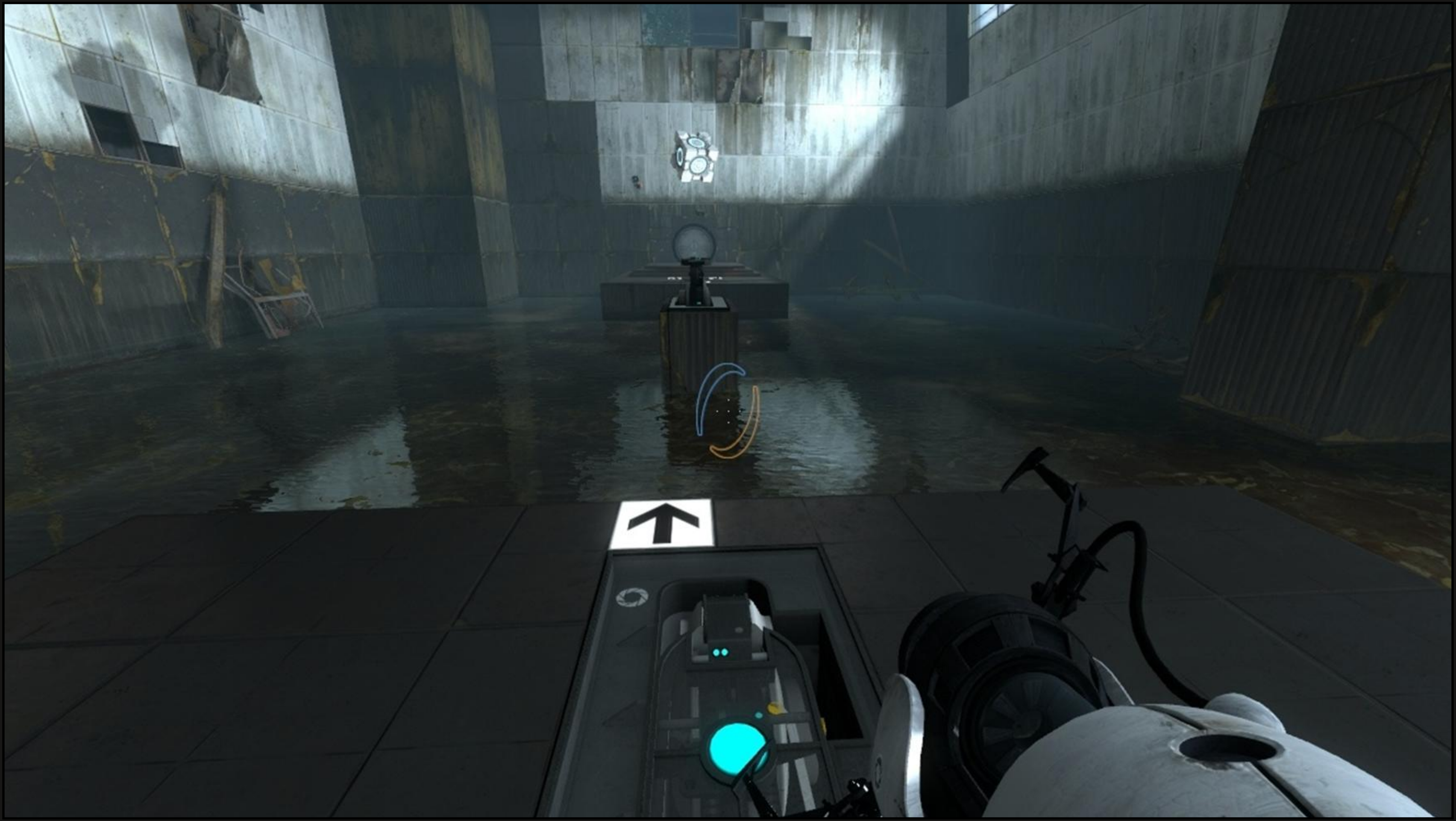
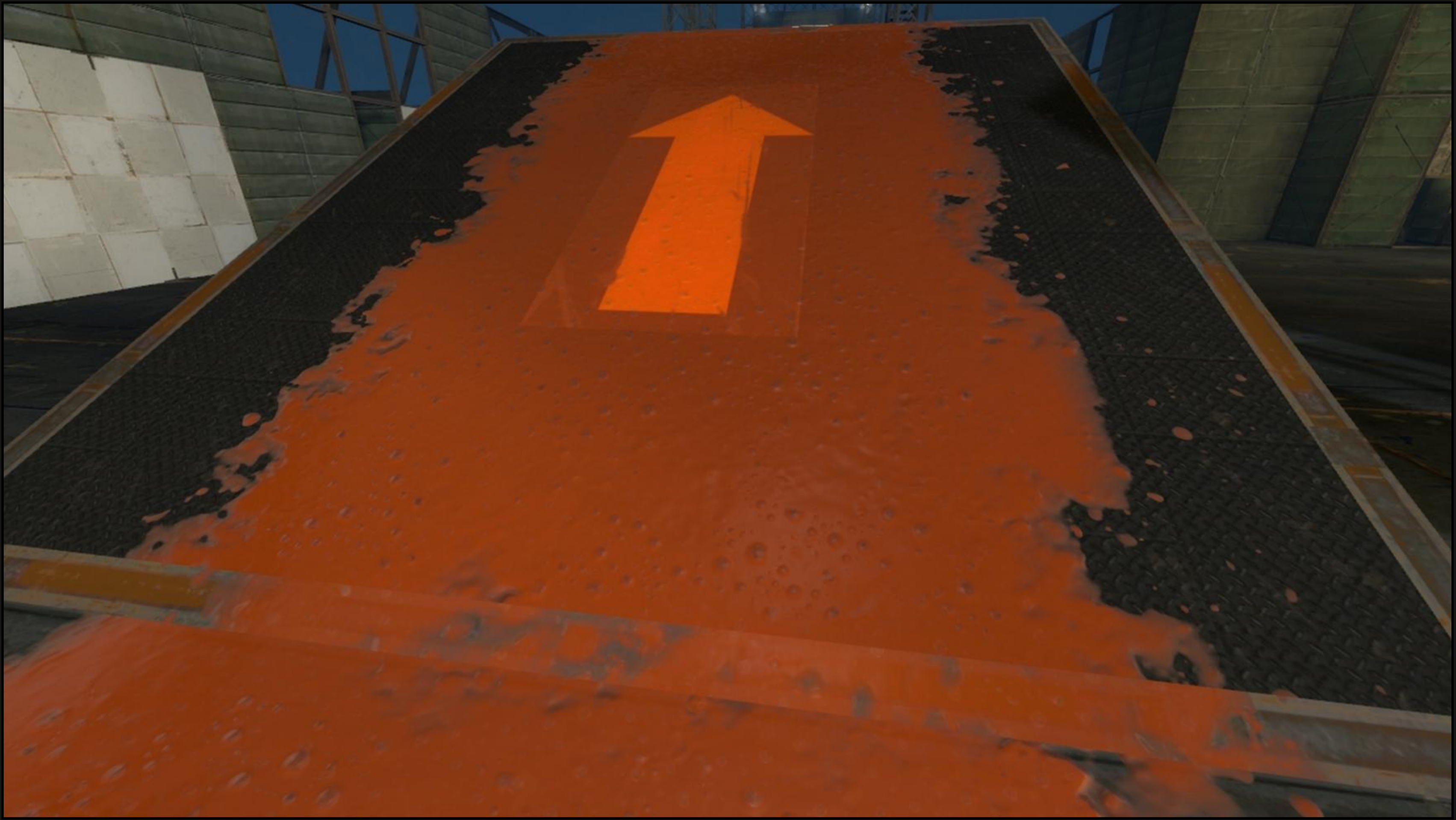
Making and Using Non-Standard Textures

Manipulating UVs through Color Data in Portal 2

Bronwen Grimes



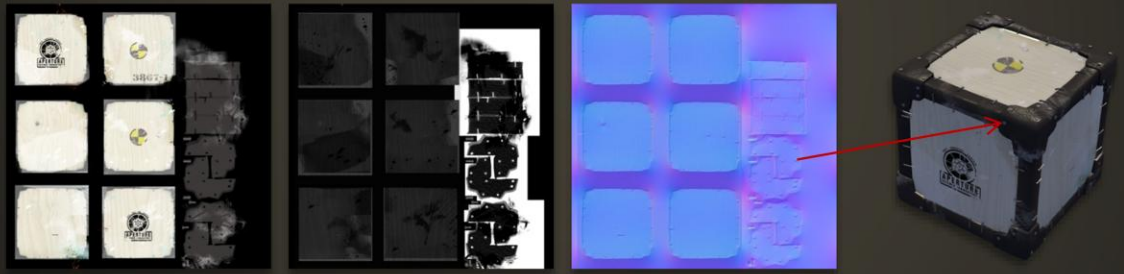
First will be a quick tour of the techniques used to render the water in Portal 2, followed by a more in-depth view on the method used to generate the flow data that is required by the shader.



Next will be a look at the gels, which are a new gameplay element in Portal 2. We'll see how to create surface-embedded sprites using a technique for making sprite-like visuals without generating any sprite card geometry. We used it to enhance the look of the gels by adding bubbles.

Beyond the “Diffuse” Map: Standard Textures

- Diffuse texture: color directly applied to the model
- Specular mask: intensity applied to a lighting effect
- Normal map: direction of surface normal modified



- In each case, the artist knows where each pixel of the texture will appear on the model: UVs are static

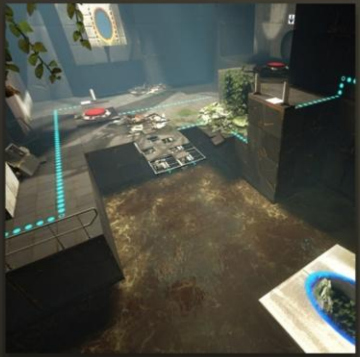
The title of this talk is “Making and Using Non-Standard Textures.”

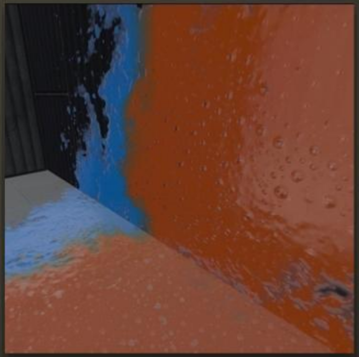
“Standard textures” refers to the types of textures that game artists are used to creating when surfacing a typical asset, like diffuse textures or normal maps.

All these textures have something in common: the artist always knows where each pixel will appear on the model.

Beyond the “Diffuse” Map: Non-static UVs

- Read UV from face verts, interpolate over face
- Modify UV in non-static way based on texture input
- Sample texture from UV position in texture space



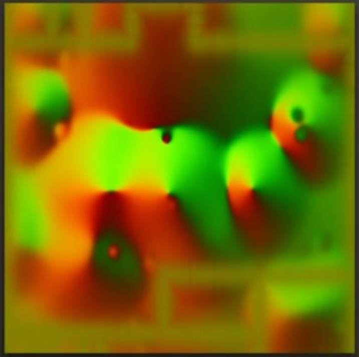


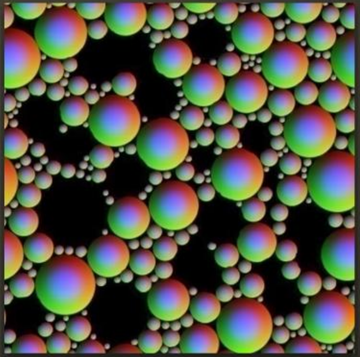
In the two techniques we’ll look at today, this isn’t the case:

The texture coordinates are modified in some non-static way, creating complex-looking surfaces on what is actually extremely simple geometry.

Beyond the “Diffuse” Map: Non-standard Textures

- Non-standard texture types used in these algorithms:
  - Flow map for water
  - UV layout map for surface-embedded sprites in gel





The non-standard textures we use to modify the UVs are as follows:

For the water, we use a flow map that represents a 2D vector field. It is used to warp the UVs to create animation.

For the gels, we use a UV layout map that describes the distribution of the embedded sprites over the surface.

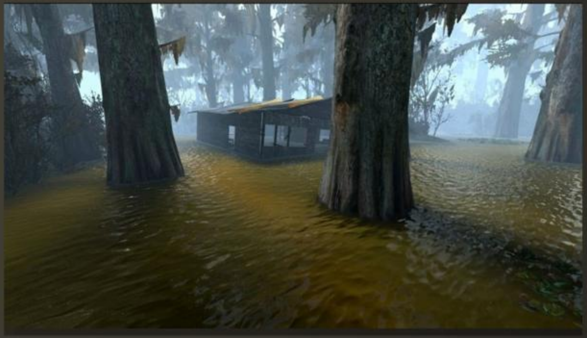
As we proceed, we will explore how these textures are used and how they are created.

Water Flow

Toxic, Roiling Liquid in Portal 2

Water in Portal 2

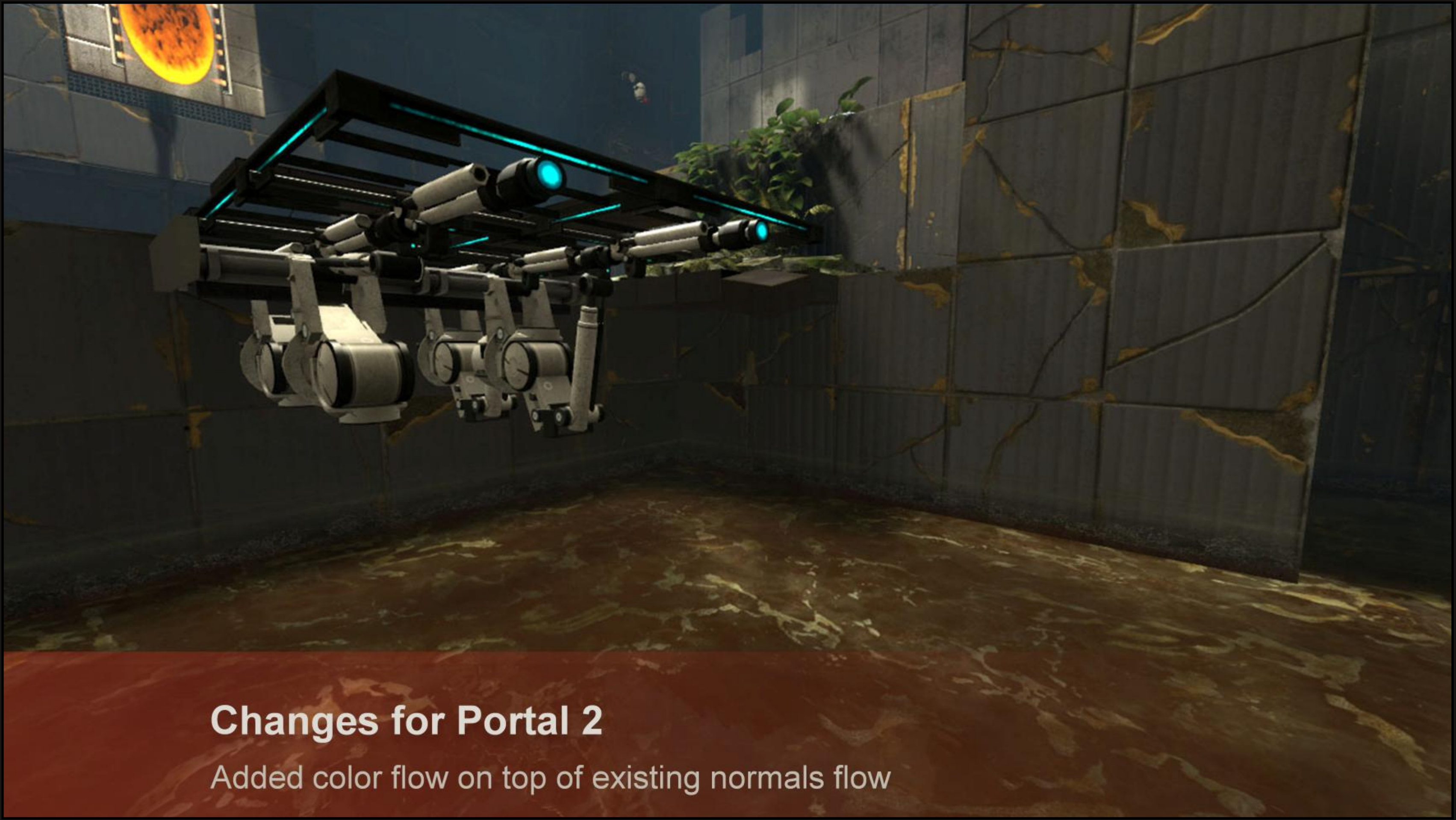
- Shader originally created for water in Left 4 Dead 2
  - Game contained an entire campaign in the bayou with its complex waterways
  - Needed the flow of the water to follow complex contours in a believable way



The water in Portal 2 is an extension of the shader developed for Left 4 Dead 2.

In that game, one of the campaigns took place in a bayou. To reach the realistic art direction of the game, the water needed to follow the complex contours of the terrain in a believable way.

We used a technique that flows a normal map over the surface to ripple the water. The shader also used a depth fog color, dynamic refractions, and either dynamic reflections or pre-baked cube maps based on the performance constraints of the area.



Changes for Portal 2

Added color flow on top of existing normals flow

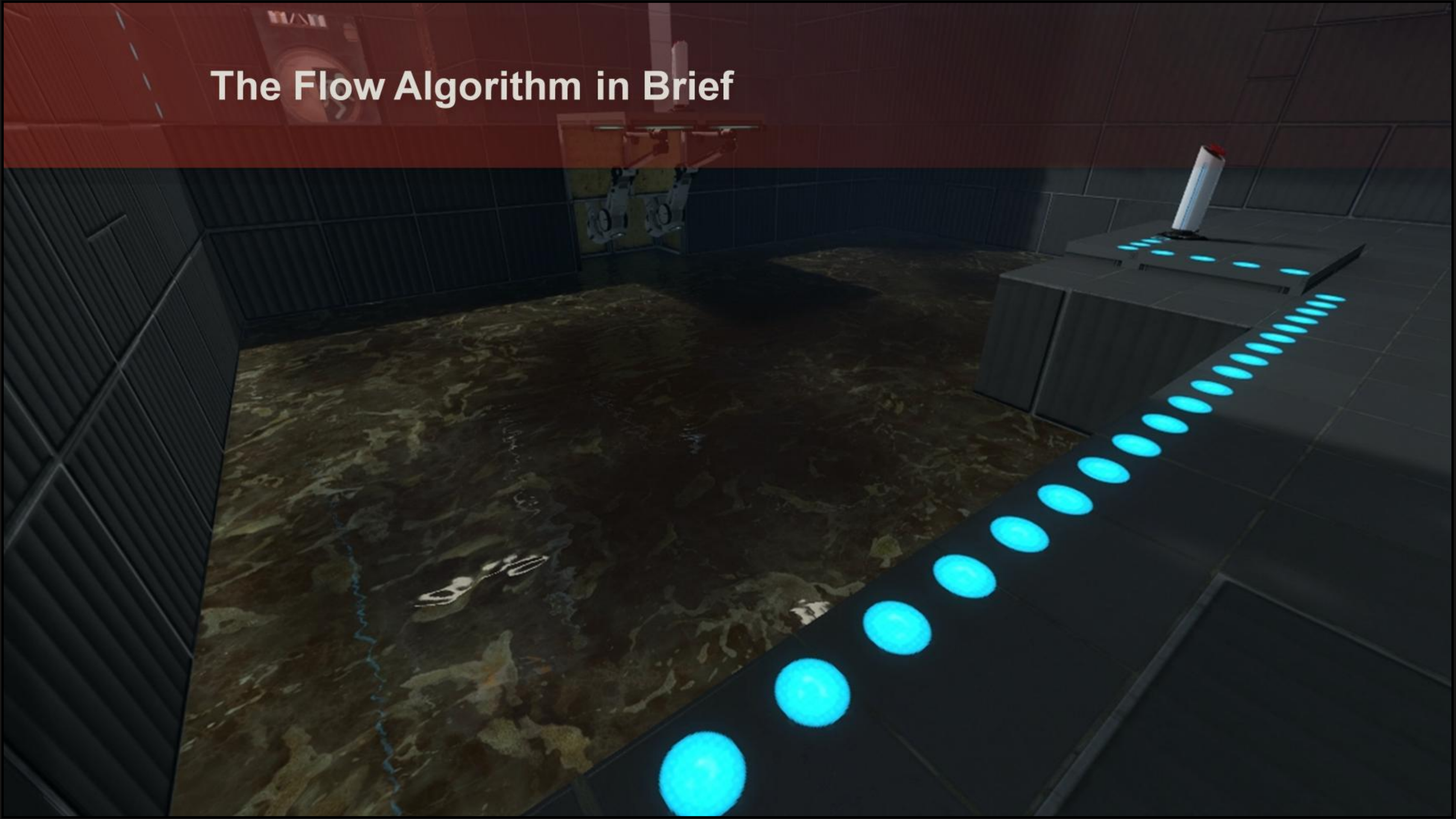
In Portal 2, the pools are rectilinear and self-contained.

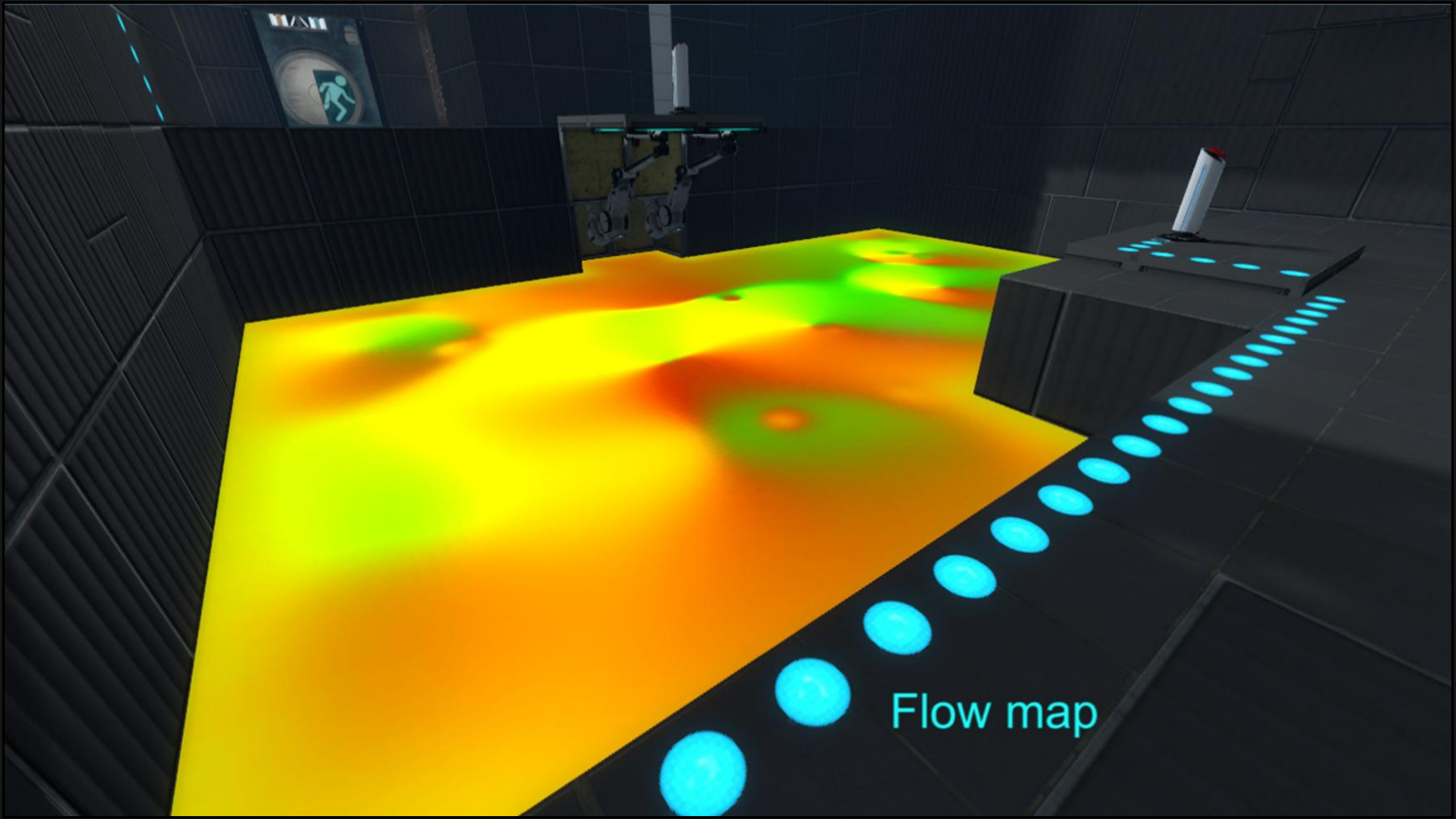
Refractions won’t solve the problem of visual interest because walls intersecting the pools are dark, simple shapes.

The depth fogging also wasn’t very helpful, since there is very little geometry that intersects the water.

We needed to add color maps to get the density of visual interest required.

The Flow Algorithm in Brief

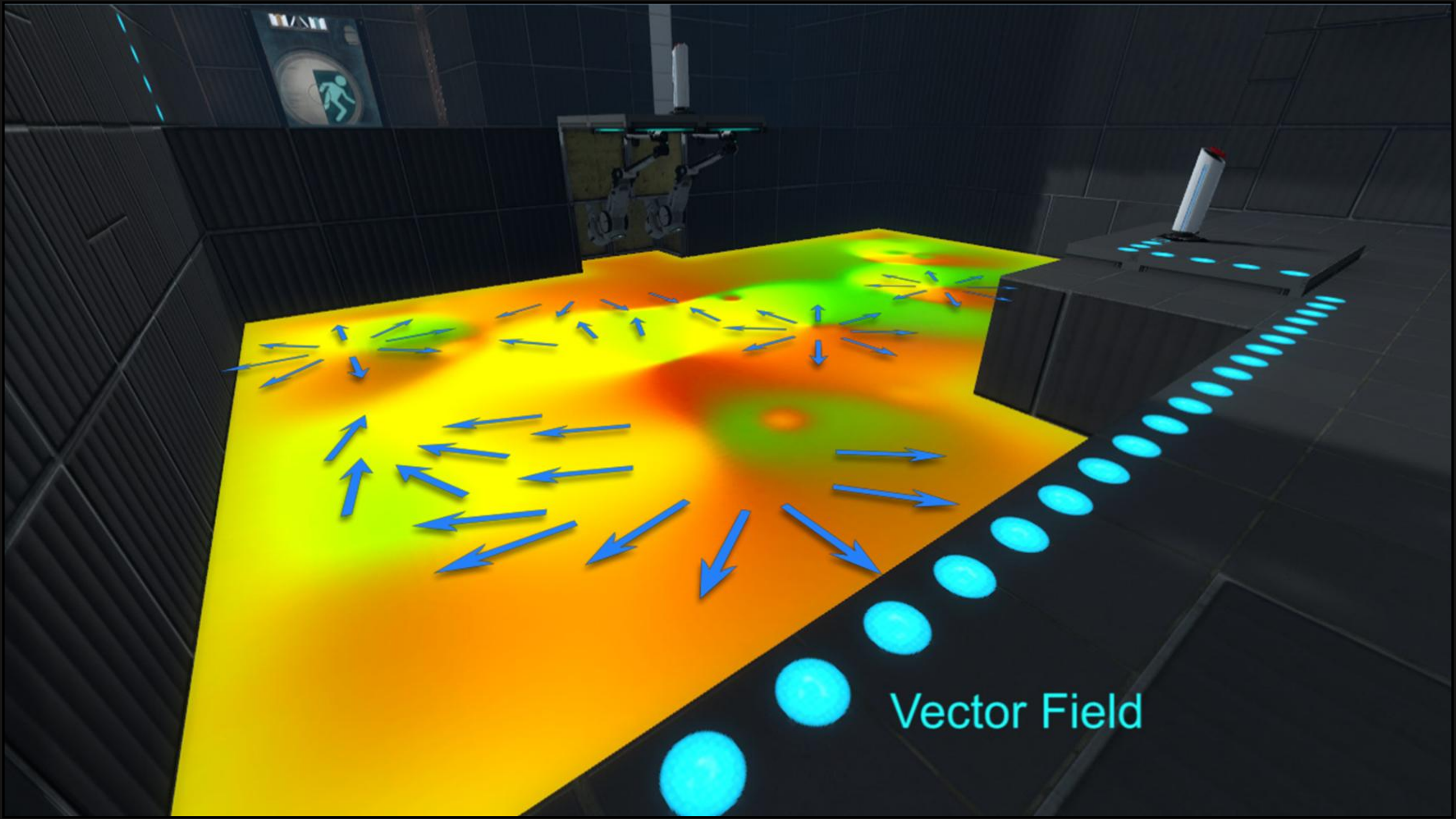




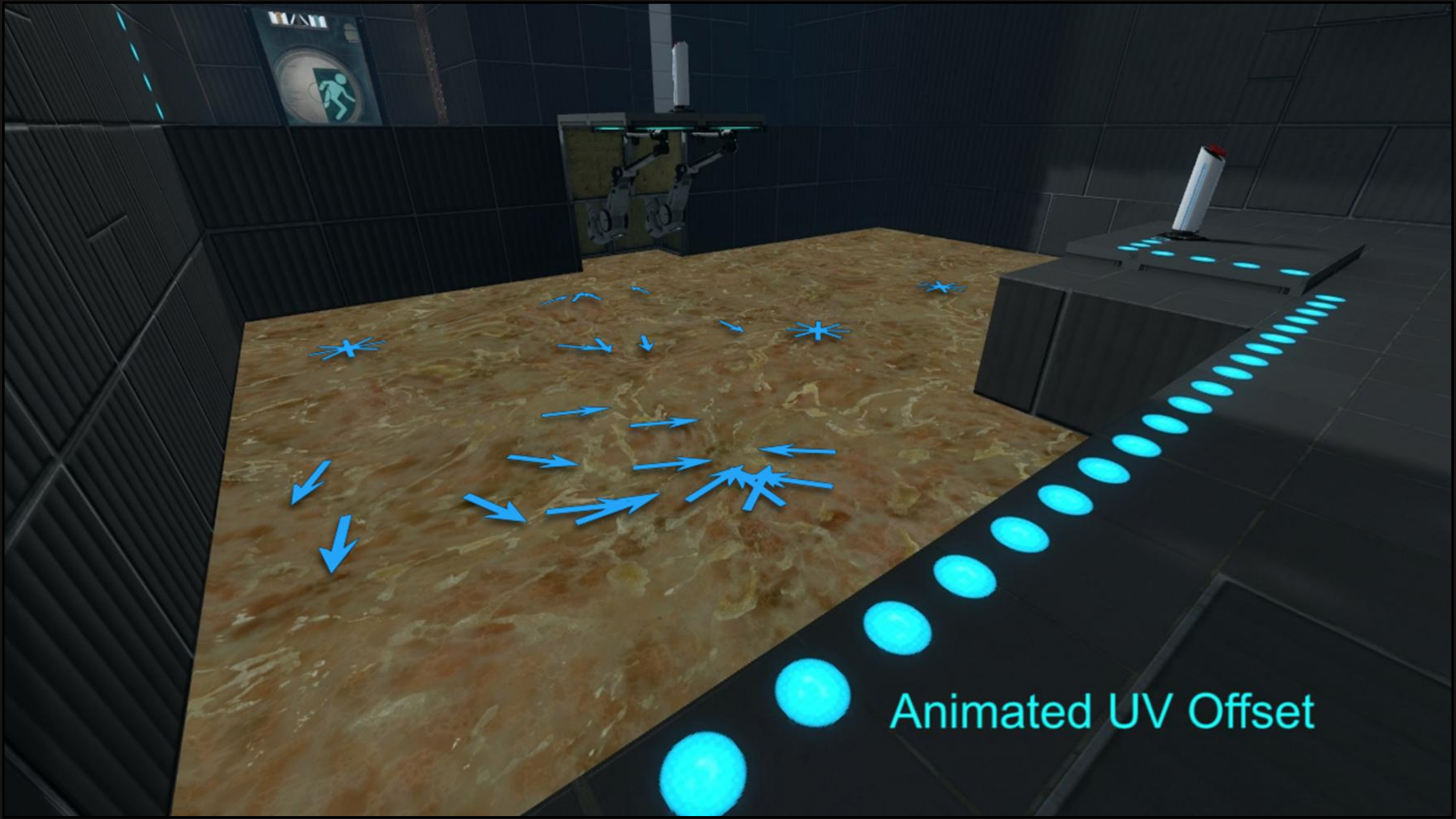
Here is the flow map as it is applied over the surface.

It represents a 2D vector field, indicating to the shader how the UVs should warp.

