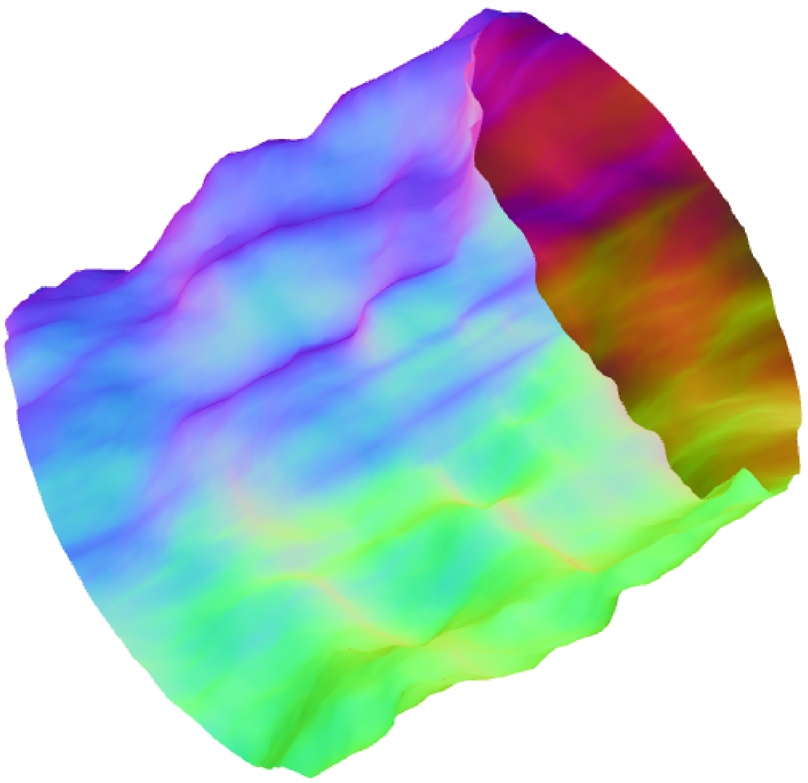
university of groningen / faculty of science and engineering / bernoulli institute

Per-patch noise functions

Bachelor's Project Thesis

Tom Apol
Supervisors: Jiří Kosinka & Gerben J. Hettinga



Normal map of a tubular mesh, displaced by patch-based Bézier noise.

Abstract

In this thesis, I will be discussing new advancements in the area of patch-based noise, in the context of displacement mapping: altering existing noise functions to make them patch-based, and continuous to a certain degree. This patch-based approach allows us to manipulate the generated noise values by changing attributes of the patch vertices. These attributes will be interpolated over the patch, and can be used in the noise computation itself.

This has been successful for several lattice-based noise functions (Value-, 2D Perlin-, and our new Bézier noise), for quadrilateral patches. The continuity of these altered functions should be the same as their original forms, being C^1 in the regular case. When applied to a mesh, it is bound by the continuity of said mesh, and further influenced by the regularity of patch-lattices over the mesh. Using the ACC2 framework (which approximates Catmull-Clark subdivision), this results in the following continuity: Patch-edge continuity can range from C^0 to C^1, while vertex continuity ranges from G^1 to C^1.

I will discuss how the continuity over patch edges, using local, edge-based coordinate systems, is maintained, and will demonstrate it with examples of these altered noise functions applied to displacement mapping.

Contents

1 Introduction

What is displacement mapping?

Displacement mapping is the act of displacing the vertices of a mesh, according to a given height-function or texture map. This displacement is generally performed along the vertex normal. Displacement mapping is used to modify the mesh itself, which contrasts to, say, bump-mapping, which only changes the normals of the surface, such that during lighting calculations it *seems* like the surface is displaced.

Why use noise?

Sometimes it is advantageous to use procedurally generated noise as a height-function when displacement mapping. For example, when the displacement needs to look organic. Examples of this would be procedurally generated terrain, or irregular organic surfaces such as human skin.

How are noise functions used at the moment?

Currently, in the context of displacement mapping, noise functions are used primarily in the following two ways:

- Generate a global texture map beforehand using the noise function (which requires the mesh to be unfolded and flattened), and then apply the displacement to the mesh.
- Solid noise: Generate a 3D noise field (i.e. a cube containing scalar values) using the noise function, large enough to envelop the mesh. The noise is mapped to the mesh by intersecting the mesh with the cube. Finally, apply the displacement as usual.

How does our project change this?

This project applies noise functions in a different way: instead of using a *globally* generated texture map or noise field, we use per-patch texture coordinates to locally define procedural noise fields per patch. This has the advantage of not having to send a lot of texture data to the GPU, nor having to flatten the mesh beforehand. Furthermore, due to being patch-based, our approach also allows us to easily vary noise parameters on a local level. (see section 2.7 for theory and 4.4 for visual examples of this)

The goal of this project is as follows: Apply displacement mapping with patch-based noise-functions on an ACC2 surface, whilst maintaining continuity over patch-edges.

2 Background information

2.1 ACC2 framework

The provided ACC2 framework approximates Catmull-Clark subdivision surfaces using bicubic Bézier patches. This framework is based on the paper by *Loop et al.* [1]. Important to note is that in the regular case, the patch surface is equivalent to the usual Catmull-Clark surface (C^2 continuity, see section 2.2). However, in the irregular case, where at least one of the patch vertices has a valency other than 4, the resulting patch surface is tangent-plane continuous along its edges (G^1 continuity, see section 2.2). This guarantees that its C^1 normal-field is continuous as well . This means that both the surface and its existing normals can be used in further surface normal computations, should the mesh be deformed from this point on.

The mesh (or rather, a series of quadrilateral patches) is implemented using a half-edge data structure. (See section 2.5.) In terms of data structure, a patch is simply a linked list of half-edges, paired with additional relevant patch-wide data/properties. In the ACC2 framework, the base mesh is comprised of quadrilateral patches, where each patch corresponds to a face of the mesh. After subdivision, each patch is still projected on the refined mesh, and is now itself comprised of smaller quadrilateral faces.

2.2 Geometric & Parametric Continuity

When discussing continuity on curves, we can think of it as a point $t \in [0, 1] \subset \mathbb{R}$ on the curve where the curve is split into two sub-curves (e.g. curve A and B). First it is important to distinguish two types of continuity: geometric (G) and parametric (C). These continuities can be ranked in terms of how continuous they are (i.e. G^n or C^n with $n \in \mathbb{N}$).
For this thesis, we will only cover G^0 to C^1:

- G^0/C^0: A curve is G^0 or C^0 at point t, when evaluating the curve at t from either sub-curve (the curves resulting from splitting the initial curve at point t) results in the same value. Another way of phrasing it is that both sub-curves geometrically line up at point t.

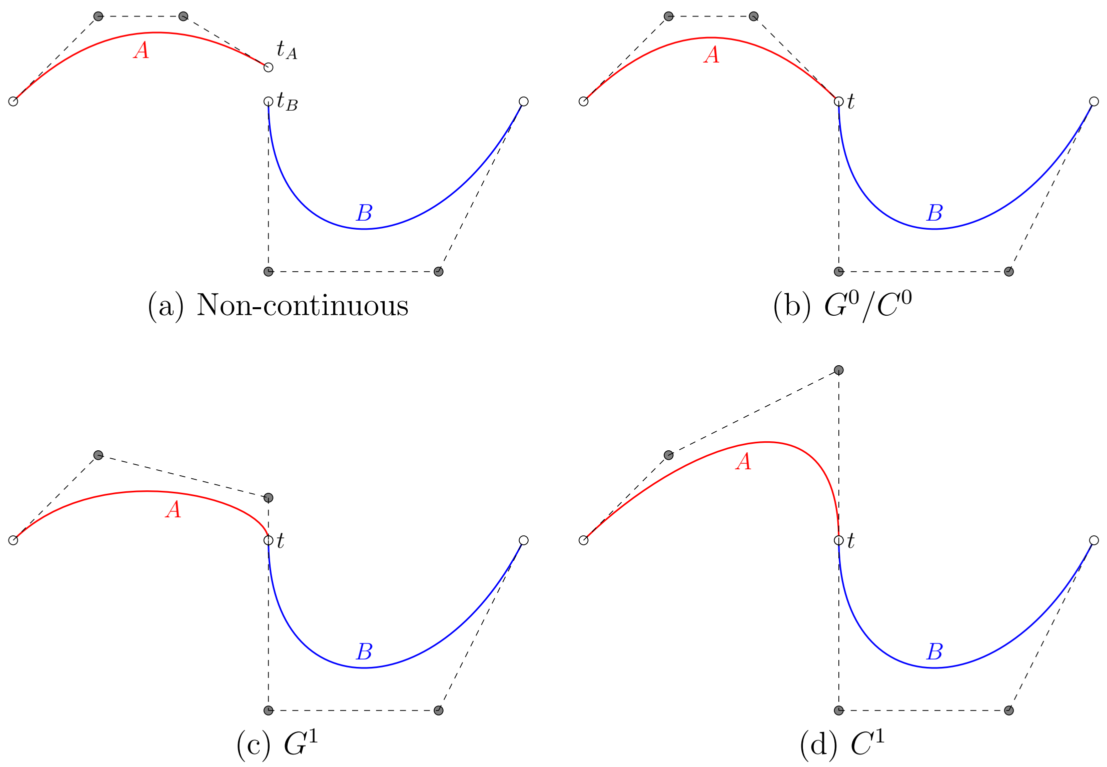
  In figure 2.1(a) and (b), we can see the difference between non-continuous and continuous curves.

