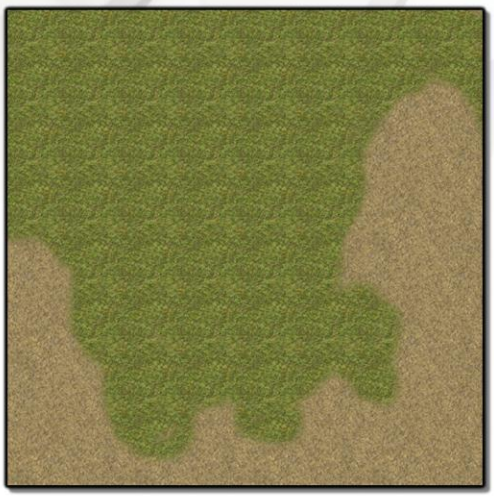
CARBON

Realtime texture quilting

Hugh Malan, Jeff Cairns

What problem does this solve?



It's difficult to avoid tiling with large single-material surfaces
The usual way this problem shows up these days is with layered texture shaders. Even though different materials show through in different planes, repeating features tend to be pretty obvious if you stand back from the wall or fly up above the terrain and look down.

What problem does this solve?

CARBON

Plausible transitions between materials need an alphablended overlay or careful mapping
It's easiest and simplest to just assign materials to groups of triangles. This creates abrupt borders between materials, which is often unrealistic. Eg: The transition from asphalt to gravel, as shown in the first image.
Creating a plausible transition in these cases means either an alpha-blended overlay, or painting transition features (like the road edge) into the texture and carefully UV mapping it.

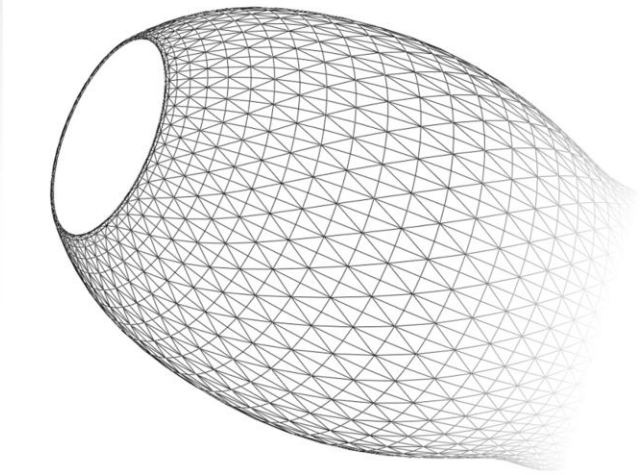
Painting effects into textures due to geometry create hard restrictions
Edge effects are not problem for uniquely-mapped objects like props and prefabs, but unique textures aren't affordable for large geometry.
Painting the features into the textures imposes hard restrictions. For instance, having a stained concrete wall texture like the one shown, means floor and ceiling have to be parallel, and must be a fixed distance apart.
It also isn't an option for scenes with lots of unique shapes. Eg: Floors

In summary, painting effects due to the geometry into textures needs careful planning and creates some difficult restrictions.

This means it's not possible for the designers or modellers to arbitrarily move geometric features around or change geometry dimensions, because it might mess up the carefully-placed textures.

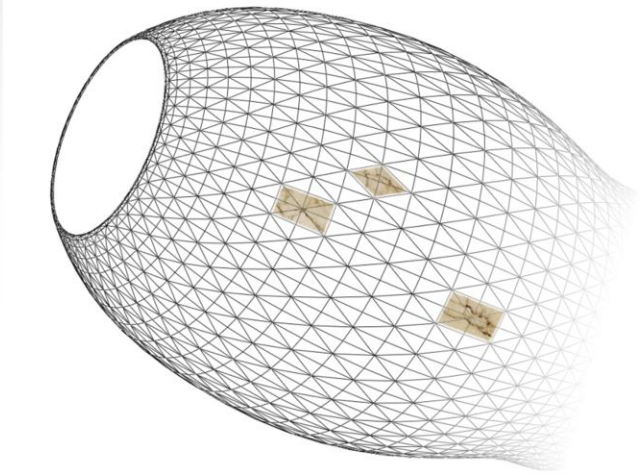
Building the environment out of instanced prefabs sidesteps this problem to a certain extent, but all the problems reappear at the places where prefabs meet.

A splat at each vertex



The tech works by placing what we're calling splats on the surface of the model. A “splat” is a region copied from the source image. We place a splat at every vertex of the model. The splat at a particular vertex covers all the triangles that reference that vertex.

A splat at each vertex



When the model is covered with a splat at each vertex it's a bit difficult to visualize what's going on - so, here's our example model with just a few of the splats shown.

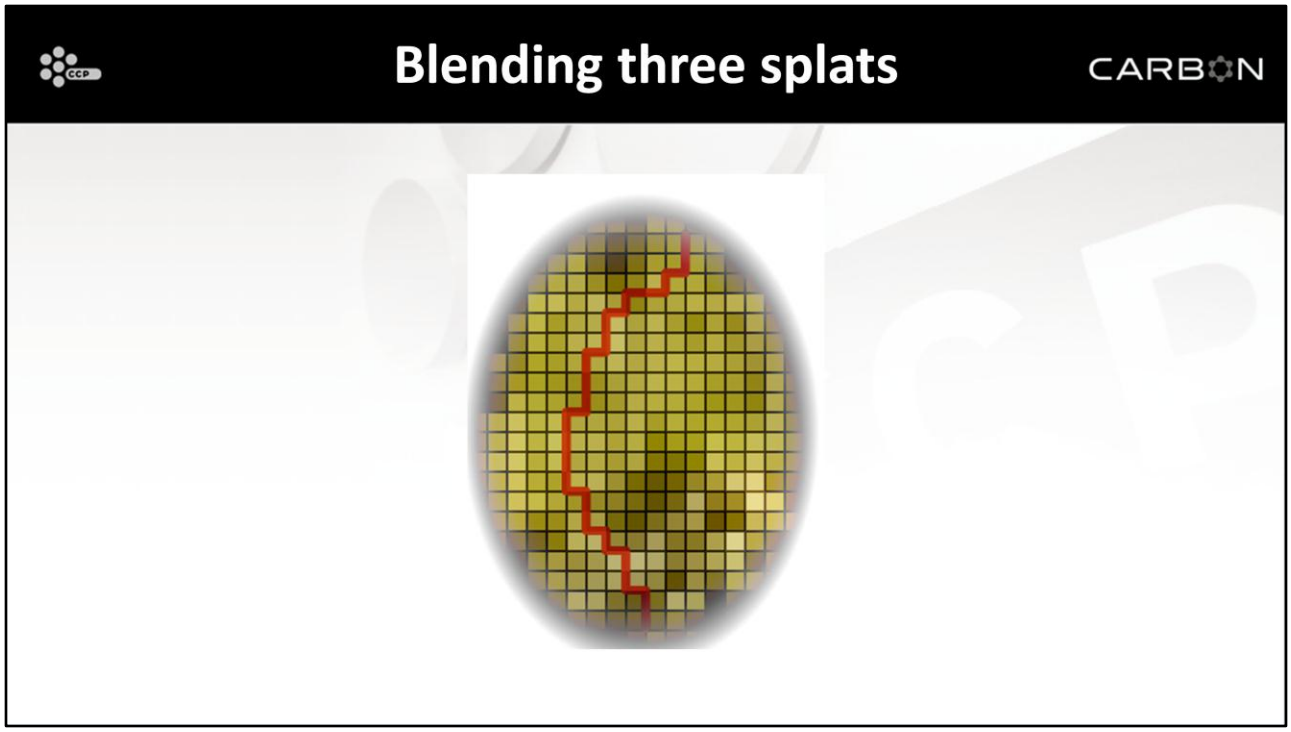
The big advantage of the splat-per-vertex approach is that every pixel and triangle is covered by precisely three splats. Fewer overlapping splats is better, because each additional splat means more texture samples and more pixel shader work. A fixed number also means the compositing logic is much simpler than if that number could vary.

Another big advantage is that at every point on the model surface there is at least one well-defined splat covering it: we never have any points that are dangerously close to the boundaries of all the three splats covering it.

We're able to encode all of our placement data in the vertex data. This has a couple of big advantages over encoding the placement data in textures. If we stored splat placement data in a UV-mapped texture (the specifics don't really matter) it means we'd have to worry about all the standard problems UV mapping brings. For instance, it's difficult to have features flow continuously over a UV seam.

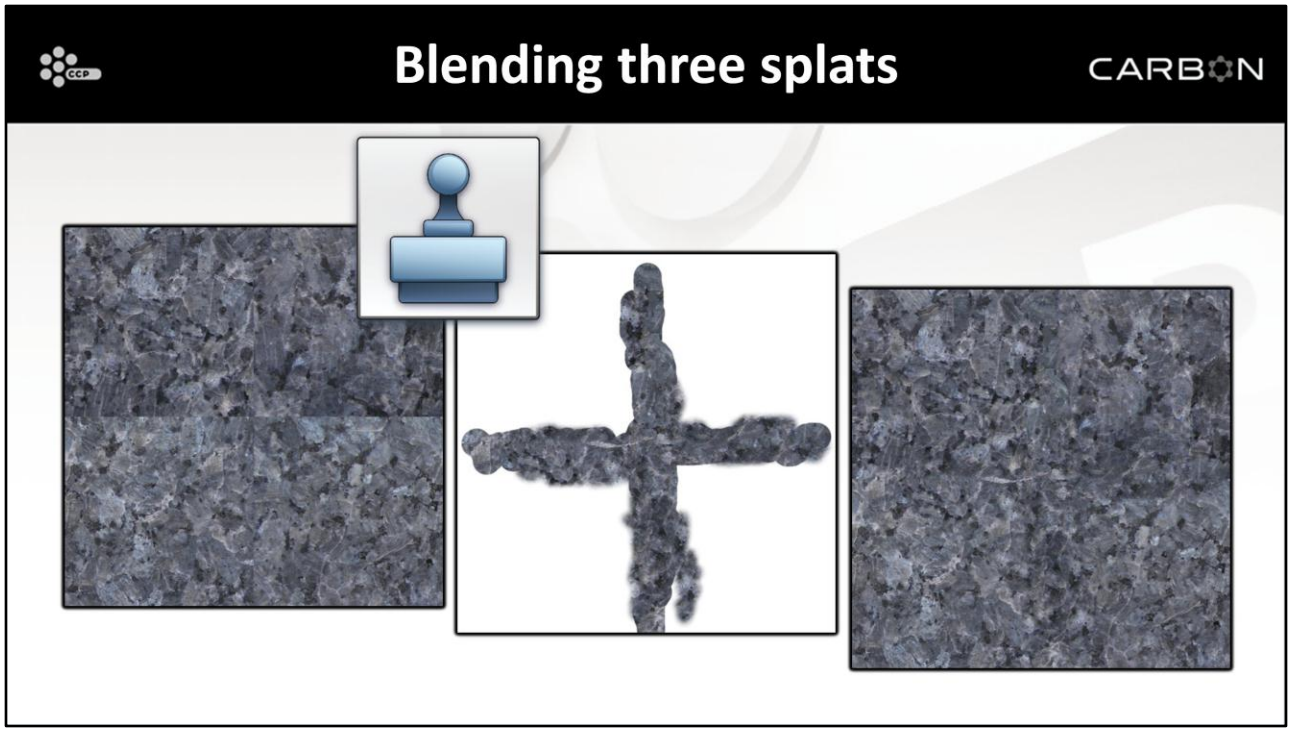
Encoding it in the verts means we can cope with any topology, and will never have seams. We will have to cope with varying vertex density – for instance, the verts on the left of the model are much closer together than the verts on the right – but that's OK.

One last thing to point out is that this approach won't add, remove or change the geometry in any way. This is pretty important when working with artist-created geometry in production.