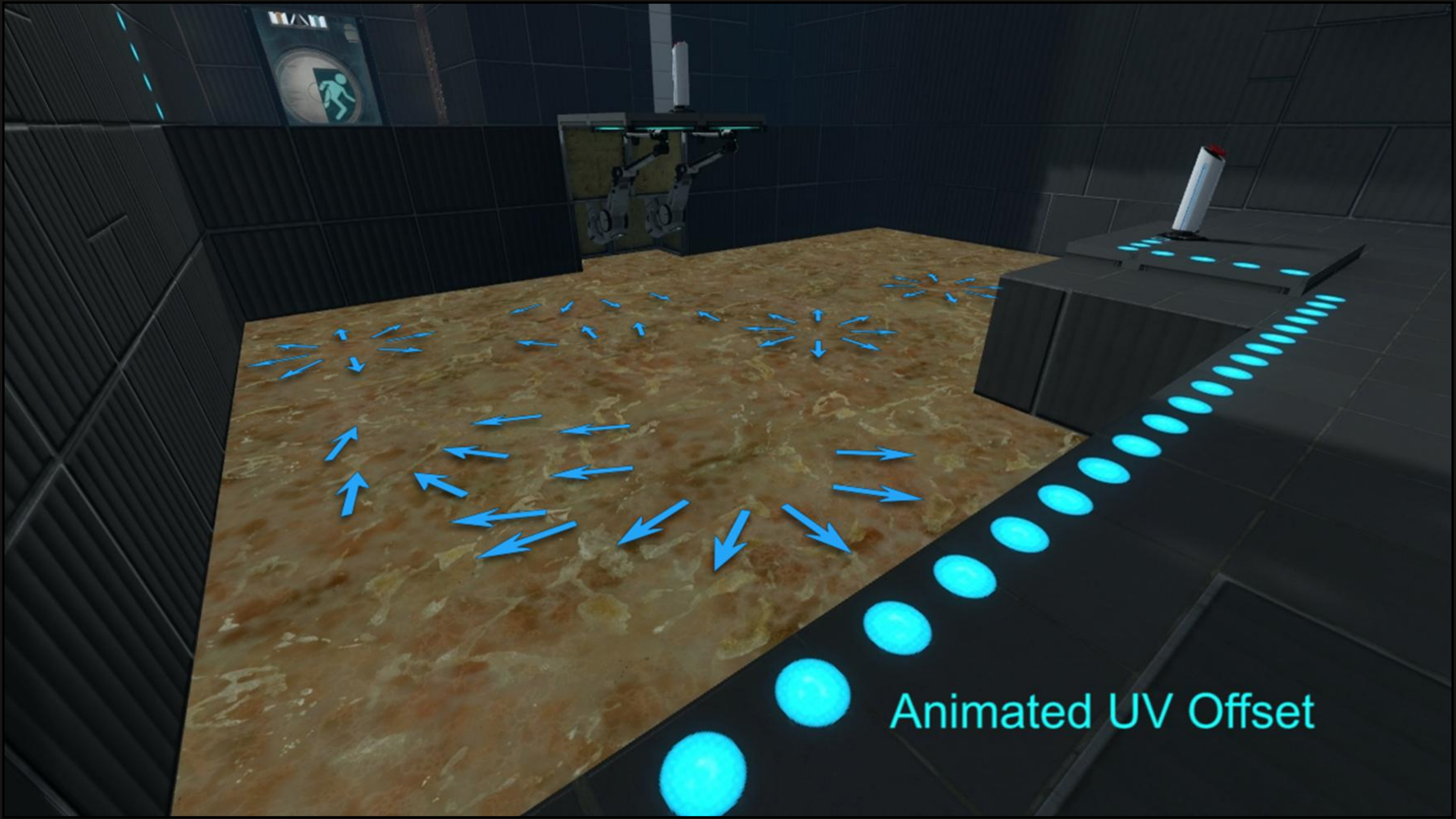
The human brain has a hard time interpreting color as direction and vice versa, so…



This is how the shader interprets the data.



The animation is accomplished by scaling the contribution of the flow map over time, from negative…

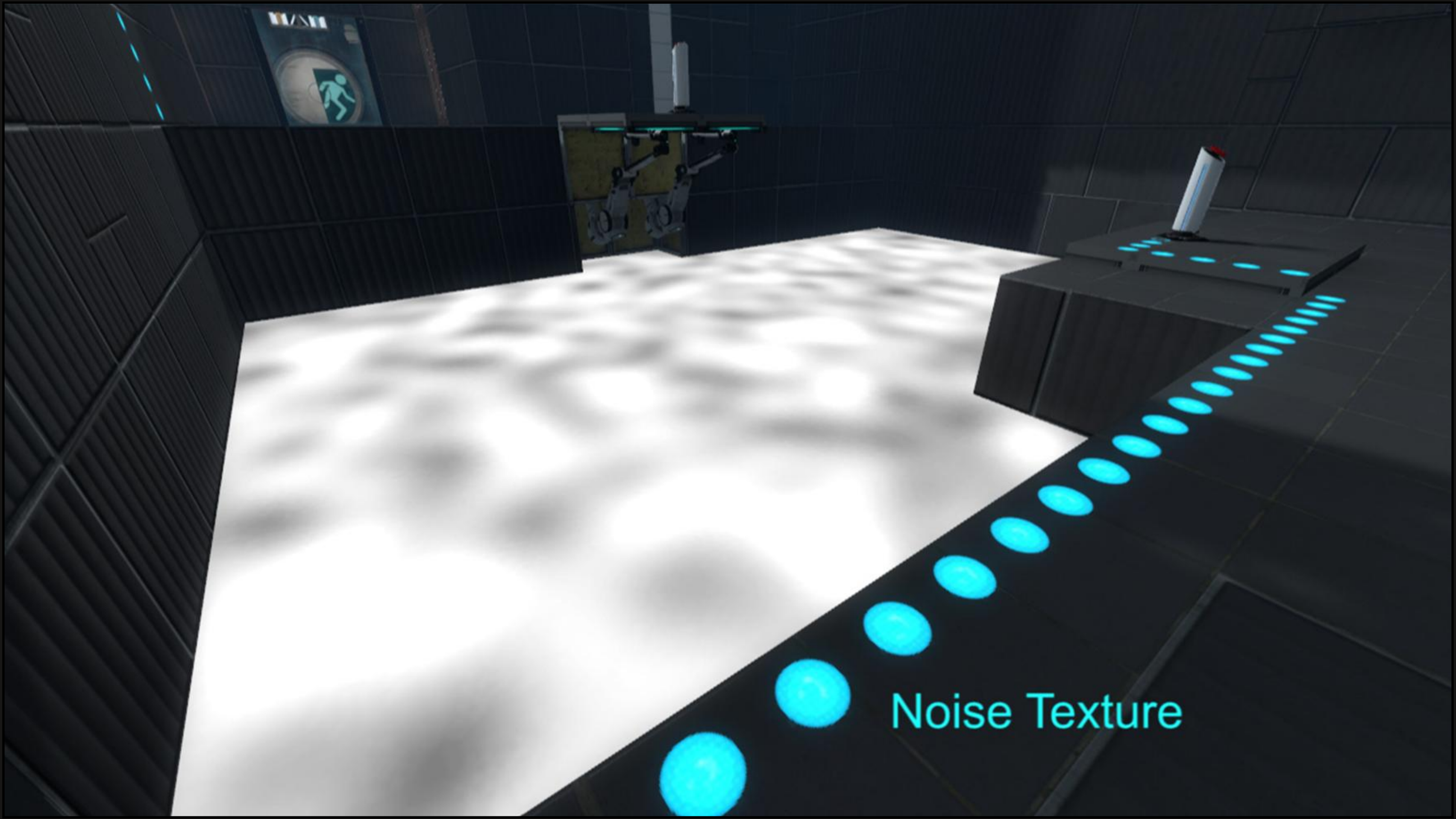


…to positive.

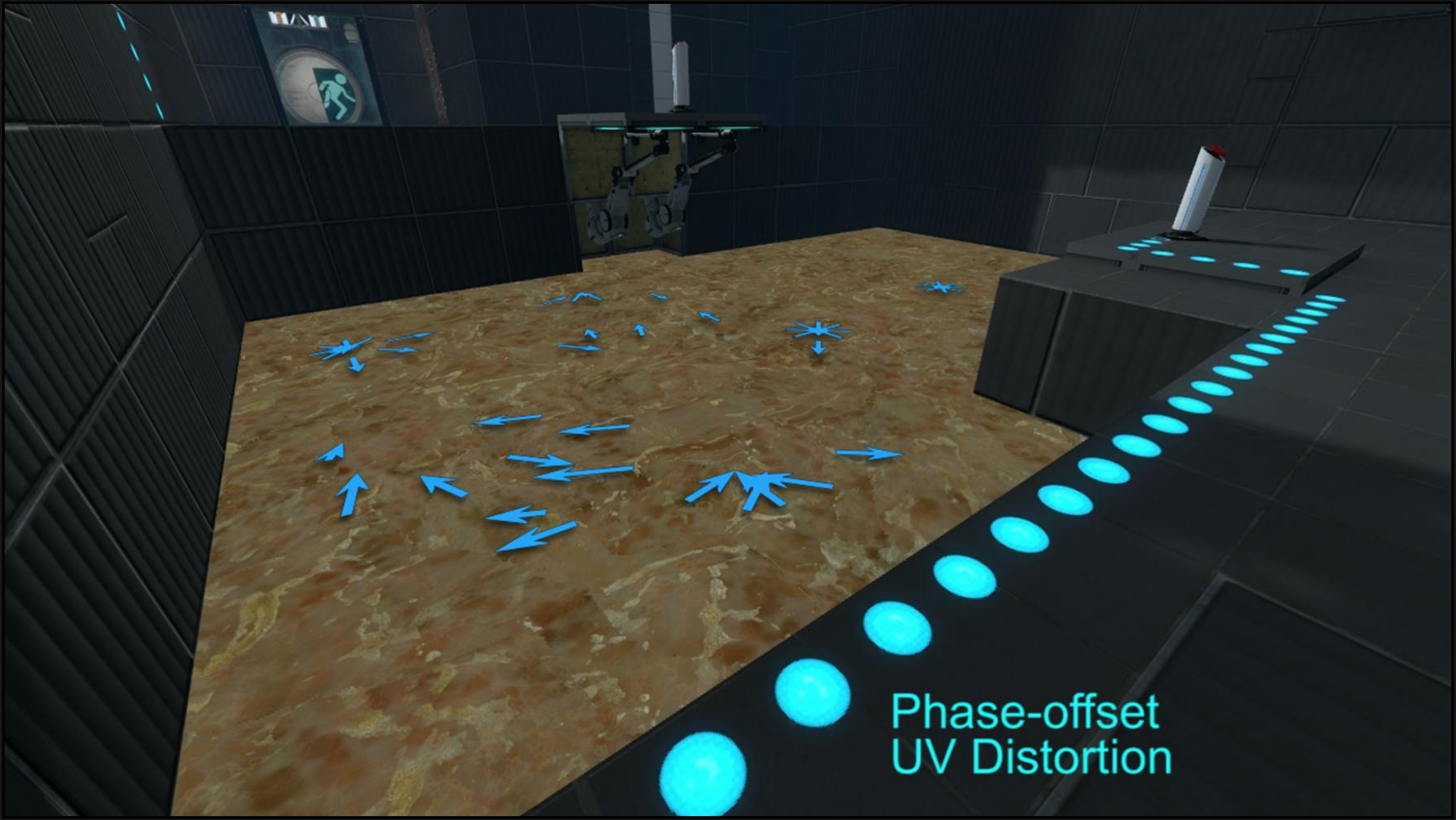
We can only warp the UVs so much before the color map is distorted beyond recognition, however.

After a certain amount of warping is applied, we have to start over.

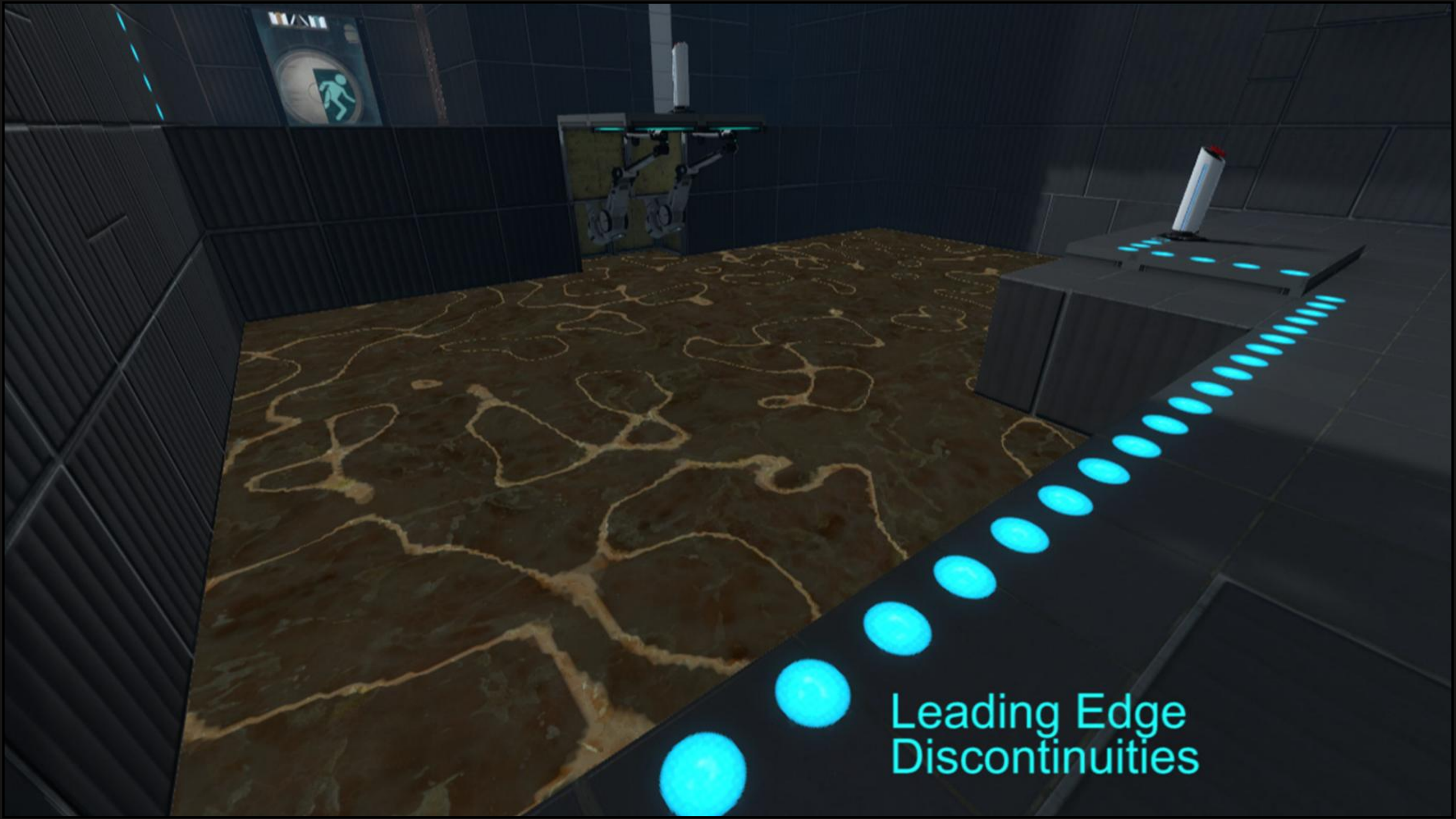
This results in a visual “pop” as the entire surface resets.



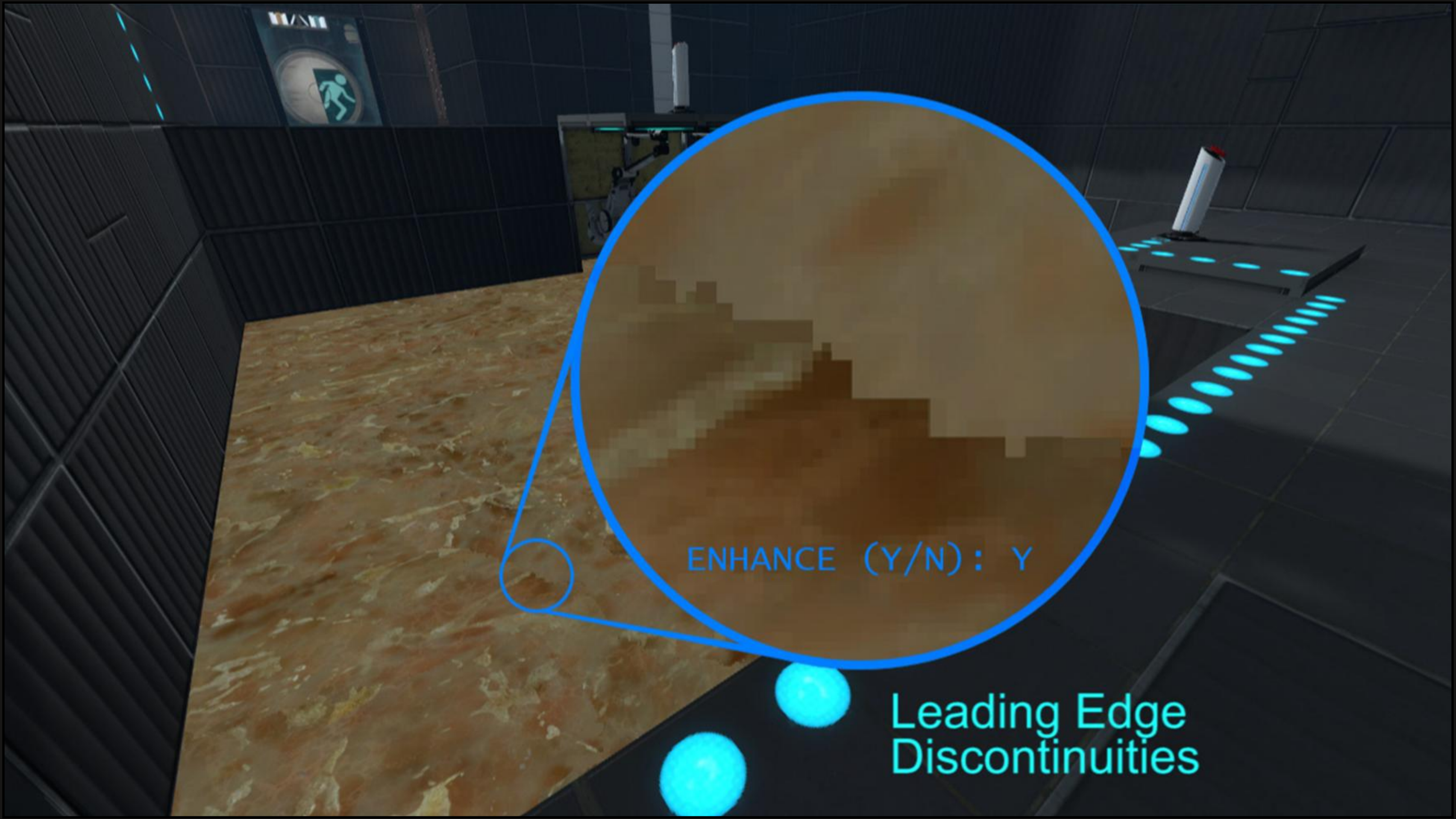
We use a noise map as a time input to offset the phase of the animation per pixel.



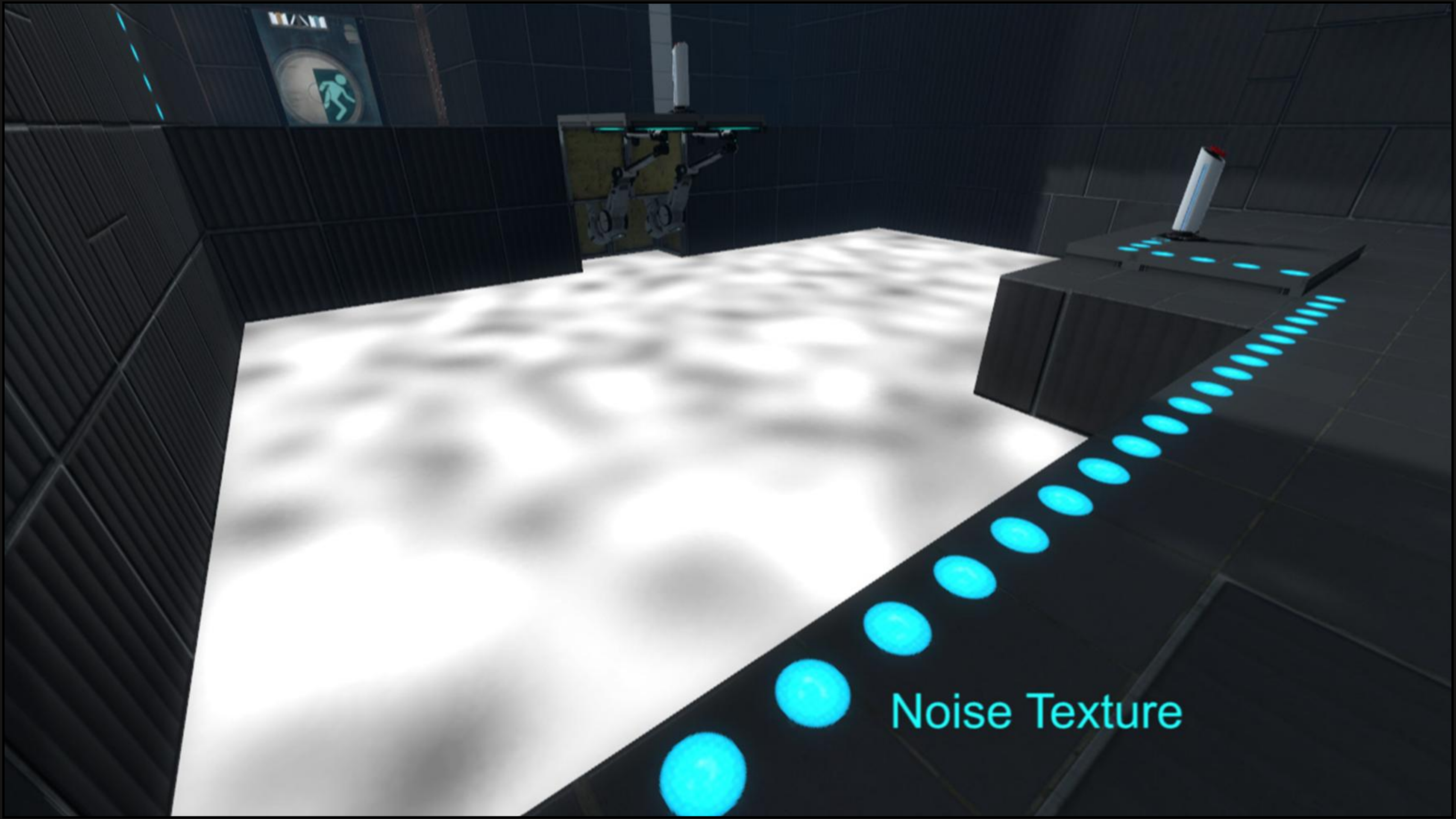
As a result, the phase doesn't all change at once, which is much less jarring.



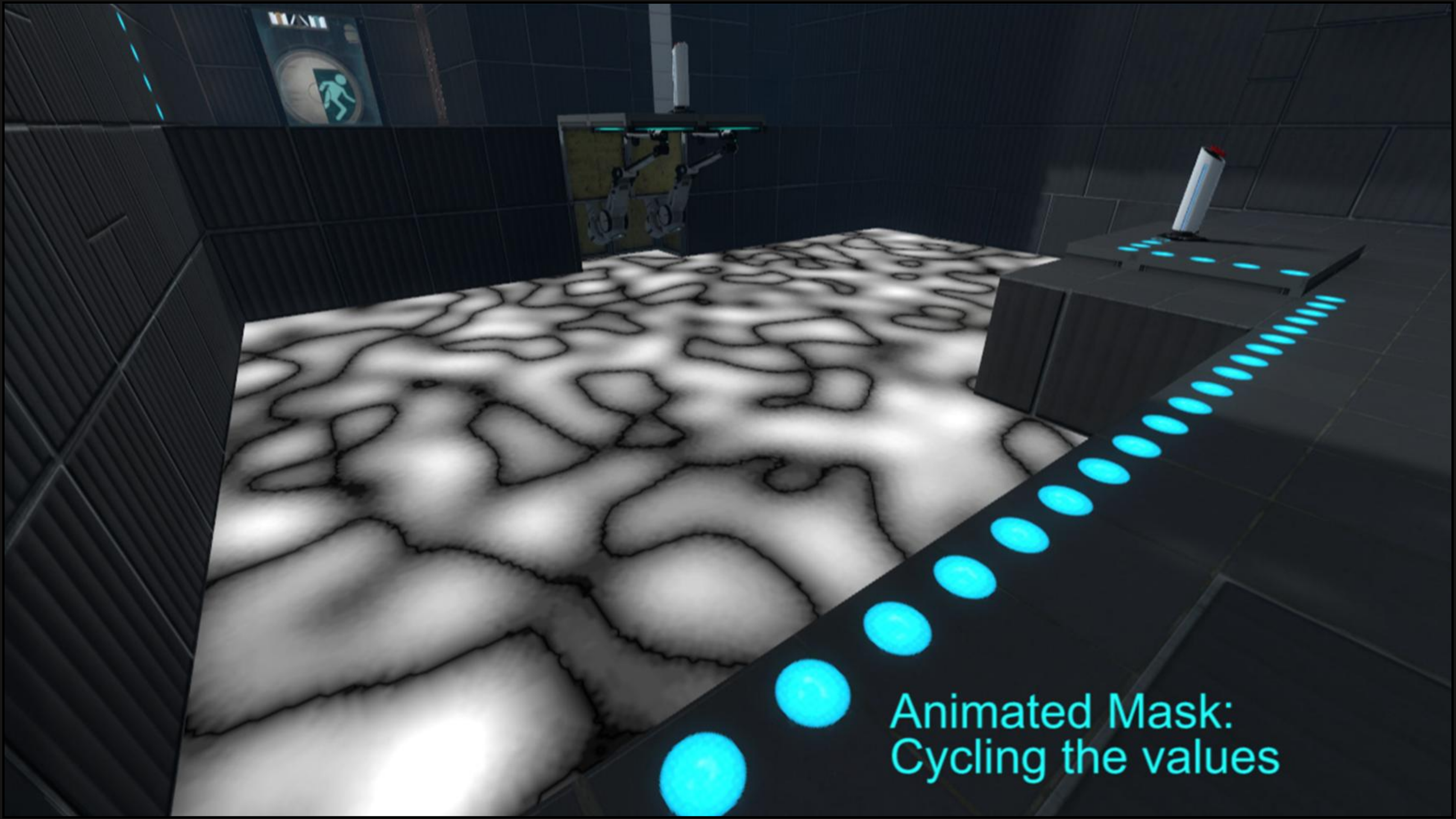
We do end up with these leading-edge discontinuities, though, at the border of the phase change.
(The edges are highlighted using an overlay to show the locations of the discontinuities.)



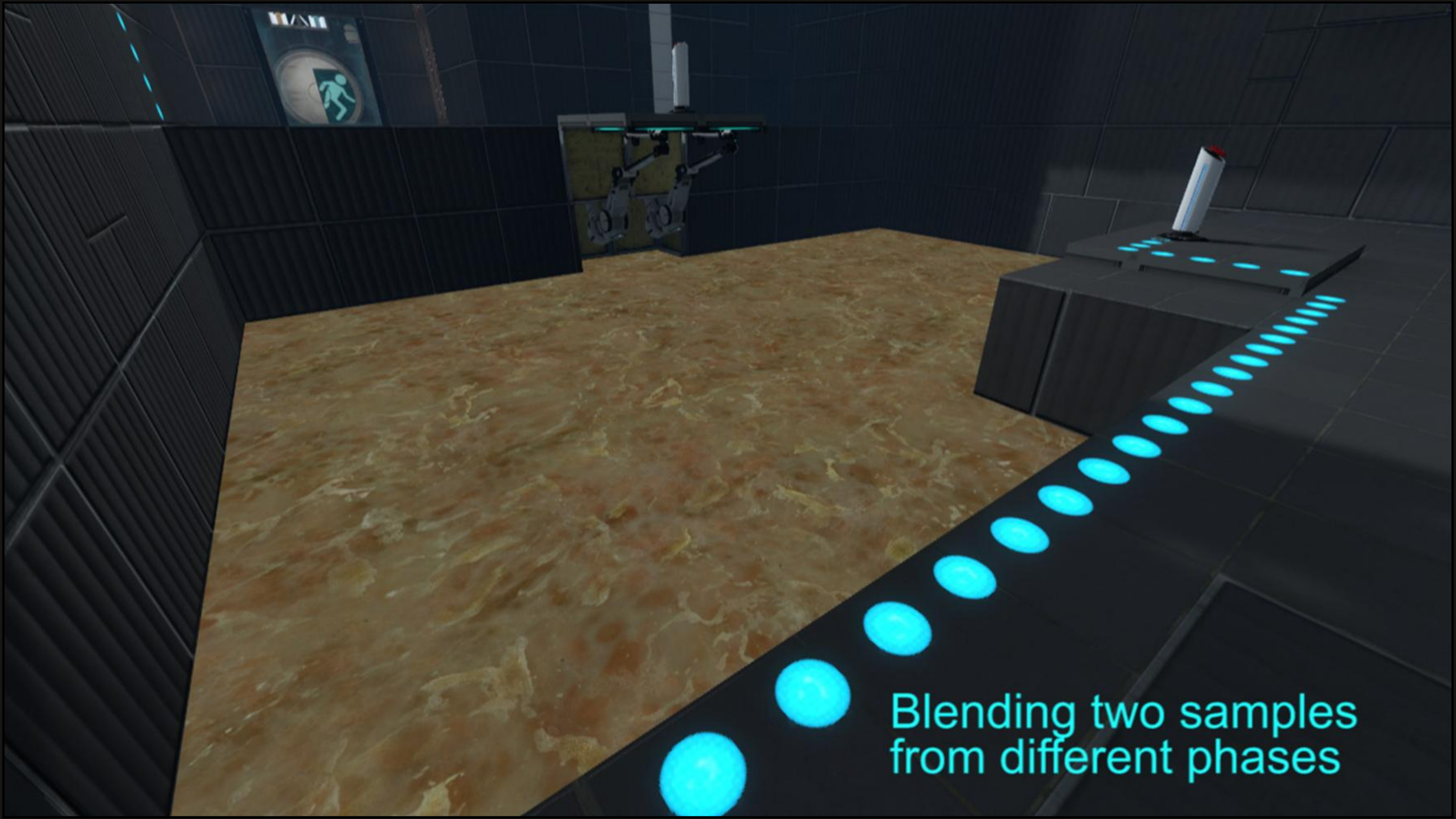
He is a closer look at the border of the phase change, enhanced for clarity.
It crawls visibly over the surface.



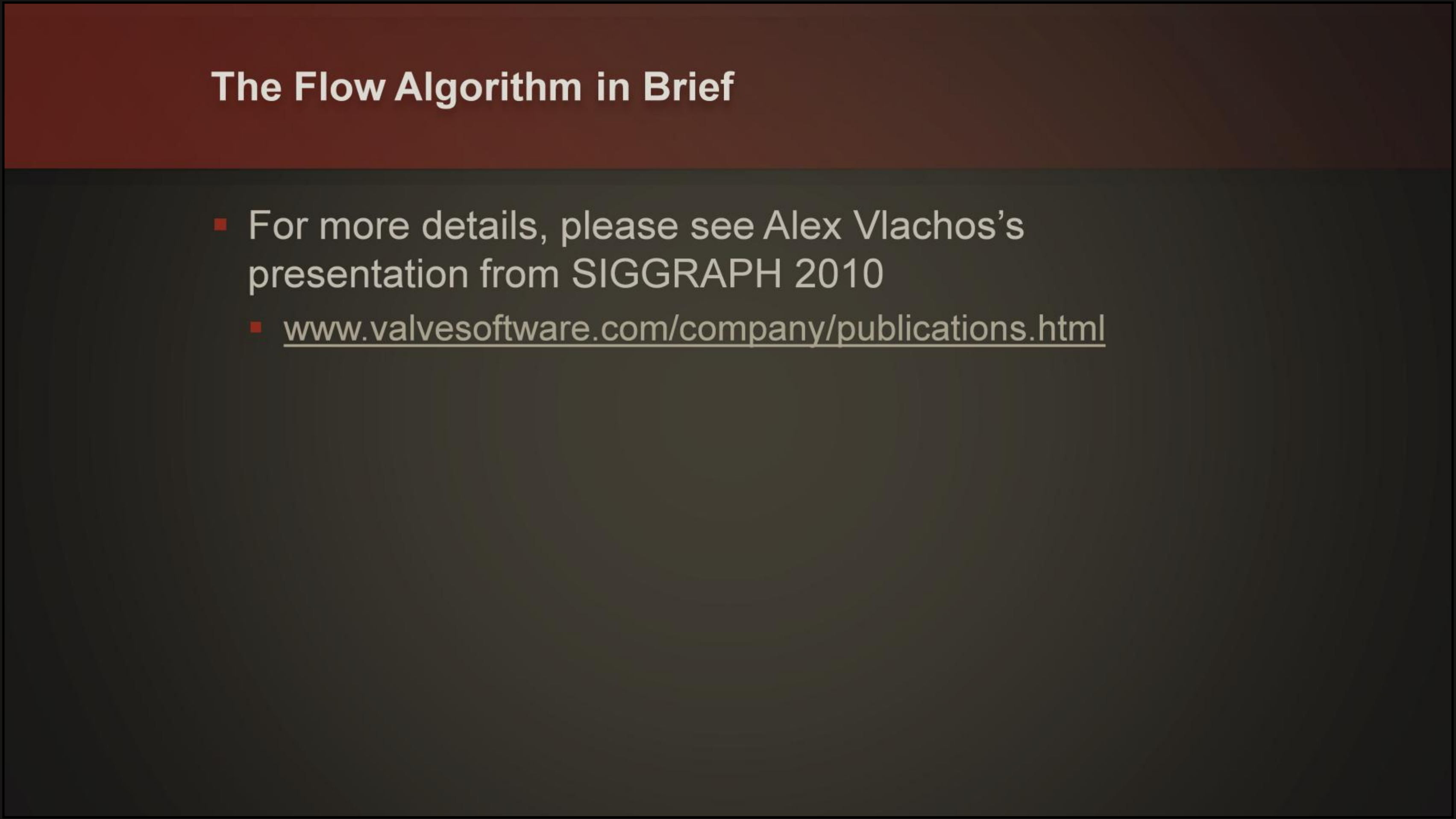
The noise texture is useful for more than offsetting the phase.