- G^1: A curve is G^1 continuous at point t, when it is G^0 continuous, and when evaluating its first derivative (i.e. its tangent vector) from either sub-curve results in a vector in the same direction.

  In figure 2.1(b) and (c), we can see the difference between G^0 and G_1 continuity: both tangent vectors are aligned (as in our example, both of the control points surrounding t and t itself are all collinear). Note, however, that the curve of (d) is also G^1 continuous.

- C^1: Similarly to G^1 continuity, a curve is C^1 continuous at point t, when it is C^0 continuous, and the tangent vector at point t is the same when evaluated from both subcurves, both in direction *and* magnitude.

We can see this in 2.1(d), where the control points surrounding t and t itself are collinear, with an equal distance between each of these control points and t.



(a) Non-continuous (b) G^0/C^0

(c) G^1 (d) C^1

Figure 2.1: Examples of non-continuous to C^1 continuous Bézier curves at point t. Each curve is split into subcurves A (red) and B (blue). The white nodes are start- or endpoints of a subcurve, while the grey nodes are the other control points. The control polygon is displayed using dashed lines between each control point.
Note that a dashed line between a control point and a start-/endpoint, in the order of the control polygon, corresponds with the tangent vector at that point on the curve, from the direction of the corresponding subcurve.

Continuity on surfaces is in essence the same as continuity on curves, but with two degrees of freedom rather than one (so a point is defined by two coordinates, e.g. u and v). We can simply see a point on a surface as a combination of two curves: one defined solely in terms u and one solely in terms of v. Now it is just the case of checking the continuity of our point on both of these curves. The final level of continuity is then bound by the least continuous part (e.g. C^1 with respect to u and C^0 with respect to v results in a surface continuity of C^0).

2.3 UV-coordinates

A UV-coordinate system is a 2D coordinate system where $UV \in [0, 1]^2 \subset \mathbb{R}^2$.

In our application, the original mesh is made up out of quads. When applying ACC2 to this mesh, these quads remain projected onto the mesh. We will call these projected quads 'patches'. These patches have their own local UV-coordinate system associated to them, similar to how an entire mesh might be flattened and assigned texture coordinates. This way points on a patch can be identified, on every patch, rather than using global coordinates of the entire mesh, without the issue of cutting and flattening the mesh itself.

Globally, we use a combination of uv-coordinates together with the corresponding patch-ID to identify a point on the mesh. Note that for a point on a patch edge, there may exist a multiple of these tuples, as multiple patches may border such a point. Furthermore, this is the case for all points along any patch edge of a closed mesh.

2.4 Displacement mapping

Displacement mapping is similar to texture mapping: texture mapping assigns colour values to vertices, which get interpolated when evaluating pixels. Displacement mapping assigns displacement values ($\in \mathbb{R}$) to vertices, which will get displaced along its normal vector. This way, a mesh can be altered at run-time. When combined with ACC2 subdivision, we can start with a coarse mesh, and subdivide it to generate new vertices on the fly, which we can then displace using displacement mapping. This way a detailed mesh can be generated without sending as much data to the GPU.

2.5 Half-edges

Our quad implementation is based on the half-edge data structure. Where a standard quad is implemented using bi-directional edges, our quads are defined as a linked list of single-directional edges, in a counter-clockwise manner.

Figure 2.2 shows a diagram for an example half-edge e (blue). Each half-edge contains a reference to:

- Its target vertex. (purple)
- The next half-edge in the list. (red)
- The previous half-edge in the list. (green)
- Its twin. (orange)
  The twin of a half-edge is its counterpart, starting from the target vertex to the vertex it originated from. This twin lies on the adjacent quad, and thus may not even exist at all if the mesh is not completely closed.

The benefit of using half-edges is that you can easily traverse the quad, as well as easily traverse adjacent quads, using the half-edge's twin reference.

2.6 Noise functions

The noise functions that are used and discussed in this thesis are the following: Value noise, Perlin noise and our newly constructed Bézier noise. In this section, only the already established Value- and Perlin noise functions will be described. See section 3.5 for an explanation of Bézier noise.

The first thing to note is that both Value- and Perlin noise are lattice-based noise functions: the feature points (or control points if you will) all lie on the vertices of a lattice-structure. This is in contrast to a point-based noise function like Worley noise, where the feature points are randomly distributed within the cells of a grid-structure.

Perlin noise is a basic and widely known lattice-based gradient noise function.

Its 2D version is constructed as follows:

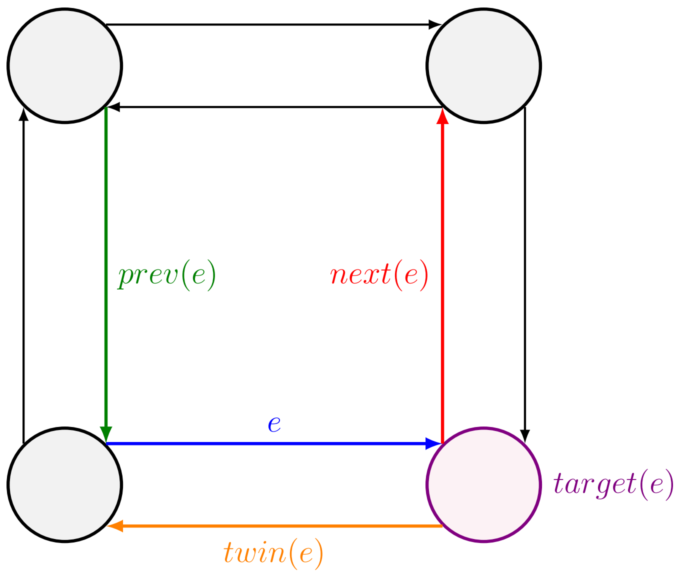


Figure 2.2: Half-edge structure, centered around the half-edge e. Note that $twin(e)$, just like all the other black twin half-edges, is part of some adjacent quad.

1. Create a gradient field, using randomly-generated gradients on grid-vertices.
2. For each point on the noise field (on which the gradient field is projected):
   (a) For each of the 4 nearest grid vertices, compute dot product between gradient vector assigned to this grid vertex and the corresponding offset vector (from the grid vertex to the point in the cell).
   (b) Interpolate these four resulting scalar values, weighted by the point's proximity to each corresponding vertex.

Value noise is a simple noise function, which is conceptually different from Perlin noise, but very similar in its construction. A 2D value noise field is constructed as follows:

1. Assign a scalar value (usually in the range of $[-1.0, 1.0] \subset \mathbb{R}$) to each vertex of the grid.
2. For each point on the noise field: interpolate the values assigned to the four nearest grid vertices.

Examples of Value- and 2D Perlin- noise are shown in figure 2.3.

Note that for both Value- and Perlin noise, the function with which these last scalar values are interpolated determines the maximum possible 'smoothness' or continuity of such a noise function when tiled. The GLSL `smoothstep` function is one such a function, which uses cubic Hermite interpolation after clamping. The resulting continuity is C^1, since at its endpoints both its 0th- and 1st-derivatives are 0. Of course, this assumes a regular grid, such that the tangents along the edge of each tile line up.

2.7 Spatial variation of noise through interpolation of parameters

The noise generation can further be influenced by using parameters whose values, instead of being static, are interpolated over the entire patch.

Value noise
2D Perlin noise

Figure 2.3: Examples of Value- and 2D Perlin noise. Both results used the same seed for their hash functions. Note that the Perlin noise here is rescaled to fit the standard [-1, 1] range.

An example of this would be the use of interpolated persistence in fractal noise. Fractal noise is a more complex noise function, which essentially layers low and high frequency noise on top of each other to create a more varied noise pattern. The amplitude of the generated noise decreases exponentially the higher its frequency. The `persistence` parameter (between 0 and 1), dictates how quickly this occurs.

Or, as described by a formula:

$$fractal\_F(x) = \sum_{i=0}^{N} (persistence^i \cdot F(2^i \cdot x))/normK$$

$$= \sum_{i=0}^{N} (persistence^i \cdot F(freq_i \cdot x))/normK$$

with $normK = \sum_{i=0}^{N} persistence^i$, F being the base noise function, x being the point's coordinates on the patch, $freq_i$ being that iteration's noise frequency, and N being the maximum number of iterations. Note that $normK$ merely serves to normalise the resulting noise (to a range of $[-1.0, 1.0] \subset \mathbb{R}$). Furthermore, $normK$ is a constant, since $persistence$ is a constant as well.

By varying the `persistence` between vertices, we can generate smooth differences in detail within the noise field.

