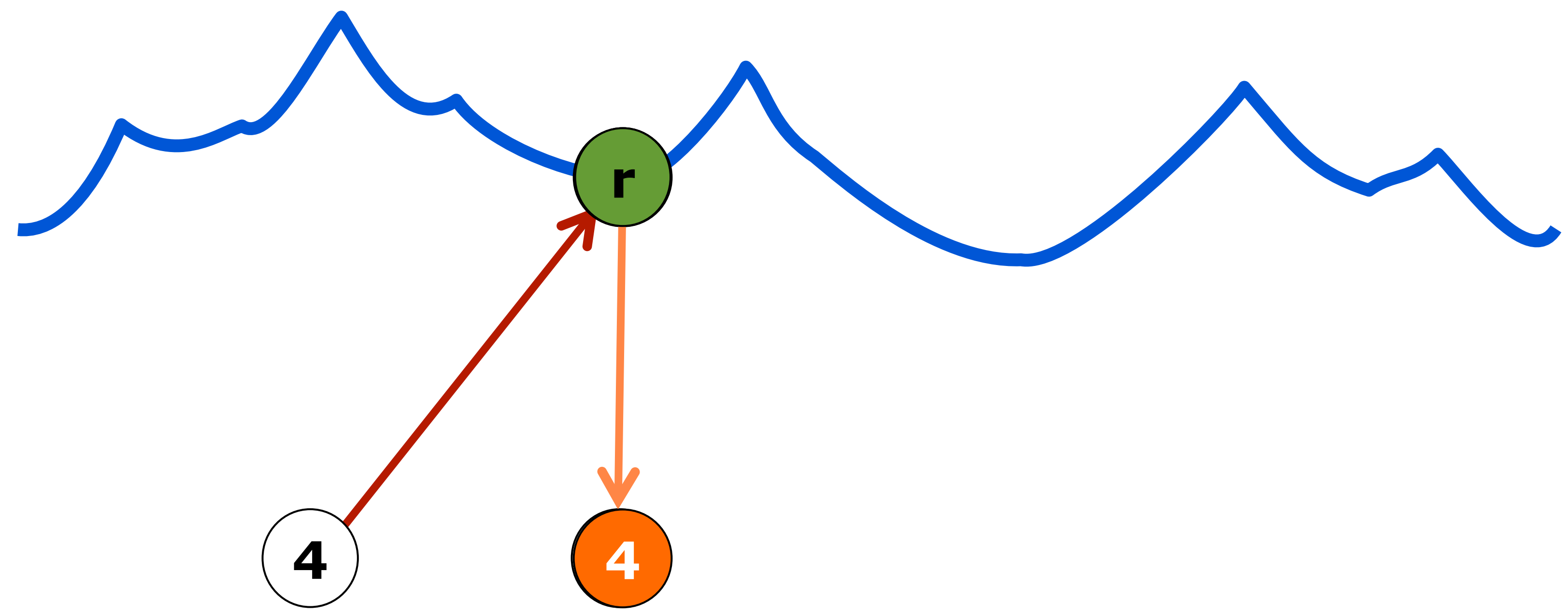
result $<u,v>+<x,y,z>$

projection $<u,v>+<0,y,0>$

final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Point queries



Cameras, player query:

given point **p** <u,v> find water position **r** <u, y, v>

Ryan Broner's search method
Similar to the Secant method, but use displacement in a $R^2 \rightarrow R^3$ map

p query $<u,v>$

q result $<u,v>+<x,y,z>$

s projection $<u,v>+<0,y,0>$

r final result position

We use a search method to the wave field instead of building a mesh and then do some form of ray casting.