We can cycle the values in the noise map at the same rate that the phase change crawls over the surface, resulting in an animated, soft-edged mask that can be used to hide the discontinuities.



We sample the color twice at opposite phases, and use the cycled noise values to blend the two phases together, resulting in unbroken color that can animate indefinitely.

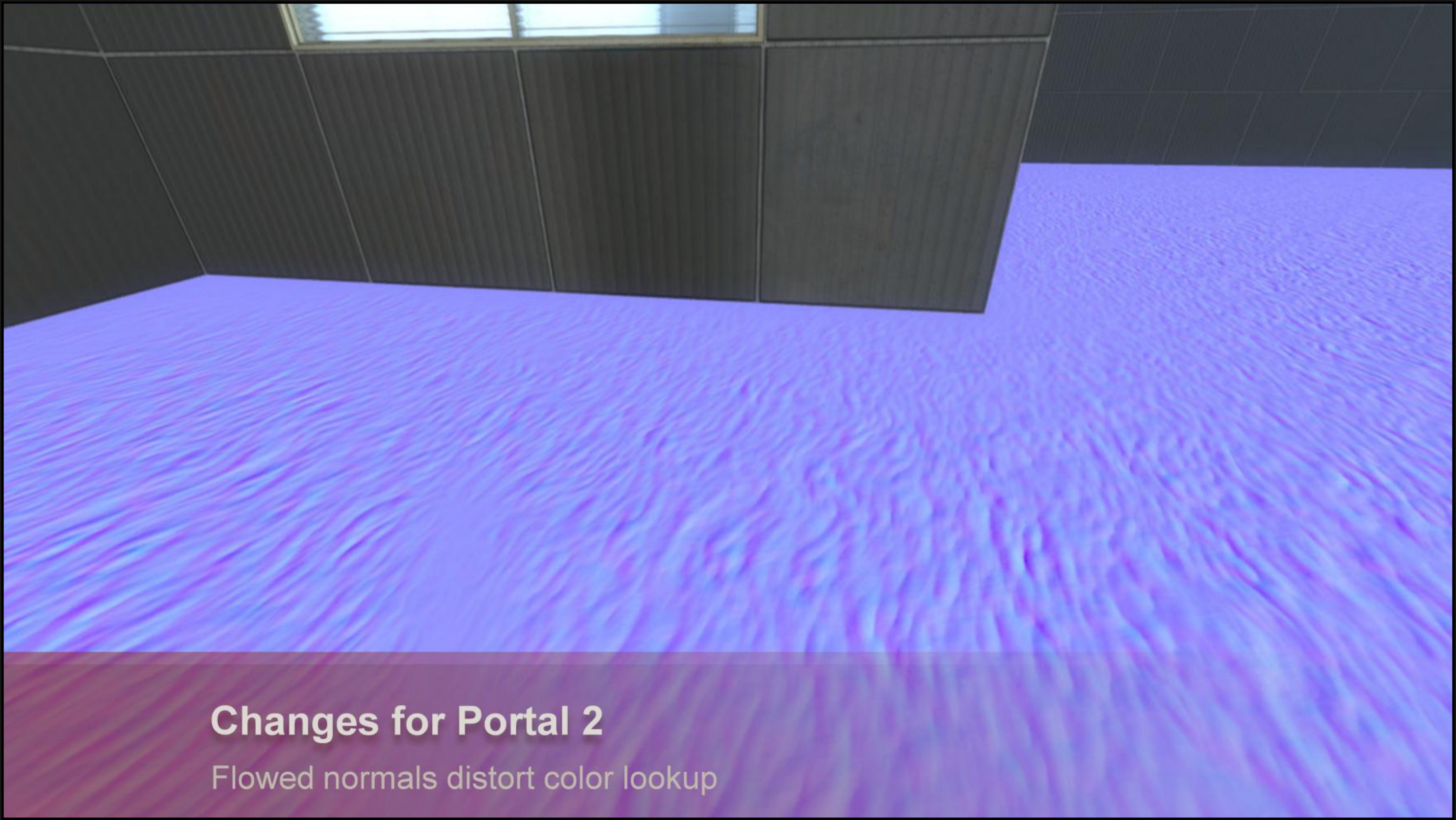


The Flow Algorithm in Brief

- For more details, please see Alex Vlachos's presentation from SIGGRAPH 2010
  - www.valvesoftware.com/company/publications.html

For deeper implementation details, please refer to Alex Vlachos's presentation from SIGGRAPH 2010. You can download the slides from our website at http://www.valvesoftware.com/publications/2010/siggraph2010_vlachos_waterflow.pdf

As development progressed on Portal 2 and the art source changed, we found ourselves with two new problems we had to solve.

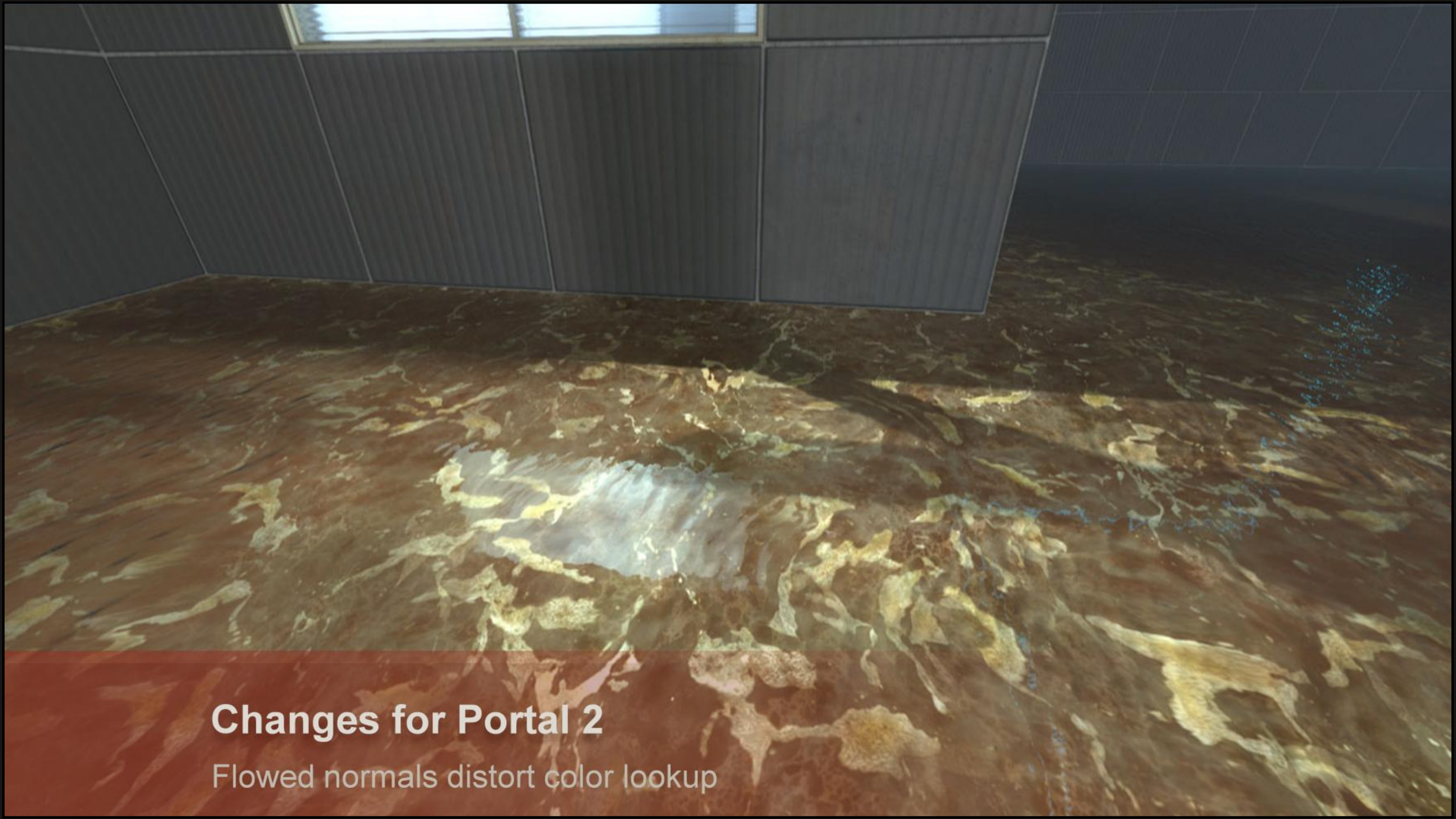


Changes for Portal 2

Flowed normals distort color lookup

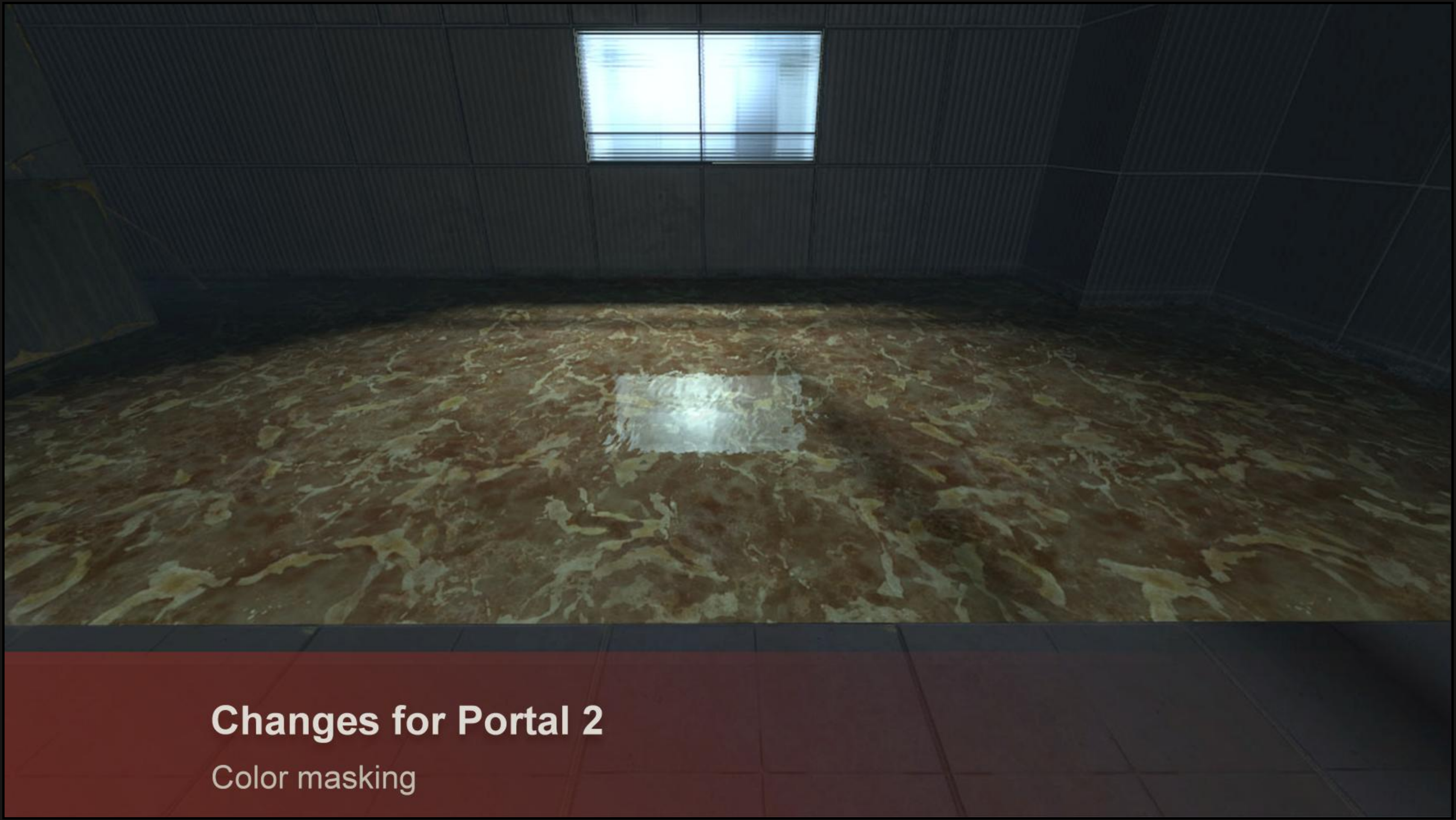
The foam patches were, at first, animated solely using the flow technique. The low-frequency motion that resulted was a poor match to the high-frequency ripples in the normal map. This was further compounded by the different speeds in the animation of the ripples and the color flow.

To resolve the two, we first compute the normal flow, then use the results to further distort the lookup into the color texture.


Changes for Portal 2

Flowed normals distort color lookup

The result is foam patches that ripple with the normal map, appearing to sit on the surface of the water instead of gliding beneath it.



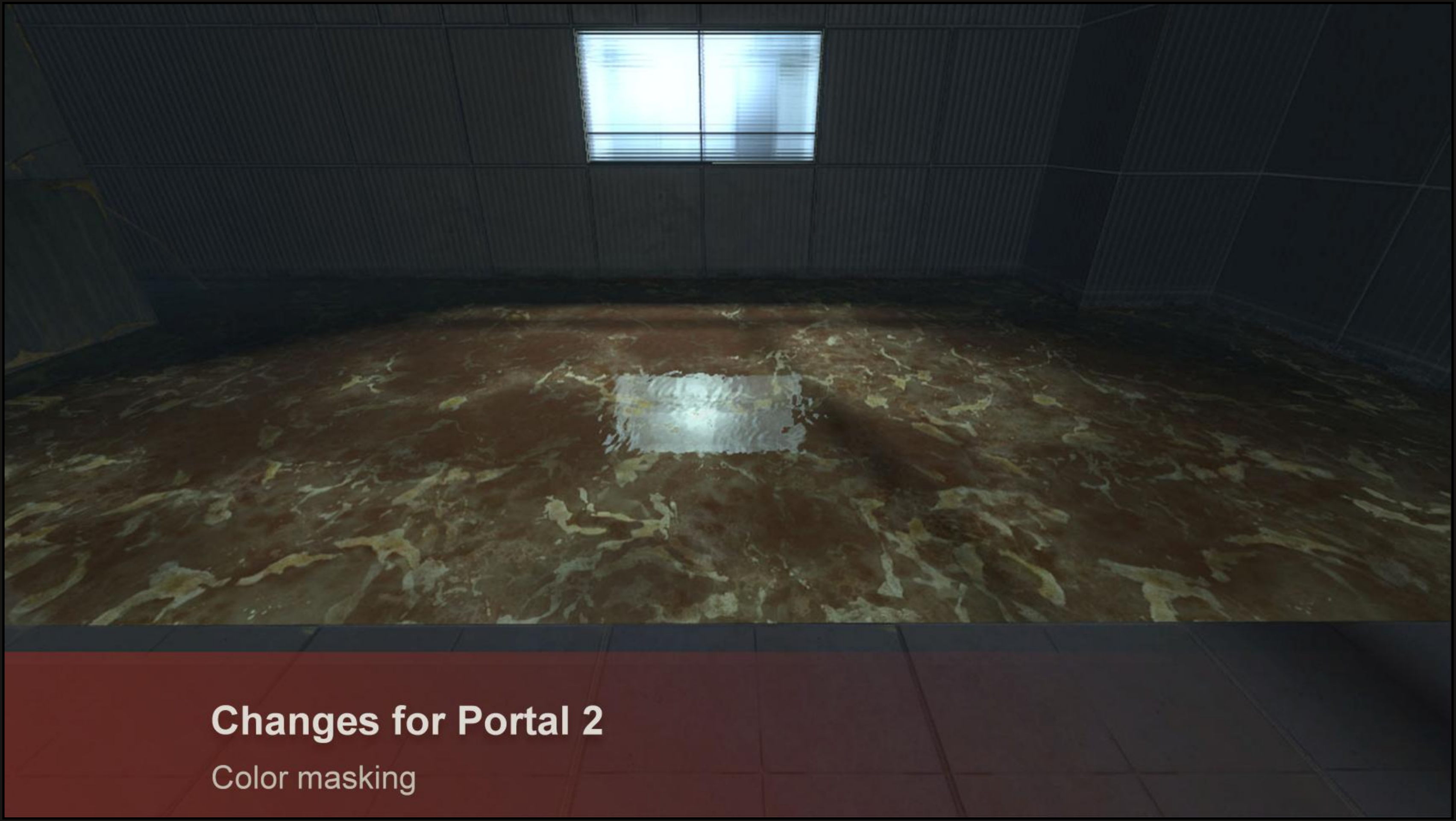
Changes for Portal 2

Color masking

There are several levels in which the water covers the majority of the ground.

We found that the color map, though full of medium-scale detail, became monotonous when applied to very large surfaces.

On top of that, it made little logical sense for the foam to be of uniform density. The currents would cause it to disperse in some areas and collect in others.



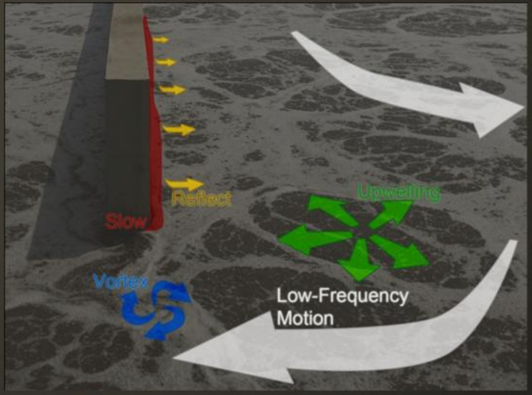
Changes for Portal 2

Color masking

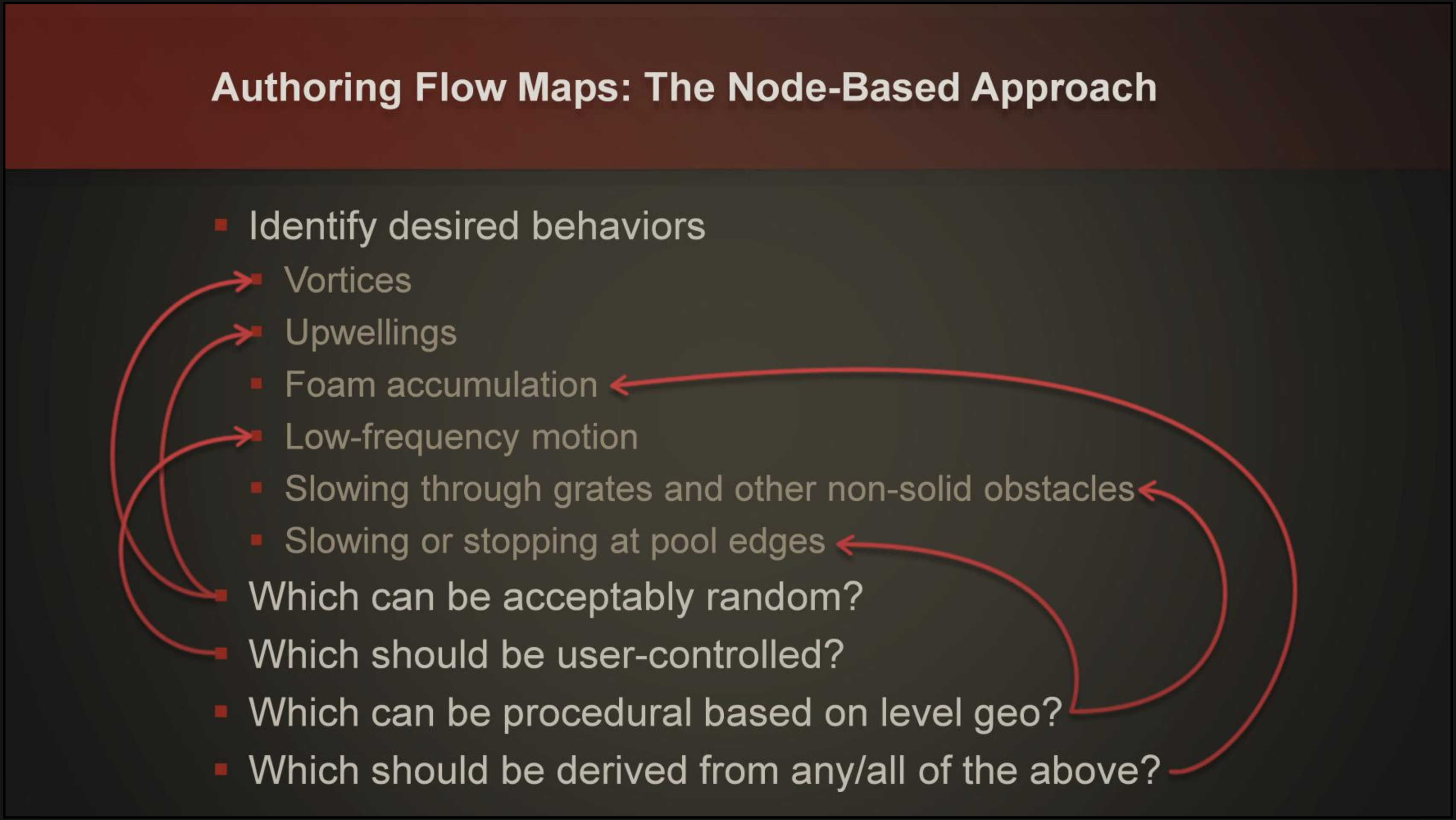
We modulate the contribution of the color map based on a texture input to break the monotony.

Authoring Flow Maps

- Painting a flow map like trying to paint a normal map
- ~~We used water simulation~~
  - No, actually, that wouldn't have worked for our timeline!
- Analysis and Abstraction: Pick key features



Flow maps encode vector information much the same way a normal map does, and are difficult to paint for the same reasons.
We needed a way to procedurally generate the flow vectors, so we turned to fluid simulation. (No, we didn't.)
Simulations can take a lot of time to generate if you need very specific results. Our approach is much more predictable, editable, and immediate.
We analyzed our reference to simplify it, picking out key features.
We will recreate these features independently then composite the results together.
We looked at sewage treatment, which keeps a constant circulation going during aeration.
The mixing process creates upwellings and vortices as water is forced up to the surface.
Around barriers and walls, the current slows and waves reflect back into the open areas of the pool.
Low-frequency motion sweeps the water around the pool in a stable configuration.



Authoring Flow Maps: The Node-Based Approach

- Identify desired behaviors
  - Vortices
  - Upwellings
  - Foam accumulation
  - Low-frequency motion
  - Slowing through grates and other non-solid obstacles
  - Slowing or stopping at pool edges
- Which can be acceptably random?
- Which should be user-controlled?
- Which can be procedural based on level geo?
- Which should be derived from any/all of the above?

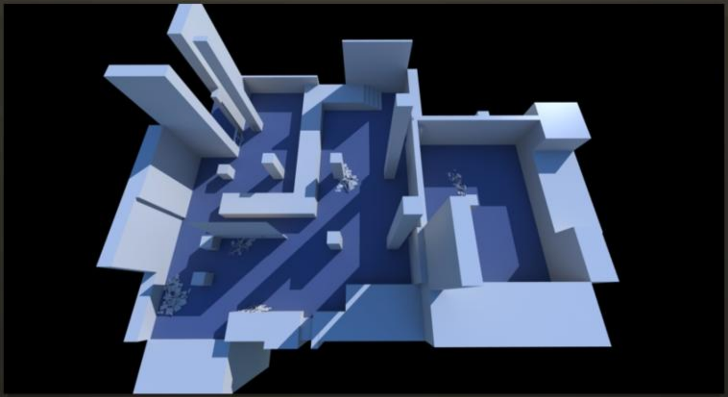
To recreate these behaviours, we needed to decide how they could be generated.

Choosing a Tool

- Workflow needs to be tailored:
  - Must be able to create templates
  - Allow procedural modification
  - Also take artist input directly
- Houdini our choice
  - Node-based geometry editor

Preparing the Source

- Export level geometry as OBJs for import into Houdini
  - We used Maya as an intermediary, since we have a plug-in that allows it to read our map format
- Keep water surface separate



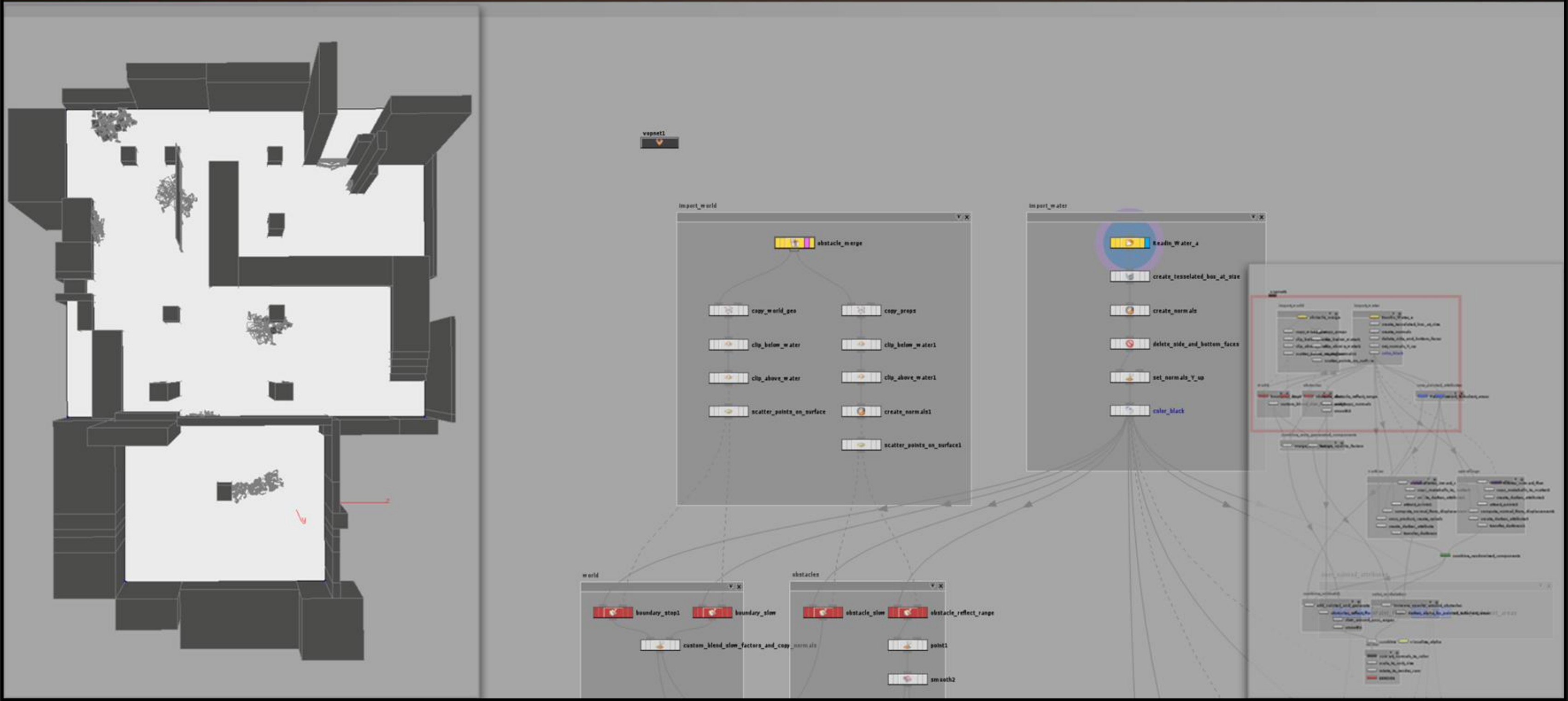
We must take our level geometry and import it into Houdini.

We used Maya to create .OBJ files, since we are lucky enough to have a plug-in that allows Maya to read our map format.

The water surface is exported as a separate object to make it easy to identify in Houdini.

The level geometry is mostly untouched: we do delete areas that don’t affect the water for performance’s sake in Houdini.

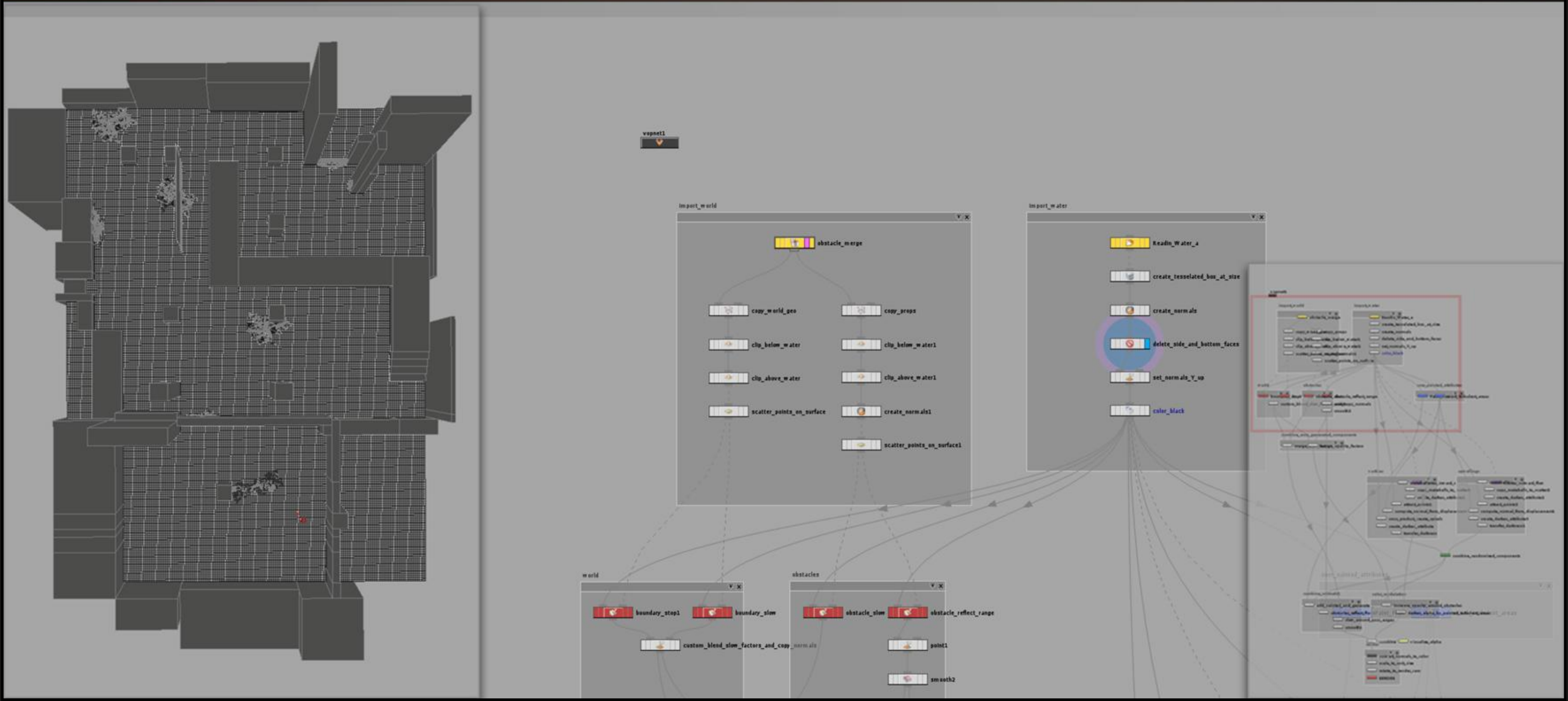
Creating the Vector Field



From here I'll illustrate the steps I outlined with screenshots from Houdini. On the right is the overall graph, which corresponds roughly to the earlier diagram. The layout goes downstream south: up at the top are the geometry inputs, and down at the bottom is the final result.

Our first step is to import our source.