In conclusion

Using these techniques and theory, notably: the ACC2 framework, surface continuity, displacement mapping, and noise functions, section 3 will describe how they will be used for generating continuous patch-based noise.
Note that the theory on half-edges is included mainly to understand the ACC2 framework better. Additionally, the theory on 'spatial variation of noise through interpolation of parameters' is mainly used in section 4.4, where results are shown featuring continuous fractal noise with interpolated values for `persistence`.

(Since the domain for parameters like `persistence` is continuous over the mesh already, as they are only tied to mesh vertices, we do not need to concern ourselves with a different implementation to account for continuity over patch edges.)

3 Per-patch noise

When creating continuous, lattice-based per-patch noise, assuming that our noise functions are continuous to begin with, there are several main issues that need to be addressed:

First of all, we have to ensure that neighbouring patches agree on the final noise value along shared patch edges. This will make the surface along patch edges C^0. Additionally, we will have to ensure that neighbouring patches agree on the tangent vectors of the noise along shared patch edges, making them either G^1 or C^1.

Furthermore, we work under two main assumptions. Our first assumption is that the lattice-based noise functions make use of a hash function based on uv-coordinates. This hash function returns a (potentially 1D) vector. (We will call this a 'hash-vector' in the rest of this thesis.) Furthermore, we assume that the feature points of these noise functions lie on the vertices of the lattice. Finally, we assume is that the mesh the noise function is applied to is locally (roughly) coplanar along patch edges.

To address these issues, we have opted to implement these strategies:

First of all, we create a continuous mesh-wide domain to replace the functionality of a global uv-coordinate system. We will use this as the input of our hash function, instead of using patch-based uv-coordinates. (see section 3.1)

Next, we restrict our feature points to the vertices of an NxN grid, where N is integer, such that it is symmetrical. This will keep the lattice continuous over patch edges, as differing patch-orientations with a non-symmetrical lattice could cause misaligned feature points along patch edges.

Furthermore, we implement a patch-edge based coordinate system for maintaining independence of the neighbouring patches' uv-coordinate system's orientation. This is in conjunction with the aforementioned constructed domain. (see section 3.2)

The next strategy is to use static hash-vectors for patch vertices, in order to deal with the fact that valency can vary for patch vertices. (see section 3.3)

Last but not least, to satisfy our 'locally coplanar' assumption: we apply our function to densely tessellated meshes, such that locally the resulting faces are roughly coplanar.

In section 3.5 we will explain our newly created Bézier noise function. In section 3.4 we will explain how we adapted the standard Value noise, while we do the same in 3.6 for 2D Perlin- and Bézier noise.

3.1 Continuous mesh-wide domain and hash-vectors

For our domain, we opted to include the patch's vertex indices into the hash function. (I.e. `hash(uv, index)`. This way, when computing the hash-vector of a grid-vertex on the patch, we simply bilinearly interpolate between the four hash-vectors of the patch vertices.

However, the uv-coordinates of a point on a patch-edge will most likely differ depending on the patch the point is viewed from, given that they may differ in orientation. Therefore, along patch edges we have opted to switch from patch-based uv-coordinates to edge-based coordinates along patch edges, when computing hash-vectors.

With these changes, we can implement continuous noise for Value noise, as well as more complicated noise functions, such as Perlin- and Bézier noise.

3.2 Edge-based coordinate system

Our edge-based coordinate system is very similar to the definition of a point on a line (i.e. only using a single parameter t). However, in our case we also need to indicate the direction of t along the edge, and thus create an agreed-upon secondary y-axis, even if the y value of points along the edge are always 0. To this end, we use one of the two vertices as origin point of the coordinate system. In our implementation, we simply chose the smallest of the two vertex indices. Finally, we also use the origin-point vertex index as the second part of our hash-function input. See figure 3.1 for a visual example of such a coordinate system.

When using this coordinate system as input for our hash function, both patches use the same input, and thus agree on the resulting output. Note however, that the direction of this output vector is based on the basis vectors of this coordinate system, and not of the patches themselves.

Furthermore, note that the resulting hash vectors along such an edge may not be unique within the patch. For example, the hash vectors generated along the edge BC are the same as the hash vectors generated along the edge BF, as both have vertex B as their origin.

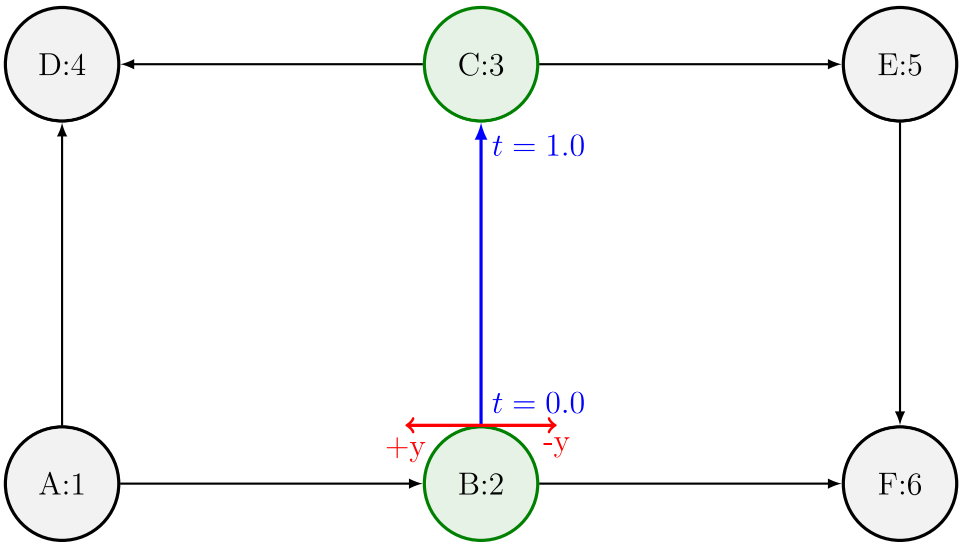


Figure 3.1: An example the directions of the edge-based coordinate systems of two adjacent patches, with the vertex indices shown next to their name. Highlighted is the edge BC, where the t-axis is highlighted in blue, and the direction of the hypothetical y-axis is highlighted in red.

3.3 Static hash-vector for patch vertices

As stated earlier, in the case of patch vertices, their valency may vary. This in turn makes it rather difficult, if not impossible, to create a (simple) solution to ensure that all connected patches agree on the objective hash-vector of the vertex. Therefore, for simplicity's sake, we opted to use the static 0-vector for patch vertices. This way all connected patches do agree on the hash-vector, and the noise value on that point on the mesh will be 0 as well. This in turn guarantees C^0 continuity at patch vertices.

3.4 Value noise adaptation

The value noise adaptation for ensuring continuity over patch edges is relatively trivial. The only thing we change is how the scalar hashes on each grid vertex are computed:

- On a patch vertex: Return 0. Note that, because of this, if $N = 1$, the noise for the entire patch will be 0 as well.
- Inside the patch: Interpolate between the hashes that the patch vertices normally would get. We ignore the 0-override here, since otherwise this would only result in a noise of 0. The continuity of the noise remains guaranteed, since we bilinearly interpolate over the local grid-cell.
- On a patch edge: Transform the uv-coordinates from patch-space to edge-space and compute the hash from there. Since this hash is a scalar, and therefore has no direction, we don't need to worry about our change of basis vectors.

See the appendix (section A.1) for the full adapted version.

3.5 Bézier noise

Bézier noise is a noise function we designed with the express purpose of being C^1 continuous along the edges of its lattice, in the hopes that this would be easier to ensure C^1 over patch edges. The unmodified Bézier noise function essentially creates a 2D Bézier-spline surface, creating a bicubic patch inside each grid-cell. The control points for these bicubic patches are determined by the hash-vectors of the grid-cell's four vertices.

Of interest is an example of how a curve of this surface is defined along the lattice, in the u direction, over two edges:

We denote our main controls points on the lattice vertices as A, B and C, which in turn form two continuous curves AB (red) and BC (blue), see figure 3.2. Each main control point is used to generate two other child-control points, e.g. for point P this would be the points P_+ and P_-. Furthermore, we denote the hash vector of a grid-cell vertex P as P_{hash}. Finally, we denote the u or v component of a point vector V as $V.u$ and $V.v$ respectively.

The child-control points of control point B would then be computed as such:

$$B_+ = B + [\frac{1}{3}(C - B).u, B_{hash}.u]$$

$$B_- = B - [\frac{1}{3}(B - A).u, -B_{hash}.u]$$

In the case of figure 3.2, $B_{hash}.u$ would be -0.8. This definition of the child-control points ensures that the tangent vector at B in the u direction is the same for both curves AB and BC, and therefore the entire curve AC is C^1 continuous at point B.

This same technique is applied for Bézier-splines in the v-direction. The final Bézier-spline surface is obtained as the tensor product between these two Bézier-splines. See figure 3.3 for a schematic of how these control points lie within a single grid-cell.