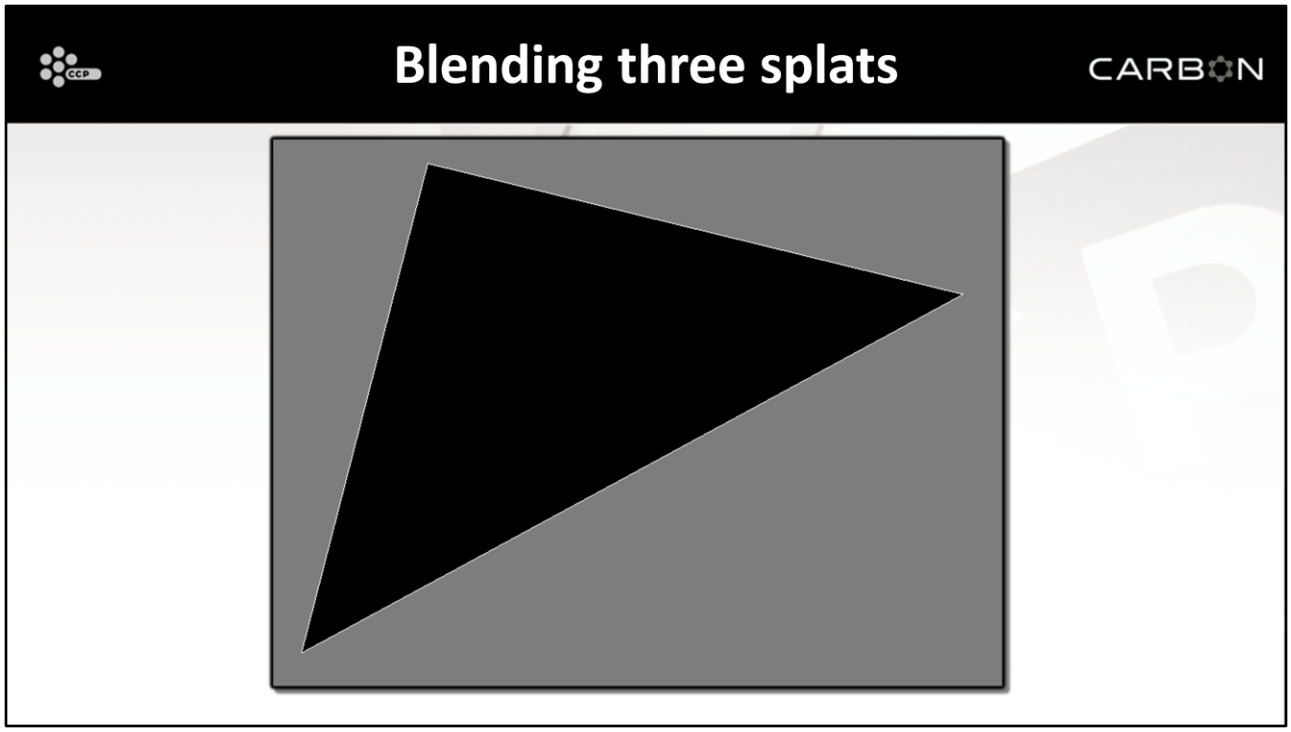
So the question is, how the pixel shader determines where the edge of one splat is and where the next one begins.

There's quite a lot of literature on texture by example and texture quilting, which describe methods for automatically finding a border that makes the transition from one splat to another as seamless as possible. Unfortunately, none of the techniques are really a natural fit to a single-pass pixel shader.



The main inspiration for this tech is the clone-paint tool. When working with textures, it is possible to clonepaint out seams or other problem features with some pretty rough work – as seen in the slide, where the hard edges on the left are covered up with a very rough clonepainted overlay shown in the middle; the final result is on the right.

The point of this slide is to demonstrate that we can get away with something simple.



The slide shows a single triangle. There will be three splats covering it. Each of the splats is centered on a different vertex.

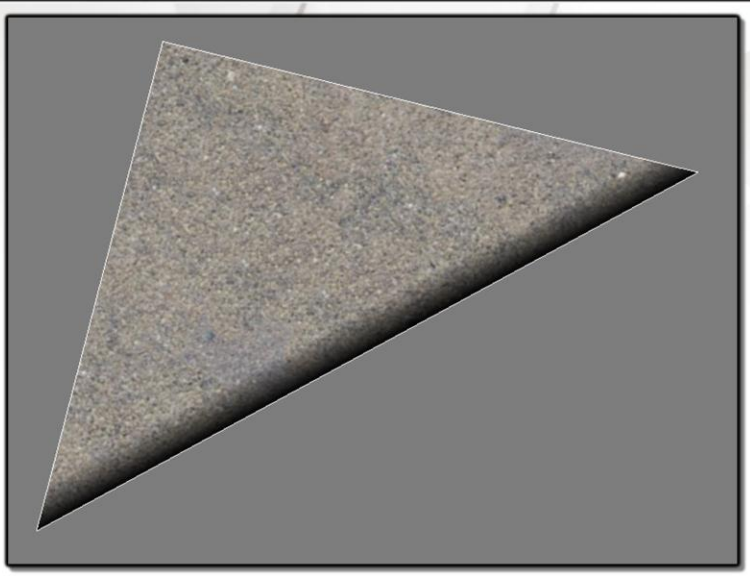
The pixel shader's job is to sample the three relevant splat textures, and efficiently figure out three blend coefficients for combining the splats.

The blend logic needs to fade each splat out at its edges.

To arrange this fadeout, we want to set up a strength factor that's 1 at the center of the splat, and 0 at the edge. Since all the splats are centered at vertices, and their boundaries are triangle edges, this is very straightforward: we can encode the strength factor as a per-vertex value.

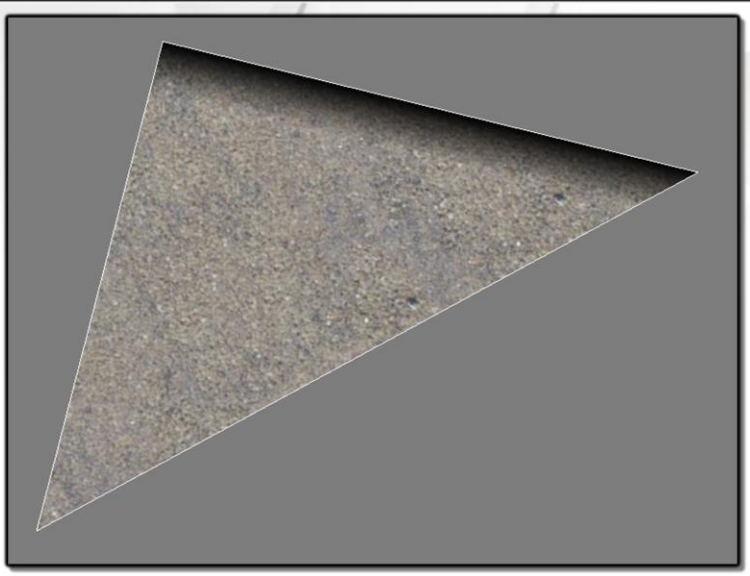
Here's each of the three relevant splats , with a fade to black indicating where the edge of the splat is.

Splat 1 of 3



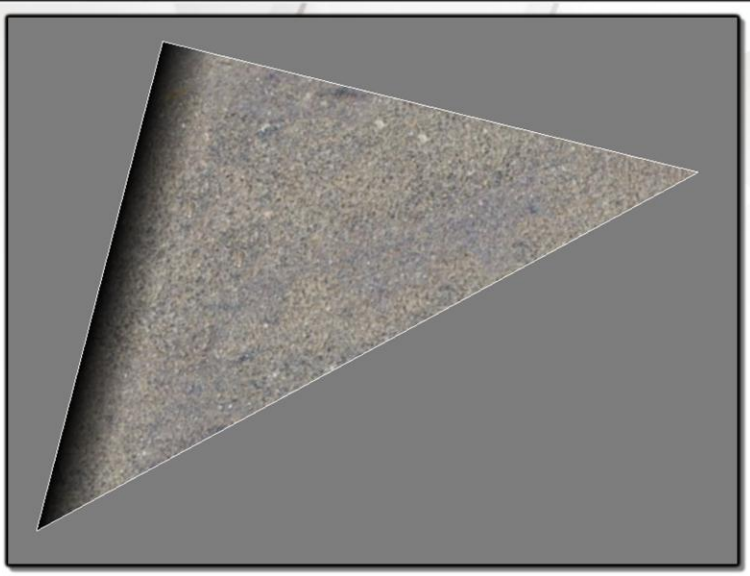
Here's the first of the three splats. It is centered on the top vertex, so it needs to fade out on the bottom right edge.

Splat 2 of 3



The second splat is centered on the bottom left vertex, and has to fade out before the top edge.

Splat 3 of 3



The third splat is centered on the righthand vertex and needs to fade out before the lefthand edge.

Color and blend maps

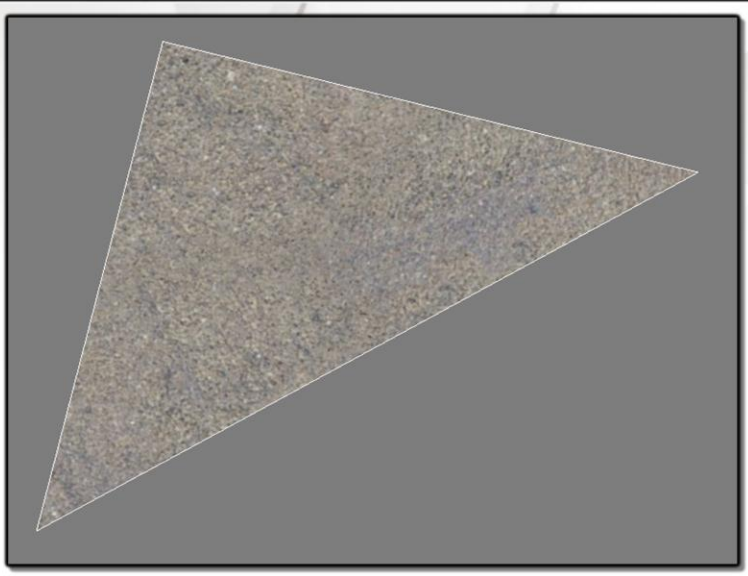
Another thing the blend logic should allow is some artist control over the blends.

We experimented with a bunch of different ways to accomplish this, and the best and most efficient solution we found was this.

In addition to the color map (as shown on the left), the artists provide a strength map (shown on the right). For each of the three splats, the pixel shader samples the strength map, and multiples it by the per-vertex strength value to provide the final strength value for the splat.

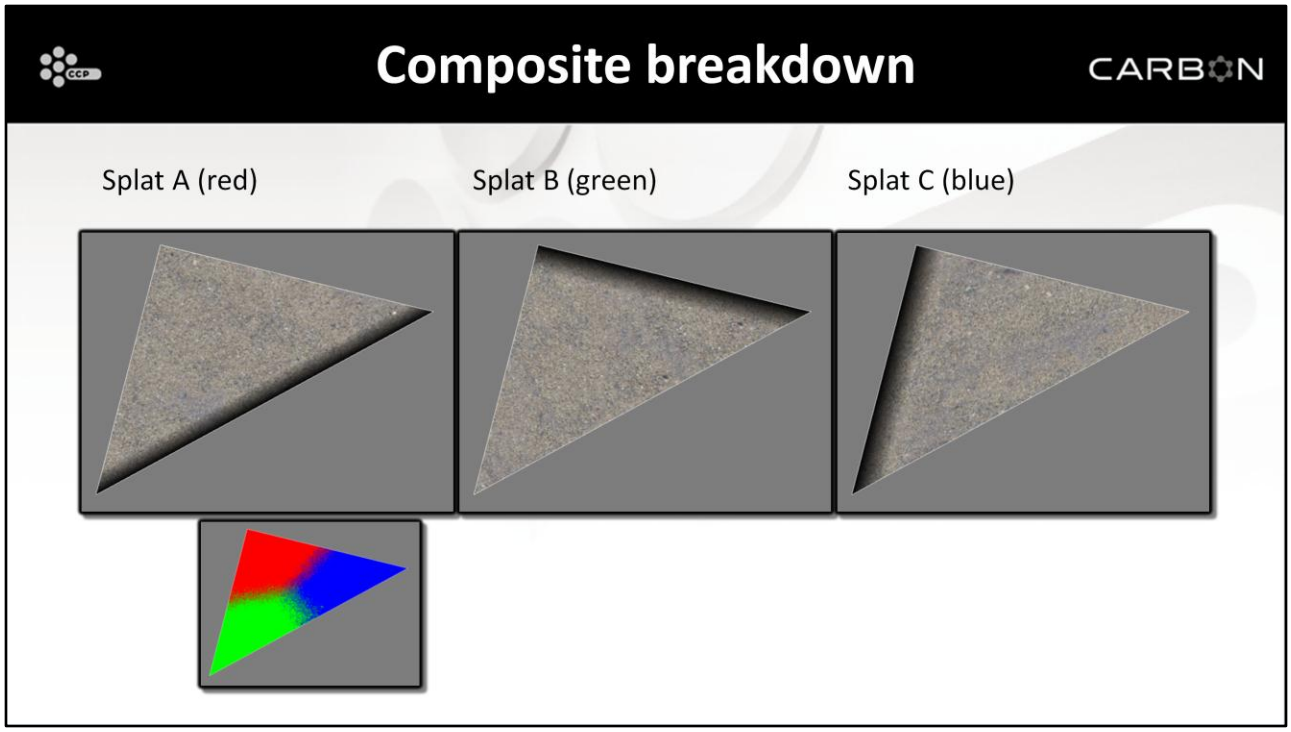
I'll show the results for this triangle, and then go over the details of how we calculate the blend coeffients from the data presented so far.