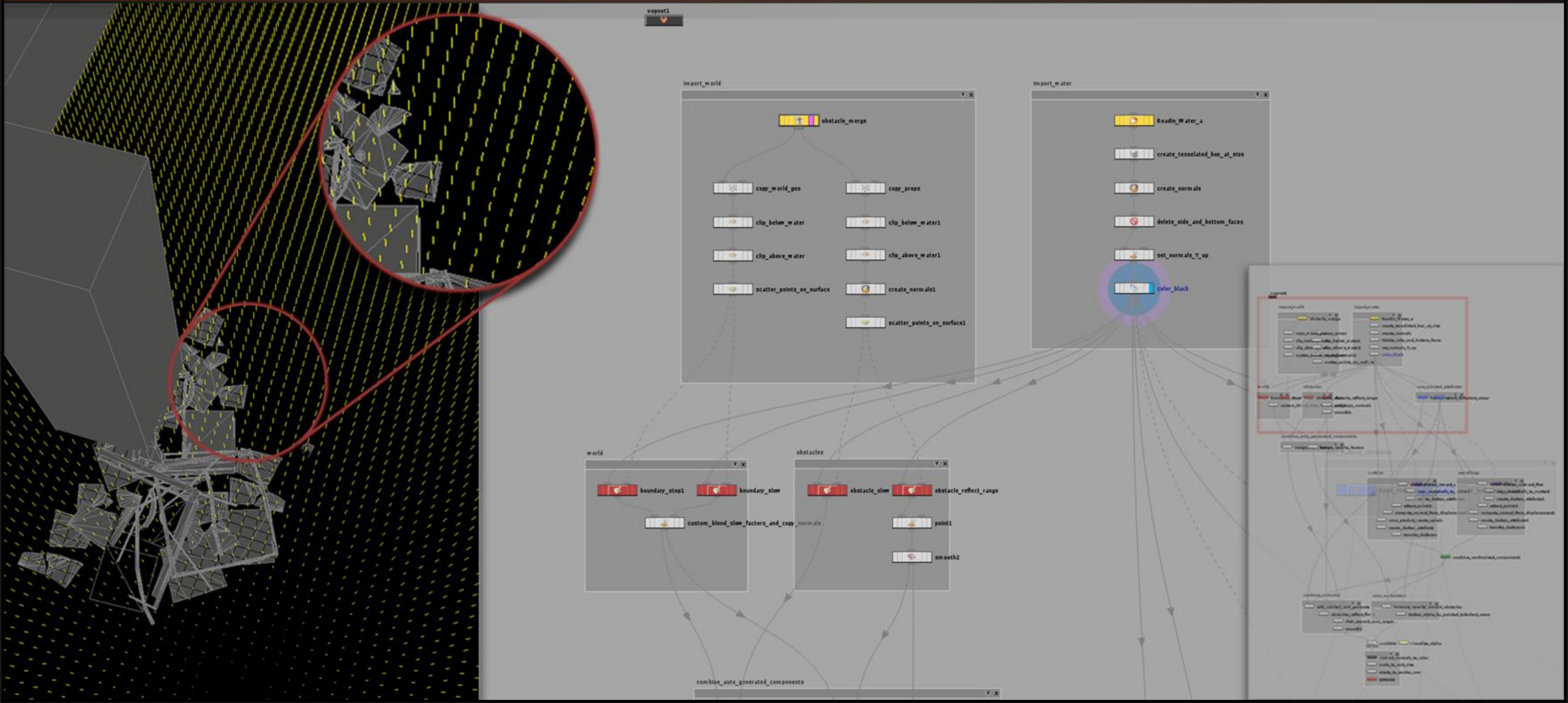
Creating the Vector Field



We immediately replace the water with a highly tesselated plane.

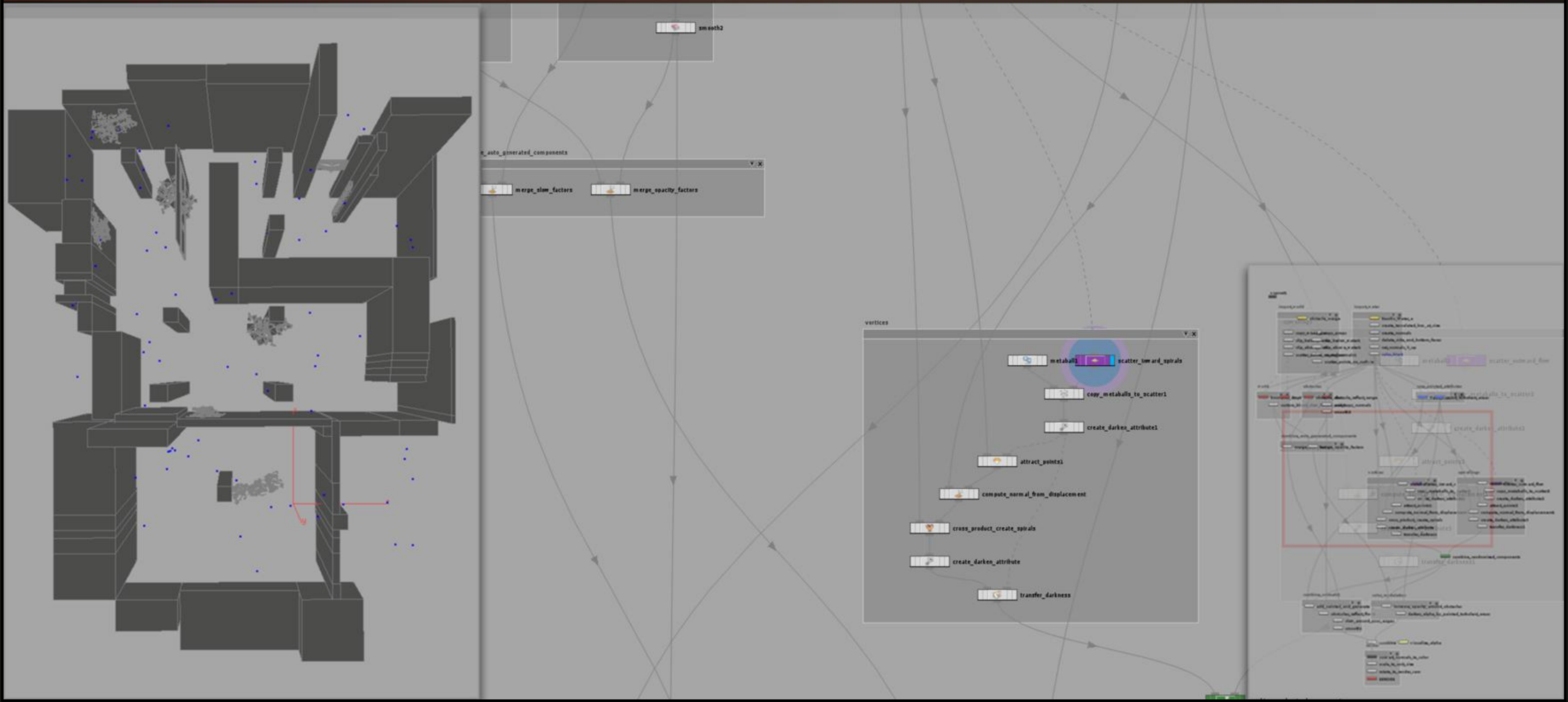
We do this because we need to generate a 2D vector field.

Vortices and Upwellings



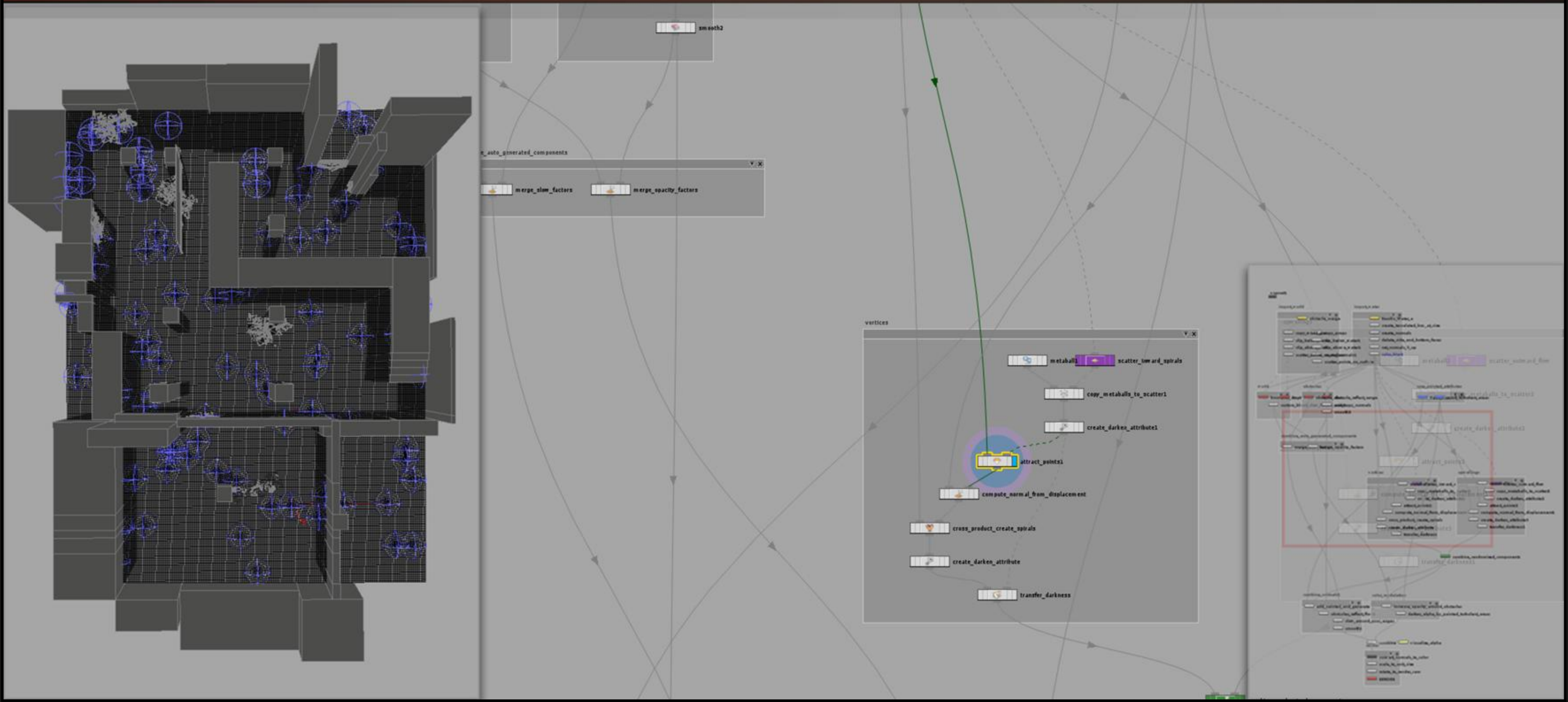
A tesselated plane gives us a ready-made vector field in the form of surface normals, which we can edit and repurpose.

Vortices and Upwellings



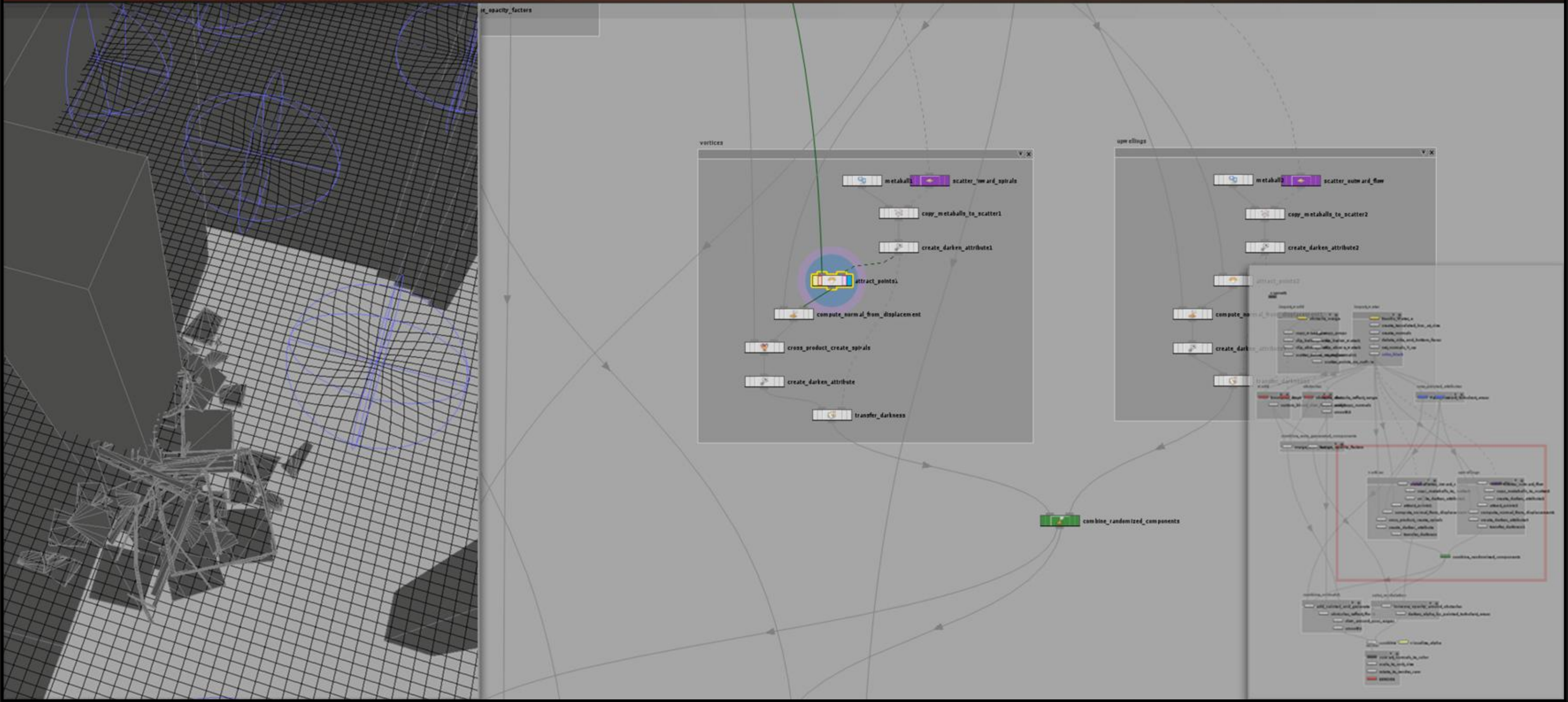
To create the vortices and upwellings, we scatter points randomly on the surface that will be the origins for these features.

Vortices and Upwellings



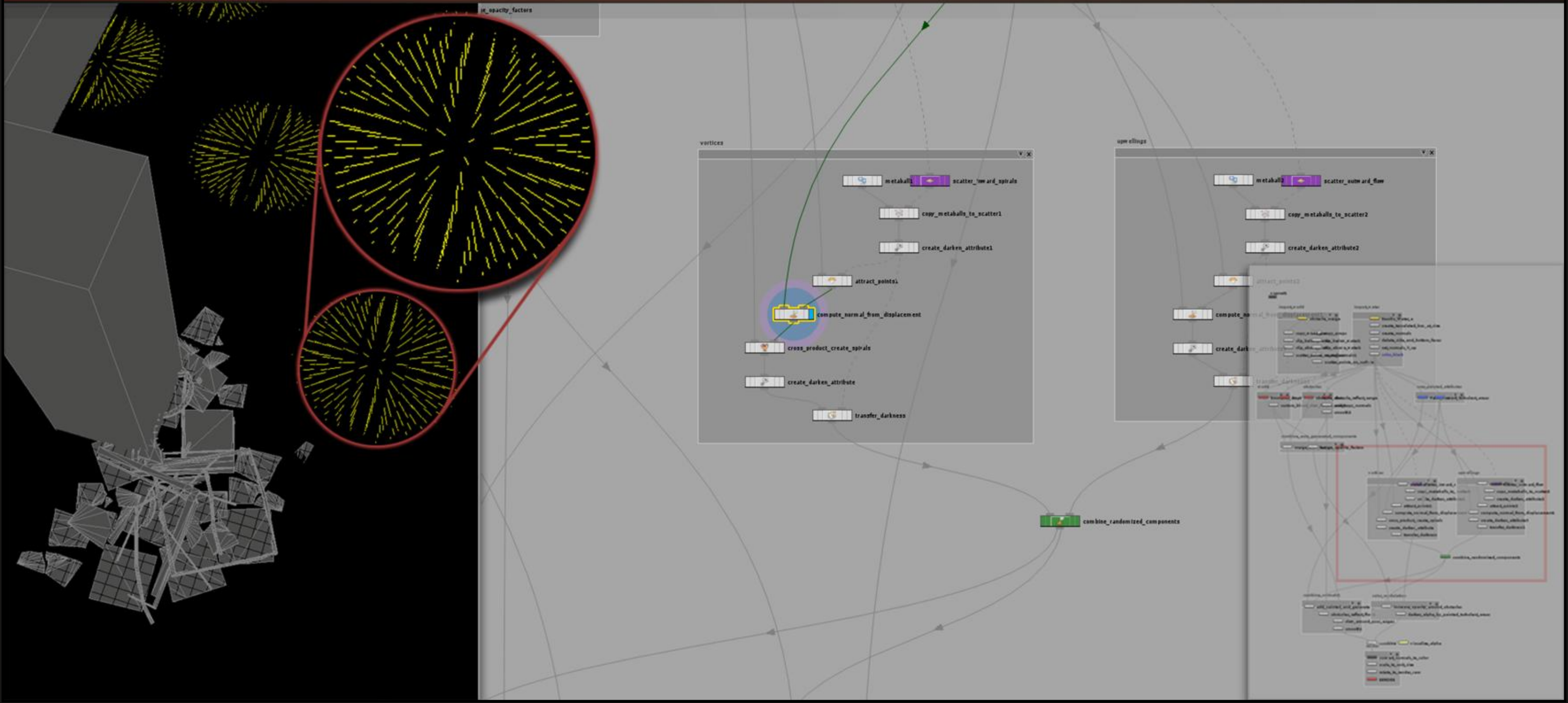
We copy metaballs to each point to use as point attractors.

Vortices and Upwellings



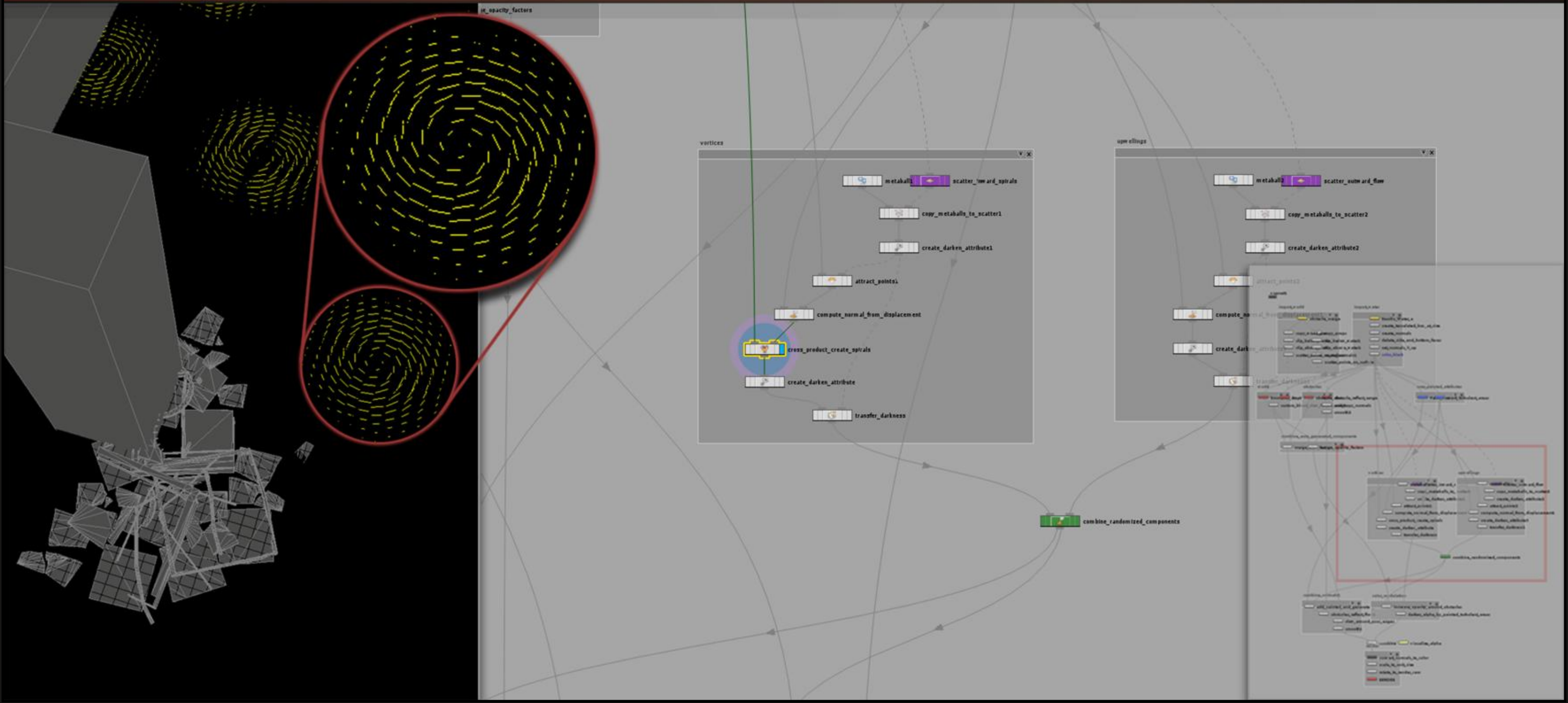
We can use the deltas between the original vertex positions and their new, attracted positions to create a new set of vectors.

Vortices and Upwellings



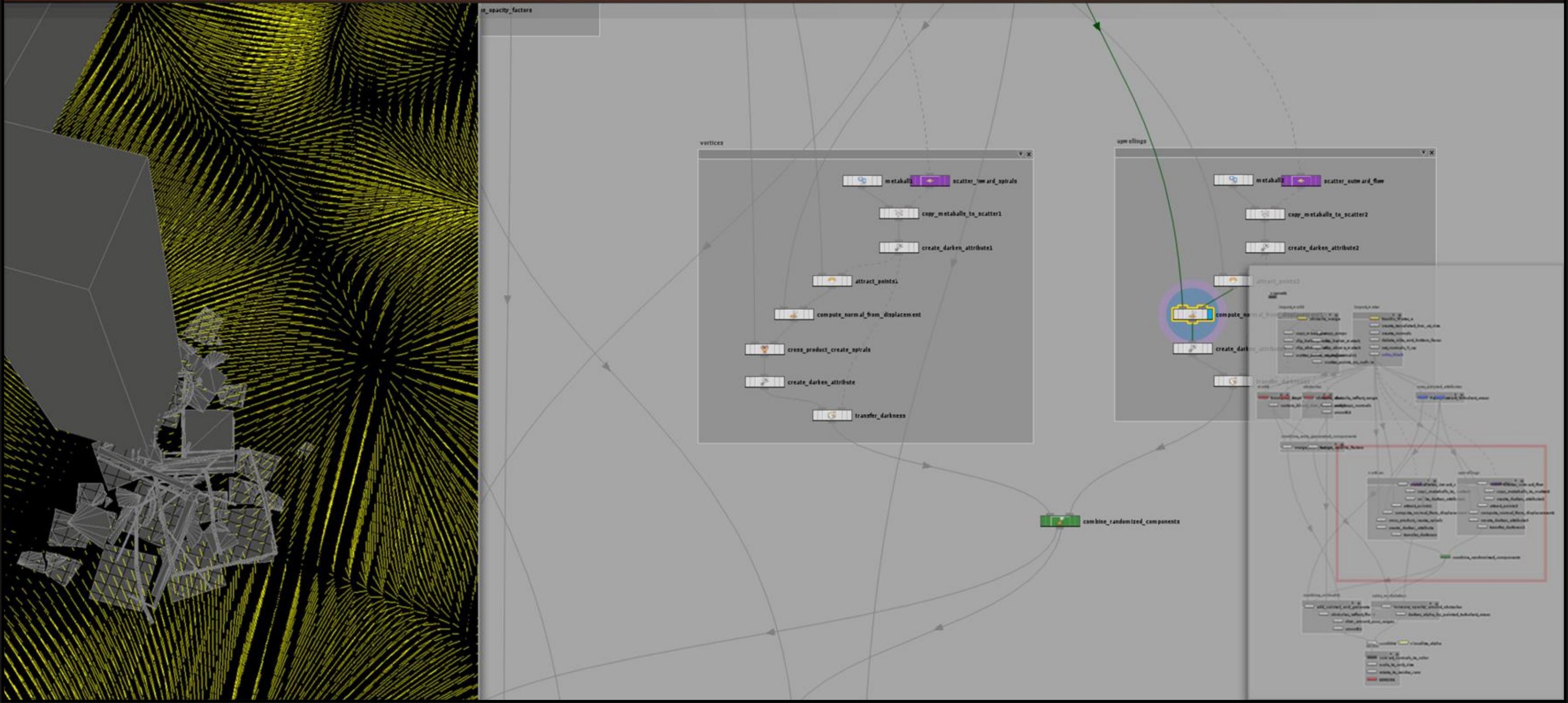
The new vectors are parallel to the water surface and have a built-in falloff: the ones at the edge are shorter than the ones near the center.
We'd stop here for the upwellings.

Vortices and Upwellings



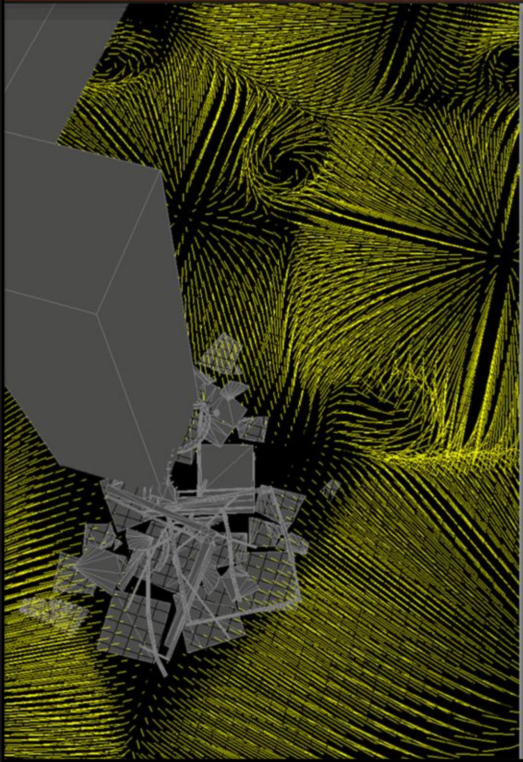
For the vortices, we use the cross product of the deltas and the original, upward-facing surface normals to create circular patterns.

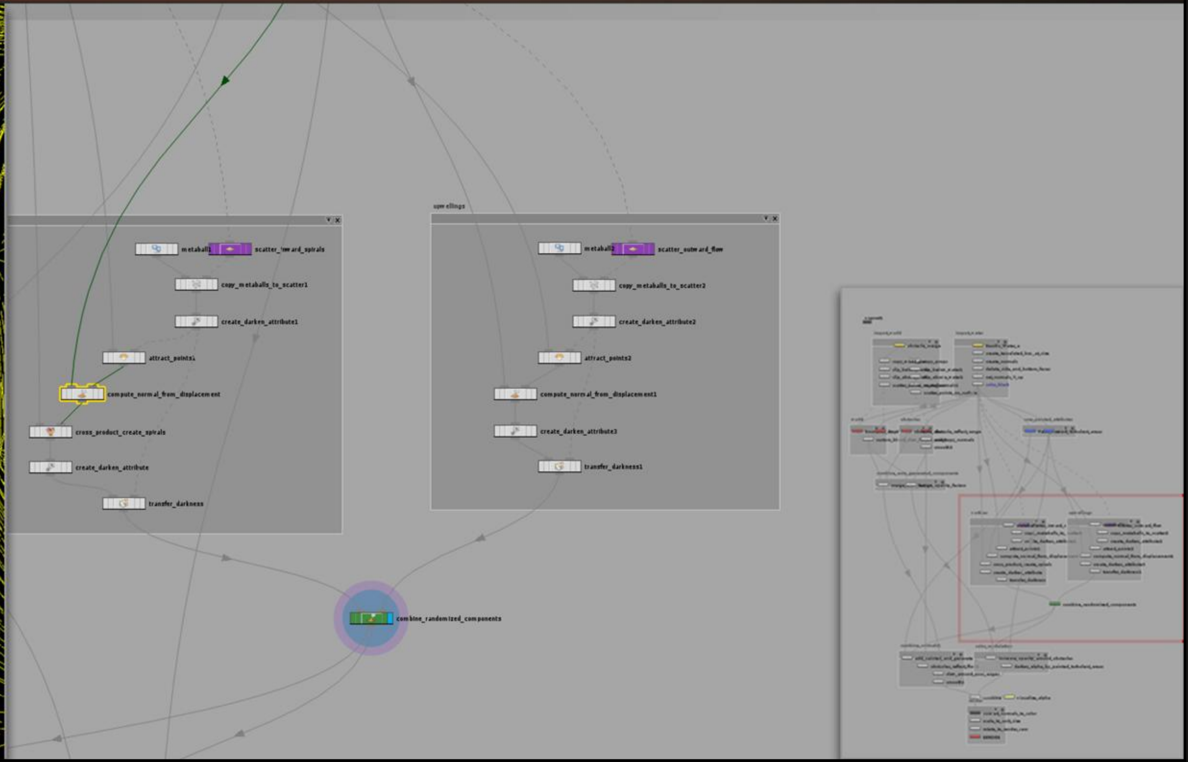
Vortices and Upwellings



The upwellings are much larger than the vortices, so that interesting interactions can happen at the borders between them.

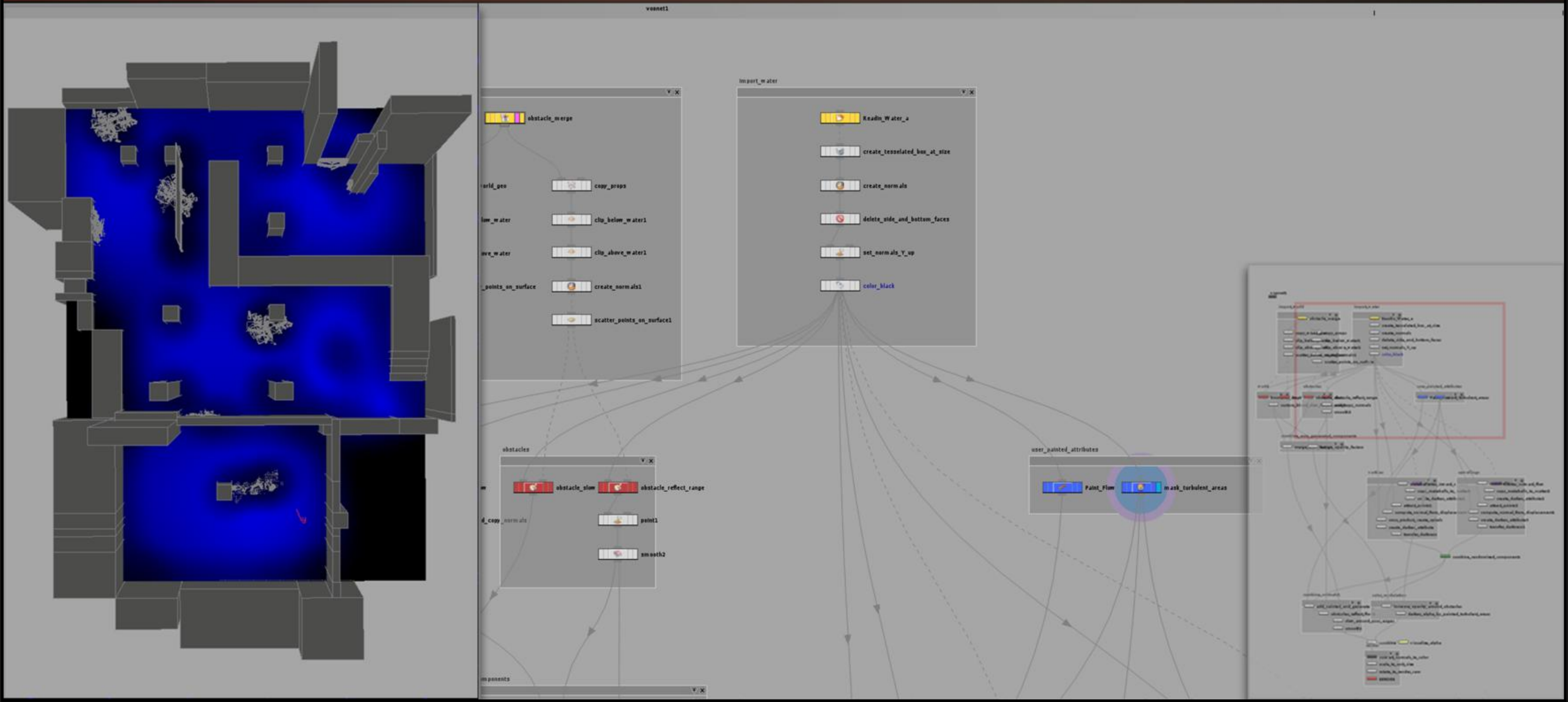
Vortices and Upwellings





The combined vortices and upwellings create a turbulent surface.