Such a Bézier-spline surface lends itself for C^1 continuity when tiling, using a regular grid. This property naturally extends to C^1 continuity over patch edges, as long as both patches use the same basis vectors for their coordinate system (i.e. they are oriented in

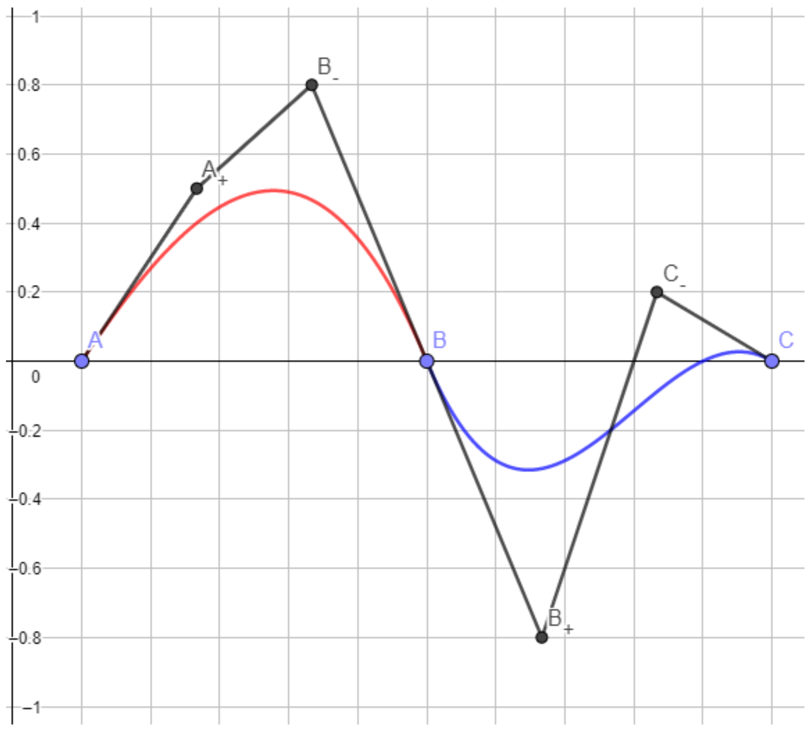


Figure 3.2: A Bézier-spline, together with its control polygon. It is constructed such that the spline is C^1 continuous at control point B.

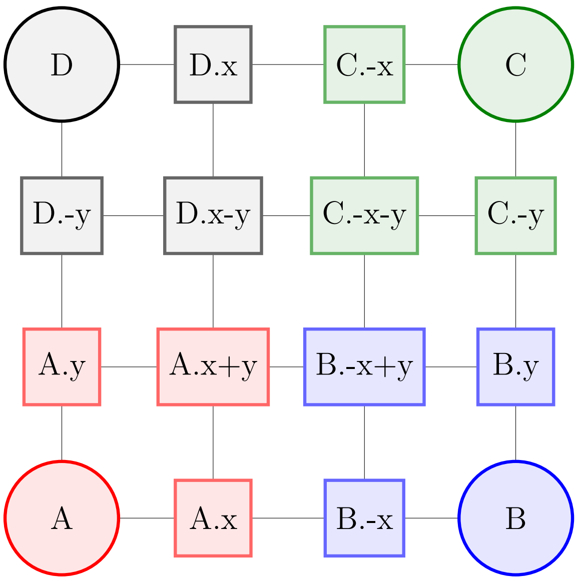


Figure 3.3: A schematic for how the control points of our Bézier surface lie within a single grid cell. The control points are colour coded to show which parent-control point controls which child-control point.

the same direction).

An example of the resulting Bézier noise can be seen in figure 3.4. Note the similarities between Bézier noise and Perlin noise. However, note that the 2D Perlin noise has been scaled to fit the standard $[-1, 1]$ range, since its normal range is $[-\sqrt{2}, \sqrt{2}]$. [2] From this we can see that Bézier noise seems to have more energy than Perlin noise, and that both noise functions seem to generate an almost identical looking pattern. Furthermore, upon closer inspection, the lattice edges are more visible with Bézier noise. This may be due to Bézier noise's internal C^2 continuity, while on the edge of its tile it is only guaranteed to be C^1 continuous.

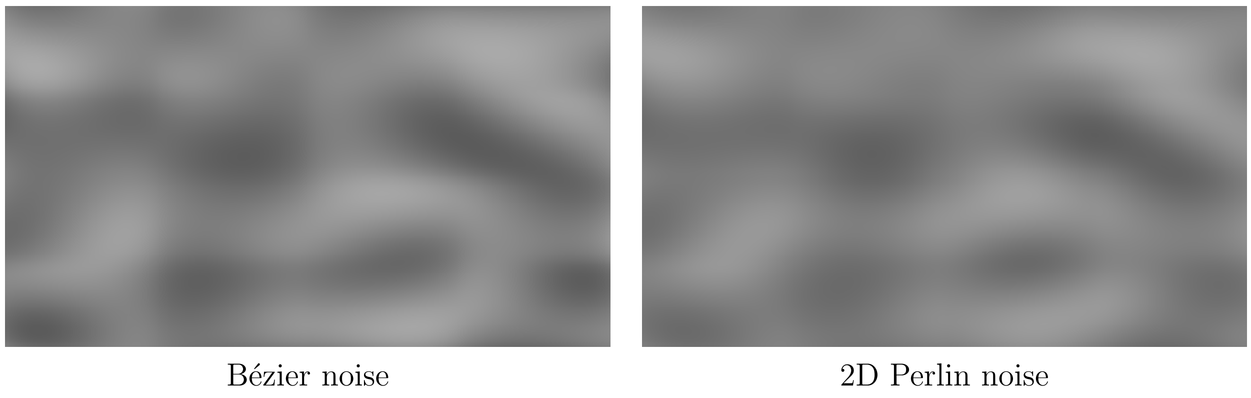


Figure 3.4: A comparison between Bézier- and 2D Perlin noise with the same seed for their hash functions. The used grid is 4x4 cells. Note that the Perlin noise has ben rescaled to fit the standard [-1, 1] range.

3.6 2D Perlin- and Bézier noise adaptation

The changes to 2D Perlin- and Bézier noise are virtually the same as the ones applied to Value noise. However, these noise functions have the added complication that the return value of their hash function is a proper vector, instead of a scalar. This re-introduces the direction of the hash-vector into the equation: if implemented in the same way as Value noise, then patches with differing orientations will use the same vector *as if* this vector is computed using their own coordinate-system's basis. Evidently, this is not the case, and as such, discrepancies between the two patches along the edge will occur.

The answer to this is simple: transform the hash-vector back from the edge-coordinate system's basis to the patch-coordinate system's basis.

Thus, we use the following decision scheme when computing the hash vectors in both noise functions:

- On a patch vertex: Return a 0-vector. Note that, because of this, if $N = 1$, the noise for the entire patch will be 0 as well. This applies to both 2D Perlin noise and Bézier noise.

- Inside the patch: Interpolate between the hashes that the patch vertices normally would get. We ignore the 0-vector override here, since otherwise this would only result in a noise of 0. The continuity of the noise remains guaranteed, since we bilinearly interpolate over the local grid-cell.

- On a patch edge: Transform the uv-coordinates from patch-space to edge-space and compute the hash-vector from there. Since our hash-vector is oriented with respect to our edge-space, we will need to transform it back to its respective patch-space in order for it to be interpreted correctly.

Note that, while Bézier noise does not generate a gradient-vector like Perlin noise does, it *does* generate two patch-orientation dependent values which it uses to generate its child-control points. Since these two values depend on the orientation of the patch, together they function like a vector, which is why we can apply the same procedure to both types of noise functions.

For the full description of the adapted 2D Perlin- and Bézier noise algorithms, see appendix A.2 and A.3 respectively.

3.7 Adapted noise function continuity

The adaptations to Value-, 2D Perlin-, and Bézier noise have interesting repercussions for the continuity along patch borders. Here we have two possible scenarios to inspect. The first one is the 'regular' case, where the *lattice* on the patch edge is continuous. The other one is the 'irregular' case, where the lattice on the patch edge is C^0 continuous, but not C^1 (i.e. the tangents of the first patch do not line up with the tangents of the second patch, see figure 3.5).

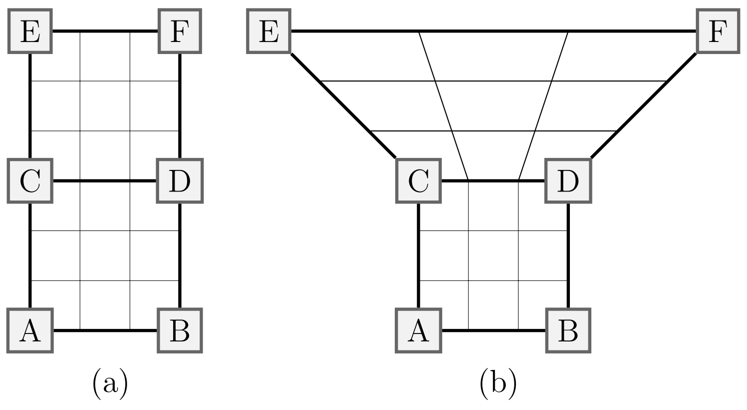


Figure 3.5: Examples of a regular lattice (a) and an irregular lattice (b) along the patch edge CD.