Mesh computation

For each ring, run an SPU job to handle patches (i % 3):

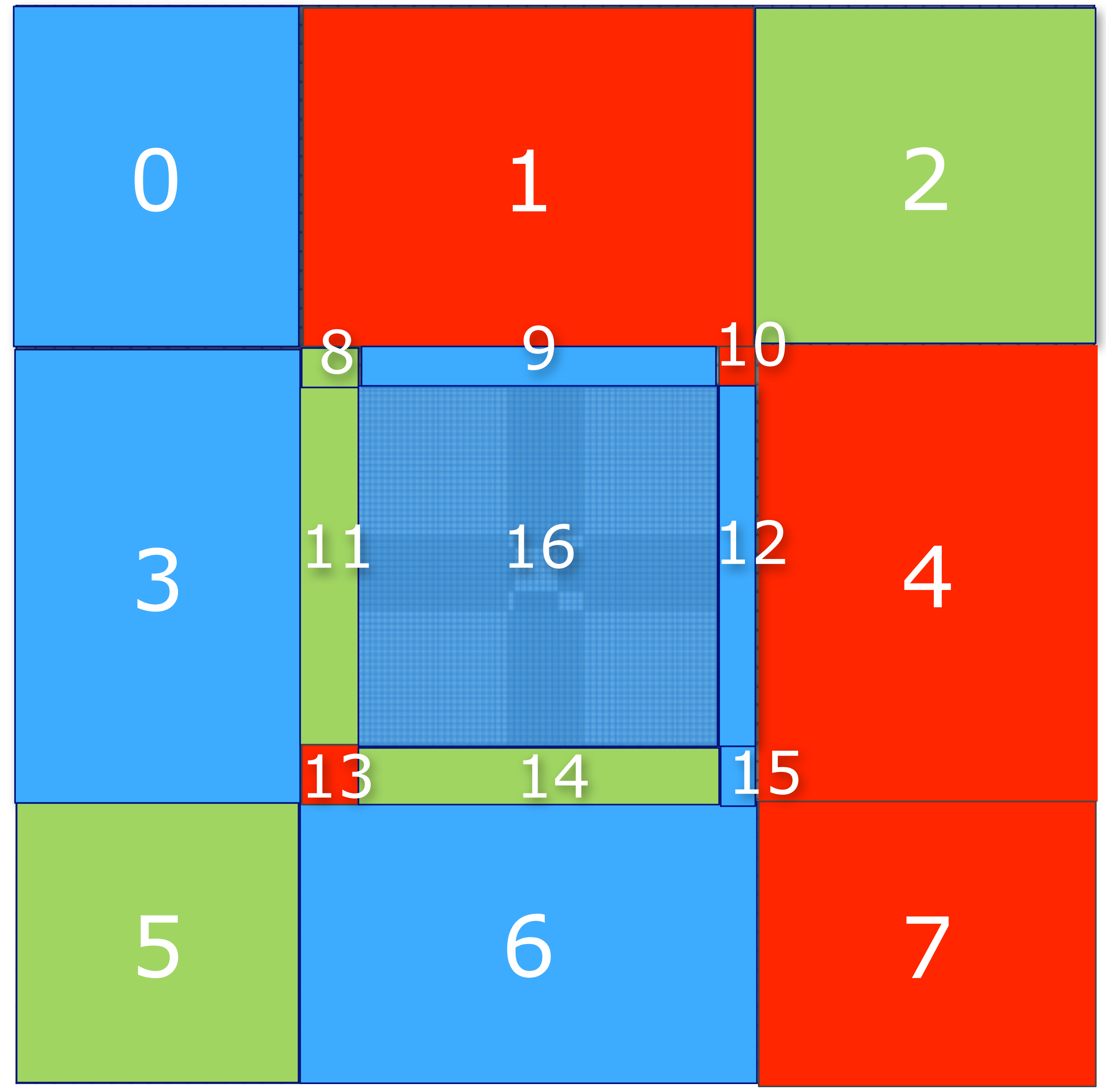
J1: (0,3,6,9,12,15)

J2: (1,4,7,10,13,[16])

J3: (2,5,8,11,14)

