# Practical Real-Time Hex-Tiling

Morten S. Mikkelsen
Unity Technologies, USA

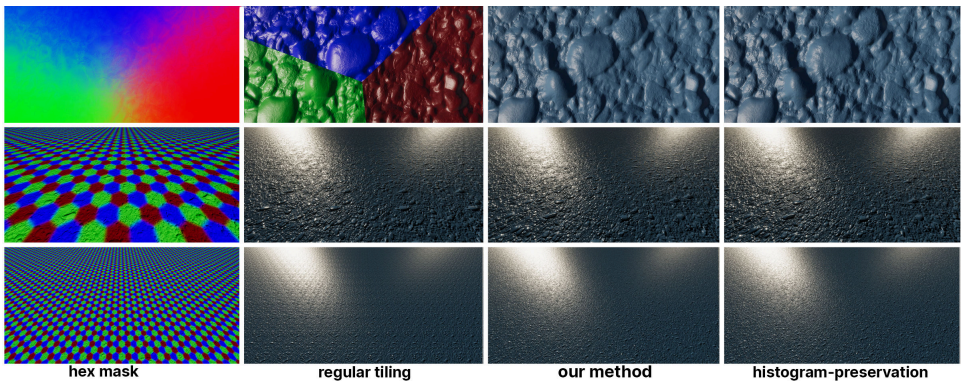


**Figure 1**. Illustration of hex–tiling, Heitz and Neyret, which alleviates repetition and is achieved by each hexagonal region sampling at a random offset. Results are illustrated at different scales in rows 1–3 respectively. The two bottom images of the second column show conventional tiling and repetition is evident at distance. In the fourth column we see the original histogram–preserving method compared to the very similar results in column three using our more simple approach to compositing hex tiles.

## Abstract

To provide a convenient, easy to adopt, approach to randomly tiled textures in the context of real–time graphics we propose an adaptation of the by–example noise algorithm of Heitz and Neyret. The original method preserves contrast using a histogram–preserving method which requires a precomputation step to convert the source texture into a transform– and inverse transform texture which must both be sampled in the shader rather than the original source texture. Thus deep integration into the application is required for this to appear opaque to the author of the shader and material. In our adaptation we omit histogram–preservation and replace it with a novel blending method which allows us to sample the original source texture. This omission is particularly sensible for a normal map since it represents the partial derivatives of a height map. In order to diffuse the transition between hex tiles we introduce a simple metric to adjust the blending weights. For a texture of color we reduce loss of contrast by applying a contrast function directly to the blending weights. Though our method works for color we emphasize the usecase of normal maps in our work because non repetitive noise is ideal for mimicking surface detail by perturbing normals.

 ISSN 2331-7418

## 1. Introduction

A key challenge in modern computer graphics is the time it takes to create a sufficient quantity of artwork to make up a full scene. Using modern workflows such as photogrammetry and sculpting alone, to create complex and expansive surface detail, is impractical both in terms of development time as well as constraints in resolution to a single mesh or texture. A common simple approach to address this is by tiling textures however repetition is noticeable. A possible solution to repetition is to use procedural textures, as analytical functions, such as proposed in Perlin [1985], Worley [1996] and Thibault [2019]. The problem with such methods is they are expensive and they lack artistic control to choose a pattern.

A by–example noise algorithm is introduced in Heitz [2018] which allows the user to choose a stochastic texture as a sample. Their method then synthesizes an infinite non repetitive output with the same appearance. This is achieved by structuring the synthesized texture space on an equilateral-triangle lattice also famously used in simplex noise by Perlin [2002]. Each vertex in the grid represents the center of a hexagonal shape which we refer to as a hex tile and is illustrated on the left in Figure 2. Each such tile is assigned a random offset when the source texture is sampled. During synthesis the sampling location represents a barycentric coordinate within a triangle of the lattice. This coordinate is then used to blend between the three corresponding hex tiles.