Regular case

For all three noise functions, the continuity is the same as their original versions: C^1 continuity along both patch vertices and patch edges. First of all, C^0 continuity is achieved by making both patches agree on the hash-vectors: the use of the static 0-hash vector on patch vertices and an edge-based coordinate system ensures this. While Bézier noise has C^1 edge (and vertex) continuity built in (when applied to this regular case, see section 3.5), Value- and Perlin noise rely on the GLSL `smoothstep` function to ensure C^1 continuity (see section 2.6).

Irregular case

Like in the regular case, the patch vertices remain C^1, since the tangent vectors there have a magnitude of 0, stripping them of direction. Patch edges, however, are diminished to C^0 continuity, approaching C^1, since the magnitude of the tangent vectors of both adjacent patches remain equivalent, while their direction varies.

4 Results

We implemented patch-based versions of these lattice-based noise functions: Value-, 2D Perlin- and Bézier noise. Note that our interface allows for the adjustment of several global noise parameters (located to the left of the viewing window):

- `Noise factor`: A floating point multiplier that serves to reduce the raw output noise value, since a displacement of 1 can be far too much for certain fine meshes. `Noise factor` $\in [0,1] \subset \mathbb{R}$.
- `Scale` (noise frequency): An integer multiplier that influences the noise frequency of both u and v coordinates of the patch. (e.g. with a `scale` of 2, $UV \in [0,2]^2 \subset \mathbb{R}^2$) The reason why it govern both is because our method requires a symmetrical square lattice.
- `Noise type`: A set of buttons to choose the type of noise to be applied.
- `Fractal noise`: A checkbox for applying fractal noise with the currently selected noise type. `Max Octaves` governs the amount of octaves (or 'noise layers'/iterations) that will be used in the fractal noise algorithm.

Furthermore, we also allow for the manipulation of vertex-based parameters (located on the right side of the viewing window). For now, the only noise-based parameter is `Persistence`, which is used for fractal noise computations.

See figure 4.1 for an example of this GUI. Note that the GUI has many other (for this thesis) relatively irrelevant parts, since this prototype was built on top of an existing, trimmed down, renderer.

Finally, note that our renderings of 2D Perlin noise are scaled to fit the standard $[-1,1]$ range, to be able to aptly compare the three implemented noise functions, since the range of N-dimensional Perlin noise is $[-\sqrt{\frac{N}{4}}, \sqrt{\frac{N}{4}}]$.[2] 2D Perlin noise thus normally has the range of $[-\sqrt{\frac{1}{2}}, \sqrt{\frac{1}{2}}]$ or roughly $[-0.707, 0.707]$.

4.1 Implemented for Value-, 2D Perlin-, and Bézier noise

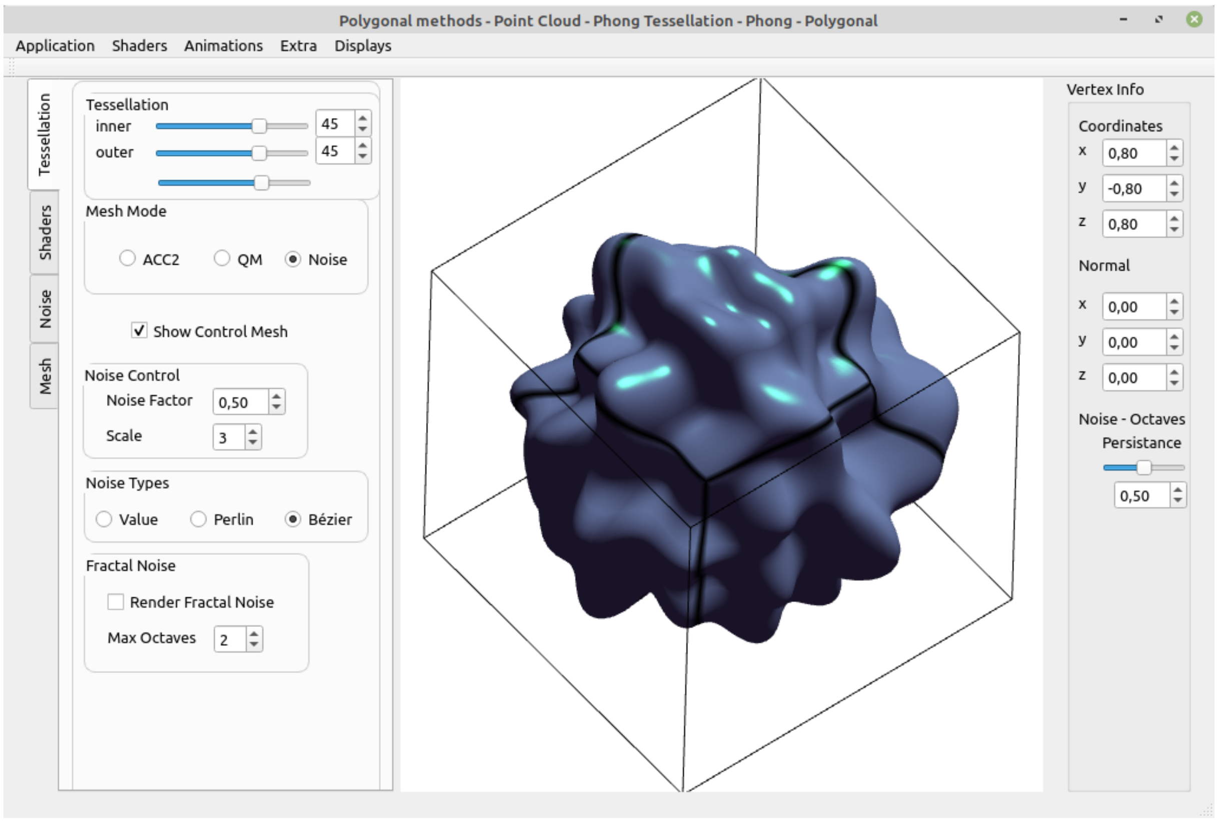
As stated before, we implemented patch-based versions of Value-, Perlin- and Bézier noise. Examples of this, applied to a simple, completely coplanar '2quad' mesh, are shown in figure 4.2 This once again shows us how similar 2D Perlin noise and Bézier noise are, as noted in section 3.5.

4.2 Examples of noise applied to different objects/meshes

These noise functions can be applied to non-coplanar meshes as well. Examples can be seen in figure 4.3. From the lighting in 4.3.(c) we can clearly see that, while patch edges certainly are not C^1, the surface is at the very least C^0.

4.3 Example of patch-orientation independence along patch edges

Along patch edges, our method is patch-orientation independent, which means that the resulting noise values do not depend on the patch's orientation. With patch-orientation, we mean the orientation of the uv-coordinate system within the patch. Our orientation


Polygonal methods - Point Cloud - Phong Tessellation - Phong - Polygonal
Application
Shaders
Animations
Extra
Displays
Tessellation
Shaders
Noise
Mesh

Tessellation
inner 45
outer 45

Mesh Mode
ACC2 QM [✓ Noise]

[✓ Show Control Mesh]

Noise Control
Noise Factor 0,50
Scale 3

Noise Types
Value Perlin [✓ Bézier]

Fractal Noise
Render Fractal Noise
Max Octaves 2

Vertex Info
Coordinates
x 0,80
y -0,80
z 0,80
Normal
x 0,00
y 0,00
z 0,00
Noise - Octaves
Persistance
0,50

Figure 4.1: A screenshot of our application, showing the user interface. Global tessellation and noise parameters are located left of the viewing window, while vertex based parameters are located to its right.