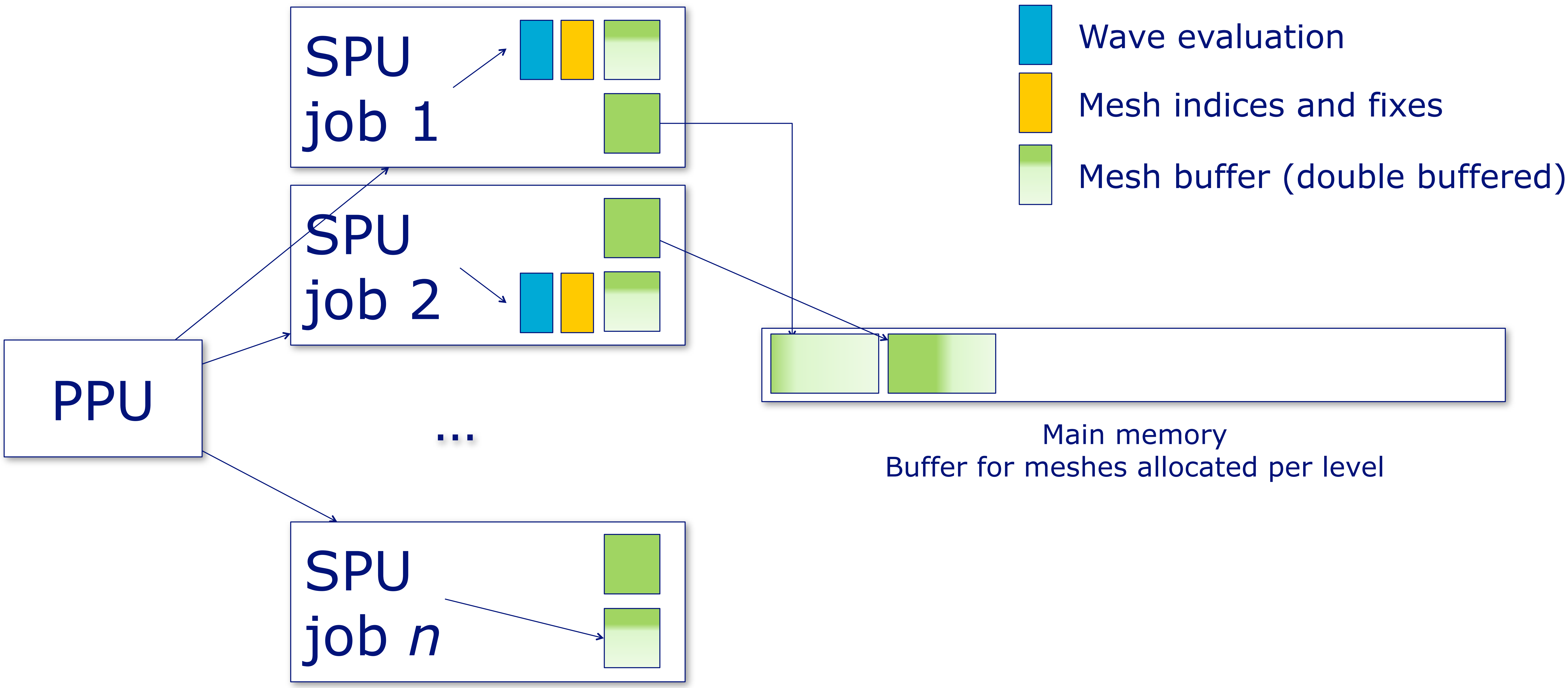
Minimize ring level computation

Double buffer mesh output



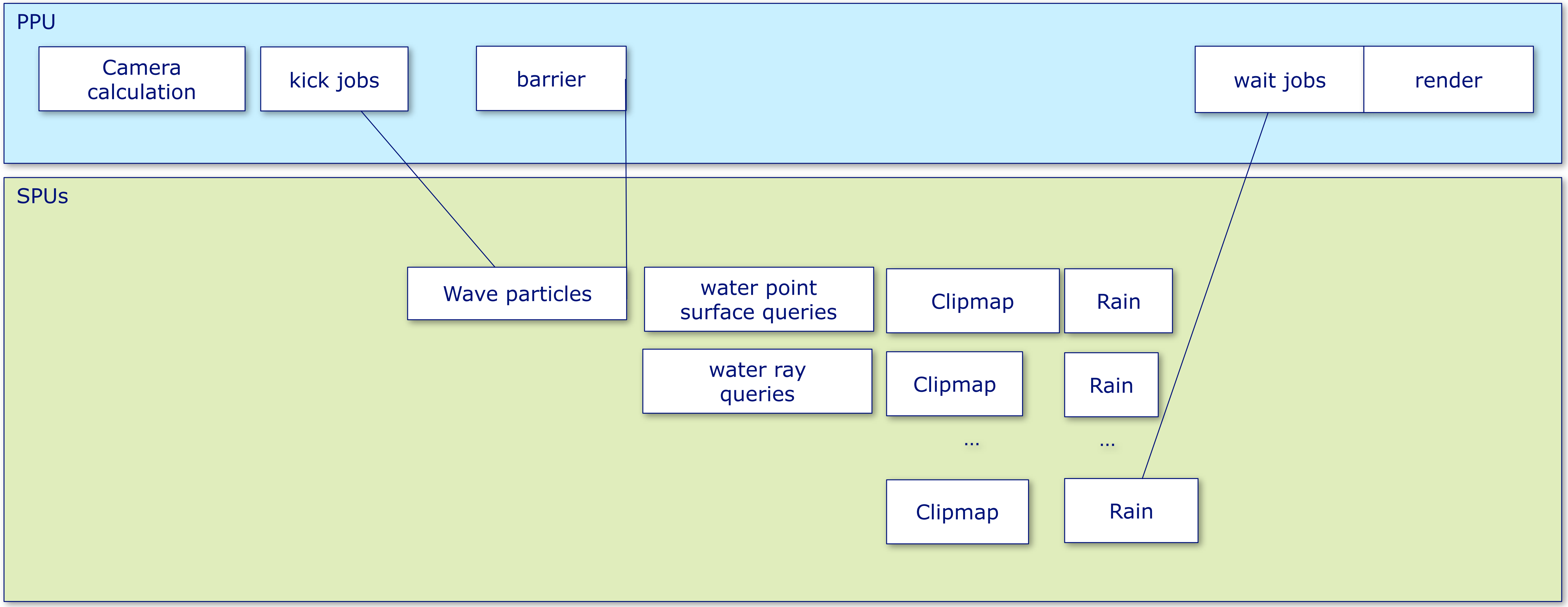
The job distribution per SPU is in round-robin order.

A job consists on a set of patches (indexed using a modulus 3) on a particular ring.



Since each job will create a perfectly seemed mesh, there is no need to stitch back the mesh.

The final mesh of the ocean, consists on multiple meshes



We only have to be careful with time. The clipmaps need the wave particles at a particular time.

We only need a single wave-particles job. This job generates a displacement grid.

To synchronize the jobs we put a barrier to wait for the wave-particles job to have finished generating its mesh.