Mask Turbulence

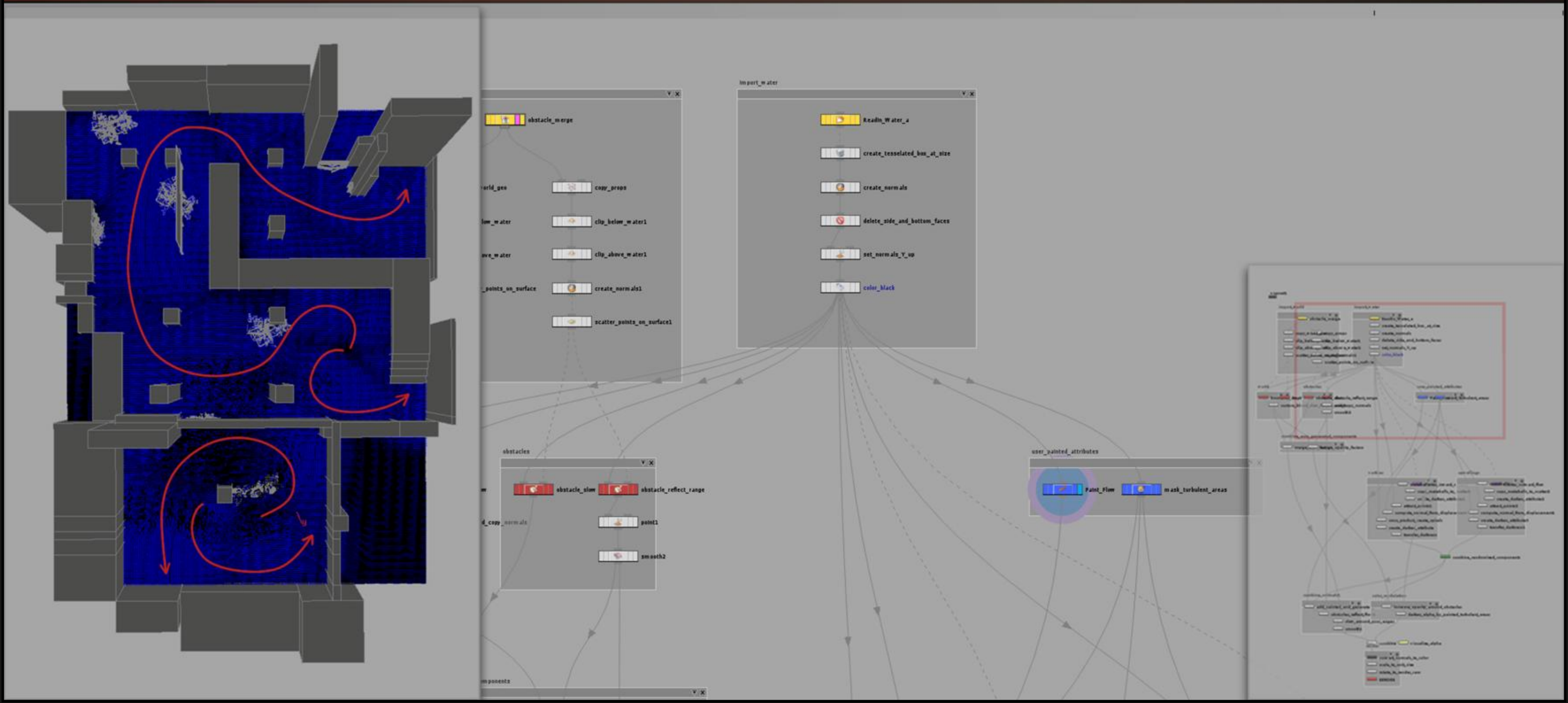


The turbulence should not be uniform over the surface, however.

The artist has the ability to paint the amount that the turbulence will contribute to the final flow.

We do this so that specific areas of a pool of water can have different characteristics, as determined by the level design and art direction.

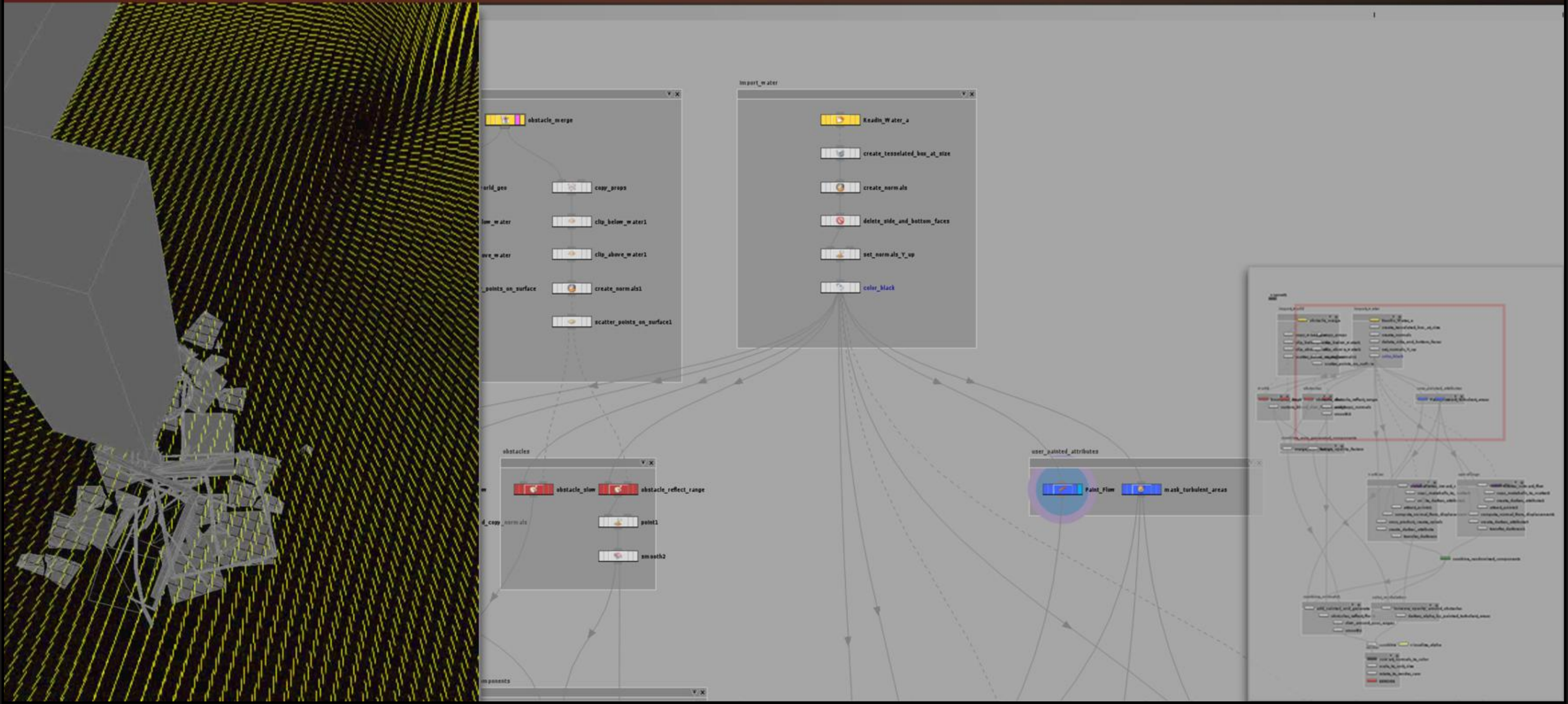
Overall Flow



Low-frequency motion is best accomplished by the user painting flow direction.

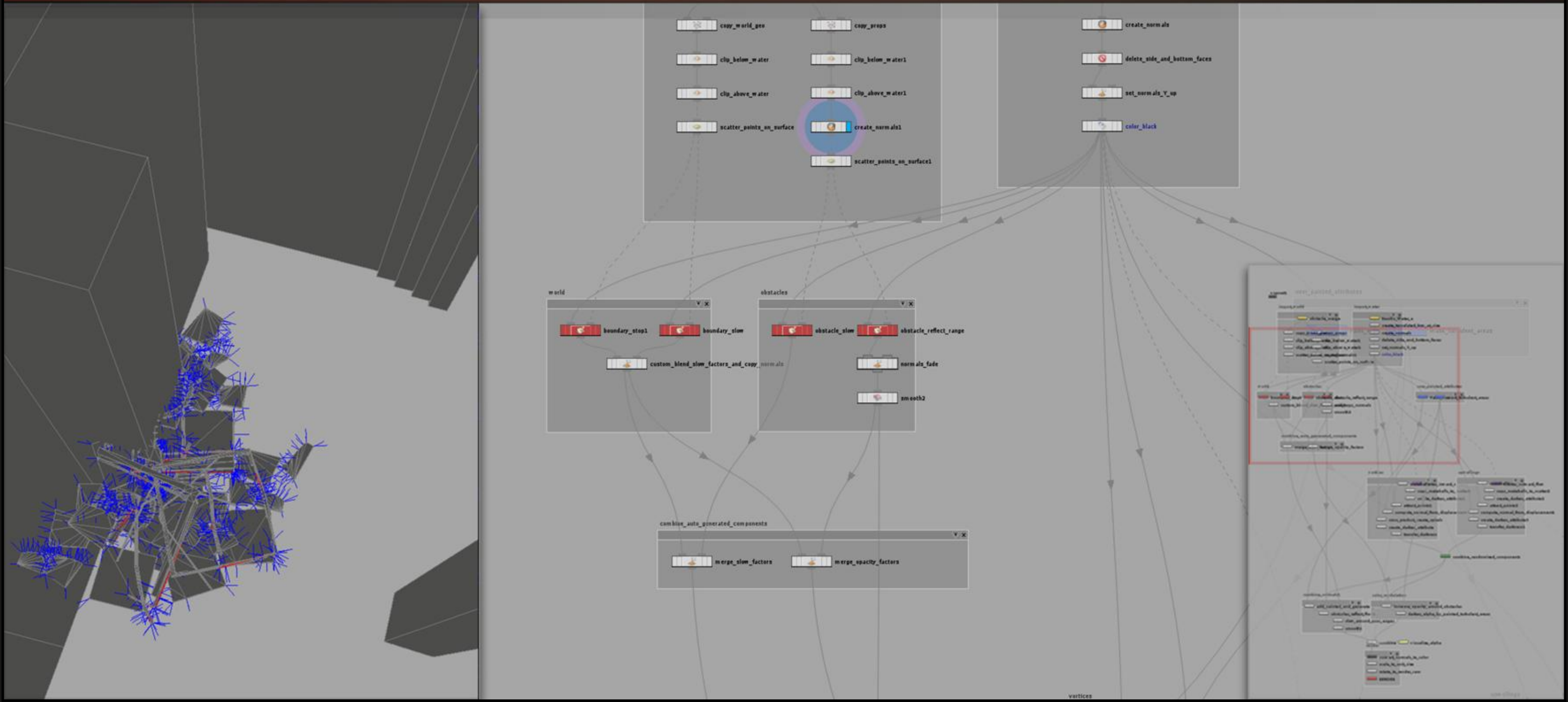
This could be randomized, but again, either the art direction or the level logic might call for giving the water a particular character, so keeping it under user control makes sense.

Overall Flow



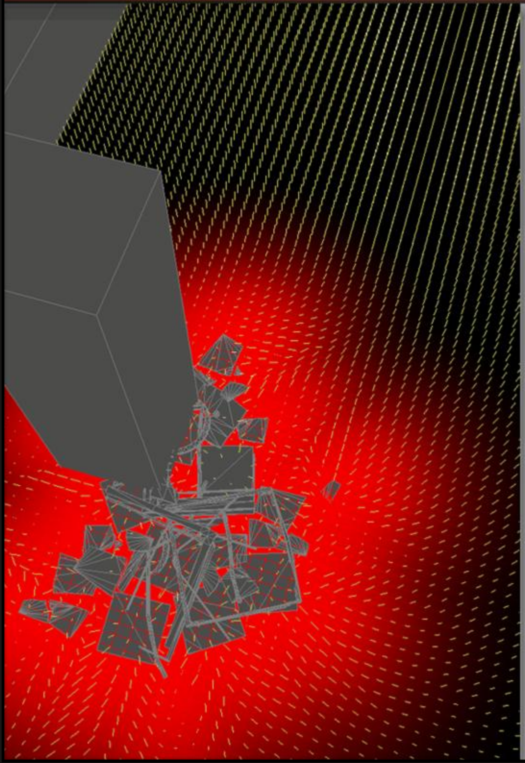
The overall flow can be painted directly; the surface normals take on the direction of the brushstrokes as the artist paints in the viewport. Painting flow for a typical Portal 2 map takes about 30 seconds.

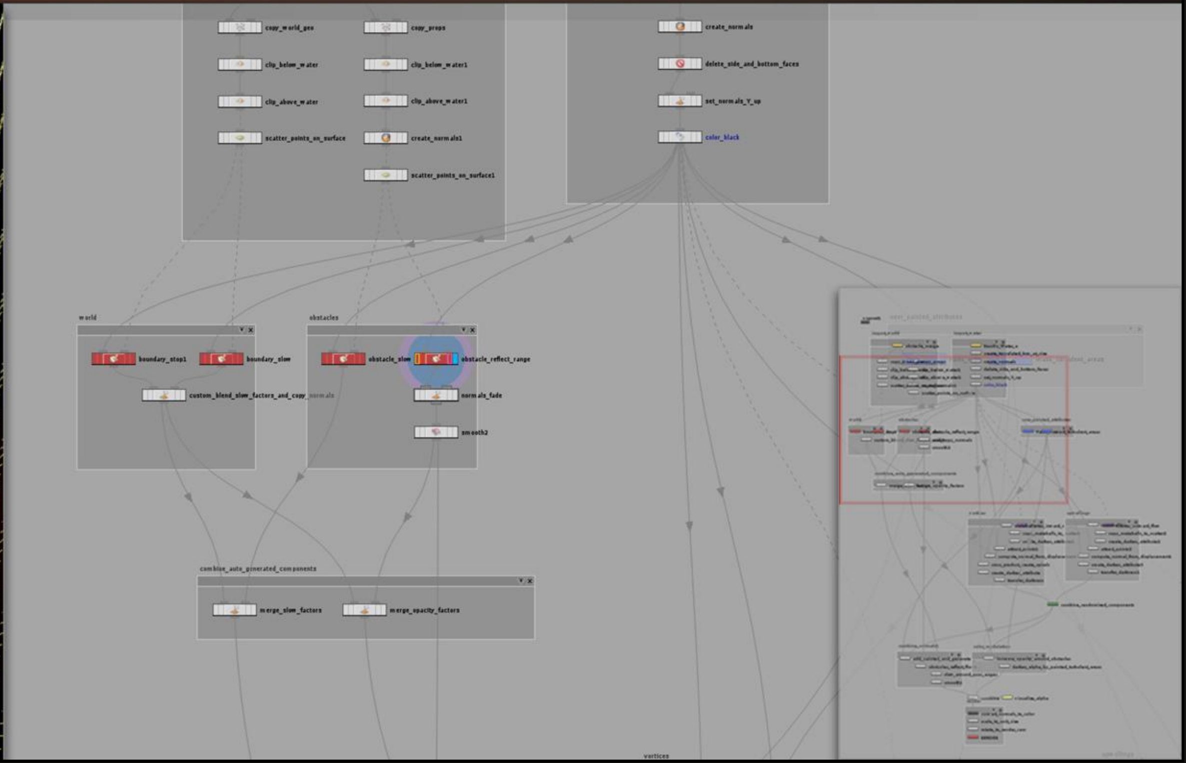
Derive Flow from Prop Geometry



The reflection or deflection of the flow vectors off the world surfaces can be accomplished by projecting the existing normals of the prop geometry into the vector flow.

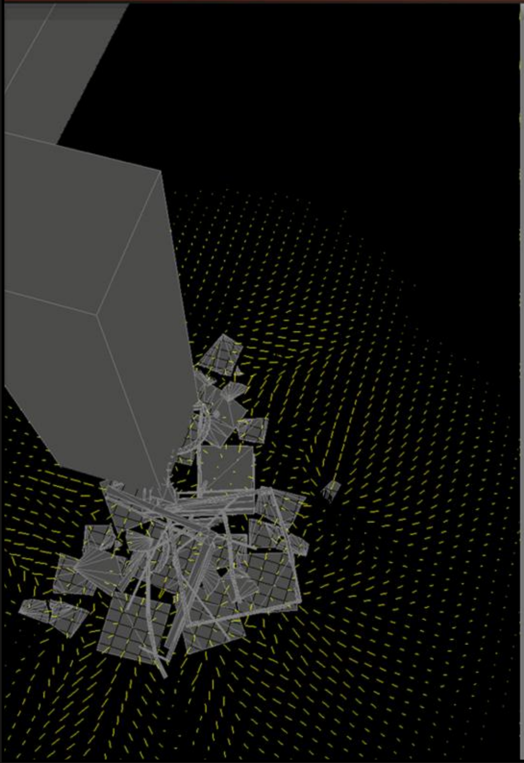
Derive Flow from Prop Geometry

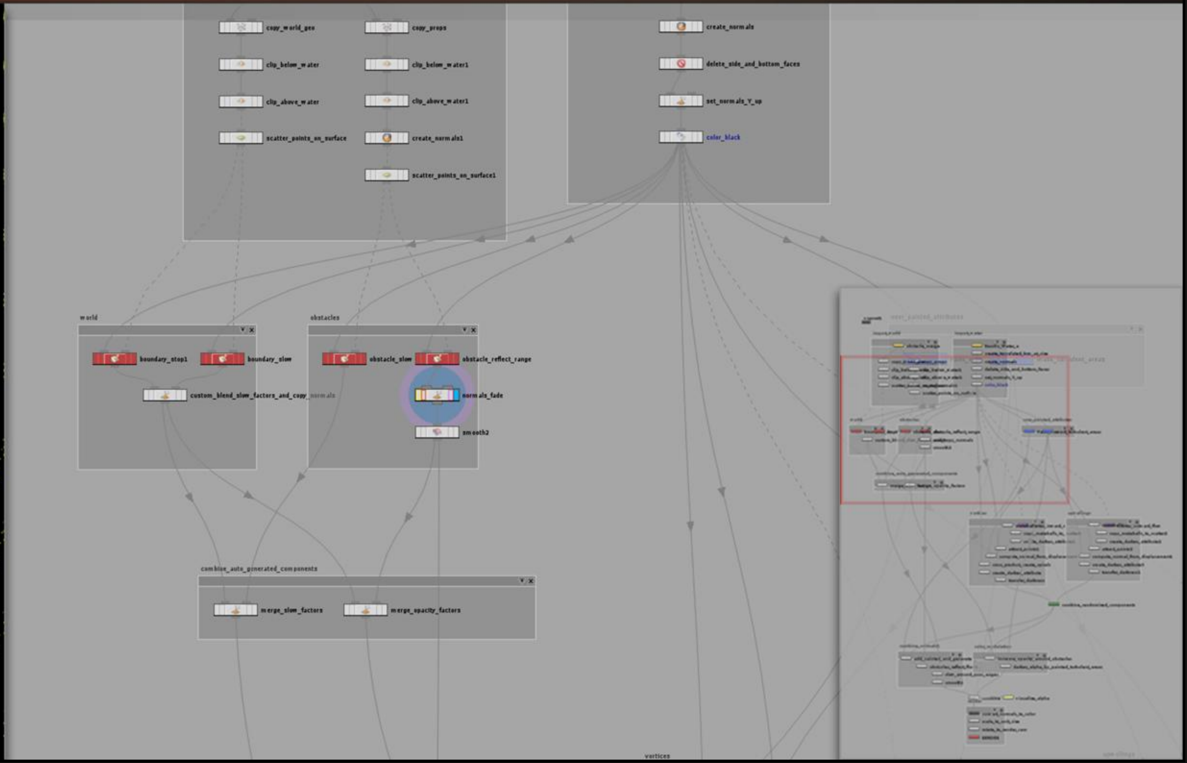




We project those normals onto the water surface with a falloff.

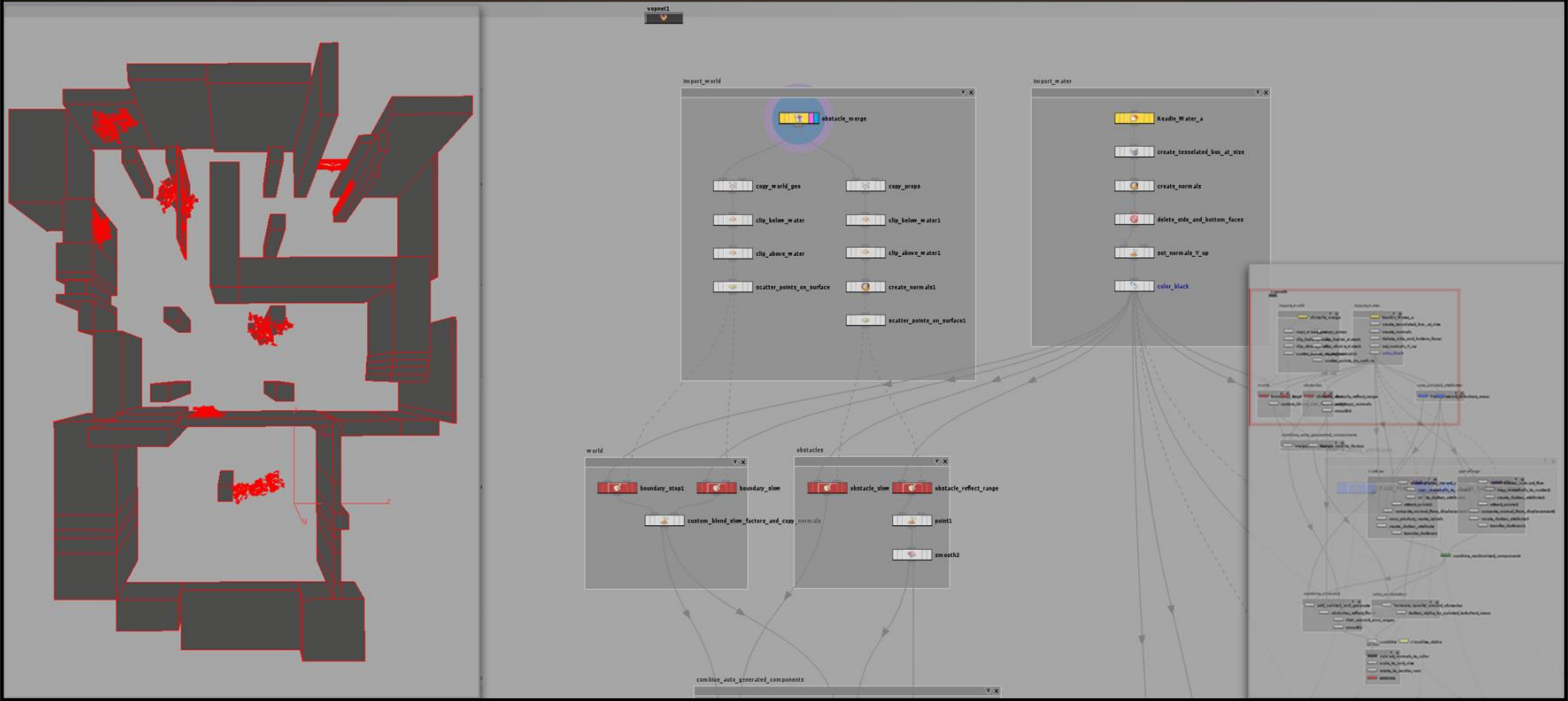
Derive Flow from Prop Geometry





The result will be combined with the rest of the flow later.

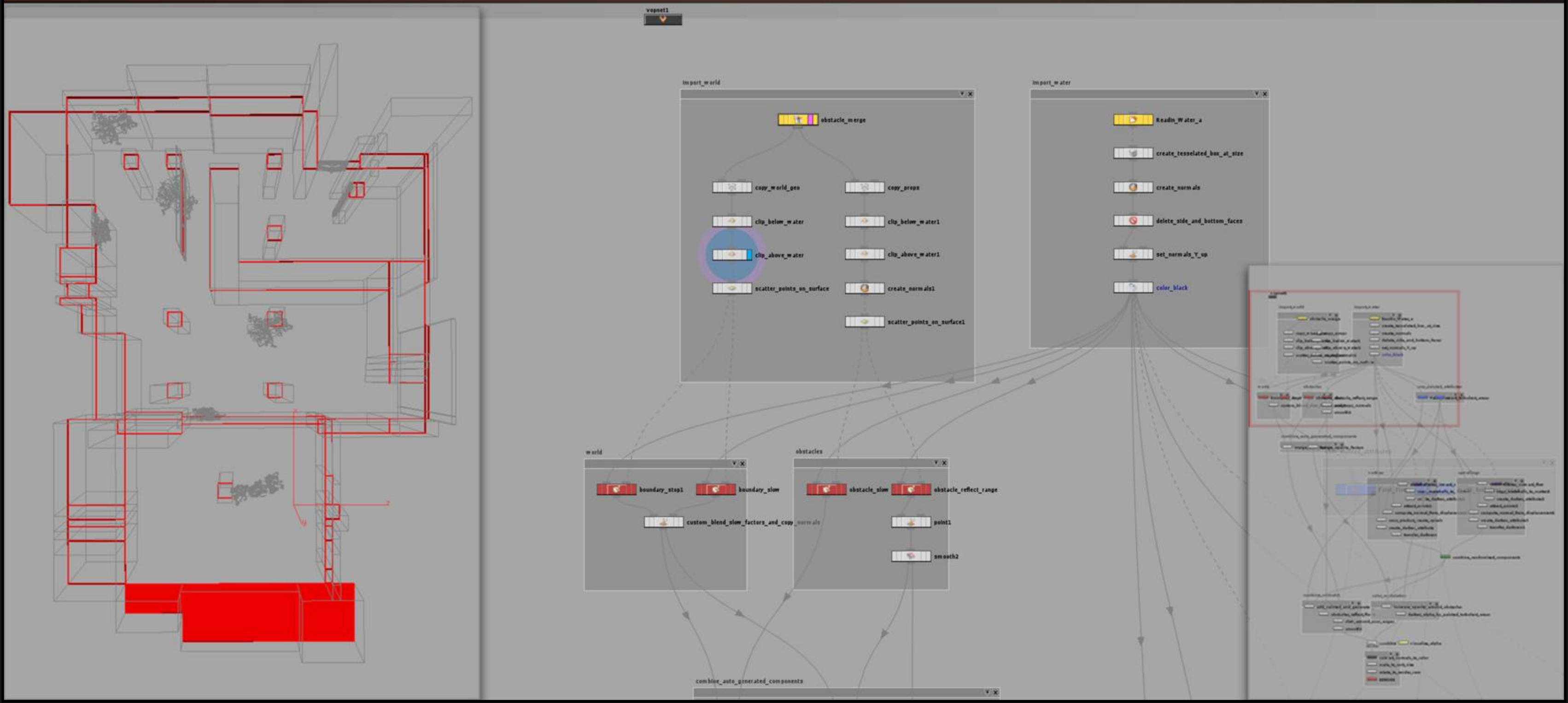
Using Geometry to Determine Speed



To slow down our flow around the edges of the level, we start with the imported level geometry.

We need to somehow project the geometry onto the water plane so that it is usable in context.

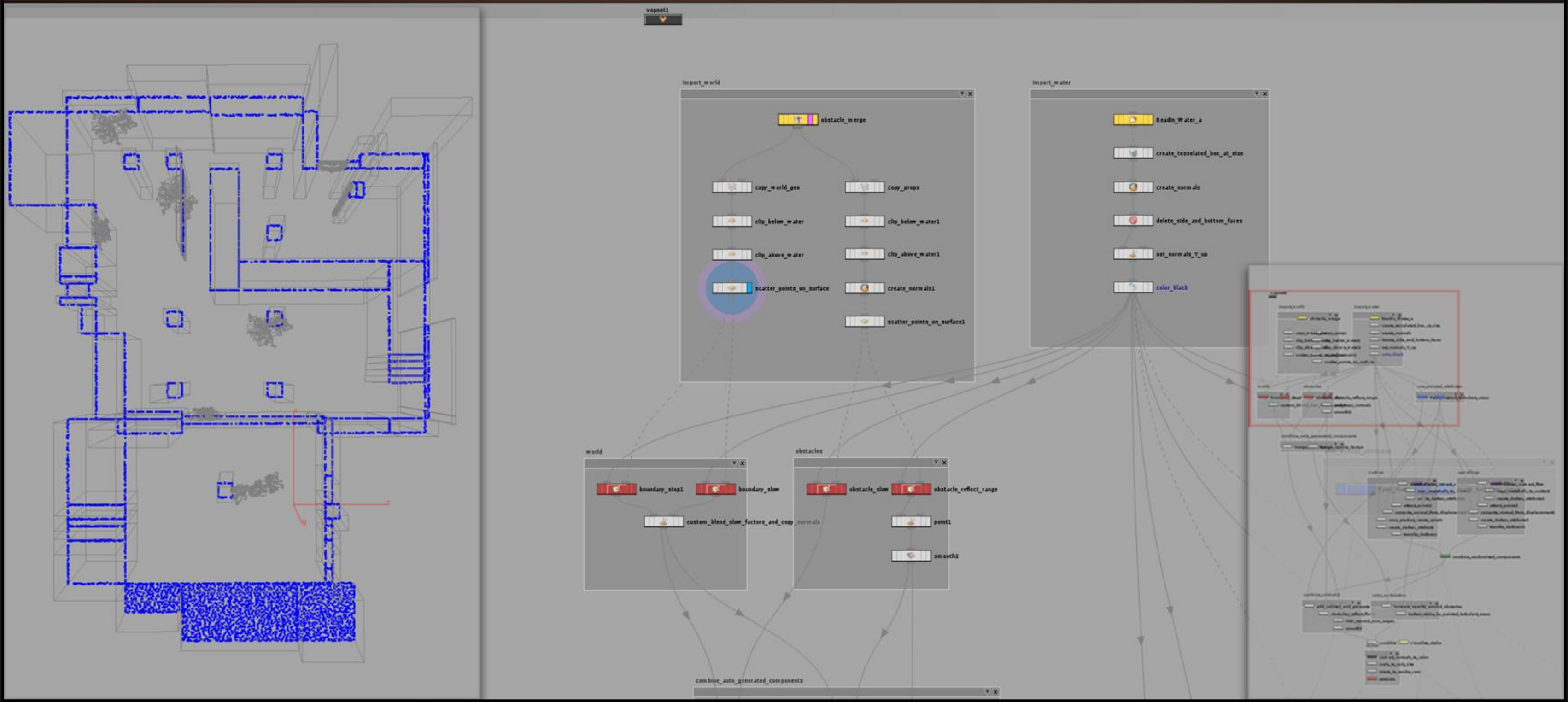
Using Geometry to Determine Speed



The first step is to clip the geometry around the water surface.

This is purely for performance, because our next step…

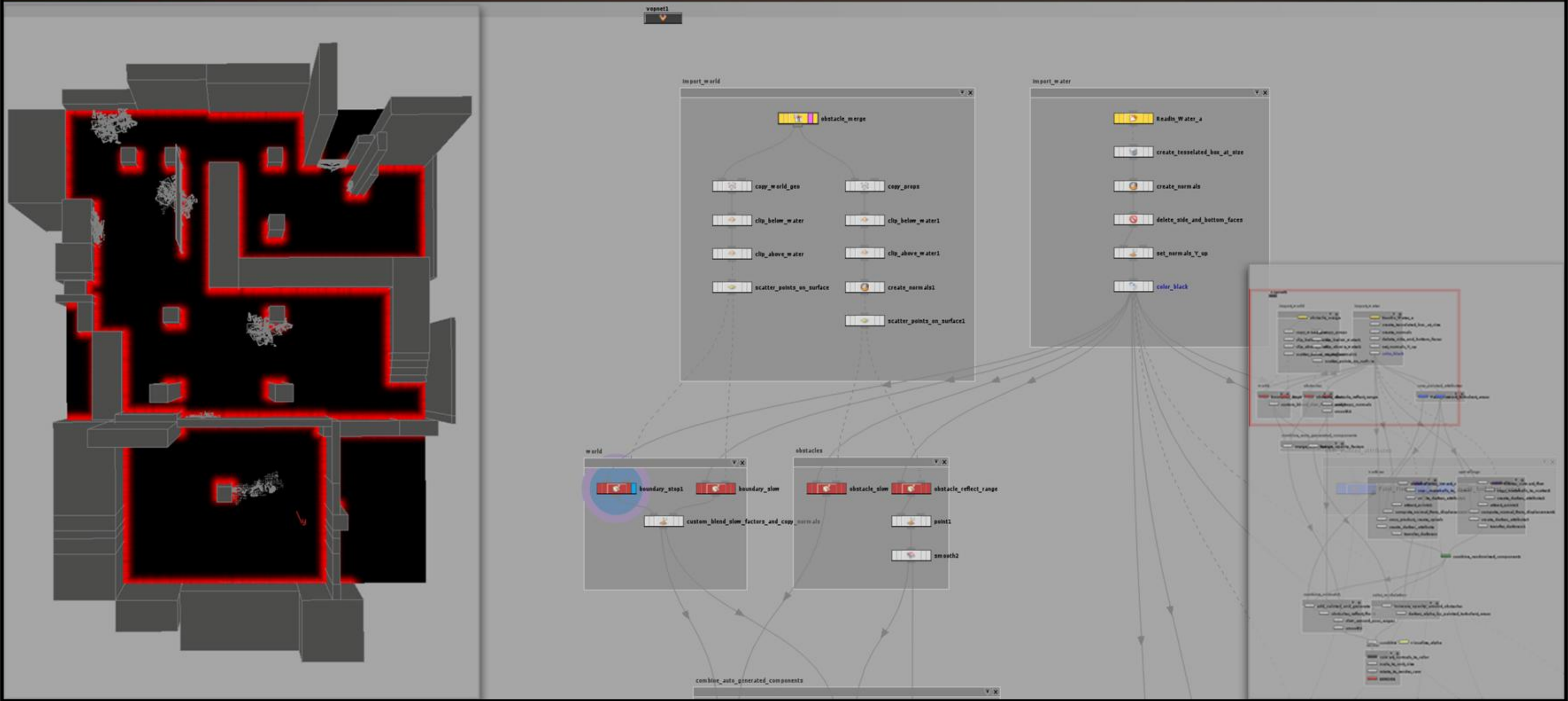
Using Geometry to Determine Speed



…is to scatter hundreds of points over the surface.