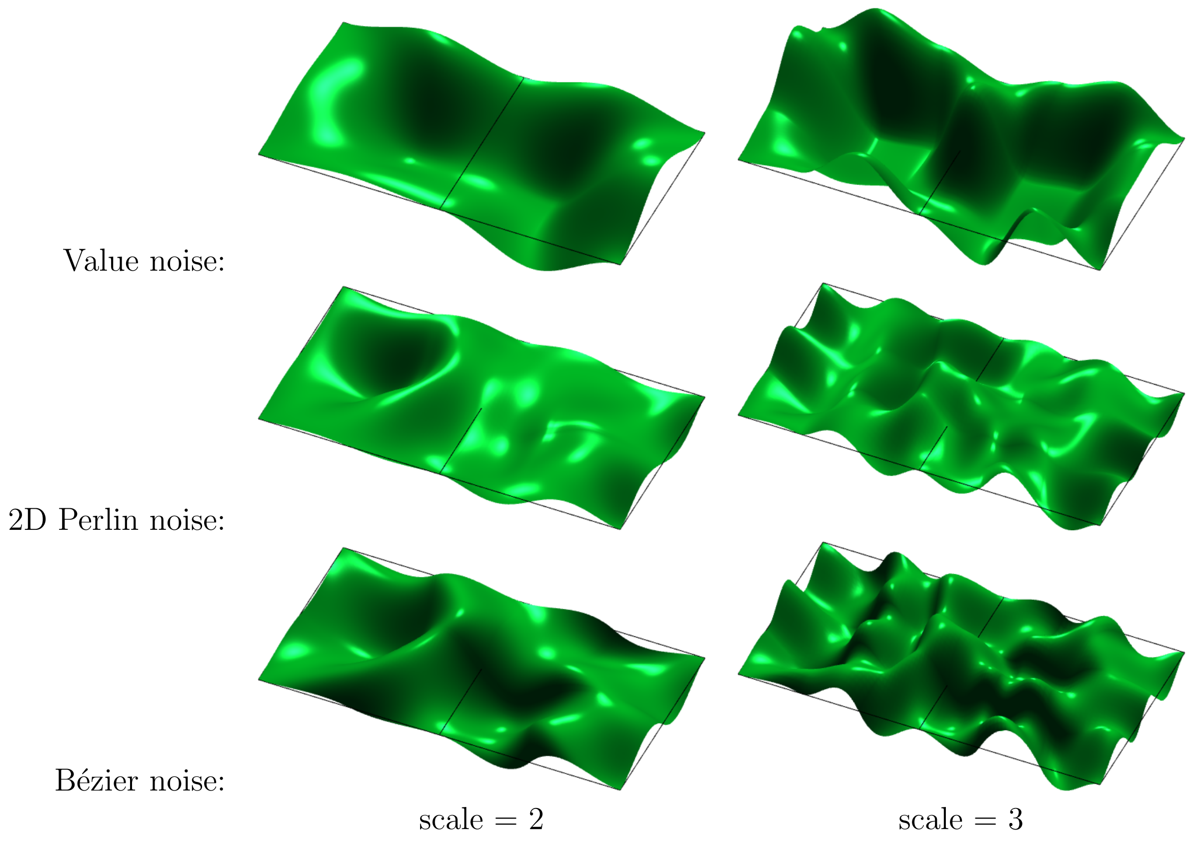


Figure 4.2: Noise functions applied to two adjacent patches and differing noise frequency (scale), including the control mesh. Note that these patches are irregular, since no patch vertex has a valency of 4.

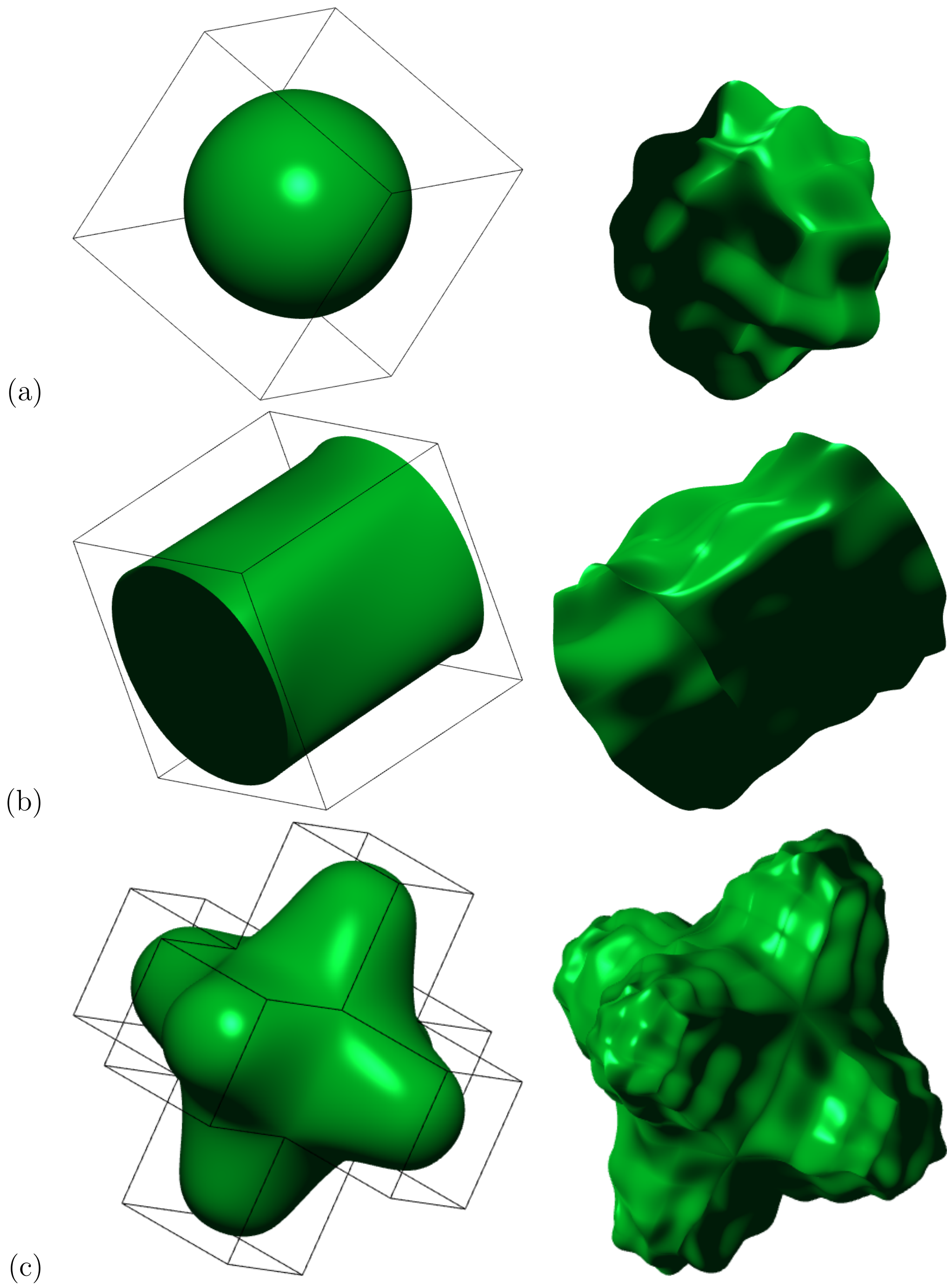
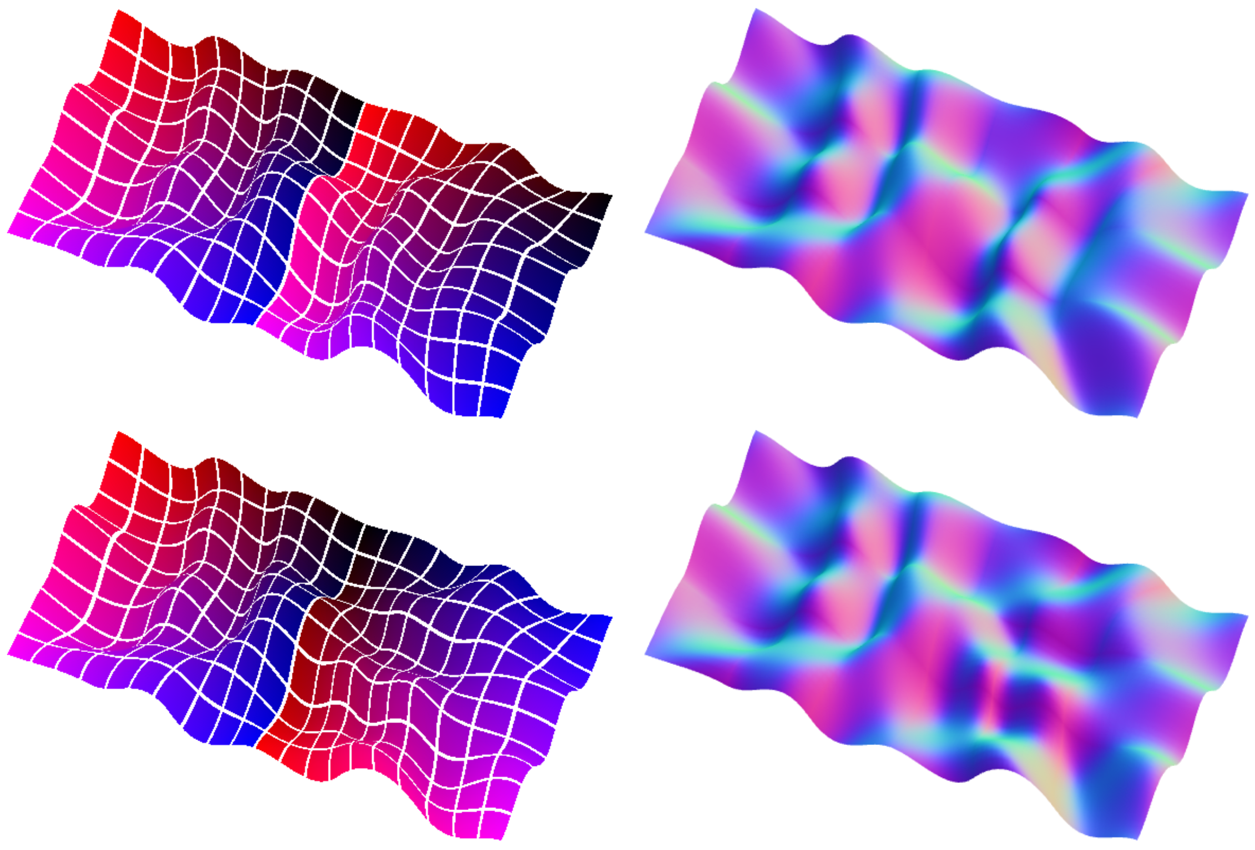


Figure 4.3: 2D Perlin noise applied to a spherical mesh (a), a tubular mesh (b), and a cross-esque mesh made of cubes (c).

independence stems from the fact that we use the (readjusted) hash-vectors obtained from edge-coordinate space instead of patch-coordinate space. (It goes without saying, of course, that the internal hash-vectors of a patch *do* rely on patch-coordinates rather than edge-coordinates. This, combined with the fact that the internal grid of a patch with `scale` > 2 is not symmetrical, means that the internal noise of said patch is very much patch-orientation dependent.)
Patch-orientation independence is important, as it is required for a smooth transition between patches in closed meshes, since closed meshes *require* some patches to have different orientations. (For example, imagine a cube whose front, side and back faces all have the same orientation. In this setup, the top and bottom face will be misaligned for 3 out of 4 of these aforementioned faces.)
Next are some example screenshots from the application: Figure 4.4 shows several examples of the patch-orientations by displaying the uv-coordinates with colour gradients (u = red, v = blue). From this, we can see that when changing the orientation of one of the patches, the resulting displacement and normal vectors remain continuous along the patch edge.