Final composite



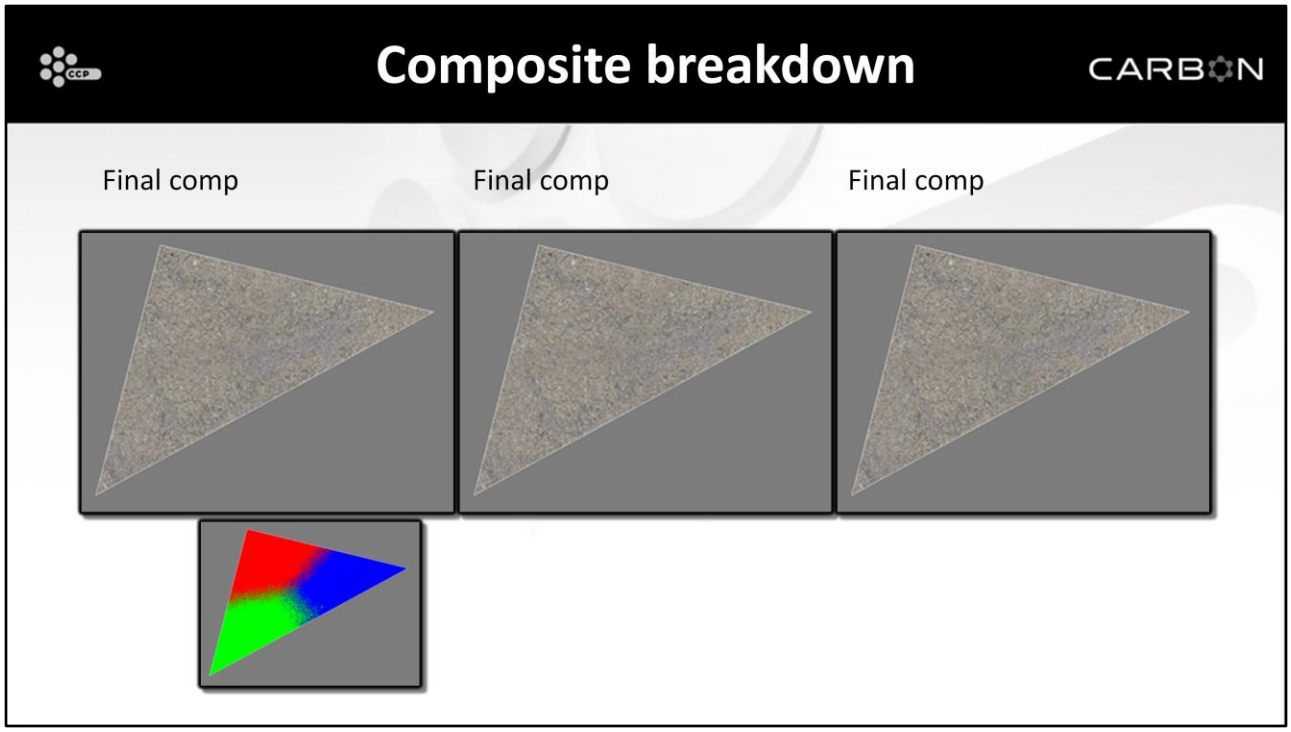
Here's the result. The three splats are blended together, and hopefully the seams are difficult or impossible to spot.

I'll show a breakdown the composite so the contribution of each splat is visible.



Here's each of the three splats...

The RGB triangle at the bottom shows the final blend coefficients. The R component specifies the contribution of the first splat; the green component the contribution of the second splat, and the blue component the contribution of the third splat.



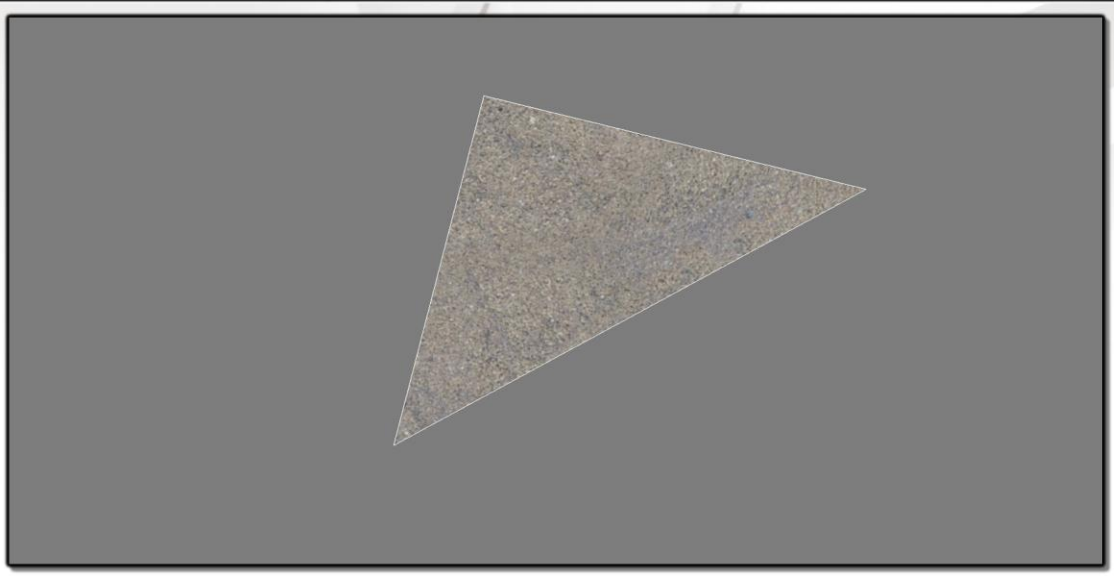
… and here's the final composite, for comparison. Flip back and forth between the two slides to see which splats contribute to which parts of the triangle.

The first splat should be used in the top left, which is 100% red in the blend image at the bottom. The second splat is used at the bottom; the third splat is used on the righthand side of the triangle.

Note that the transition between the different splats is interesting and complex, it's not a simple linear blend or anything.

Extension to the rest of the mesh

CARBON



Now we'll see how it looks when it's extended to the rest of the mesh. Here's the example triangle we've been looking at ...

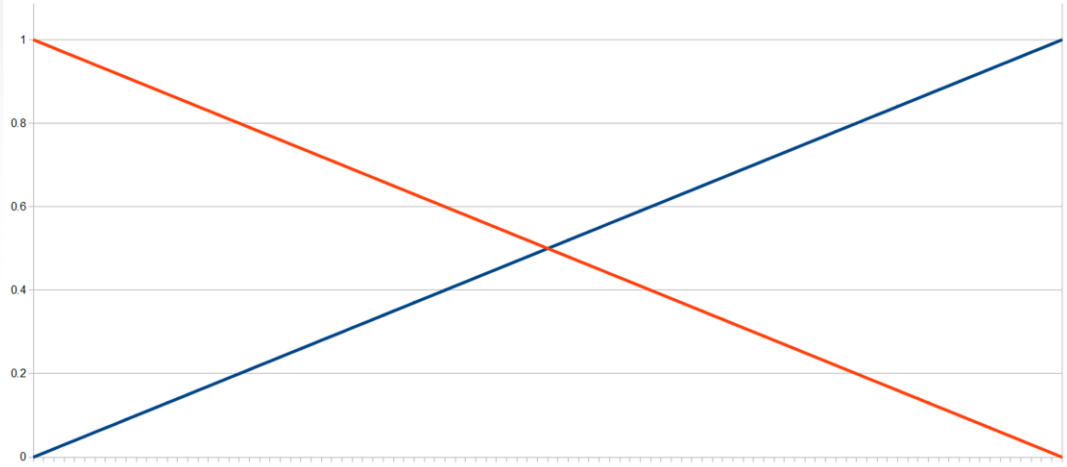
Extension to the rest of the mesh

CARBON

… and here's the rest of the mesh. Hopefully the blends and seams are difficult to see. Some repeated features are visible, but they're not periodic, and they appear in random orientations so it's a lot more difficult to pick out than texture repeats.

I'll now go into the details of how the splats are blended in the pixel shader.

Initial splat strength

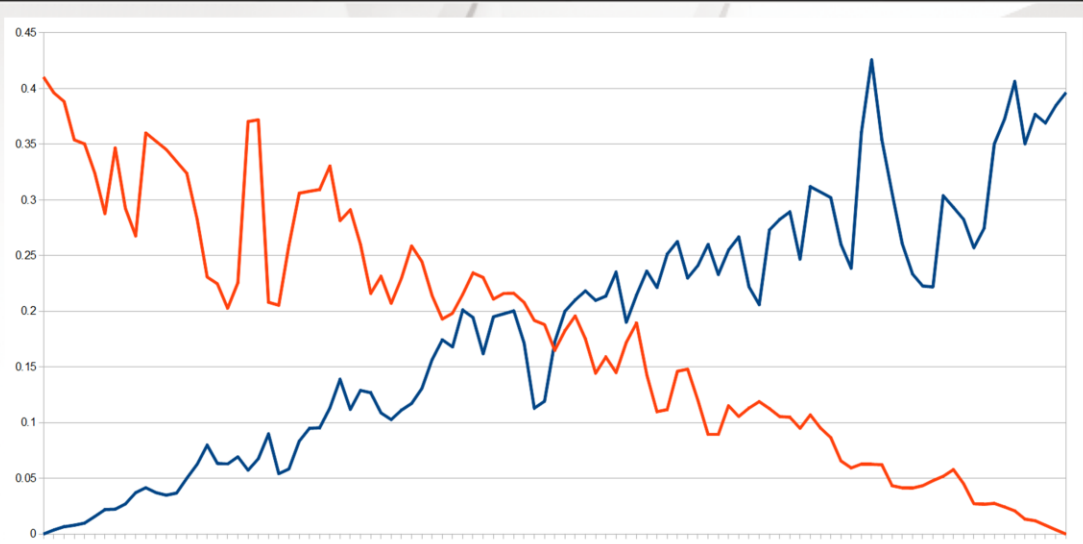


I'll work through an example with strength values on a 1d instead of 2d domain. This example only shows how two raw strength values are converted to normalized weights; but it should be pretty clear how to generalize to three or any other number of contributing strength values.

So for each of the three splats we've got a per-vertex value that's 1 at the center of the splat and linearly 0 at the edge, and we've also an artist-provided texturemap that indicates the strength of each point.

This slide shows the per-vertex value. On the far lefthand side we're at a vertex; the splat centered there has a the orange strength graph. On the far righthand side we've moved to the other vertex; the splat centered there has the blue graph. Points along the X axis correspond to points on the path between the two vertices.

… scaled by artist blend map

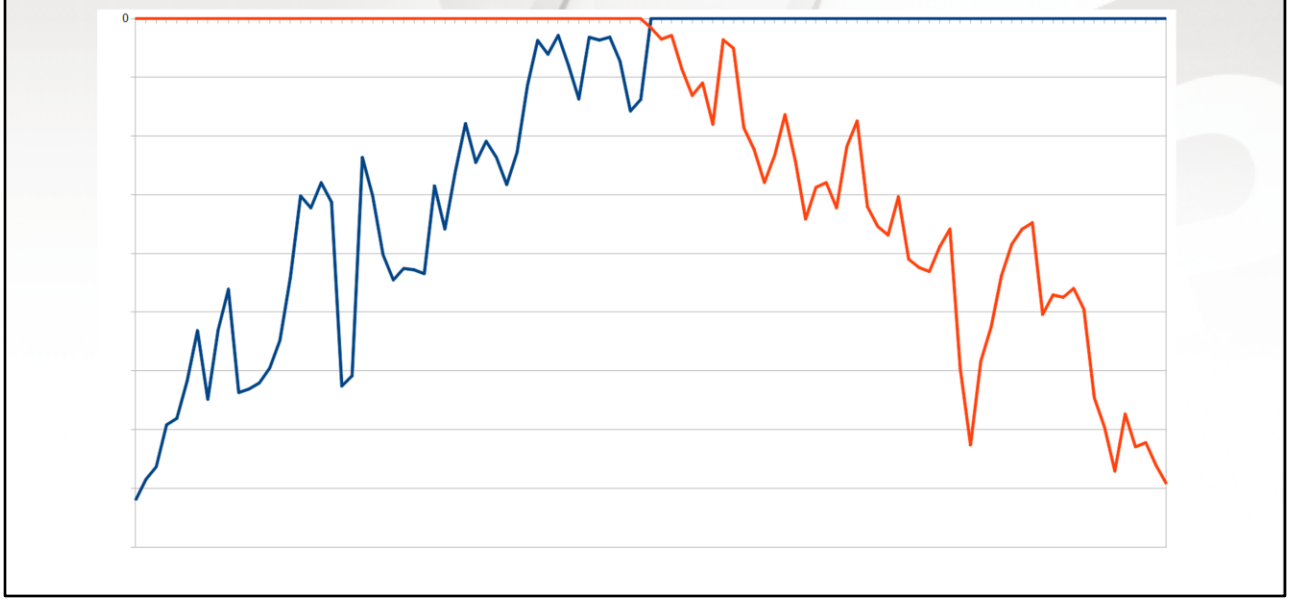


We multiply the per-vertex value by the value sampled from the artitst-provided fitness texturemap. This slide shows the result.