This gives us a nice, dense distribution of points whose attributes can be projected onto the water surface.

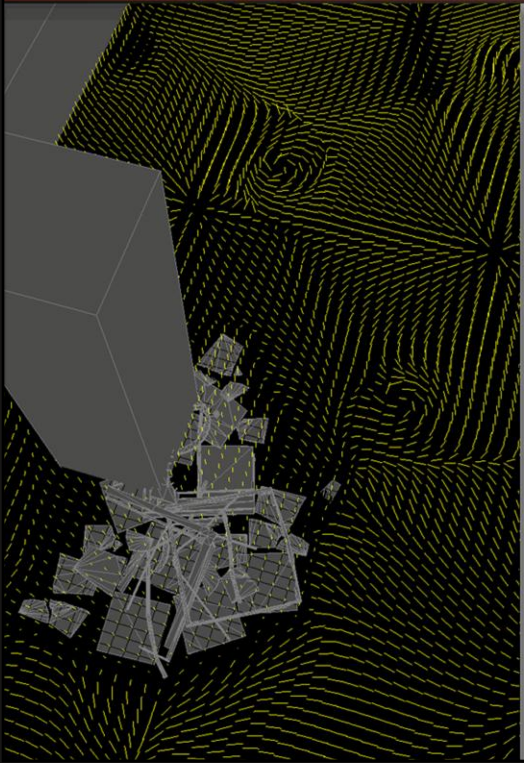
Using Geometry to Determine Speed

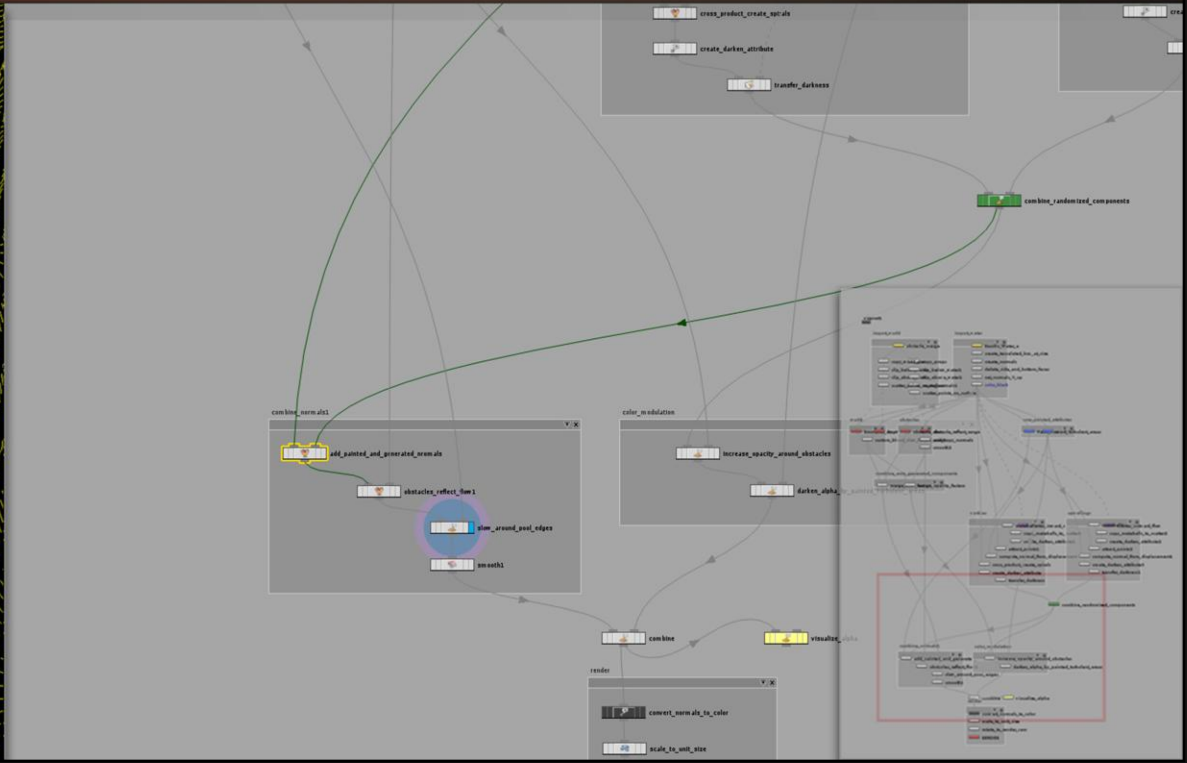


We assign the points a simple color and project it.

We will use this as a mask to affect flow and foam accumulation.

Combine Flow





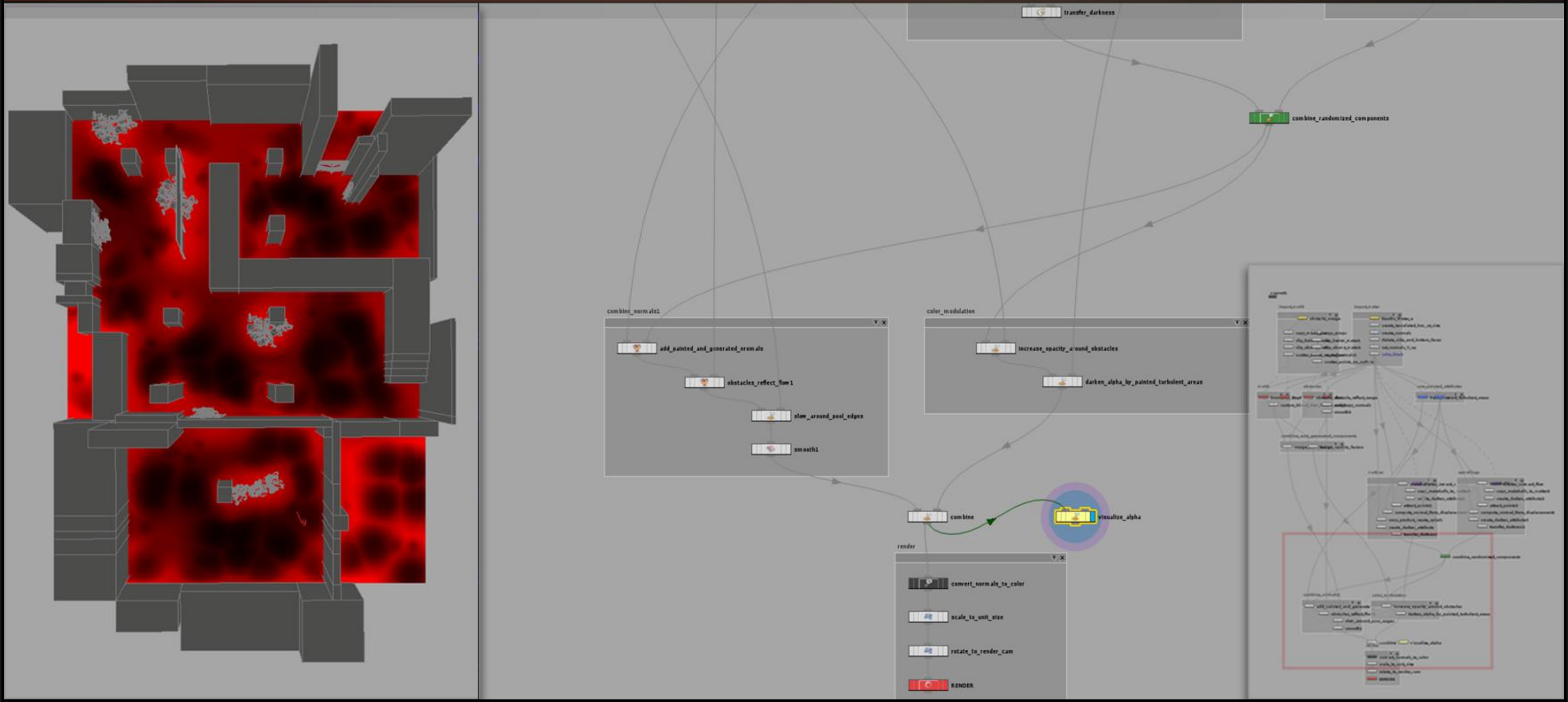
Combining the turbulence, overall flow direction, and slowing gives us a vector field that looks like this.

We will keep only the x and y components of the vectors in the field.

This means the more parallel a vector is to the water surface, the faster the flow will be. Vectors that point up indicate still water.

The red mask from the geometry edges is used to lerp between the original surface normals and the combined flow components, thus slowing the water around the edges of the pool.

Combine Opacity



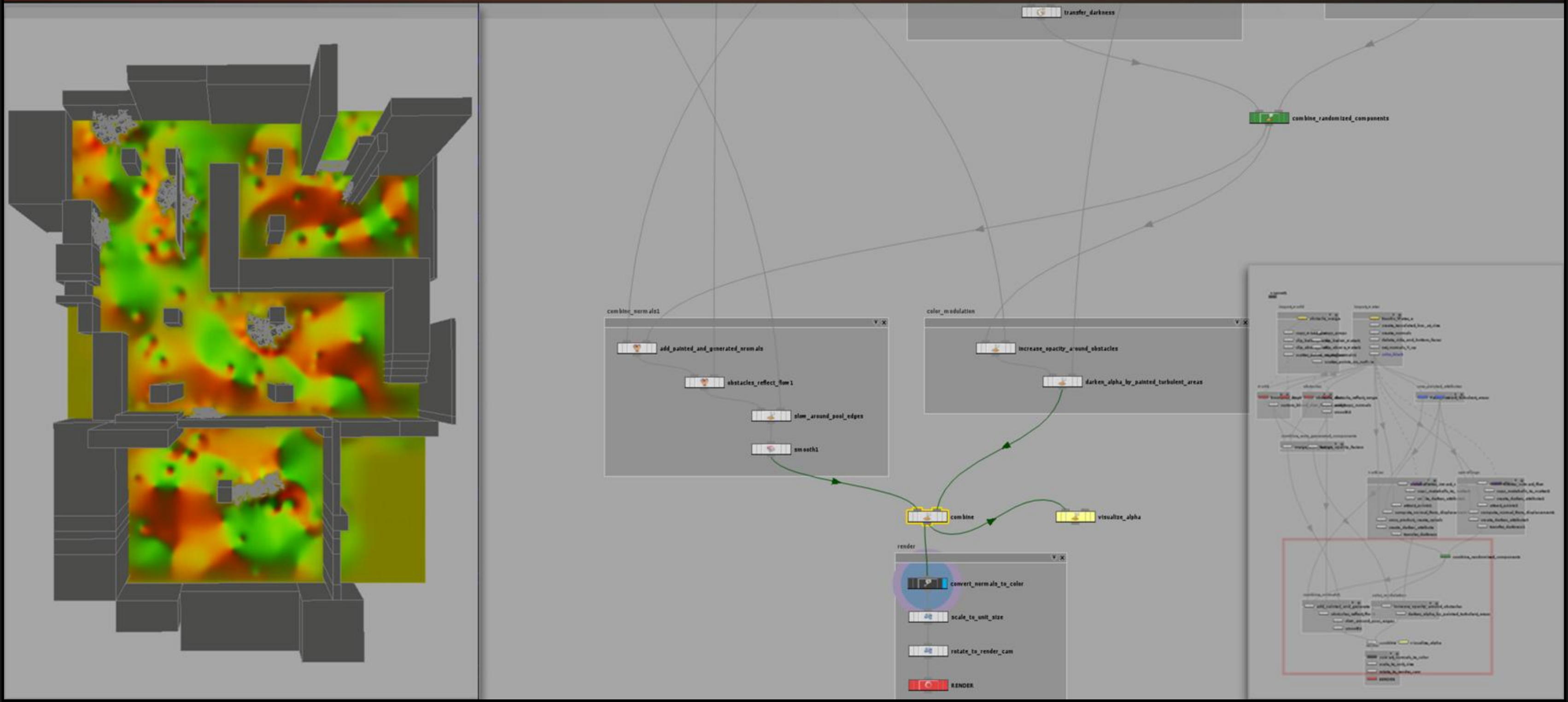
The opacity of the foam on the surface is affected in a couple of ways. This will be stored in the alpha channel of the output but we'll visualize it here in red.

First, it is reduced based on proximity to an upwelling.

Then it is further reduced by the painted turbulence mask: the more turbulent the water, the less foam will have collected.

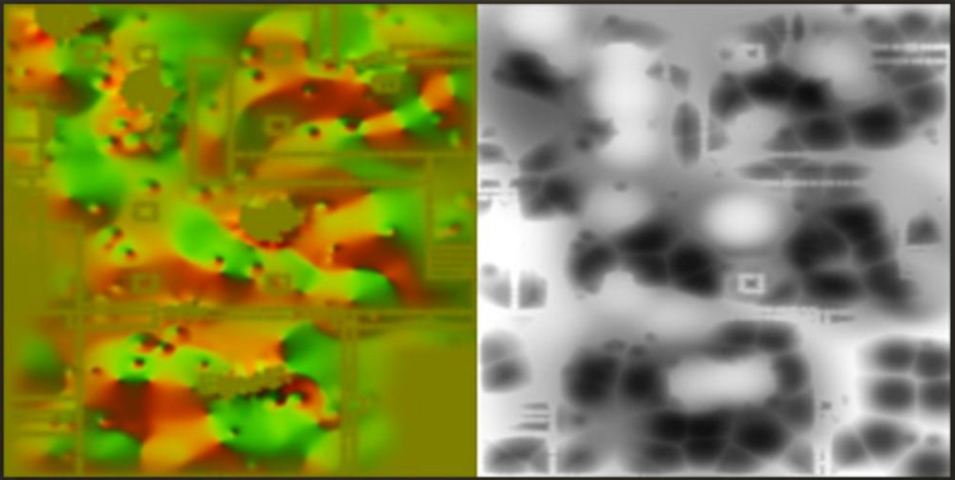
Lastly we make sure the opacity is increased in areas the foam should collect, in slow areas and around the pool edges.

Final Color



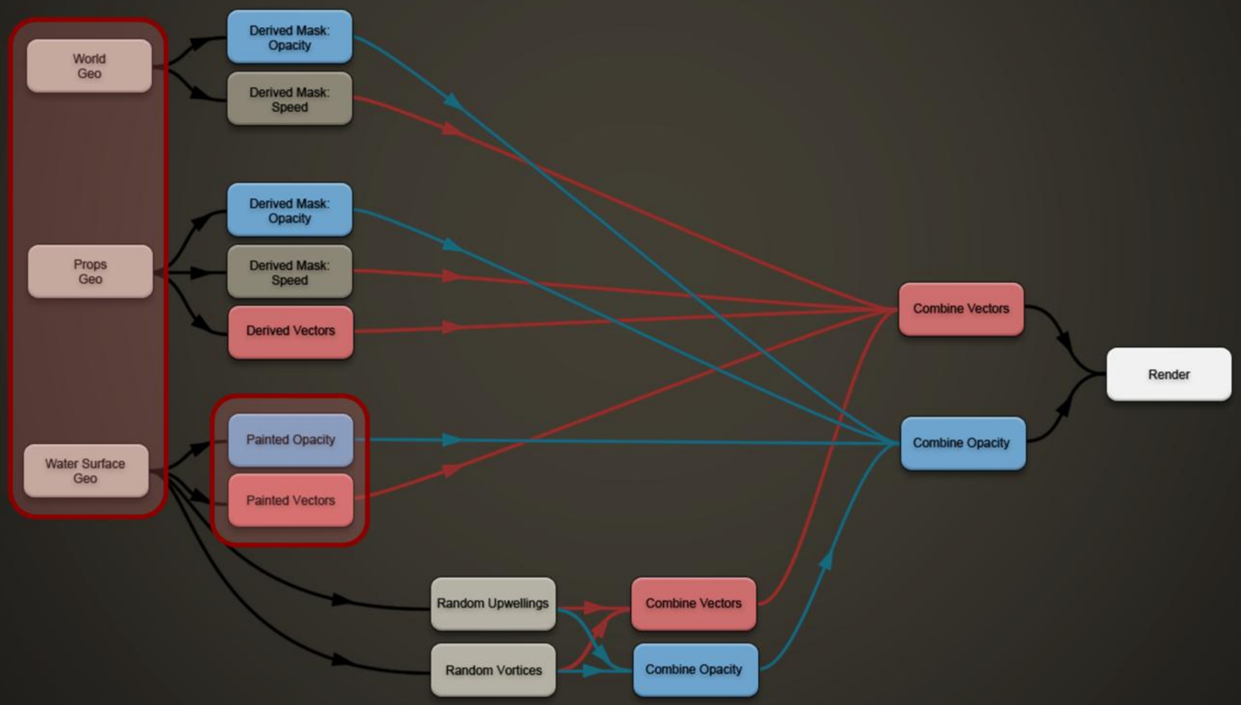
The final result converts the x and y components of the vector field into color much the way we would store a normal map. (C.rg = N.xy * 0.5 + 0.5)
The blue channel isn’t used because we don’t care about keeping these vectors normalized; again, the shorter they are, the slower the flow.

Final Texture



The final result is a 256x256 DXT5 texture where R and G represent the flow and A represents the opacity of the color map.

Templatizing and Organizing



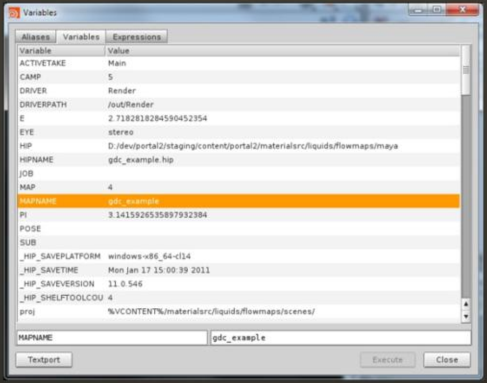
The best part about this technique is how easy it is to use to generate new textures.

We need only swap out our inputs.

The only other thing that might get touched are the turbulence masking and overall flow direction, but that's optional.

Templatizing and Organizing

- Houdini has the concept of variables
- Assign variable names to all input and output file names
- All it takes to generate a new flow map is to switch out the MAPNAME



Variable	Value
ACTIVETAKE	Main
CAMP	5
DRIVER	Render
DRIVERPATH	/out/Render
E	2.7182818284590452354
EYE	stereo
HIP	D:/dev/portal2/staging/content/portal2/materialsrc/liquids/flowmaps/maya
HIPNAME	gdc_example.hip
JOB	
MAP	4
MAPNAME	gdc_example
PI	3.14159265358979323846
POSE	
SUB	
_HIP_SAVEPLATFORM	windows-x86_64-cl14
_HIP_SAVETIME	Mon Jan 17 15:00:39 2011
_HIP_SAVEVERSION	11.0.546
_HIP_SHELFTOOLCOU	4
proj	%VCONTENT%/materialsrc/liquids/flowmaps/scenes/

MAPNAME gdc_example

Houdini has the concept of variables built into the interface.

We can assign variable names to the inputs and even generate them from a single variable, requiring us only to name our input files correctly then switch out a few characters under our MAPNAME variable.

The updated content causes a cascade effect through the network, recreating the masks based on the new input and re-distributing the randomized components, instantly creating new content that is customized for the new level.

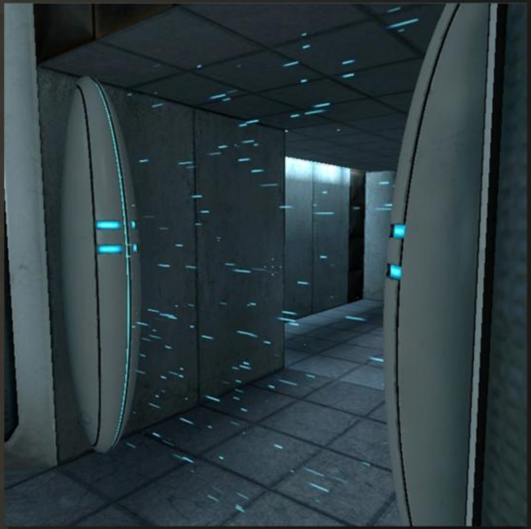
Results

- Complex, realistic motion
- Flexibility to achieve art direction
- Extremely simple geometry: 4-point planes
- Flow maps don’t need high resolution: 256^2 DXT5 on consoles, even for large levels
- 2 artists, 30 levels, 1.5 days

Because we were able to templatize the process, we could complete customized flow maps for 30 levels in just a day and a half.

The Aperture Science Material Emancipation Grid

- Originally we did not plan to change the effect from Portal 1
  - Used in Portal 1 to avoid problems when players carried objects into new chambers
- Used more extensively as a puzzle element in Portal 2
  - Closes existing portals when player passes through
  - Blocks portal placement through grid
  - Destroys level objects on contact



We didn't just use the flow algorithm on the water, though.

One of the gameplay elements that has been around since the first Portal is the Aperture Science Material Emancipation Grid.

It was used in Portal 1 primarily to stop players from bringing cubes from puzzle to puzzle, or using them to jam the elevators between puzzle chambers.

In Portal 2, they are used extensively as puzzle elements because they have several useful properties:

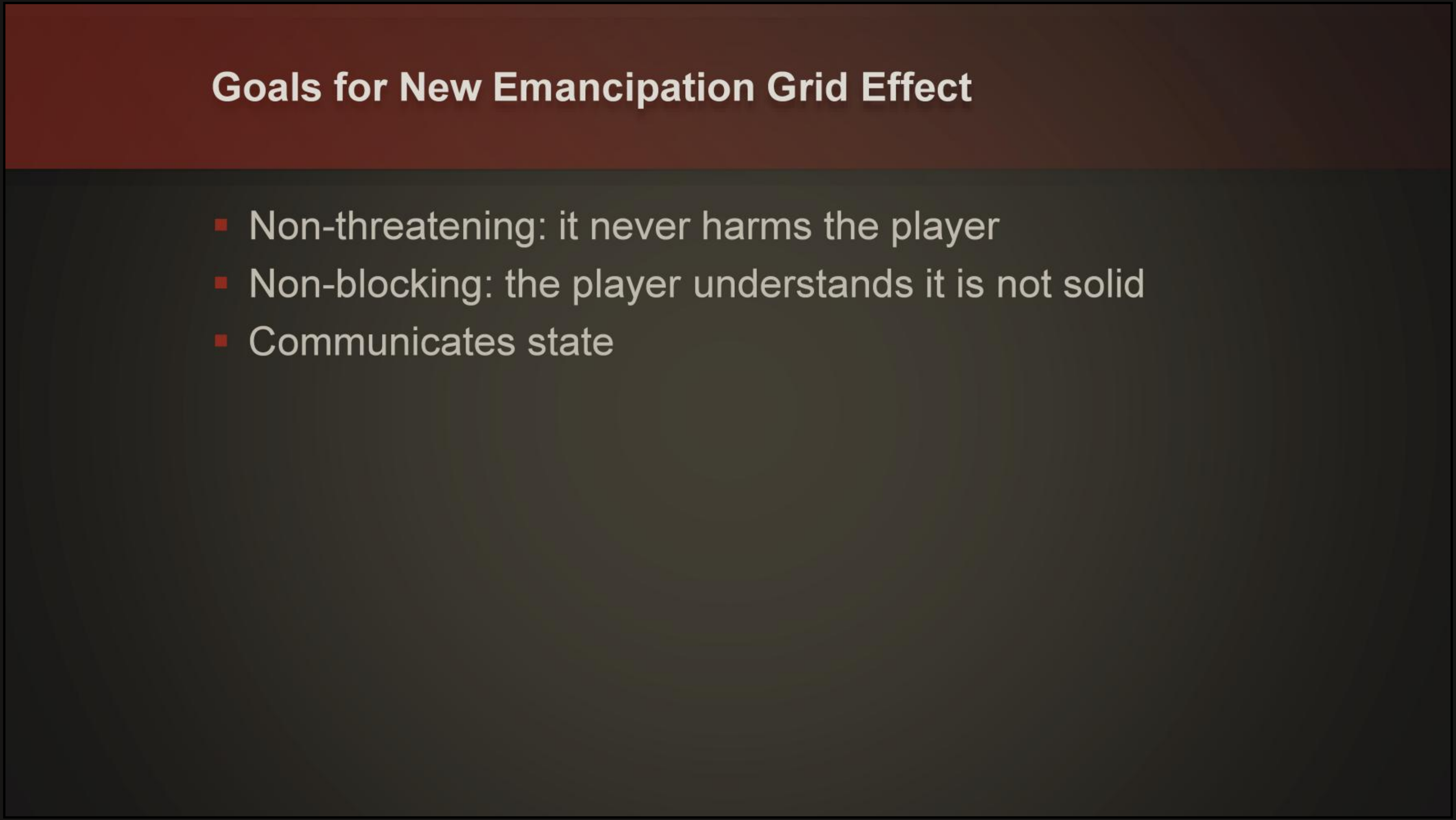
•When a player passes through one, any portals they have placed are closed, which is useful particularly in co-op. It can force players to think through the order in which they navigate the environment and place their portals.

•Portals cannot be placed through a grid, allowing the level artists to control where a player can stand to place a portal where it is needed. In some cases, reaching that location is a part of the puzzle.

•Puzzle objects like cubes cannot be carried through them; they are destroyed. Players may have to figure out how to get a puzzle object around or over the grid to solve a puzzle.

As you can see in the screenshot, the original effect was particle-based and not very dense. Playtests revealed that players often walked through them without registering their presence, and were confused by the consequences.

Players were also not well able to report the three properties of the grid, which indicated to us we were not communicating when a player/grid interaction was occurring.



Goals for New Emancipation Grid Effect

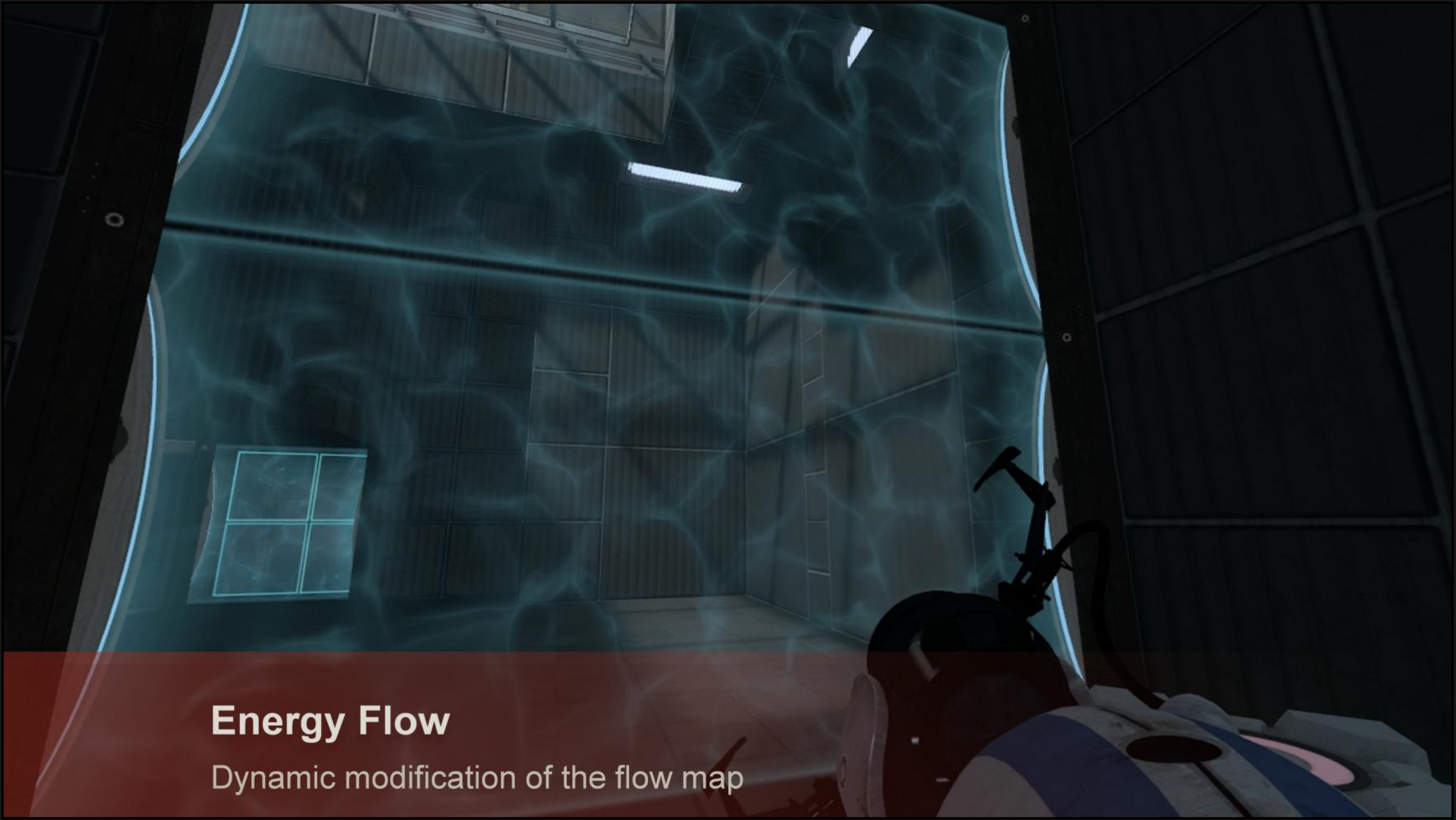
- Non-threatening: it never harms the player
- Non-blocking: the player understands it is not solid
- Communicates state

We decided to revisit the effect.

It had to be non-threatening: players *must* pass through them to progress, so shouldn't have to question whether it is safe.

It had to appear non-solid: it's no help if it looks non-threatening if players never even try to walk through it.

It had to communicate to the player when an important interaction was occurring.

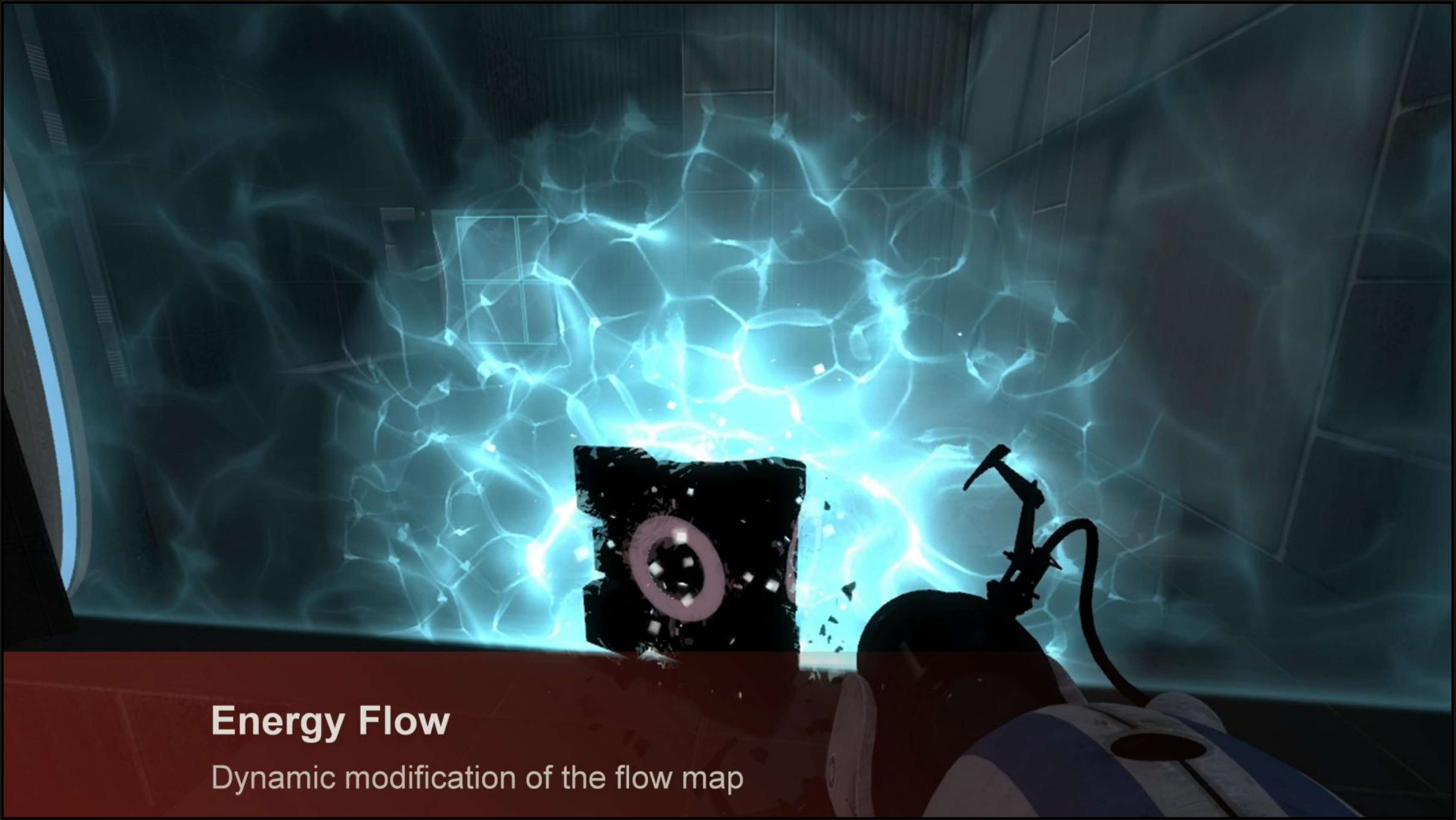


Energy Flow

Dynamic modification of the flow map

We used the flowmap technique on an additive surface. To keep it looking non-threatening, we used a cool color scheme and a color map that recalled the kind of caustics you might see in a shallow swimming pool.