Performance

Ocean

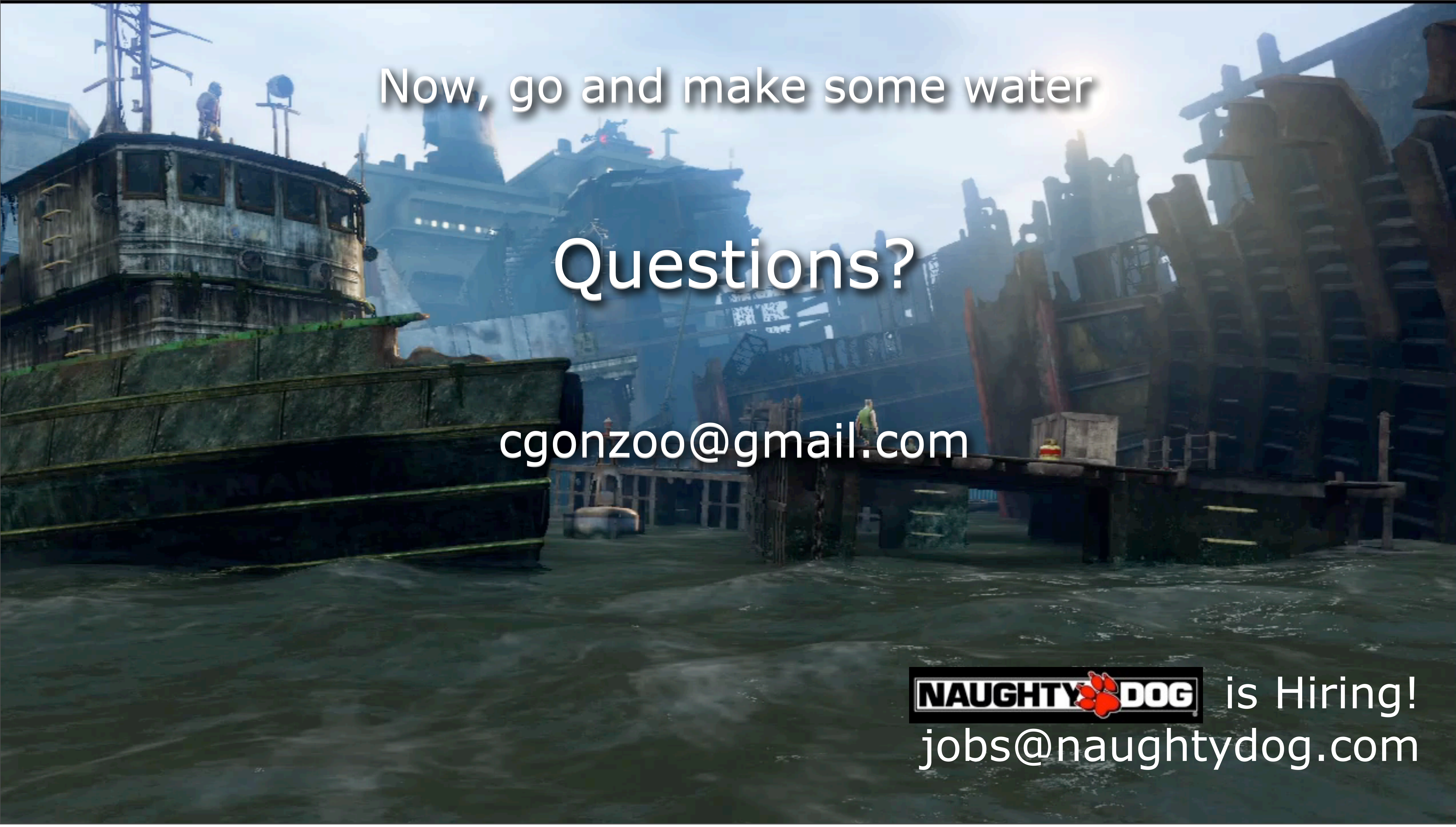
- 0.9 ms wave particles
- 0.1 ms water query (point and ray)
- 8.0 ms tessellation + wave displacement
  - 5 SPUs
    - average 7 rings = 21 jobs

~2.7 ms rendering

- average 50+ visible patches

1 Mb double buffer memory


UNCHARTED 3
DRAKE'S DECEPTION


Now, go and make some water
Questions?
cgonzoo@gmail.com
NAUGHTY DOG is Hiring!
jobs@naughtydog.com

This is a shot of the first pond we generated for U1. This was out test bed for flow, foam and interaction of Drake's
clothes