One basic idea is to select the splat with the highest strength. This gives a very abrupt transition between the splats though, we'd like a more gradual transition that can be controlled by the artists.

We experimented with a lot of different ways to accomplish this; here's the method that worked out the best.

Biased by maximum value

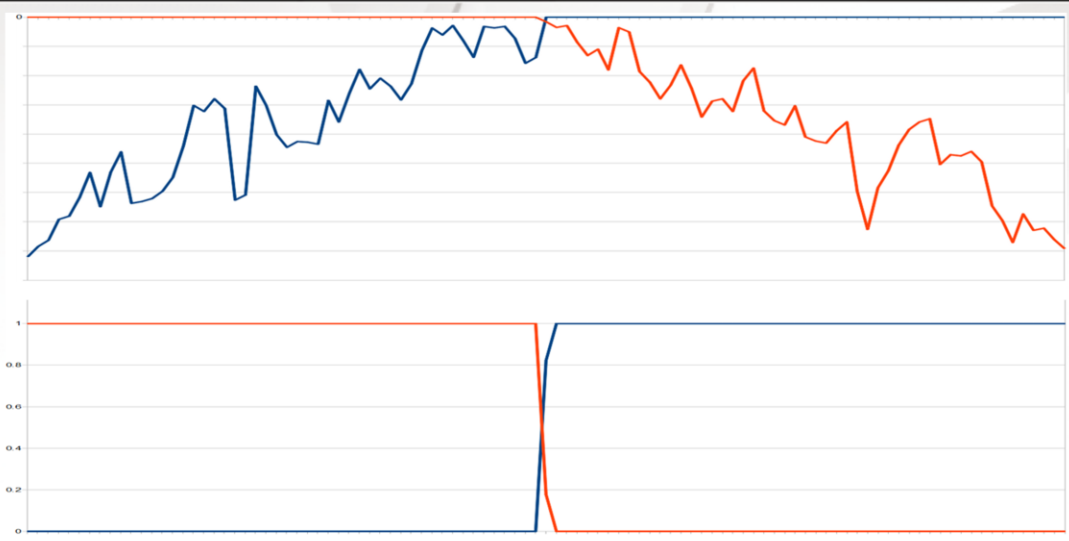


First, we offset by the maximum of the contributing values.

Then we specify a lower bound; we then clamp each of the values to that range and normalize. The lower bound allows us to loosely control the size of the transition range, and the size of this range controls whether we have a very abrupt transition or a more gradual transition. The lower bound value is controlled by the artists.

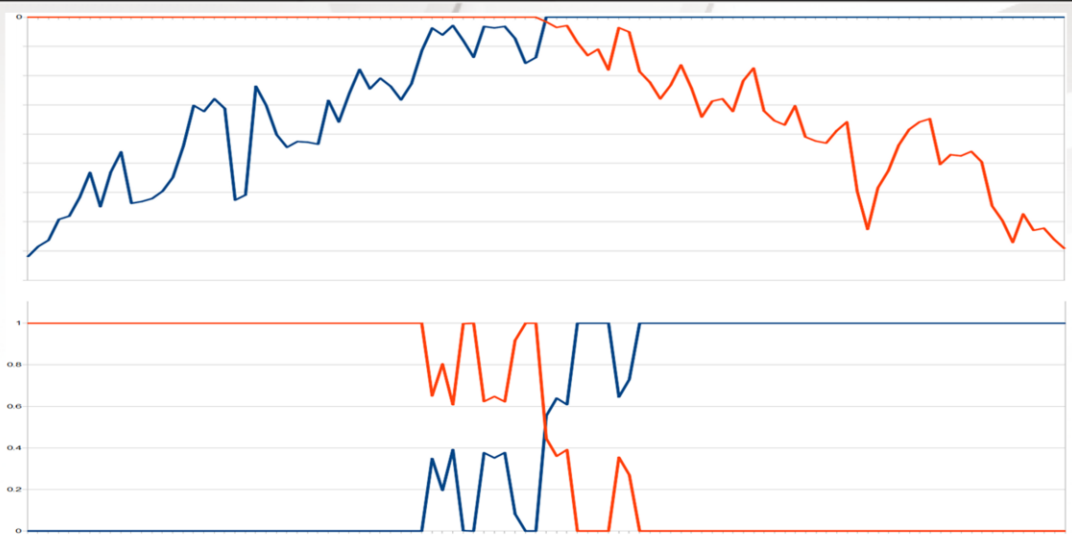
I'll show some examples.

Transition range = 0.01



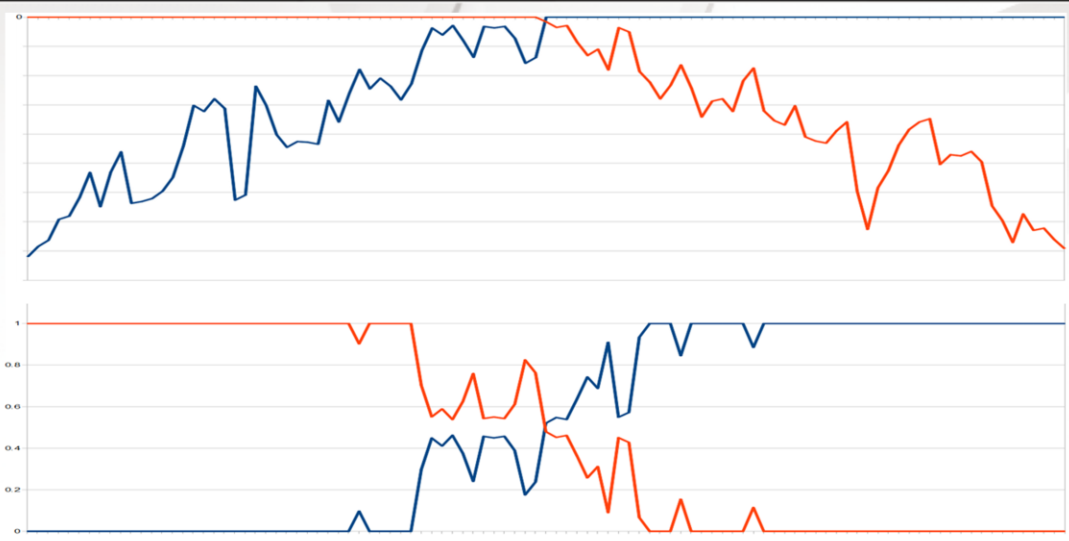
This slide shows a very small range, which gives a very abrupt transition. Intuitively what we're doing is magnifying the region near 0, and the transition value specifies whether we're using a smaller or larger region.

Transition range = 0.04

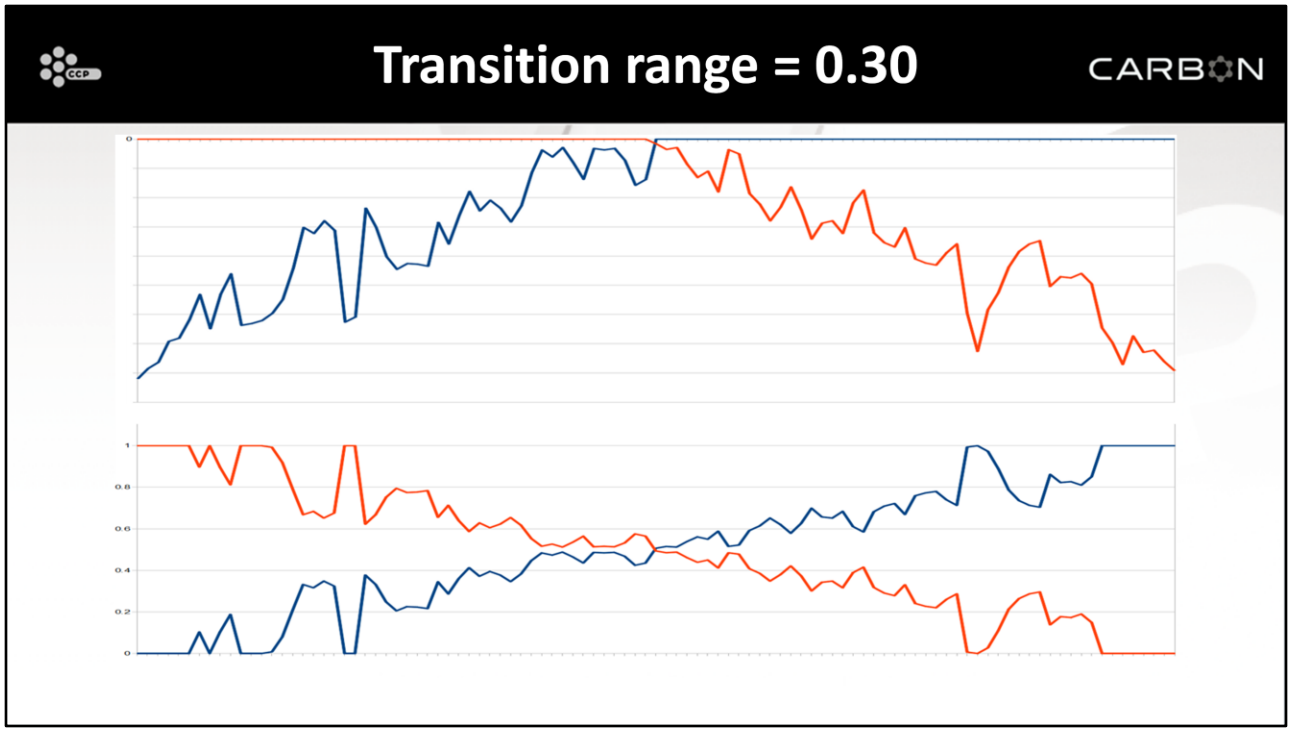


Here's the result with a slightly larger range value. The transition happens over a wider area.

Transition range = 0.10



A larger range value gives an even wider transition.



And a relatively high range value gives an even wider transition. This example shows the other extreme: a very large area will have several splats contributing, which tends to lead to an indistinct and vague look.