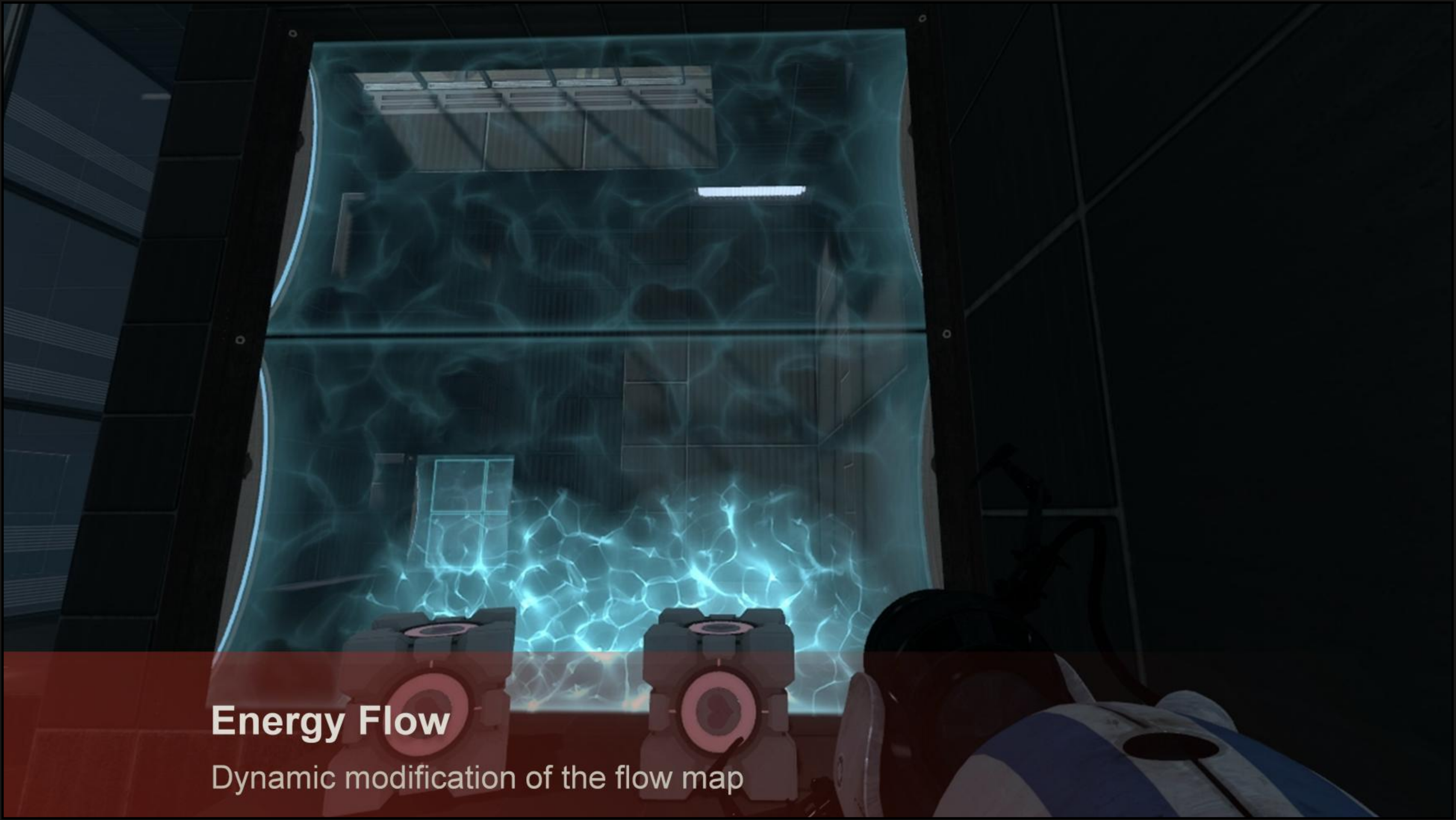
Shooting the grid lights the entire surface, communicating its blocking ability.



Energy Flow

Dynamic modification of the flow map

To communicate impending destruction of a level object, the surface lights when one is brought near. The flow also changes, sucking the energy of the field towards the offending object.



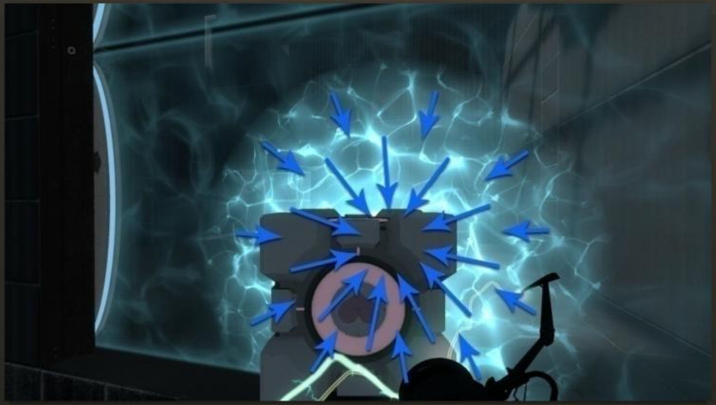
Energy Flow

Dynamic modification of the flow map

We had enough free interpolators to pass the positions of two level objects at a time; this is enough for most maps, which don’t contain large numbers of puzzle objects.

Energy Flow

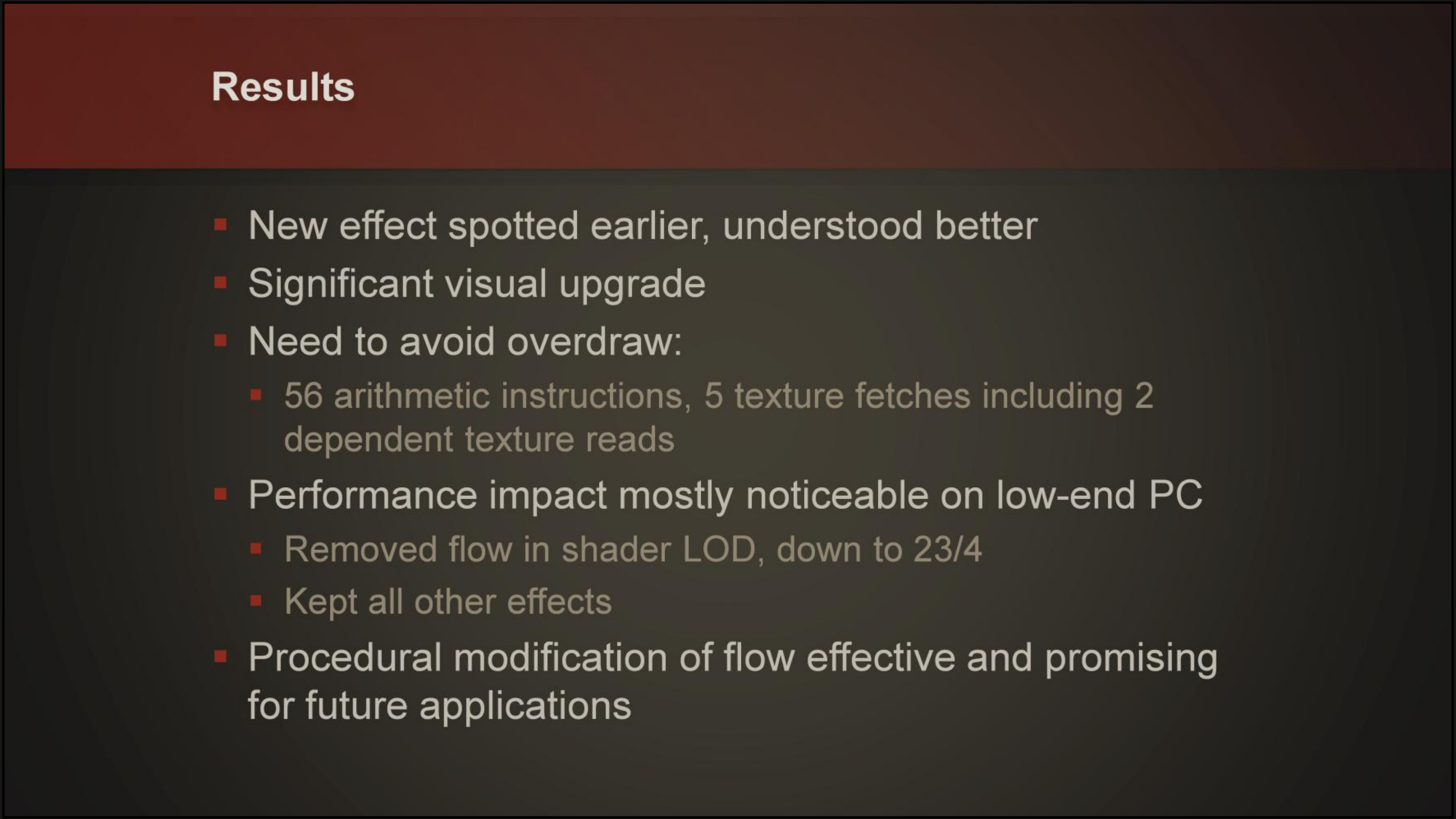
- Take game input to communicate state:
  - Position of “fizzleable” objects projected into tangent space
- Use UV-position relative to object to generate delta, use as flow: just like upwellings in Houdini



To create the lit vortices when an object approaches, we use game input for the two closest level objects.

The position is projected into tangent space, then used to generate a set of deltas over the surface, just like the upwellings we created in Houdini.

The result is a dynamic flow that responds to game state.



Results

- New effect spotted earlier, understood better
- Significant visual upgrade
- Need to avoid overdraw:
  - 56 arithmetic instructions, 5 texture fetches including 2 dependent texture reads
- Performance impact mostly noticeable on low-end PC
  - Removed flow in shader LOD, down to 23/4
  - Kept all other effects
- Procedural modification of flow effective and promising for future applications

Playtests revealed that players spotted the new effect earlier and understood it better.

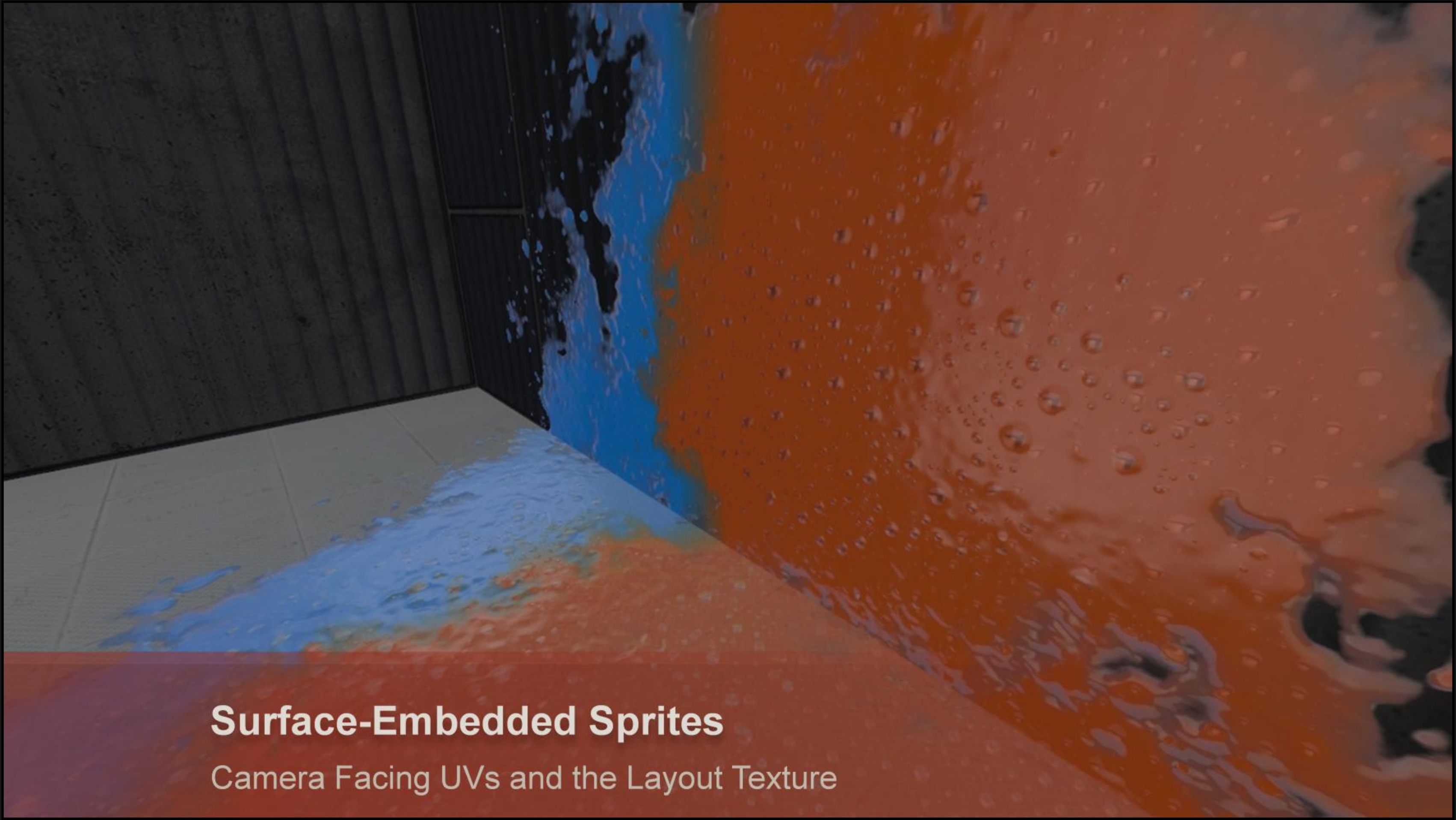
It was a significant visual upgrade, but also came with a cost:

We needed to avoid overdraw. The pixel shaders consists of 56 arithmetic instructions and 5 texture fetches, including two which are dependent texture fetches. In a couple of cases, level design needed to change so that you couldn't look through several emancipation grids at once.

The performance cost was felt mostly on low-end PC and Mac, so we used a shader LOD that discarded the flow but kept the other effects.

The procedural modification of the flow was very effective, and is promising for future applications. One can imagine a stream dynamically flowing around player characters.

In all, the flow algorithm gave us a huge visual boost, both to the emancipation grid and the water surfaces.



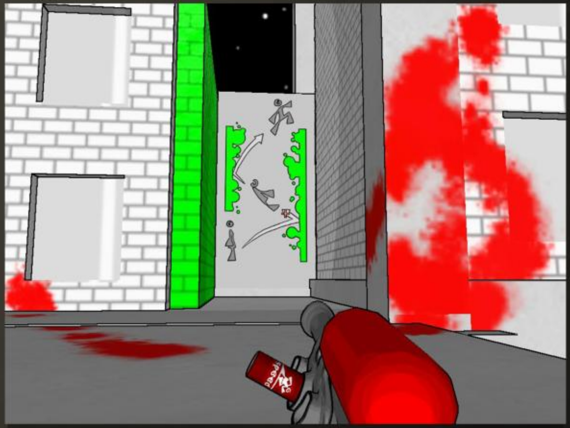
Surface-Embedded Sprites

Camera Facing UVs and the Layout Texture

Water isn't the only material that covers a lot of area in Portal 2. The gels, a new gameplay element, can be used to procedurally paint a lot of level.

Tag: The Power of Paint

- DigiPen Student project “Tag”
  - https://www.digipen.edu/studentprojects/tag
- Change the properties of any surface by painting it
- Fits well with Portal’s game play
  - Traversal of an environment as a puzzle
- Fiction and visuals needed integration



It started with a DigiPen student project called Tag: the power of paint.

The base mechanic: use your paint gun to paint any surface in the game world.

Different colors confer different surface properties. A green-painted surface will make you bounce, a red one, give you super-speed.

The game is similar to Portal in an two important ways:

•Both games treat traversal of the environment as a puzzle.

•Both give you a tool to change the way you interact with your environment that enables you to get from point A to point B.

We enjoyed the game so much that we hired the creators to implement the mechanic in Portal 2.

Though the mechanic was a great fit for the Portal universe, the fiction and the visuals still needed integration.

Propulsion and Repulsion Gel

- Goals for the effect:
  - “Science-y”
  - Communicate thickness, elasticity
  - Do it all in texture space: no silhouette changes
  - Small footprint: we’re painting *everything*

In the fiction, huge stores of a mysterious, gelatinous substance are available for your unauthorized use.

Visually, to integrate with the Portal universe, the gel needed to be “science-y”.

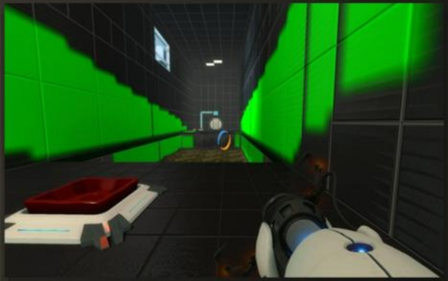
For gameplay, indicating a change in material properties was important. The visuals should somehow communicate thickness and elasticity.

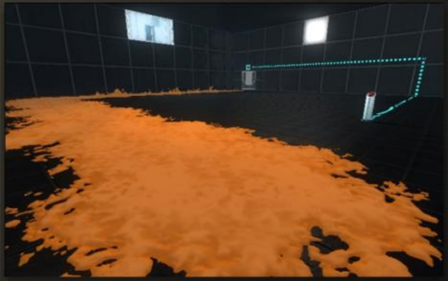
Performance-wise, we can’t procedurally generate meshes; we have decals in the Source engine, but they are short-lived to avoid growing memory use beyond hardware limits; gameplay can’t depend on them.

We wanted to create something as a material effect only, using tiling textures whenever possible, so that the memory footprint would be small.

Early Paint Iterations in Portal 2

- Tag and Portal 2 have fundamentally different requirements: need cheaper paint
- 1st iteration painted to lightmap coords, displayed color unmodified
  - Lightmaps well understood, perf-wise
  - Technology already in place
- 2nd iteration used a tiling texture that combined with the paint, then thresholded
  - Didn't meet goals for fiction/visuals





Tag and Portal 2 have fundamentally different requirements.

Tag used a unique parameterization for texturing their levels. Their art style allowed them to get away with this, so they could paint directly into the diffuse textures of their world.

Portal 2 is a more realistic environment, with several large levels. We could not afford to give up tiling textures.

The first iteration of paint shows what we are working with, under the hood. From here on in I'm going to refer to the low-resolution application you see here as the paintmap.

We use the same coordinates as our lightmaps, which allows us to use our own unique parameterization. It's much lower-resolution than Tag's implementation, however.

This solution was preferred because it was easy to understand its impact; we knew already how much lightmaps cost, so were able to predict the impact of doubling the required memory. We were able to adjust our texture budgets accordingly. It was also existing tech, which meant much of the work was already done.

Obviously, the visuals are not ideal.

The second iteration used a tiling normal map with an alpha channel. The alpha was combined with the alpha from the paintmap and thresholded to give the splatter effect.

While it was an improvement in resolution from the naked paintmap, it didn't meet our goals for the fiction or visuals.

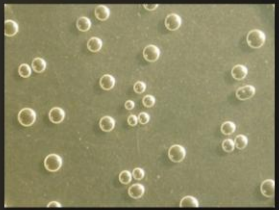
What's more, it obscured the surfaces beneath, which caused gameplay problems.

In Portal 2, surfaces either accept portals, or they don't. You can distinguish them visually. The gels do not interfere with your ability to place portals. This is because we wanted to prevent players painting surfaces necessary to solving the puzzle and getting stuck. It doesn't matter if the gel does not physically block portal placement, though, if you can no longer determine whether the underlying surface accepts them; you'll be stuck just the same.

Consequently our next implementation needed to be more translucent.

Inspiration

- Translucent thermoplastic elastomers
  - Like window clings
- Room-temperature vulcanizing elastomers
  - Like silicone rubber
- Bouncy balls
  - High elasticity
  - Do a reverse ball, air in an elastomer!
- Create a bubble-filled, gel-like splatter



Our original goal for the visuals was for the gel to look “science-y”. And what’s more science-y than plastics?

We took inspiration from translucent thermoplastic elastomers, which are great because they’re translucent but also fun: you can make toys out of them…like window clings!

Since heat is not part of the fictional application, we also looked at the less translucent room-temperature vulcanizing elastomers, like silicone rubber.

And how about communicating the elasticity of the substance? One of the gels causes you to bounce off the surface much higher than you can jump, which made us think of super bouncy balls.

How about embedding some in the paint, creating a bunch of reverse balls: air in an elastomer?

Bubbles would be great for communicating thickness, too, helping sell the idea that the gel is applied thickly enough to change the surface properties significantly.

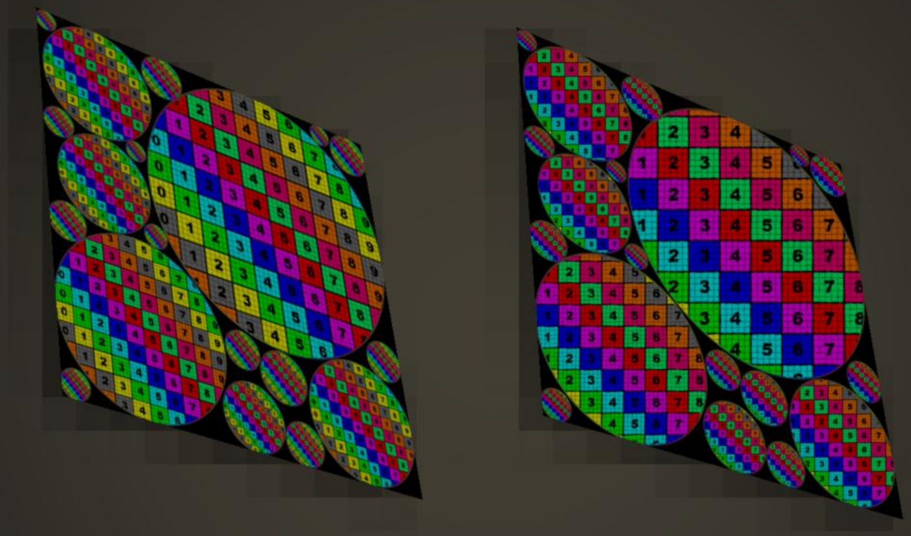
Bubbles

- Should look round from every angle
- Need a solution for sprinkling them through the gel without spending extra verts
- Need a sprite-like behavior that happens only in the pixel shader

If we're going to make bubbles, though, we've got to do it right. They must look round from every angle, like a sprite would.

But we've already nixed decals, which are far cheaper than sprinkling around several thousand camera-facing planes. So sprites are out.

We knew already we wanted the effect to happen only in the material. What we needed was a sprite-like behavior that would happen in the pixel shader.



Surface-Embedded Sprites Example

Transforming contiguous UV areas to be camera-facing

The technique forces the contents of the UV islands to appear to face the camera.

This is a quick visualization of the technique created in RenderMonkey. On the left, the contents of the UV islands are untransformed. On the right, they are transformed to appear camera-facing.

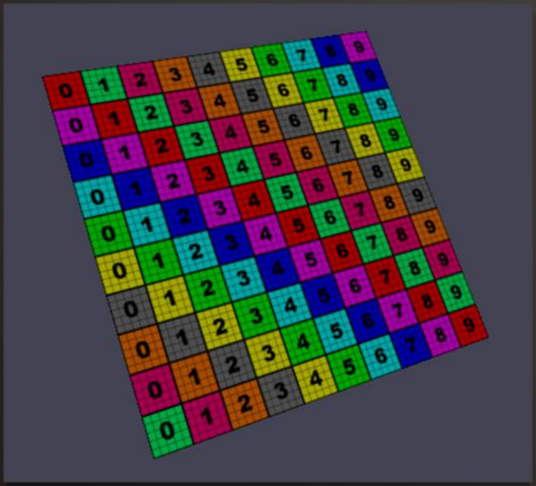
You can see that the "sprite" always clips at the edges of the layout islands, as though you are looking through a porthole at the texture.

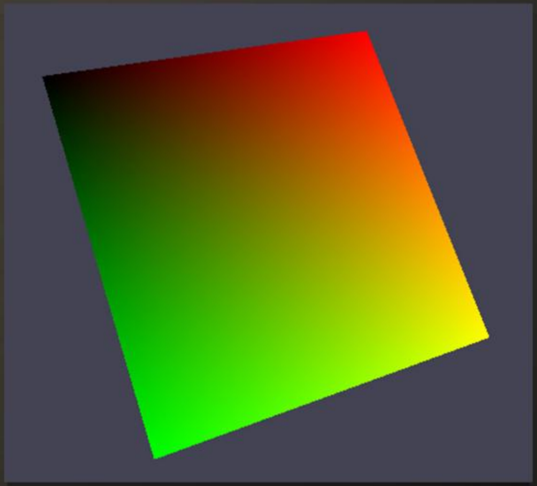
When we use this technique, we'll make sure the texture is padded well enough to stay away from the boundaries.

On top of that, we can fade the texture out at really low angles and use a mask with soft edges to reduce the abruptness and severity of the clipping.

Visualize UV as RG(B)

- R values represent the U-component of the UV vector
- G values represent V



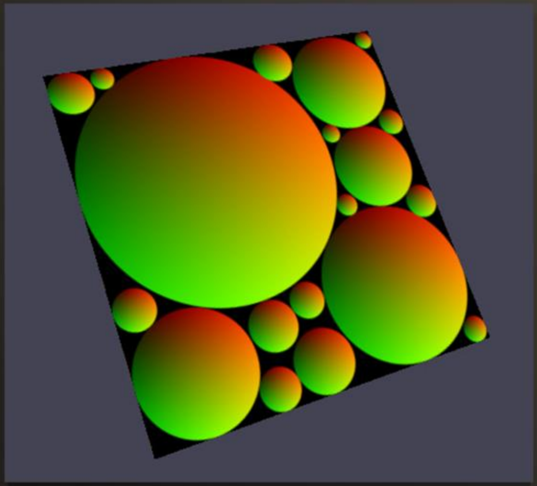


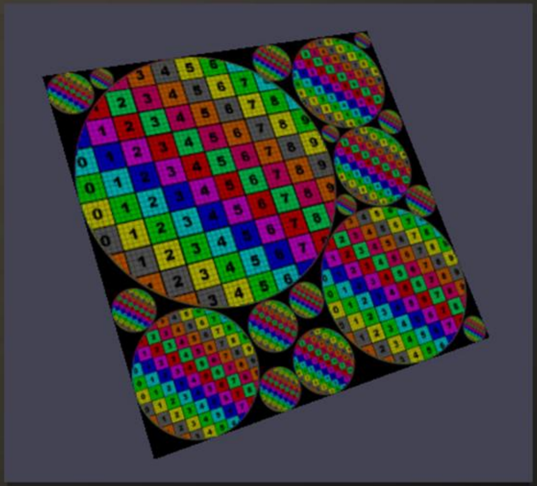
On the left we have a textured plane, unwrapped between 0..1.

On the right, we are visualizing the texture coordinates of the plane as color.

Visualize UV as RG(B)

- Do the reverse: create a texture where the red and green channels represent U and V
- Use the sampled RG as texture coordinates





Doing the reverse is just as simple.

The previous technique (flow maps) used vectors read from a texture to displace existing UVs. In this technique, we're going to flat-out replace them.

This allows us to control, through a texture, the UV layout on our surface.

Camera-Facing UVs

- Construct a matrix in tangent space to remove the perspective transformation

VS

$$\begin{bmatrix} \overrightarrow{v_s} \cdot \mathrm{T} \\ 0 \\ \overrightarrow{v_f} \cdot \mathrm{T} \end{bmatrix}$$

$$\mathrm{M}'_{[1]} = \mathrm{M}_{[0]} \times \mathrm{M}_{[2]}$$

$$\mathrm{M}'_{[0]} = \mathrm{M}_{[2]} \times \mathrm{M}'_{[1]}$$

$$\mathrm{M}'' = \begin{bmatrix} \mathrm{M}'_{[0,0]} & \mathrm{M}'_{[0,1]} \\ \mathrm{M}'_{[1,0]} & \mathrm{M}'_{[1,1]} \end{bmatrix}$$

To get the textured areas to appear to face the camera, we're going to do a transformation that's very similar to what you'd do to a sprite's geometry.

But we're going to do it in tangent space, not world space.

We construct a matrix in tangent space that uses information about the camera as entries.

v-sub-s is the camera side vector

v-sub-f is view forward vector, not to be confused with the camera forward vector. The camera-forward vector is static per frame. Instead, we're going to take the vector from the view position to each pixel on the surface, so that it varies over the surface.

T is the world-to-tangent matrix; as I said, this matrix needs to be applied in texture space.

The missing entry is filled by taking the cross product of the two known vectors. We then orthonormalize (incidentally also forcing the view side vector to vary over the surface), producing matrix M'.

M'' is M' truncated: we don't need any z-entries or anything that would affect the z-entries: we're working with two-component texture coordinates.

It's important to keep M'' small by removing any entries we don't need: we're passing it from the vertex shader to the pixel shader, and interpolators aren't free!

Camera-Facing UVs

- Apply the matrix to generate the transformed UVs

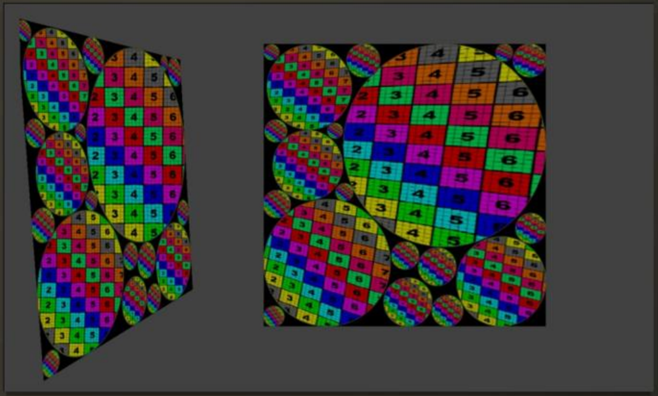
PS

$$\vec{t'} = \vec{t} - [0.5, 0.5]$$

$$\vec{t''}_{[0]} = M''_{[0]} \cdot \vec{t'}$$

$$\vec{t''}_{[1]} = M''_{[1]} \cdot \vec{t'}$$

$$\overrightarrow{t_{final}} = \vec{t''} + [0.5, 0.5]$$



The texture coordinates are pre-transformed so that the center of the UV island is [0,0]

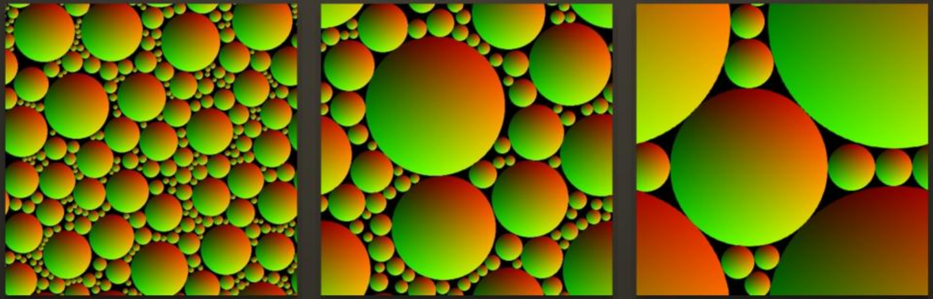
This allows the origin of the transformation to be the center of the island, so the sprites will appear to be pinned at their centers instead of their top left corners.

The matrix is applied simply using the dot product.

t’ is then post-transformed so that the center is once again [0.5,0.5], so that the texture inside the UV islands are not offset.