(a) UV-grid orientation (b) Normal field

Figure 4.4: Bézier noise applied to two adjacent patches, shaded by its local patch uv-coordinates (a) and its normal field (b). The patches of the top mesh have the same orientation, while the right patch of the bottom mesh is turned 90 degrees counterclockwise.

4.4 Examples of vertex-based persistence variation regarding fractal noise

Using persistence values as vertex attributes, we can interpolate over each patch to create a 'persistence field'. This allows us to smoothly vary persistence over the patch, to be used in the fractal noise function. Figure 4.5 shows the impact of this on spherical mesh, where the persistence attribute of the frontal middle vertex varies between 0 (which essentially is equivalent to using the non-fractal variant of the noise function), and 0.99.

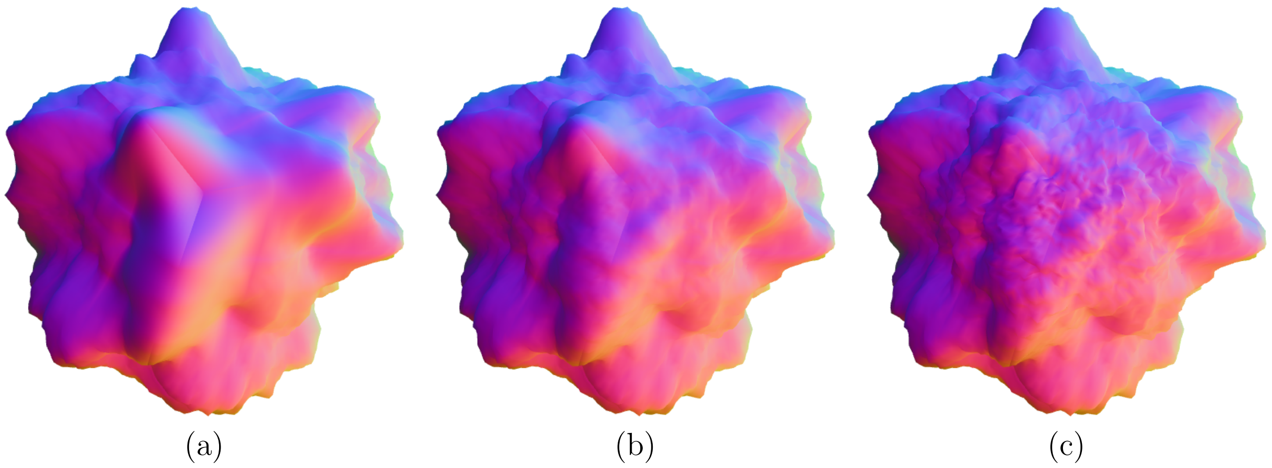


Figure 4.5: An illustration of fractal Value noise applied to a spherical mesh, with varying values for the persistence attribute of the frontal middle vertex. These values are as follows: (a) = 0, (b) = 0.5, (c) = 0.99. The rest of the vertices have a standard persistence value of 0.5.

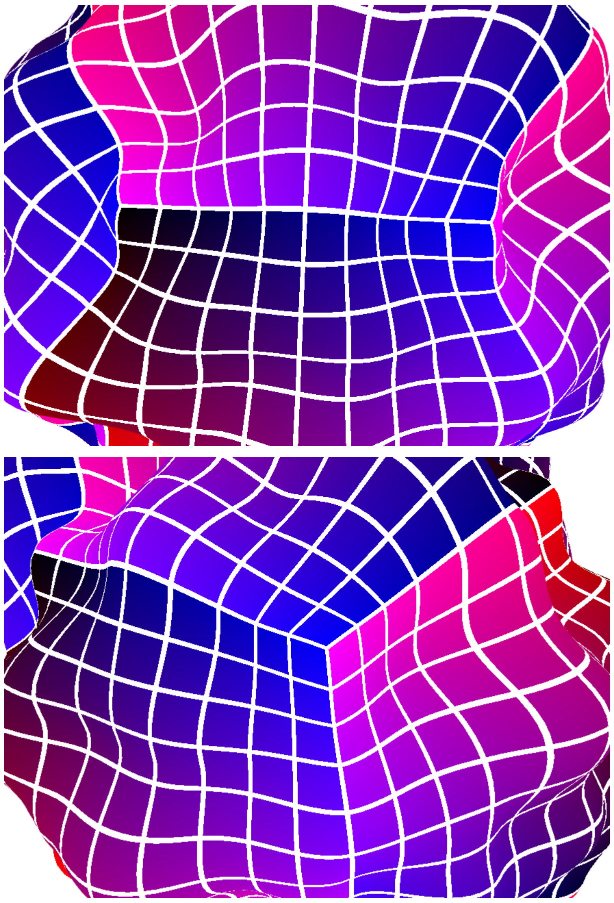
4.5 Continuity and graphical artefacts around patch vertices

One of the main properties of the ACC2 framework is that the resulting base surface is G^1 along patch edges of a patch that has at least one irregular vertex (i.e. a vertex with a valency not equal to 4). However, note that this only applies to surface continuity, and to tangent-*planes*, rather than vectors, since we are working with surfaces. As such, the G^1 base surface continuity does not guarantee a regular lattice over a patch edge. However, since *regular* patch edges have C^2 surface continuity, the normal field is guaranteed to be C^1 continuous, and thus we *are* guaranteed to have regular lattices along these patch edges.

When applying our noise functions to this base mesh, the noise function continuities described in section 3.7 (C^1 continuity for regular lattices, C^0 to C^1 continuity for irregular lattices) are then bound by the surface continuity of said base mesh, which leads to graphical artefacts along patch edges. Furthermore, since our ACC2 framework does not guarantee lattice continuity along irregular/extraordinary patch edges, we will have to assume the 'irregular lattice' case for these patch edges.

Therefore, along regular patch edges, the applied surface continuity will be C^1. Along irregular/extraordinary patch edges the applied surface continuity will be C^0, approaching C^1, and G^1 on patch vertices, since it is bound by the G^1 base surface.

Figure 4.6 shows one such aforementioned artefact along a patch edge. The spherical mesh here only contains these irregular patches, since it is derived from a cube, making the base surface continuity along patch edges G^1. What is clear to see is that the edges are distinctly visible in that area, owing mostly to the irregular lattice. Furthermore, we can see that the lattice over the edge becomes less regular the closer it gets to a patch vertex. Finally, we can see that very close to the patch vertex itself, the normal colours blend together rather than staying in high contrast to one another. This signifies that the continuity on the patch vertex is indeed G^1.


(a) Lattice

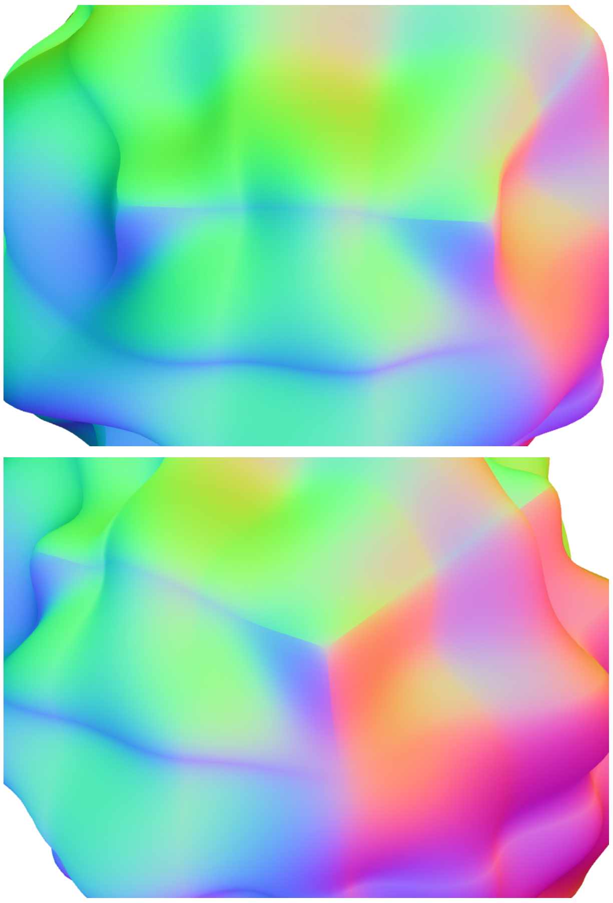

(b) Normals

Figure 4.6: A zoomed in view on both a patch edge and patch vertex of a sphere mesh on which 2D Perlin noise is applied. (a) uses the uv-map to show a rough estimate of each patch's lattice, while (b) shows the normal map across the mesh.