The code

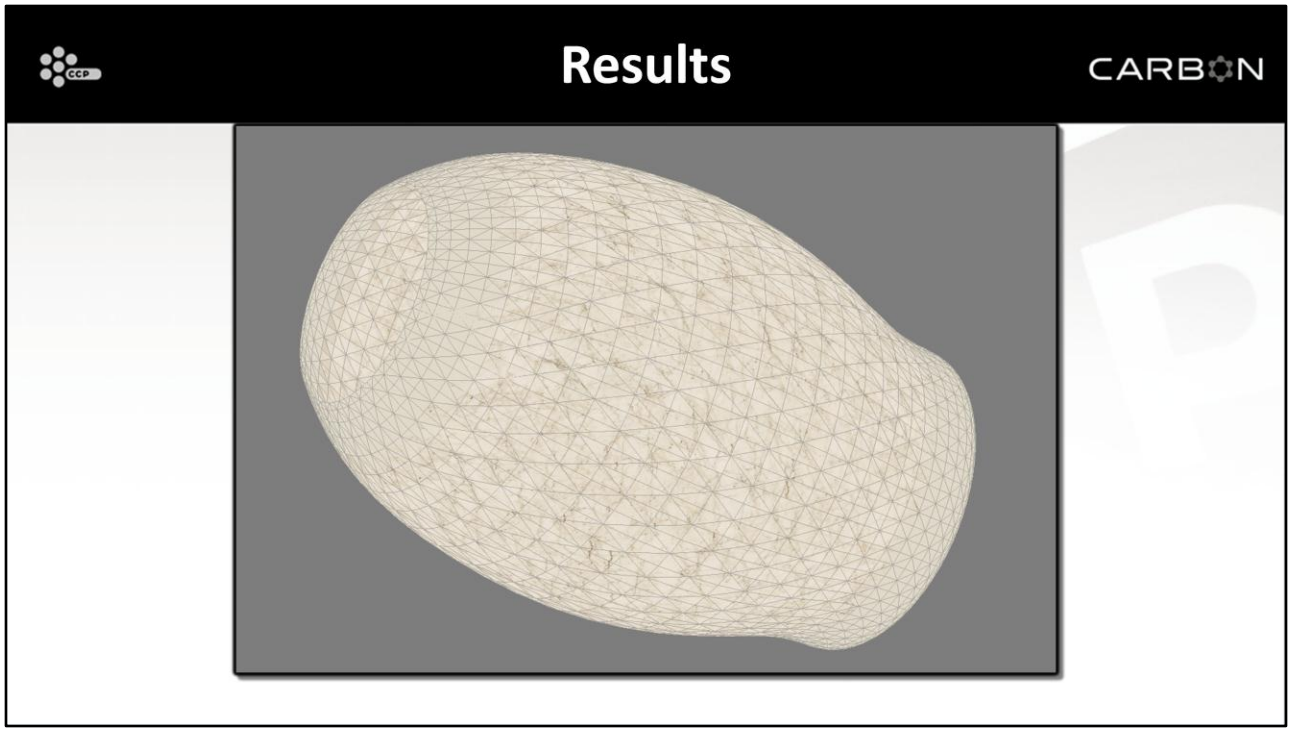
CARBON

```
vec3 fitnessVec=vec3(f0, f1, f2);
float maxVal=max(fitnessVec.x, fitnessVec.y, fitnessVec.z);
vec3 rawWeightVec=max(0,
      fitnessVec+transitionRange-maxVal)*fitnessVec;
vec3 normWeight=rawWeightVec/dot(rawWeightVec, 1);
```

Here's the code. The three fitness values are f0, f1, f2, in [0, 1]. The transition range value is transitionRange. normWeight contains the final normalized blend coefficients, used to combine the three splat diffuse color/normal/specular color/etc.

Note that there's an extra scale by fitnessVec in there, applied to rawWeightVec. We found that this additional scale improved the quality of the transition, there's no mathematical justification for it.

The normalization will fail if all fitness values are zero, and the resulting NaN will probably break the rest of the shader. This can happen if the fitness map the artists provide contains any regions of zero value. Clamping to some minimum value in the shader is possible, but it'd be better if the texturemap just avoided black.

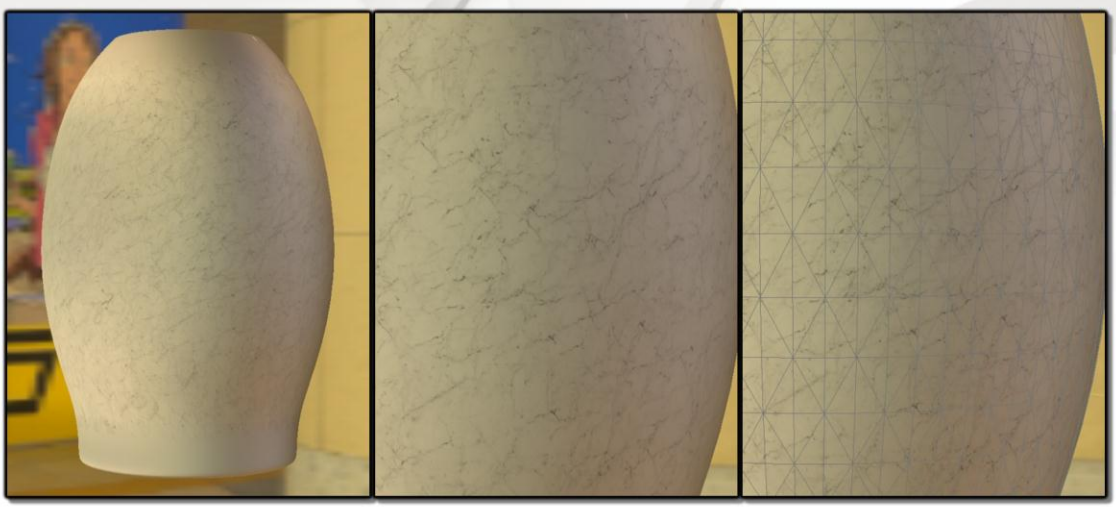


Here's how the splat tech looks in action, when applied to the model we were looking at earlier.

If we UV mapped the model we'd have to choose between UV seams, or distorted and stretched UVs. Fewer seams means more distortion and vice versa.

But with the realtime texture quilting tech, each splat is defined with its own planar projection, so we get a consistent texel size everywhere and no seams anywhere.

Results



Here's another few views of the model. The wireframe is shown to make it clear what the neighbourhood for each splat is.

Results

Another closeup view. Notice that the dark line features flow from splat to splat rather than being localized within each individual splat.

Tiling

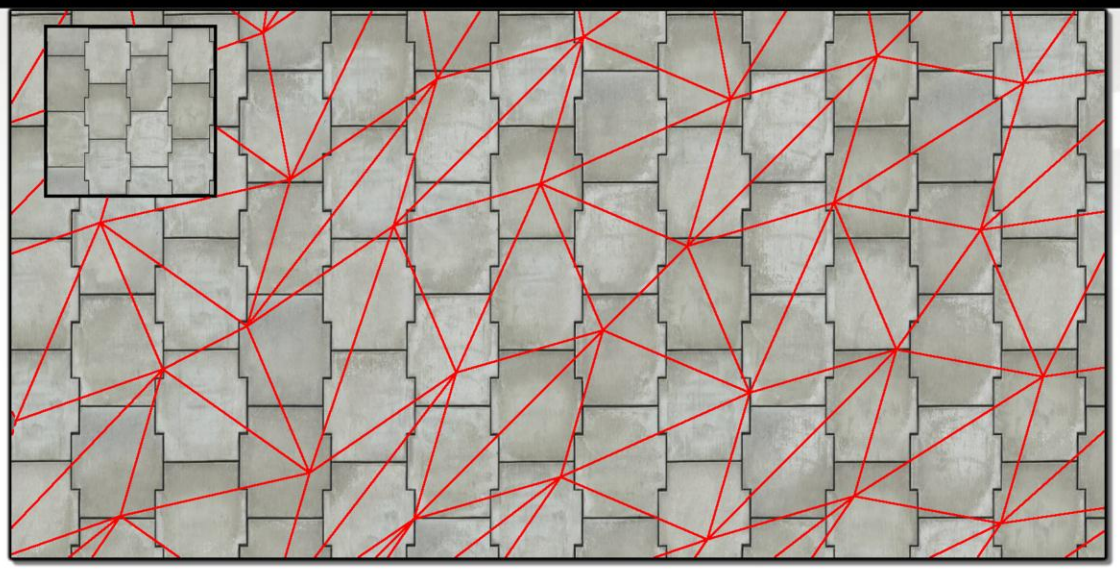
Another use for the quilting tech is to get rid of periodic tiling. This slide shows a plane using the single texture.
The splat orientation and the center UV for each splat were initially set up to simulate a planar projection.

The U coordinate was then offset by a random multiple of 0.5, and the V coordinate by a multiple of 1/3. This corresponds with the source texture being basically a 2x3 pattern.

The sharp features in the source texture don't need to follow the 2x3 grid exactly; it's fine for the dark gaps between plates to vary by a few pixels. If they vary by too much there will be noticeable artifacts, like dark edges fading in or out instead of forming a continuous outline.

Features are repeated, but there's absolutely no periodic tiling, unlike regular layered textures.

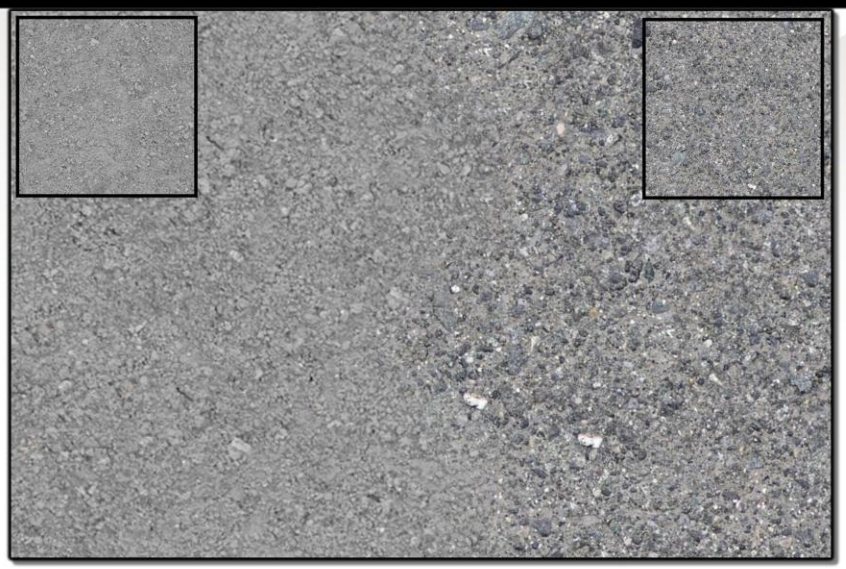
Tiling



This slide shows the underlying mesh, to show the splat size and where the transitions between neighbouring splats are.

Transition between two materials

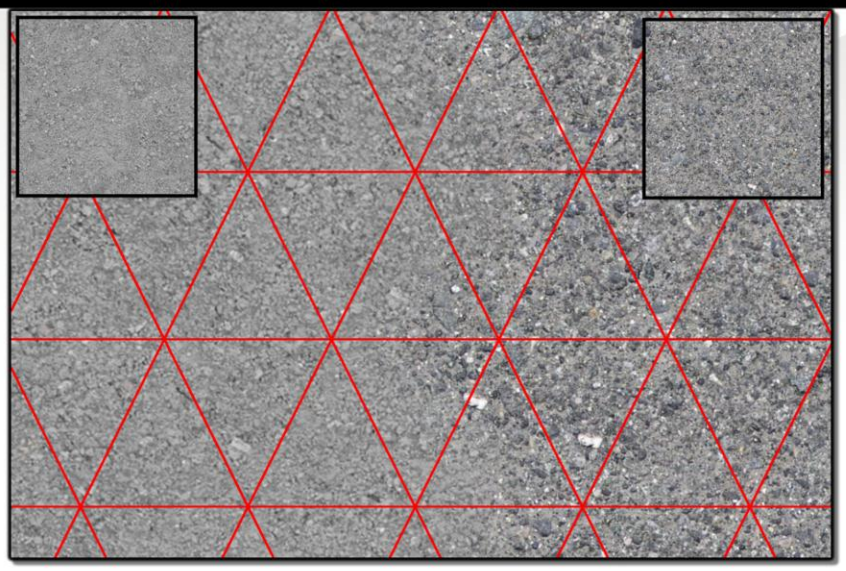
CARBON



Now, we'll extend that last example to multiple splats. There are two splats used here; they're shown at the top left and right. Verts on the lefthand side use of the mesh use the lefthand splat texture and verts on the right use the other. We get a fairly plausible transition, not as good as an artist would produce if they hand-painted it, but much better than the hard edge. Of course, this tech will cope with any combination of materials – and also, like I mentioned on the last slide, it'll cope with any topology and any kind of curve without distortion or seams appearing.

Transition between two materials

CARBON



Here's the underlying mesh, to show the splat size and where the transitions between neighbouring splats are. This example is a worst-case in a few ways: the source textures were photos and there is still a bit of lighting baked in there, which meant the splats can't be rotated randomly. Also, the mesh is very regular. If the verts are randomly moved around it makes border patterns much harder to spot.