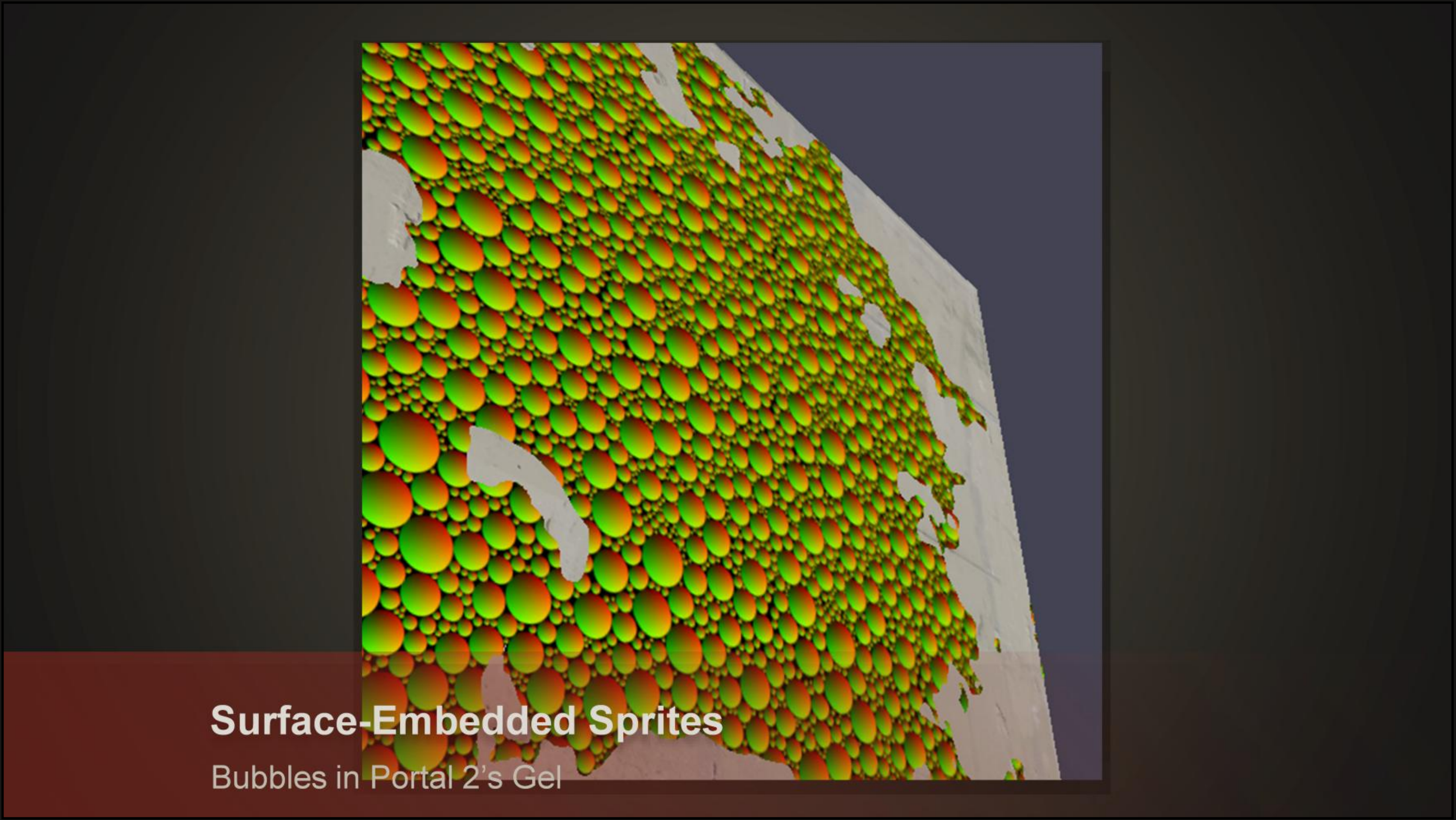
The Layout Texture

- Not difficult to hand-author
  - requires filling areas with a four-point linear gradient
- Procedural options abound
  - Placing RG-gradient textured geometry in a 3d content editor (Houdini, Maya, Max) then rendering to texture
  - Blit using any graphics API (Processing, Python)



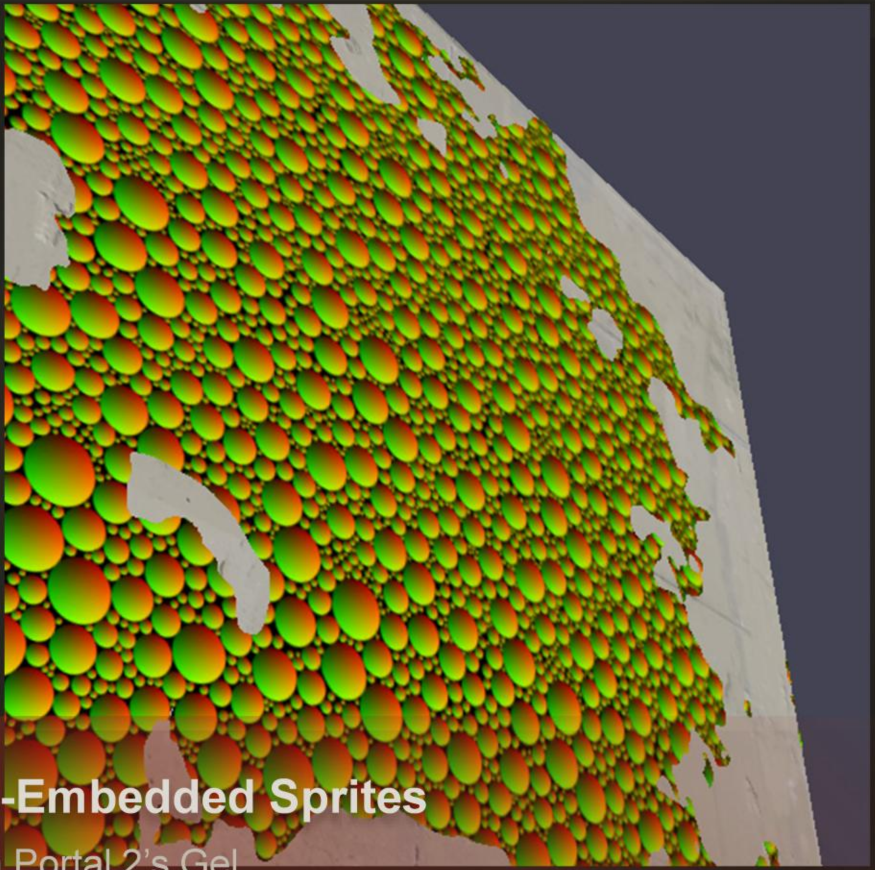
Unlike the flow maps, UV layout textures are not difficult to hand author. They're also easy to procedurally generate in all sorts of applications.

These layout maps were generated in Processing (a java-based application) using a combination of randomized placement and the Appolonian Gasket fractal to generate tiling layouts with various size settings.



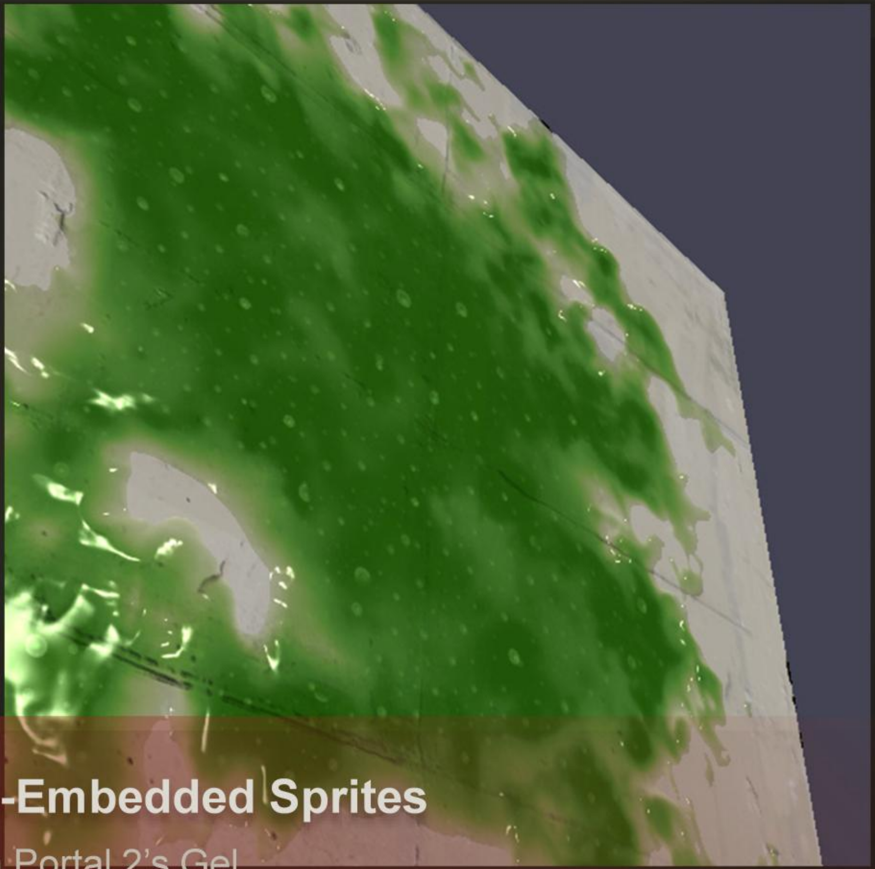
Surface-Embedded Sprites

Bubbles in Portal 2's Gel



To illustrate the use of the surface-embedded sprites in the gel, we'll look at the original prototype that was created in RenderMonkey.

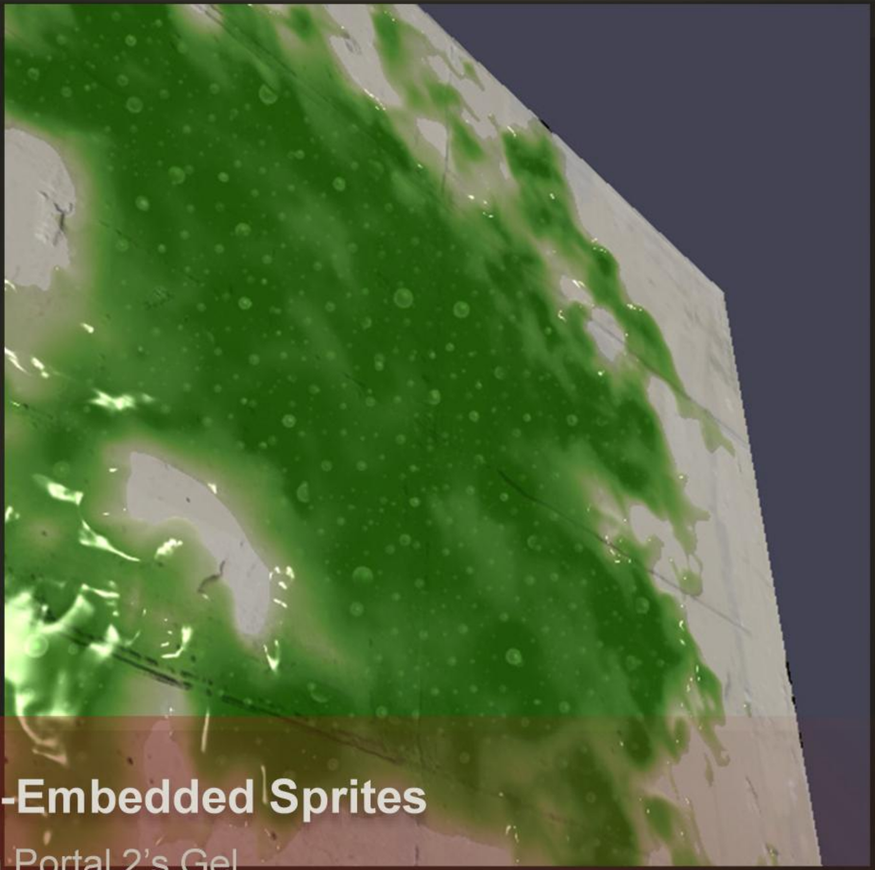
We first tile the layout texture over the surface.



Surface-Embedded Sprites

Bubbles in Portal 2’s Gel

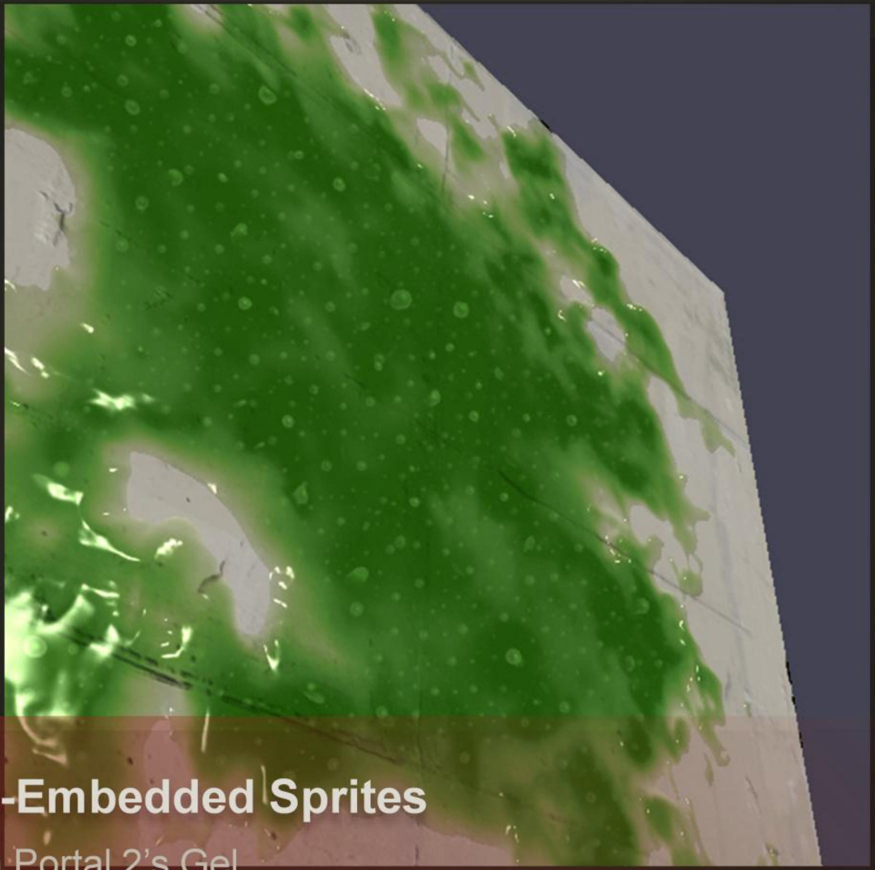
We use the layout texture to look up the bubble texture, which we use to affect normals, reflections, and opacity; here it is, pre-transform.



Surface-Embedded Sprites

Bubbles in Portal 2's Gel

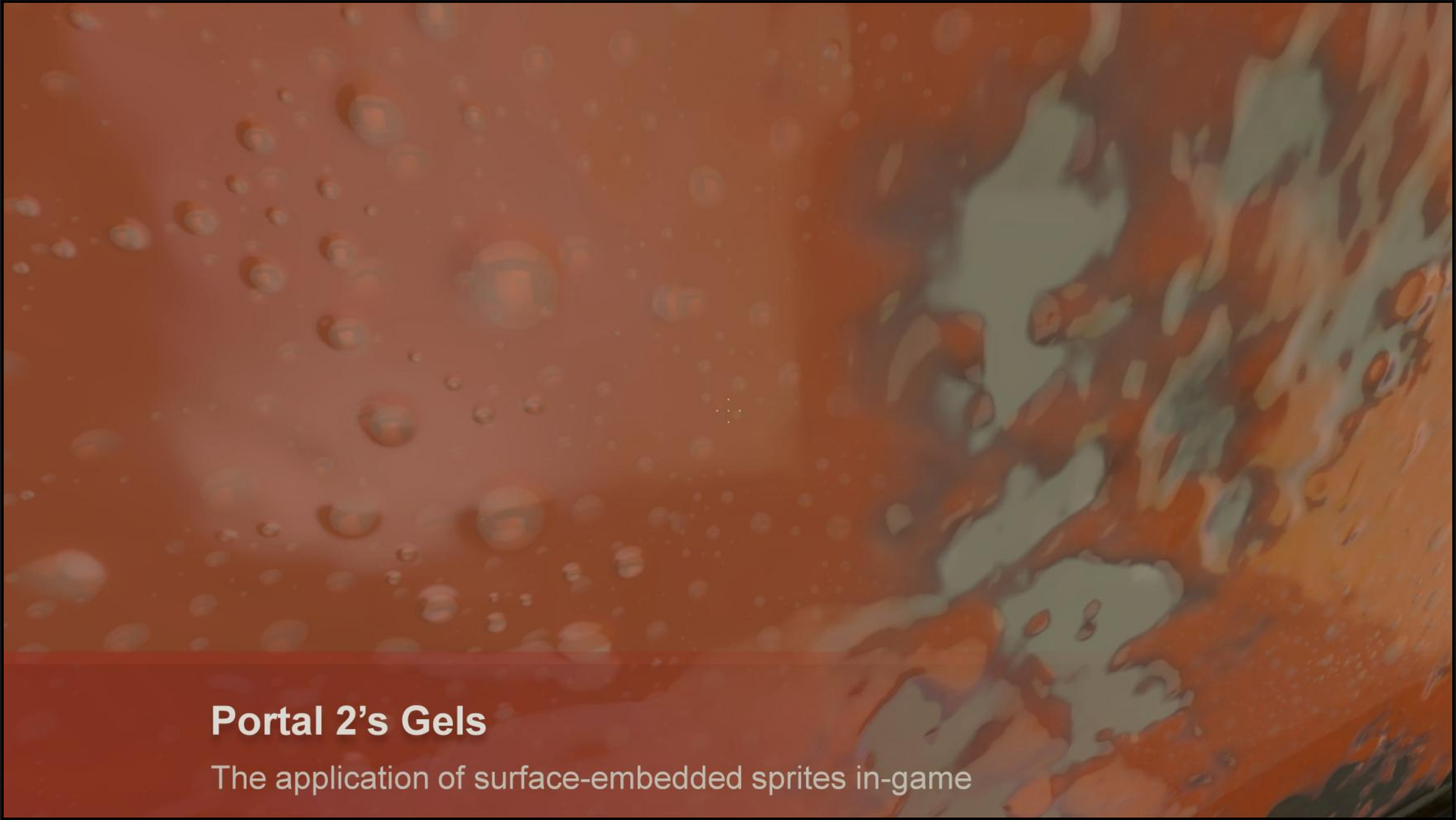
Here it is, post transform: the bubbles regain their roundness.



Surface-Embedded Sprites

Bubbles in Portal 2’s Gel

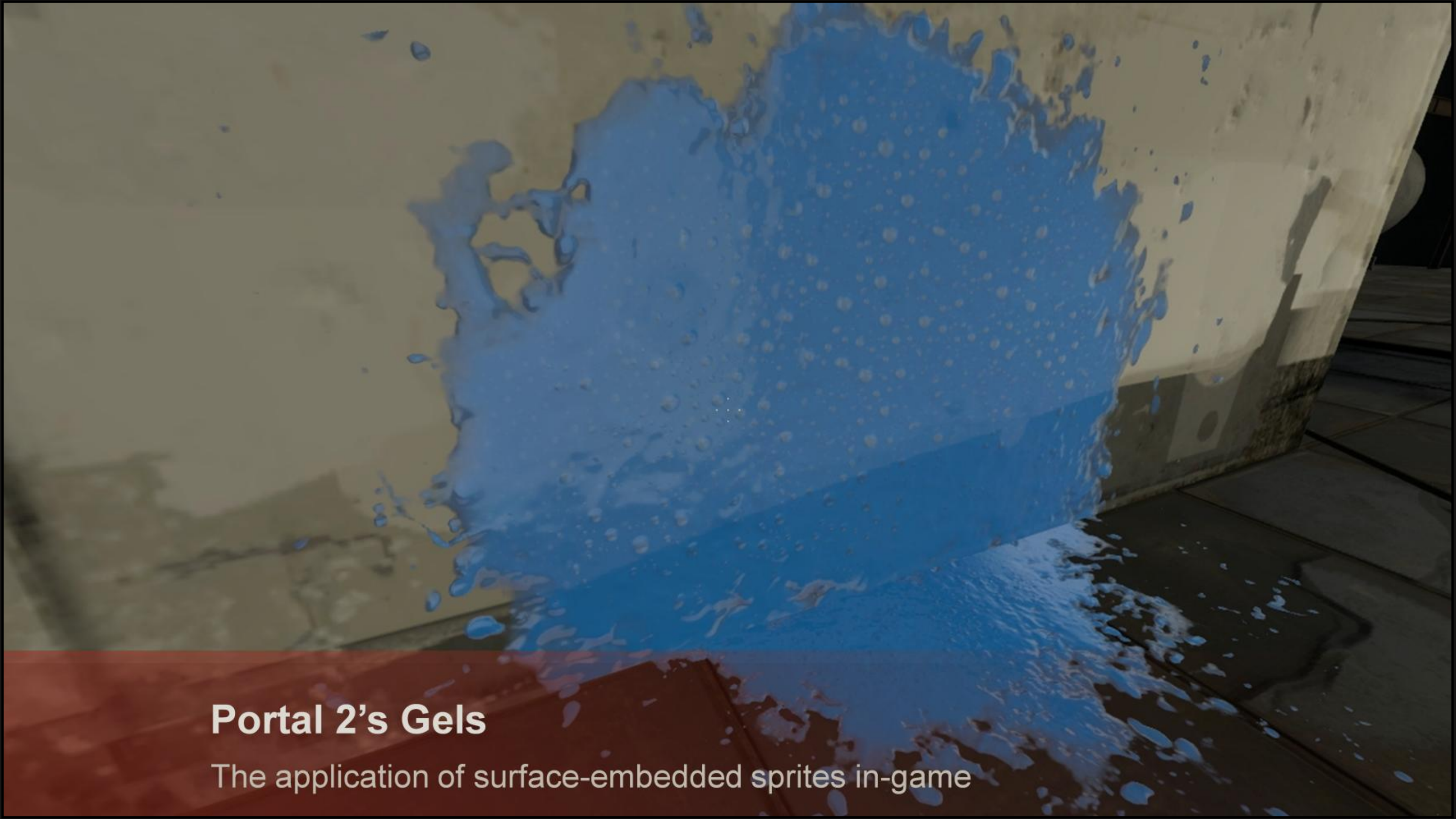
We can also affect the lookup by refraction so that the bubbles sit better *inside* the surface.



Portal 2’s Gels

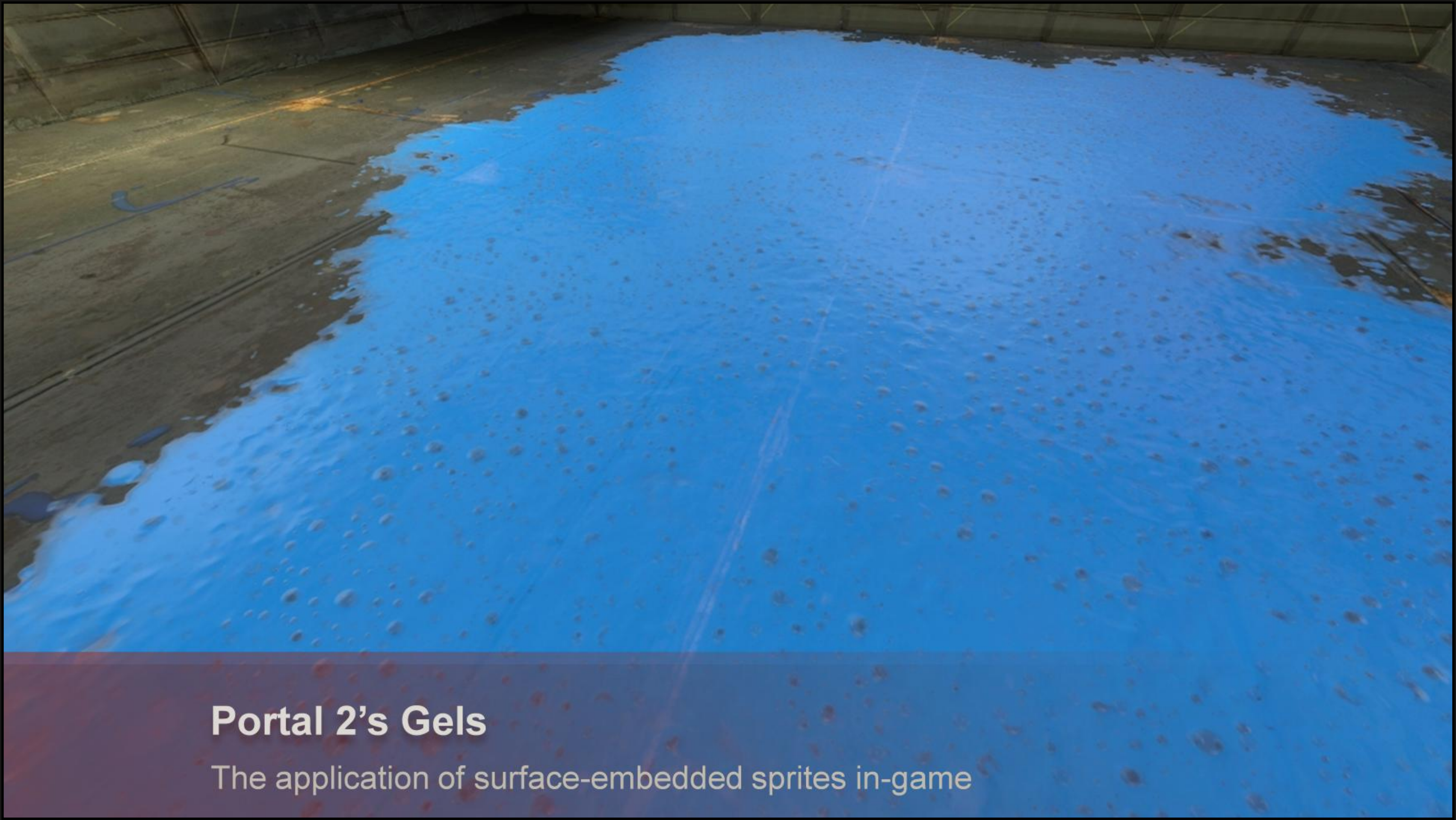
The application of surface-embedded sprites in-game

Previously we’ve seen the gel prototype. Here are some screenshots of the final result in-game.



Portal 2’s Gels

The application of surface-embedded sprites in-game



Portal 2’s Gels

The application of surface-embedded sprites in-game

It’s important that the gel is an interesting-looking surface. With patience, players can cover most of a level in gel.

Result

- Interesting surface that, sometimes, covers the entire screen
- Extremely low transform cost

The transform cost is very low, especially since we’re using this technique on Portal chambers: very blocky geometry with few verts.

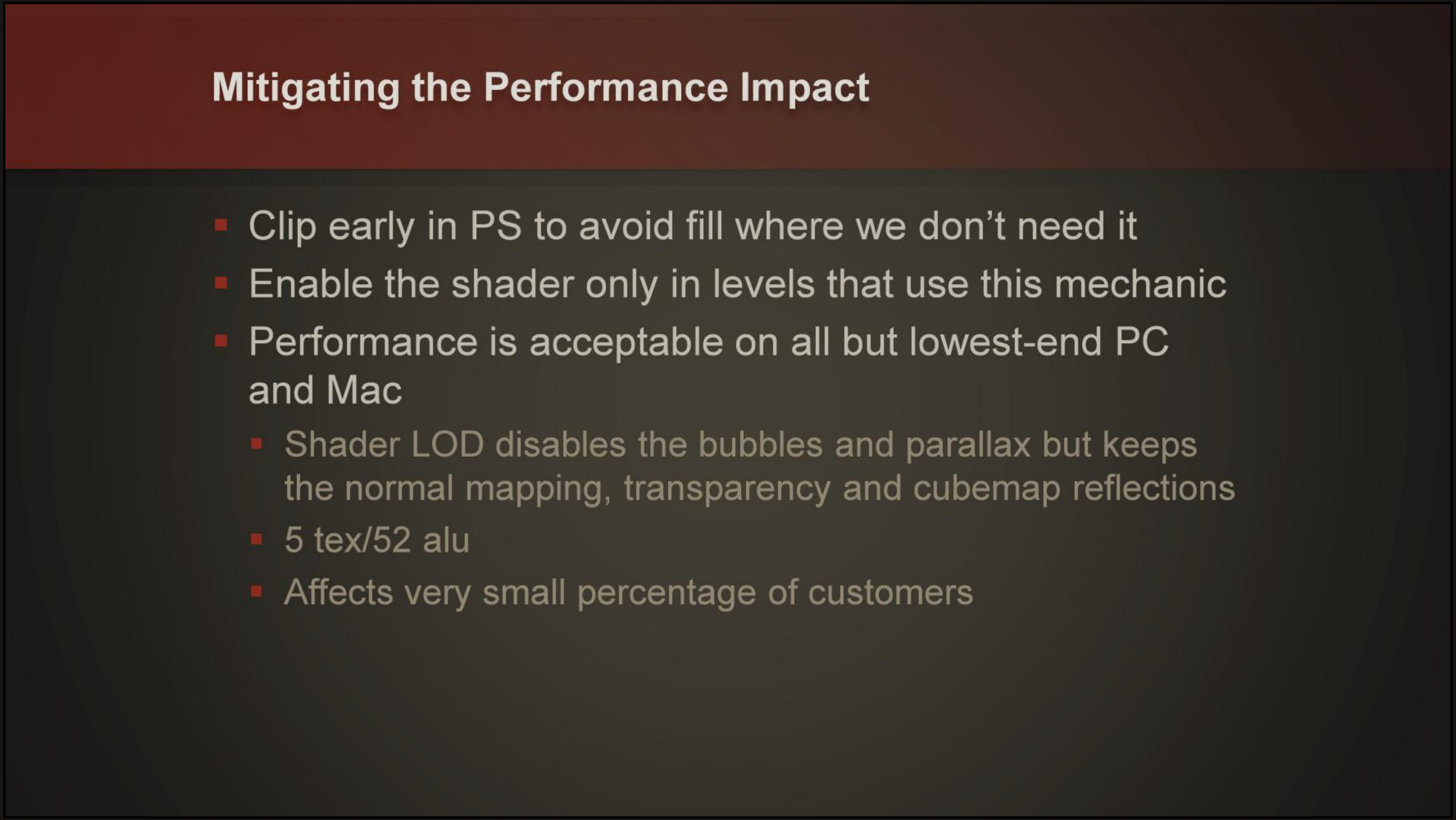
Result

- High fill cost: 171 arithmetic instructions, 15 tex
  - WHAT! 15 texture lookups?!
  - Not all of those lookups are for the embedded sprites (We're getting away with all sorts of other fanciness)
    - We do two layers of bubbles to increase the sense of depth through parallax, so all the cost is doubled
    - There is the combination and thresholding of the opacity maps
    - There is single-step parallax on the normal map
    - There is refraction of the underlying surface

On the other hand, the fill cost is very high.

Not all of those instructions are used for the embedded sprites, though, and that the cost for the sprites is doubled due to the two layers of bubbles, which was an artistic choice.

We're doing a lookup on the normal map, single step parallaxing on the normal map, refraction of the underlying surface, cubemap reflections off the bubbles and off the surface of the gel…



Mitigating the Performance Impact

- Clip early in PS to avoid fill where we don’t need it
- Enable the shader only in levels that use this mechanic
- Performance is acceptable on all but lowest-end PC and Mac
  - Shader LOD disables the bubbles and parallax but keeps the normal mapping, transparency and cubemap reflections
  - 5 tex/52 alu
  - Affects very small percentage of customers

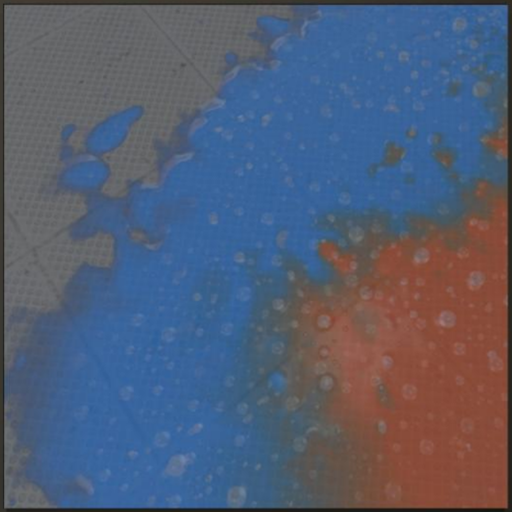
Because the fill cost is so high, we take steps to mitigate it. We clip early in the pixel shader when alpha is below a certain threshold to avoid extra cost where we don’t need to spend it.

The shader is enabled on world geometry only in levels that use this mechanic.

Performance is acceptable on all but lowest-end PC and Mac, so like the emancipation grid, we fall back to a shader LOD that is less complex.

Conclusions: Sludge and Gel

- Both techniques use specialized textures to modify UVs through dependent texture fetches



Conclusions

- Dependent texture fetches can be costly
  - Particularly if the coherence of the fetches is very low
- Can still be a good choice if it is relatively cheaper than other options
  - Water flow might not even have been possible given any other technique: Flow geometry? Lots of sprites?
  - Bubbles would have been too expensive to use if sprites had to be generated and transformed
- Both techniques shown require specialized textures
  - But they can be authored with relative ease

Thank You!

Bronwen Grimes, Valve
bronwen@valvesoftware.com

Links of interest:
https://www.digipen.edu/studentprojects/tag
www.valvesoftware.com/company/publications.html