5 Discussion

5.1 Comparison with other approaches

First of all we will compare our approach to two other ways of generating noise:

'Standard' surface noise: Computing a noise map using global uv-coordinates. The positive side of this method is that it is conceptually simple and easy to implement. One of the downsides, however, is that it is set-up noise: the global uv-coordinates will have to be precomputed. Furthermore, the surface will have to be flattened in order to map it to the standard $[0,1]^2 \in \mathbb{R}$ range. For closed meshes, this requires tears/cuts to be made along the surface, which can result in irregularities around those areas.

Solid noise: Computing a global noise map using 3d mesh coordinates. We create a continuous 3D-noise field (e.g. a cube) around the mesh and map each point on the mesh to a point in that noise field. The upsides to this method is that there are no tears or irregularities within the map, with guaranteed smooth transitions between patches. Lagae et al. [3] show that this can be achieved without any setup procedure. However, one important downside is that it does not offer the same level of control on a local vertex/patch level as patch-based noise does, as the noise field is not inherently linked to the surface of the mesh.

5.2 Up- and downsides of our method

Upsides:
Our method has several upsides as well.

For one, we perform noise and uv-coordinate computation on the fly: since our method is designed to be applied to tessellated patches, we can recompute our uv-coordinates at run-time, and thus our noise as well. This can be rather useful for applications where the mesh itself changes shape during run-time, such as animations or mesh-design programs.

Furthermore, our method only uses local patch data. This mainly is influenced by our choice for using tessellated base meshes. Furthermore, the extra patch data that we use is stored on a patch-vertex basis, which being sparse in and of itself is also shared by multiple patches, resulting in a low memory cost.

Along regular patch edges, our method results in at least C^1 continuity.

Downsides:
Of course, our method has several downsides and caveats too.

First of all, we are limited to using symmetrical square lattices. This restricts the scaling factors of both our u- and v coordinates to integers of the same value.

Furthermore, we have created this method for lattice-based noise functions with feature points on lattice vertices. It is doubtful that this method works in its entirety for other kinds of noise functions, especially ones that require access to feature points outside of the inspected grid-cell.

As for continuity, while C^0 continuity along extraordinary patch edges could be considered acceptable, these patch edges are practically unavoidable when dealing with closed quad-meshes. This in turn will result in visible irregularities in the normal field along these edges.

Finally, this method has only been implemented for quadrilateral patches, while more complex meshes may use triangular patches as well.

6 Conclusions

We were able to implement continuous patch-based noise functions for quad meshes. This includes Value noise, 2D Perlin noise, and our new Bézier noise.

Along patch edges with a regular lattice/grid, these patch-based noise functions should have the same continuity as their non patch-based counterparts. However, in the case of an irregular lattice/grid, this continuity drops to C^0, while approaching C^1.

These implementations use both the guaranteed tangent-plane continuous tessellated mesh provided by the ACC2 framework, as well as the GLSL `smoothstep` interpolation function with Value- and Perlin noise. In this setup, the earlier noted continuity is bounded by the base surface's continuity, resulting in C^1 continuity for regular patch-edges and C^0 to G^1 (approaching C^1) continuity for irregular patch-edges.

However, there remain some caveats to this:

We have applied this method only to lattice-based noise functions with feature points on lattice vertices, and it is doubtful that this method works for other noise functions that require access to feature points outside of the inspected patch.

Furthermore, our method assumes a densely tessellated (at least G^1) mesh as base. The less co-planar the faces are along the patch edges, the more our method will approach C^0 continuity along patch edges.

Our method works only with symmetrical, square lattices. This in turn restricts our `uv-scale` factor to integer values.

Finally, our method has been designed for quadrilateral patches.

7 Future work

The results of this research allow for a number of avenues to explore.

One such avenue would be to look into noise functions that do not restrict their feature points to the vertices of a lattice, and whether our edge-based coordinate system would be useful in those scenarios. An example of such a noise function would be Worley noise. Worley noise randomly distributes its feature points *within* the cells of a grid-structure. Furthermore, it allows the use of feature points *outside* of the current cell. This gives rise to the edge case where feature points are checked that lie completely on an adjacent patch. This conflicts with our (self-imposed) limitation of only being able to access the data of the current patch.

Another possible, though admittedly strange, avenue is to explore the possibility of continuous patch-bound noise functions: noise functions whose domain is only a single patch, or in general a limited number of patches, but remain continuous over all patch edges.

A final possible avenue is to implement this method for triangular patches, rather than quadrilateral patches. While the shape of the patch's lattice may differ, the general approach regarding patch orientation and edge-coordinates should still hold.

Bibliography

[1] C. Loop, S. Schaefer, T. Ni, and I. Castano, “Approximating subdivision surfaces with gregory patches for hardware tessellation,” in *ACM SIGGRAPH Asia 2009 papers*, pp. 1–9, 2009.

[2] R. Chen, “Perlin noise range.” Weblog, June 2017. (accessed: 7 Februari 2022). Available: https://digitalfreepen.com/2017/06/20/range-perlin-noise.html.

[3] A. Lagae, S. Lefebvre, G. Drettakis, and Ph. Dutré, “Procedural noise using sparse gabor convolution,” *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)*, vol. 28, pp. 54–64, July 2009.

A Appendix

A.1 Complete adapted Value noise algorithm

The entire adapted form of the Value noise algorithm:

1. Generate a grid consisting of NxN grid cells, where N is integer and equal to the noise frequency (or `scale` in our GUI).
2. When computing the scalar hash for each grid vertex:
   - On a patch vertex: Return 0. Note that, because of this, if $N = 1$, the noise for the entire patch will be 0 as well.
   - Inside the patch: Interpolate between the hashes that the patch vertices normally would get. We ignore the 0-override here, since otherwise this would only result in a noise of 0. The continuity of the noise remains guaranteed, since we bilinearly interpolate over the local grid-cell.
   - On a patch edge: Transform the uv-coordinates from patch-space to edge-space and compute the hash-vector from here. Since this vector is a scalar, and therefore has no direction, we don't need to worry about our change of basis vectors.
3. Interpolate between the hash-vectors of each vertex of the local grid cell. The interpolation function used here determines the resulting continuity.

A.2 Complete adapted 2D Perlin noise algorithm

The entire adapted form of the 2D Perlin noise algorithm:

1. Create an NxN grid, where N is an integer and equal to the noise frequency.
2. For each vertex on the grid, create a 2D gradient hash-vector in the following way:
   - On a patch vertex: Return a 0-vector. Note that, because of this, if $N = 1$, the noise for the entire patch will be 0 as well.
   - Inside the patch: Interpolate between the hashes that the patch vertices normally would get. We ignore the 0-vector override here, since otherwise this would only result in a noise of 0. The continuity of the noise remains guaranteed, since we bilinearly interpolate over the local grid-cell.
   - On a patch edge: Transform the uv-coordinates from patch-space to edge-space and compute the hash-vector from there. Since our hash-vector is oriented with respect to our edge-space, we will need to transform it back to its respective patch-space in order for it to be interpreted correctly.
3. For each point on the noise field (on which the gradient field is projected):
   (a) For each of the 4 nearest grid vertices, compute dot product between gradient vector assigned to this grid vertex and the corresponding offset vector (from the grid vertex to the point in the cell).
   (b) Interpolate these four resulting scalar values, weighted by the point's proximity to each corresponding vertex.

A.3 Complete adapted Bézier noise algorithm

The entire adapted form of the Bézier noise algorithm:

1. Create an NxN grid, where N is an integer and equal to the noise frequency.
2. For each vertex on the grid, create a 2D hash-vector in the following way:
   - On a patch vertex: Return a 0-vector. Note that, because of this, if $N = 1$, the noise for the entire patch will be 0 as well.
   - Inside the patch: Interpolate between the hashes that the patch vertices normally would get. We ignore the 0-vector override here, since otherwise this would only result in a noise of 0. The continuity of the noise remains guaranteed, since we bilinearly interpolate over the local grid-cell.
   - On a patch edge: Transform the uv-coordinates from patch-space to edge-space and compute the hash-vector from there. Since our hash-vector is oriented with respect to our edge-space, we will need to transform it back to its respective patch-space in order for it to be interpreted correctly.
3. For each point on the noise field:
   (a) For each of the 4 nearest grid vertices, using each grid vertex's associated hash vector, create the 3 child-control points that lie on the edges of this grid-cell. (see figure 3.3)
   (b) Using the Bernstein basis functions, compute the tensor product of the grid cell for this point.