Custom material transition

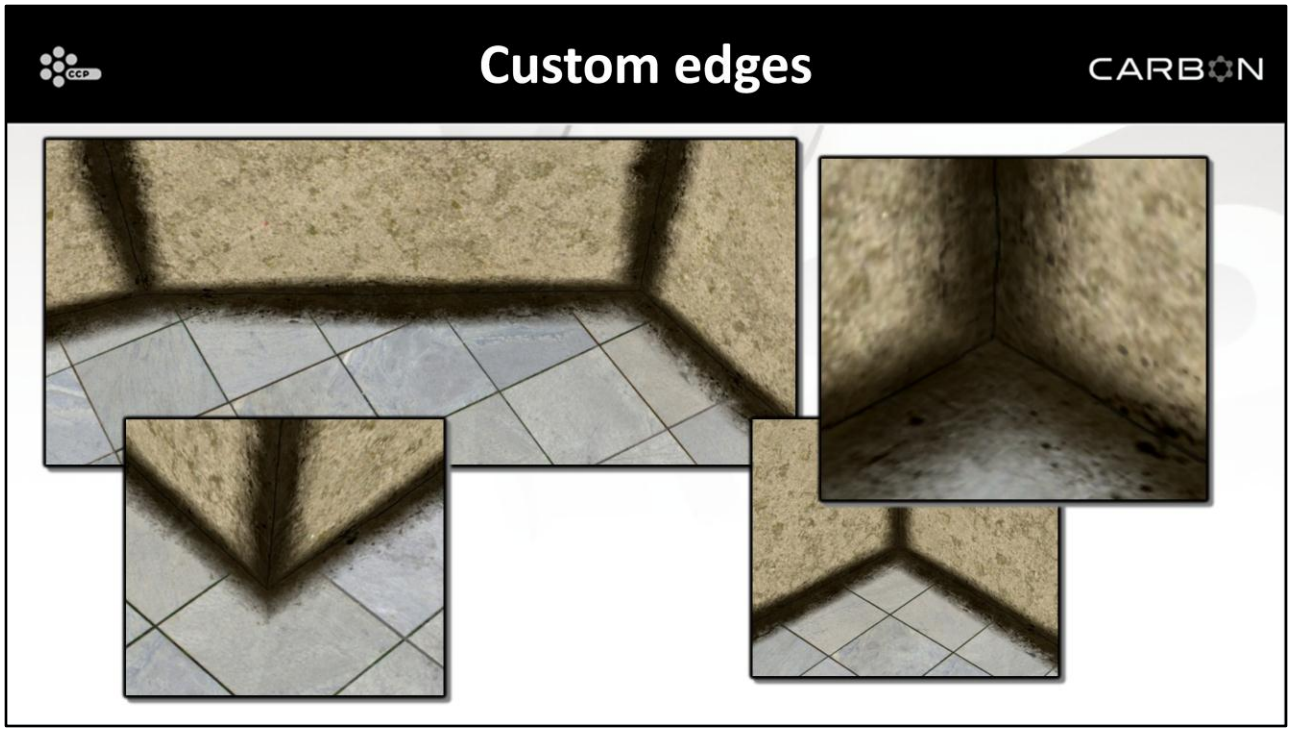
It's possible to paint up a custom transition texture. This image shows a transition from road to dirt using a custom transition texture – the three splat textures are shown on the left of the slide. The transition splat is in the middle. It repeats in U, it doesn't have any built-in bend. But it can be used to provide roads with any sort shape, if the curvature isn't too high.

(Compare with the image in the next slide showing wireframe.)

Of course, this applies to arbitrary 3D models too.

Custom material transition

Here’s the wireframe to show the splat size and where the transitions are.



Using a custom transition splat for an arbitrary 3D model means you could provide a specific textures for concrete corners so there's consistent weathering or grout for all concrete corners, regardless of shape.

Here's how it looks. The splats used here are shown in the corner. Up until now I haven't said anything about how the splats get mapped onto the geometry, because the obvious option of a simple planar map was all that was required.

This example goes quite a bit further . Instead of thinking of the shader as supporting a splat at each neighbouring vertex, we're thinking of each triangle as having three independent planar-mapped textures applied to it, each of which has a strength function encoded in the verts. This is just a change in how we look at and structure the data: it uses the same existing shader tech, the only changes are to how the data is built.

In the model , we have splats applied to verts on 90 degree geometry creases – such as the floor and wall. For those splats, it's best to have two different mappings; one for the floor and one for the wall – a single projection will cause noticeable stretching, and corners sharper than 90 degrees will get progressively worse, of course.

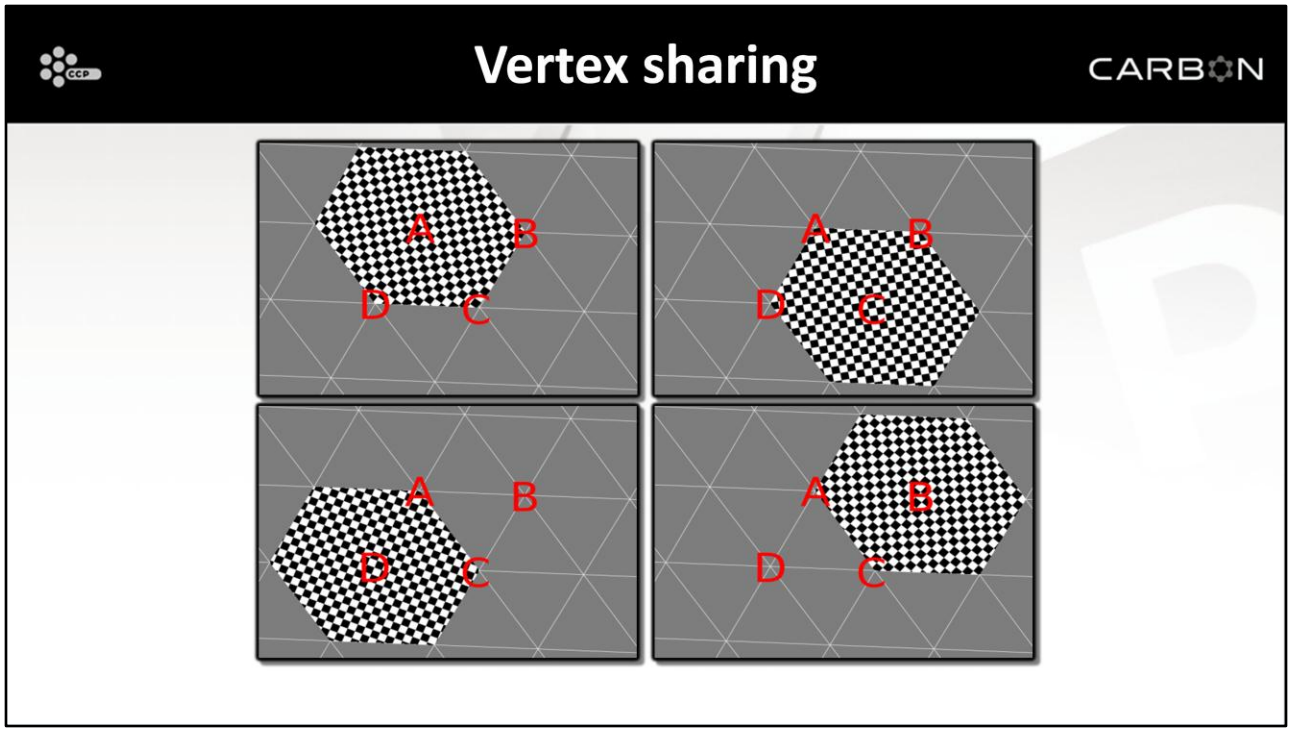
The texture being applied to the corners is just a thin strip. It'd be nice to support an alpha channel to indicate which parts of the texture are relevant. It's possible to accomplish this using the existing tech, by modulating the fitness map by alpha. This means transparent areas with alpha=0 will have fitness zero which means those areas won't be used. Having maps with fitness=0 is dangerous for reasons described earlier; it will be OK so long as at least one of the contributing splats has a nonzero fitness value.

Things get more interesting in the corners. Notice that there's only a splat for edges, there isn't a

special splat for three-way joins. However, in each of the corners we can see three creases come together without any problems. What splat is being applied to the corner vertex?

The answer is: there isn't one. On the triangles that touch the corner, there are two edge splats and one space-filling wall or floor material splat.

What this means is we can have as many edges coming together at a point that we like, with any angles, with freedom to choose any splats for each of those edges, and they'll be smoothly quilted together like you see here.



I'm going to talk about some implementation details now, because there's a pretty serious problem that needs to be addressed.

We have three splats covering each triangle. We'd like there to be precisely three channels of splat data in the interpolators. We also want verts to be shared.

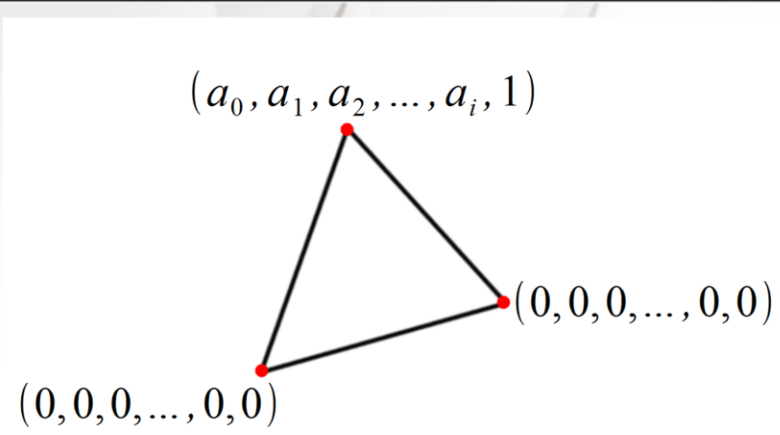
Unfortunately, if we implement this splat tech the obvious and easy way and store the Uvs for each splat in the vertex stream, there'll be a problem.

The slide shows a small piece of geometry, and the Uvs for a couple of the nearby splats. If we're encoding the Uvs for the splat at vertex B in a particular UV channel, then we can't use that UV channel for the Uvs for the splat at vertex D: that UV takes different values on either side of the edge AC. So the verts on edge AC need to be split. The same reasoning can be applied to every single edge of the mesh.

So the problem is, it looks like no vertex sharing is possible, that the vertex buffer will be three times the triangle count, and that the post-transform vertex cache will never ever get hit.

The solution we found was to use a sort of homogenous-value trick, which allows most verts to be merged and also allows some pretty serious vertex data compression. Here's how it works.

Vertex sharing



$Interpolated\ value\ is\ (ka_0, ka_1, ka_2, \ldots, ka_i, k)\ for\ some\ k \in [0,1]$

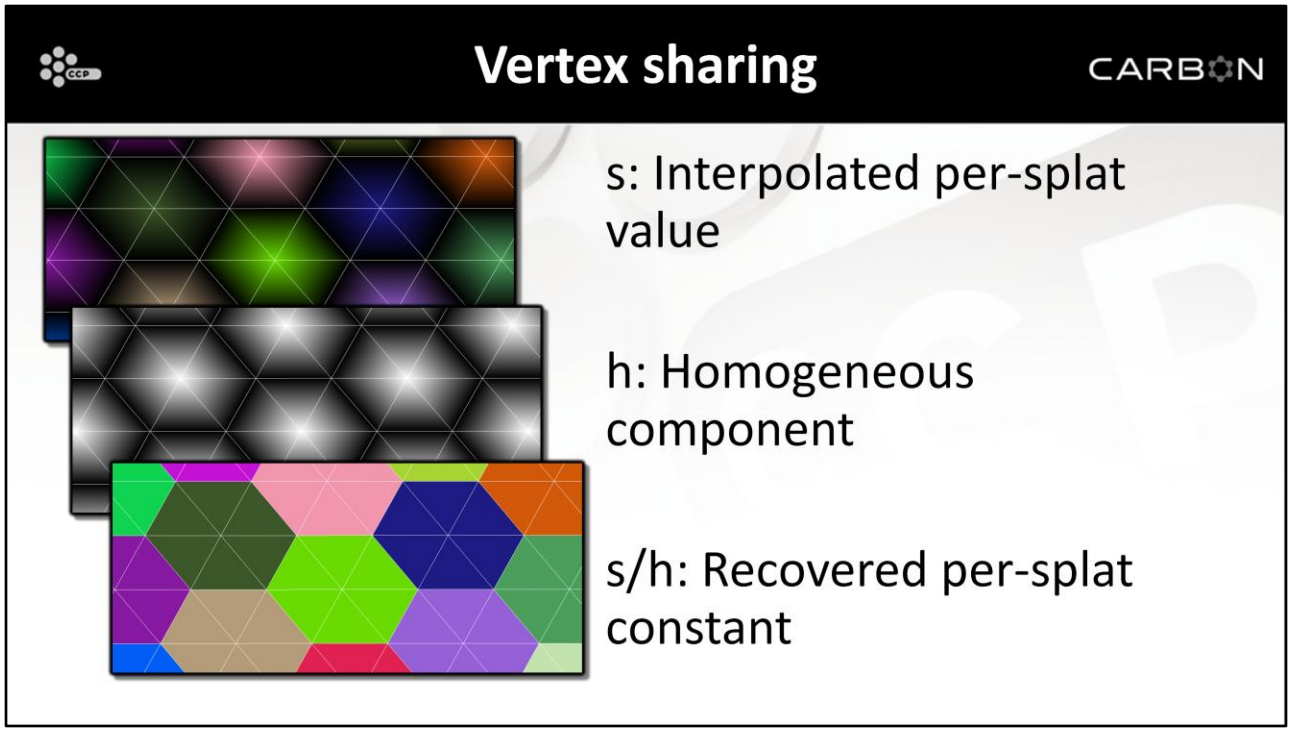
Suppose we have the triangle shown. The vertex program outputs the values (a0, a1, … ai, 1) at one vertex, and (0, 0, … 0, 0) for the other two verts.

In the pixel shader we'll have the interpolated value (ka0, ka1, … kai, k) for some k in the range [0, 1]. Divide by k to find (a0, a1, … ai).

What this gives us is that ability to specify a constant value for the whole triangle by controlling the values at one vertex only, with the other two being zero. Instead of storing UVs for the splat, we store the mapping from world to UV, which will be constant over the whole splat.

The key advantage we've gained is that the verts which are all zeros can be shared.

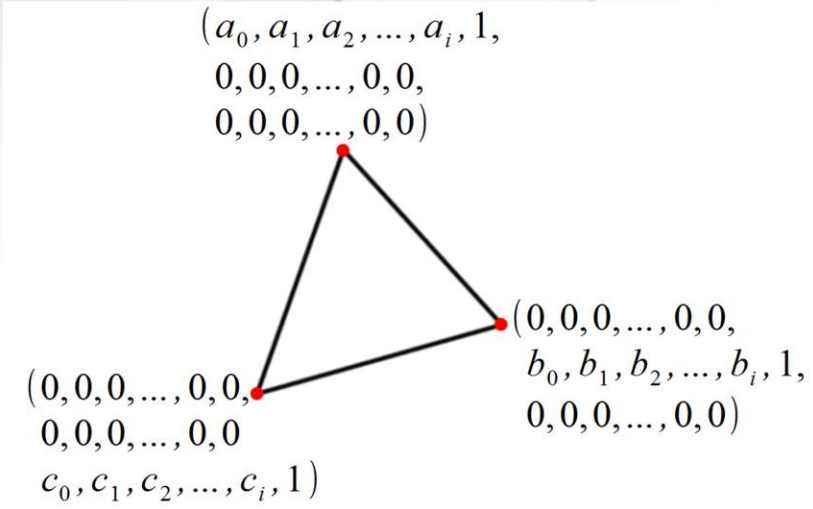
It is possible that the pixel center lies precisely on the bottom edge of the triangle, giving k=0 and making the homogenous divide fail. This case is possible, but if the interpolators are floats it should be so extraordinarily rare it's probably not worth worrying about. Another reason not to worry about it is that the splat weight will normally be zero on that edge, so the resulting blended color should be completely independent of that splat.



Here's an example of a mesh with complete vertex sharing, using the divide trick to provide a per-splat constant.

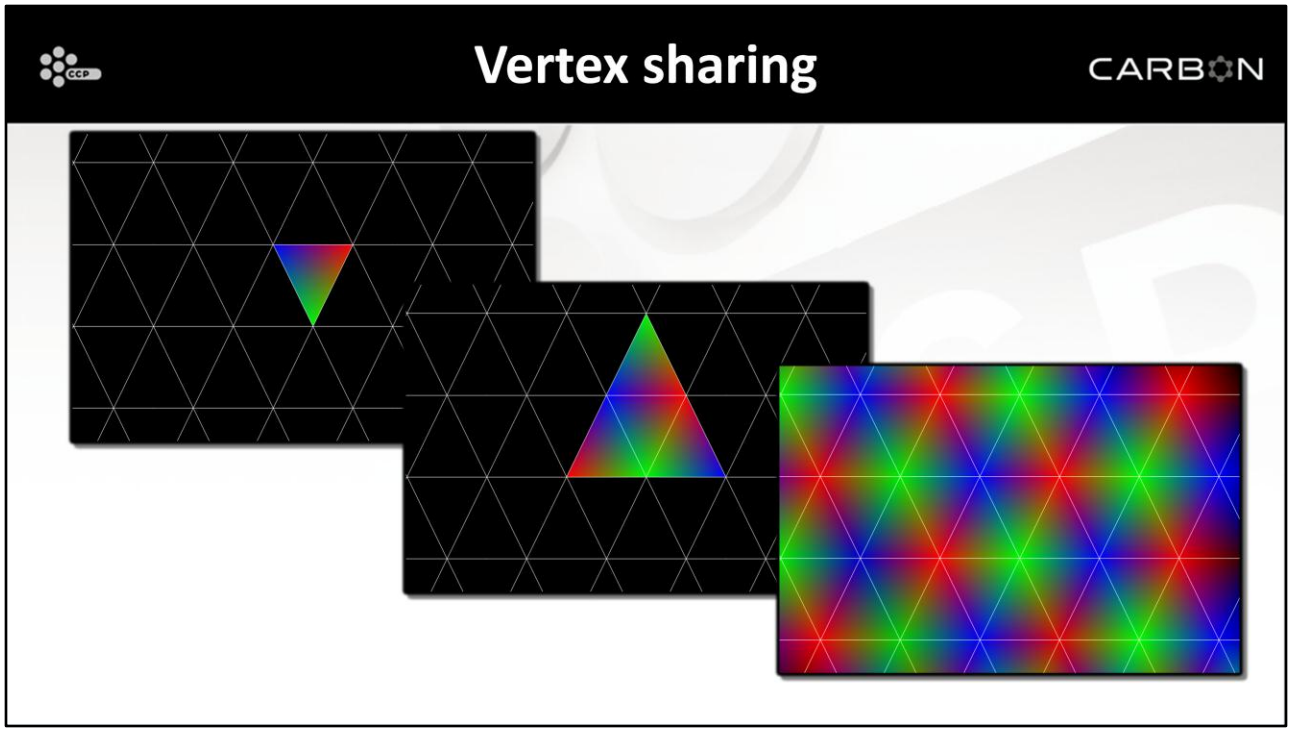
We'll use this method for all three of the splats covering each triangle.

Vertex sharing



Here's a triangle. The vertex program outputs the data shown – we're just triplicating the divison trick.

If we can arrange the data so that for each vertex one channel has interesting values and the other two are filled with zeroes, as shown in the slide, then it's an easy win to throw away 2/3 of the data. Ie: The per-vertex data contains data for a single channel, plus a few bits specifying which channel it pertains to.



That is easier said than done, with an arbitrary 3D model.

The approach I'd recommend is to build up a list of the 3 relevant mappings for each triangle – that is, the linear transformation mapping from world space to UV for each of the three splats covering each triangle.

Our problem is how to assign those three mappings to the three verts of the triangle, and how to choose the channel for each to maximize vertex sharing, or conversely to minimize the number of entries in the vertex buffer.

Just using a simple greedy algorithm worked out fine. It's undoubtedly possible to improve on it with a more sophisticated and expensive method but it's probably not worth the effort.

Here's an illustration of the basic idea. To keep things simple I'll assume that every vertex has a single consistent splat assigned to it, so when the question of which vertex a mapping should be assigned to is answered, and the problem pretty much reduces to the question of which channel each splat should be assigned to.

Anyway, start with any random triangle, and assign the splats to different channels any way you like. In the slide s the homogeneous component of each of the three channels makes up the red, green and blue color component. On the top we see how things look with our first triangle – note that at each vertex, one homogeneous component is 1 and the other two are 0.

Then, repeat the following process until exhaustion. For each unprocessed triangle, look at the sets of coincident verts for each of the three corners. Or to put it another way, have a look for vertex sharing possibilities. The most restrictive case is where two verts already have channel assignments, so that's the one we'll start with. For example, the neighbours of the first triangle are like this; for two of the

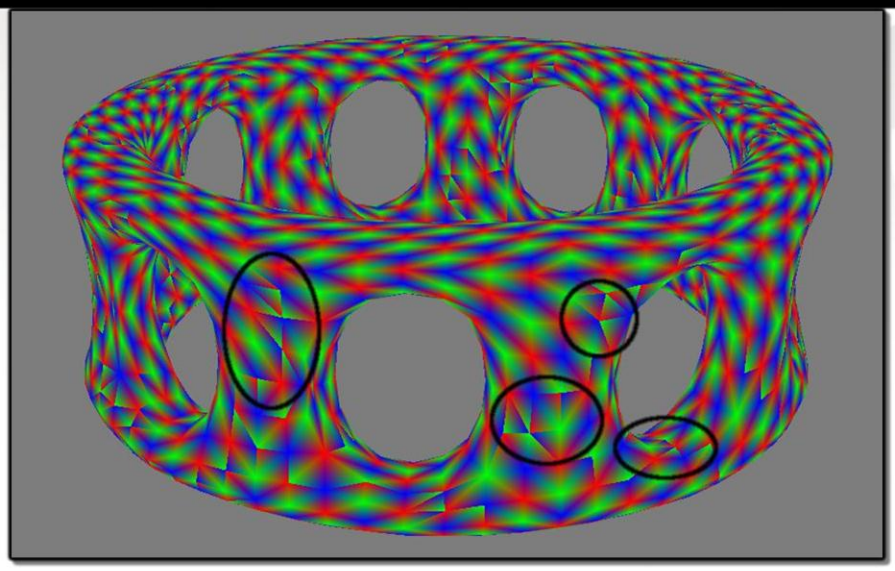
verts the channel is defined so (if they're different) the remaining channel can be used for the free vert.

The image on the middle layer shows how things look after one step.

The image at the bottom shows the end result. For this example we had a pretty simple regular mesh; with a more complex mesh things don’t always work out so cleanly.

Even with more complex meshes, it tends to be enough to take care of most verts and triangles. Continue assigning the splats for the last triangles, minimizing vertex splits, until the mesh is complete.

Vertex sharing



Here's an example showing a completed mesh – I've highlighted a few of the places where verts had to be split.

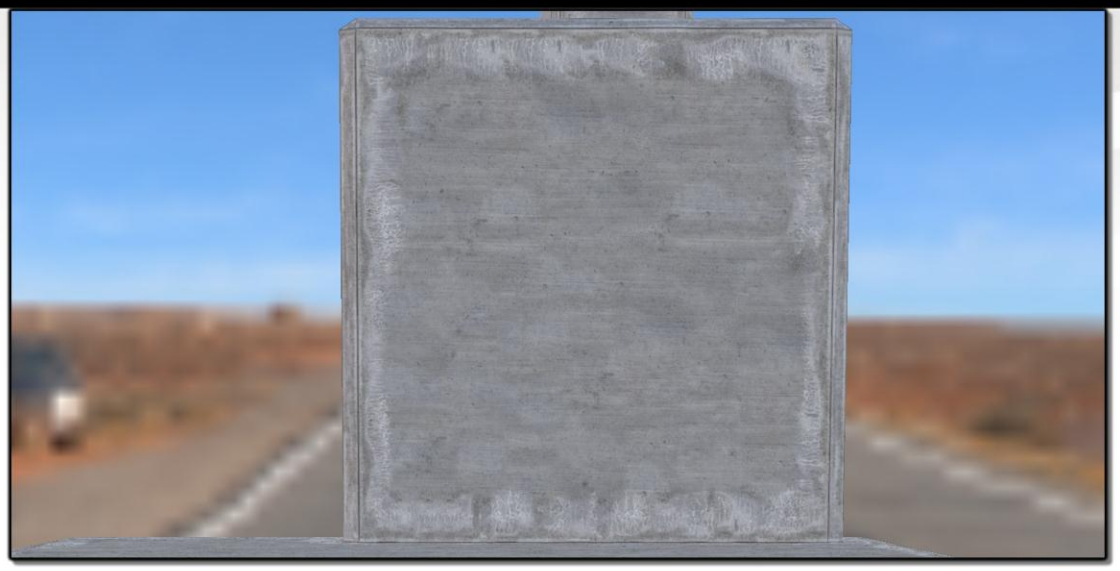
Edge effects for 3D models

Finally, here's a complex 3D model that's been textured using exactly two splats. One splat provides the grooved concrete used on the faces. The other gives the dark line along the edge, and the localized discoloration near the edge. Most of the color variation is due to lighting.

These two splats support all the different shapes and bevels and corners of the model. There are no periodic features on the faces or along the edges.

Weathering along edges and related effects would normally have to be painted into the texture, or achieved with careful alphablended layering. With texture quilting of splats, these effects are much simpler to achieve.

Edge effects for 3D models

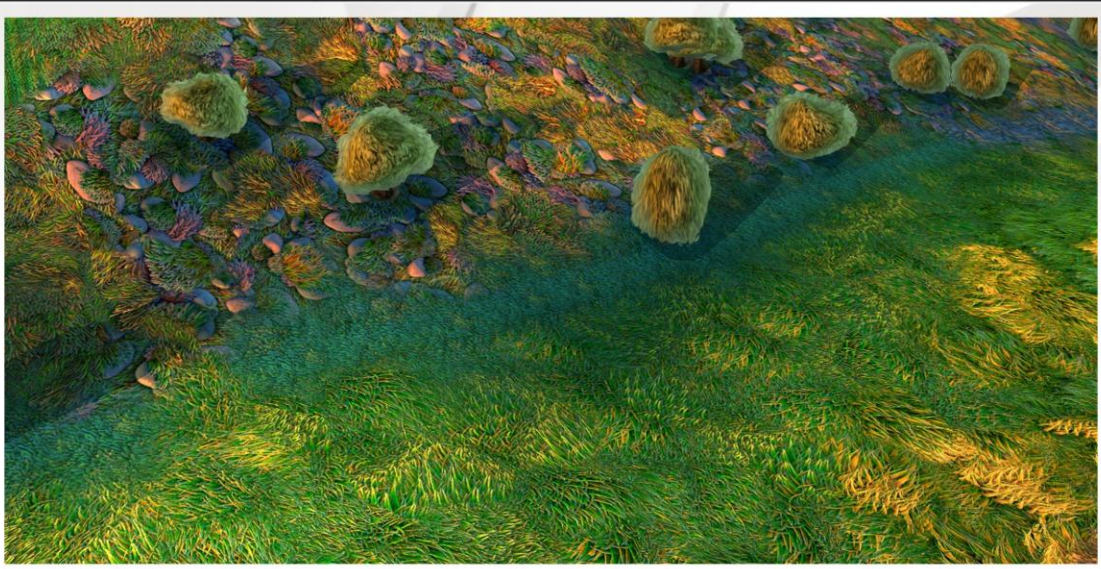


Edge effects for 3D models

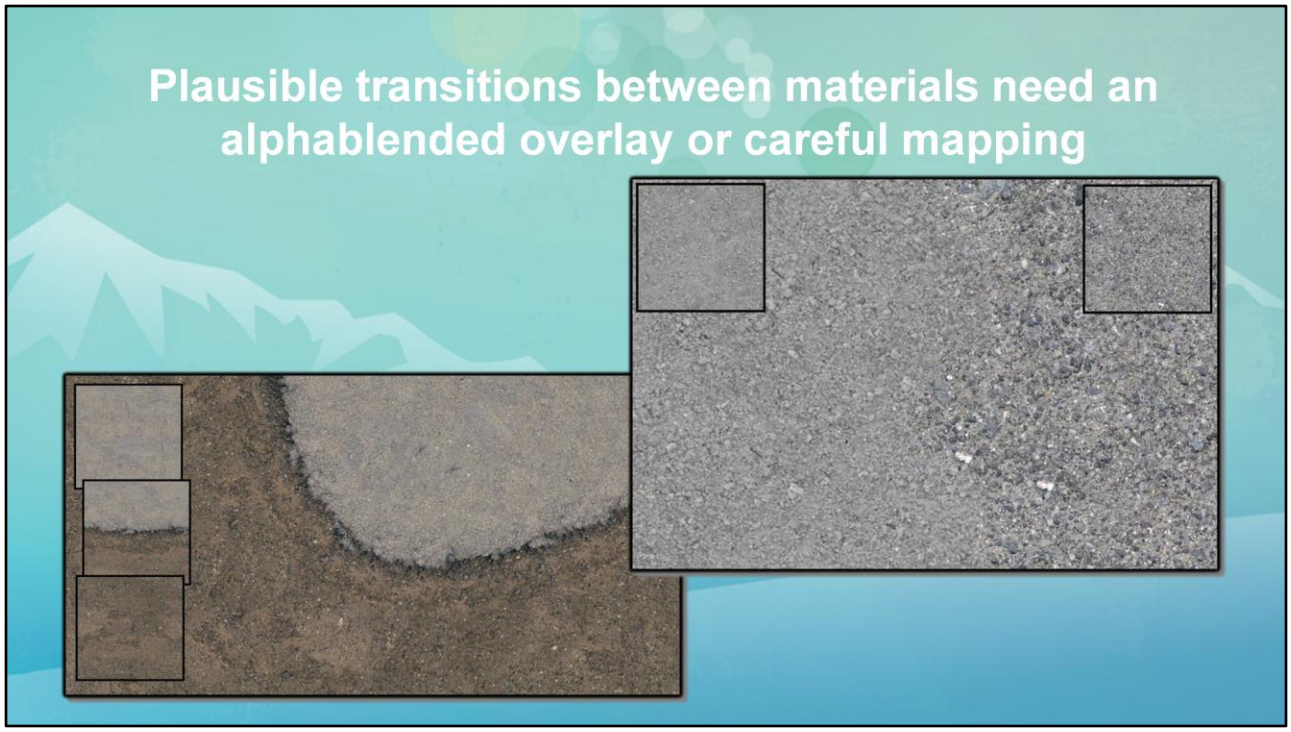
Use in production

Use in production

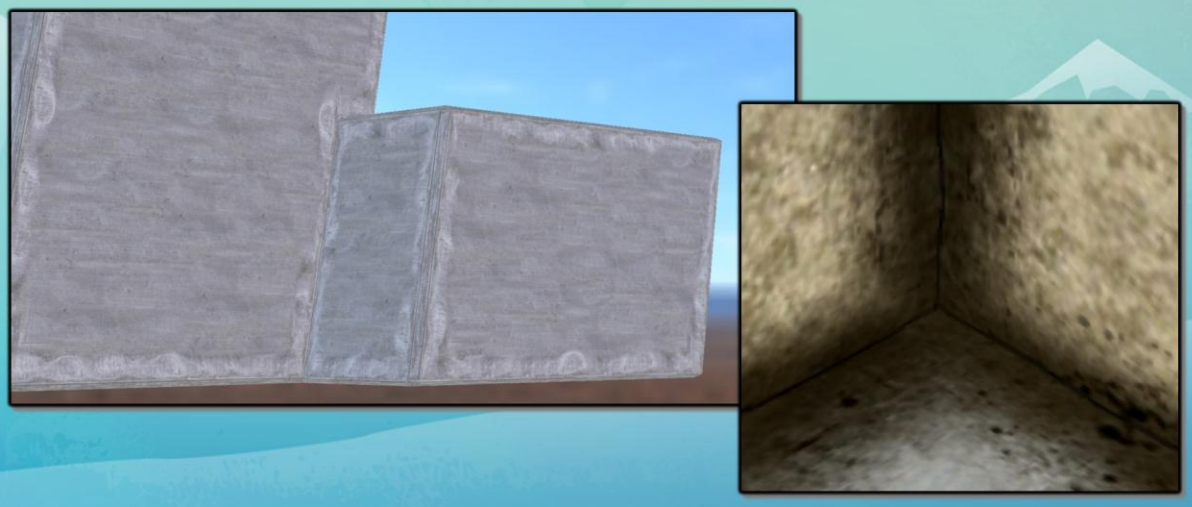
CARBON



It's difficult to avoid tiling with large single-material surfaces


Plausible transitions between materials need an alphablended overlay or careful mapping

Painting effects into textures due to geometry creates complex restrictions



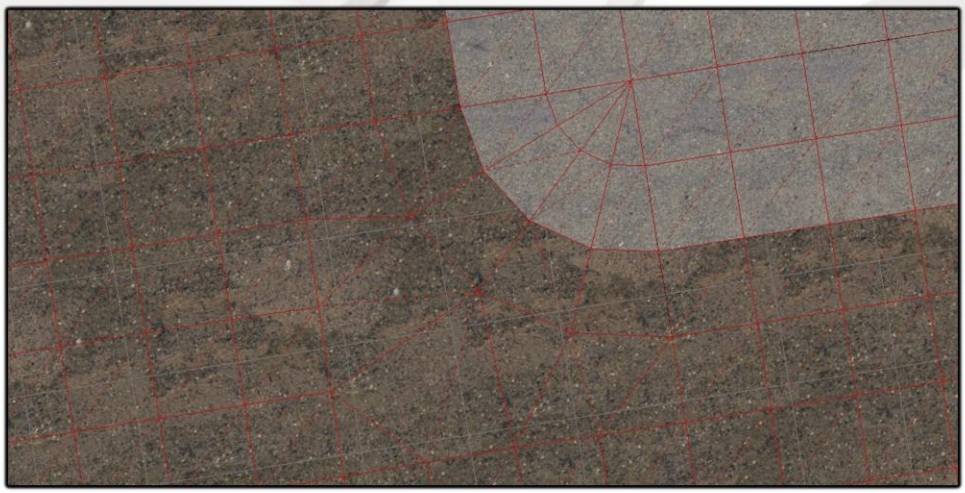
Thanks to…

CARBON

- James Bird
- James Kett
- Michael McLarty
- Steve Hodgson
- Stuart Jollands
- Tahir Rashid


CARBON
?

BONUS – Authoring tools



It would be possible to set up the splats on a model by demanding that the artists place and orient each splat on each vertex by hand, but this is seriously labour-intensive. However, any other option means procedurally generating the splat data. Any sort of procedurally generated content will suffer from some common problems. A set of new, custom tools are required, which means a learning period for every artist who has to use them. Also, art direction requests will sometimes need programmer involvement to be fulfilled, and art-directing programmers is far from ideal. Lastly, when problems appear in the final procedurally-generated result it's often not clear how to change the source content to fix them.

The best option I can see is as follows. Set up a standard, repeating texture for each splat, and build the model using a basic "block-out" of materials – no filleting or overlays or transitions. It's UV mapped so the texel size matches the texel size of the splats. The model is the source for the splat generation code.

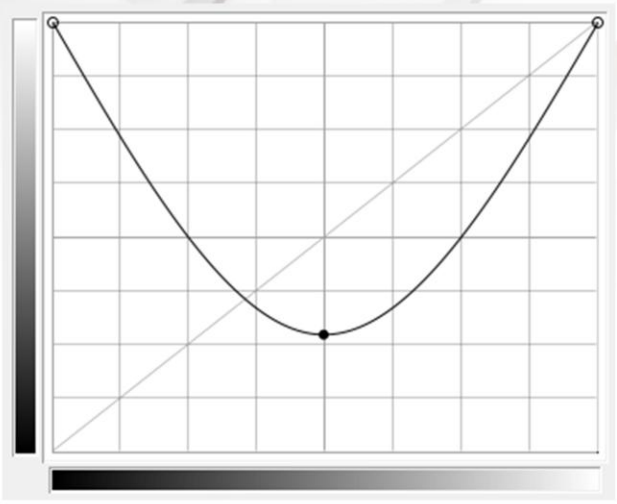
The slide shows how this method was used for the road edge transition image – compare it to the result shown in slides 33 and 34. Note that there's a completely hard edge between the two materials, and the textures repeat. (This is most obvious for the brown ground texture; it'd be much more obvious if the slide showed a larger area.)

The main advantages of this approach is that the artists work with very familiar tools: tiled textures and a simple UV-mapped model, which will match the final result as far as can reasonably be expected. The splat tech has been specifically designed to not add or remove verts or turn edges or otherwise retessellate the model, which means the artists have complete control over the resulting geometry as well.

It doesn't address the question of how to create and maintain the splat layout hints such as blend

softness, or to enable the use of the custom earth-to-asphalt transition splat used on slides 33/34, or support for splats on creases, or many of the other possibilities for laying splats out: these are fundamentally questions which need tech artist time and probably project-specific tools to answer.

BONUS – Fitness map setup



Creating the fitness map by converting the colormap to greyscale means that lighter features are stronger and dark pixels weaker. This means that in blended areas the lighter colors will tend to dominate, which means the blend areas will be slightly lighter than elsewhere. It'll be more or less noticeable depending on the color balance of course, but if it wasn't noticable it wouldn't get a slide.

One approach that seems to work reasonably well is apply the curve shown here to the greyscale image.. The reason for choosing a curve like this is so the resulting fitness map isn't correlated with light or dark colors, so the resulting blend areas shouldn't be lighter or darker than anywhere else.

For the images in this presentation the fitness maps were set up automatically. The code takes the color image, converts it to greyscale, and does a highpass filter. It calculates fitness value from that image with the formula:
fitness=abs(-1 + (number_of_pixels_darker_than_this_one/total_number_of_pixels)).