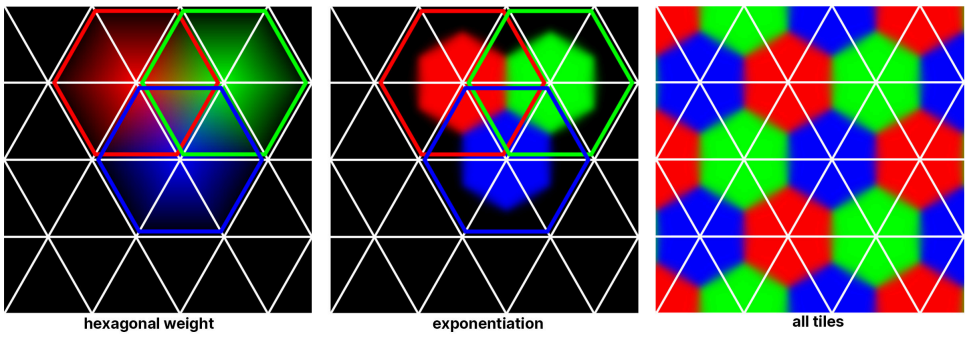


**Figure 2.** Left: Barycentric coordinate used to establish the hexagonal weight. Middle: Exponentiation applied to the hexagonal weight which separates the hex tiles. Right: Shows all distributed hex tiles of the lattice.

It is pointed out in Heitz [2018] that simply blending these three sampled colors will result in loss of contrast and introduce transitional colors that are distinctly not present in the source texture provided by the user. It is argued the problem to linear blending is that the statistical properties of the input are not preserved, i.e. its histogram. In order to blend in a way that preserves the histogram a precomputation step is performed which does a histogram transformation on the channel data and store it in a new texture $T$. A second texture $T^{-1}$ is generated which is used to perform the

inverse transformation. In the fragment shader the texture $T$ is queried three times (once for each hex tile) followed by an approximate variance–preserving blend function. The second texture $T^{-1}$ is then used to transform the result back into a color.

The histogram–preserved blending technique applied to hex–tiling achieves impressive results. However, to replace the original texture with the textures $T$ and $T^{-1}$ in the background, ie. so these are not explicit within the shader or on the material, requires a considerable amount of foundational effort in order to adopt this technique. In the following we explore a simple adaptation of the method which is straight forward to use inside any shader. For normal maps our approach offers very similar results with the additional benefit of being able to randomize rotation of the hex tiles. For textures of color there is a trade–off in quality when observed side by side though in our experience the compromise is acceptable.

## 2. Our Method

In our work we require a solution that will allow us to adopt hex–tiling instantly in any shader. To achieve this we constrain ourselves to sampling the source texture directly. As mentioned in Section 1 the barycentric coordinate $(\omega_1, \omega_2, \omega_3) \in \mathbb{R}^3$ within a triangle of the lattice is used to blend the samples $x_1, x_2, x_3 \in M$, i.e. one per hex tile, where the dimension of $M$ corresponds to the number of channels and $\omega_1 + \omega_2 + \omega_3 = 1$.

It is pointed out in Burley [2019] that even with the histogram–preservation enabled, though diminished, ghosting remains a problem. To further alleviate the issue the author exponentiates the weights $\omega_i^\gamma$ and divides by the sum of the exponentiated weights. This is illustrated in the middle picture of Figure 2 and more closely resembles a blend mask. The same approach is applicable when blending the samples $x_i$ without histogram–preservation and we adopt this in our solution too. To improve on this we diffuse the transition between adjacent tiles by modulating each exponentiated weight $\omega_i^\gamma$ by a metric $\delta : M \to \mathbb{R}_+$ applied to each corresponding sample $x_i$.

$$\omega_i' = \frac{\delta\left(x_i\right) \cdot \omega_i^\gamma}{\sum_{j=1}^{3} \delta\left(x_j\right) \cdot \omega_j^\gamma} \tag{1}$$

thus we leverage characteristics retained in the image data to diffuse the transition between adjacent hex tiles as illustrated in the upper–left image of Figure 1.

For tangent–space normal maps, specifically, the image data represents directional data rather than color. Though a histogram–preserving approach will work it has less relevance in this context since normal maps tend to form a single wide cluster and then occupy the range semi–evenly.

*Normal Mapping.* As described in Mikkelsen [2020] the tangent–space normal represents the surface normal to an implicit height map $H : \mathbb{R}^2 \to \mathbb{R}$ with partial deriva-

tives $\left(\frac{\partial H}{\partial u}, \frac{\partial H}{\partial v}\right)$ where the normal $\vec{n}$ to the graph of $H$ is given by

$$\begin{aligned} \mathbf{f}(u,v) &= (u, v, H(u,v)), \\ \vec{n} &= \frac{\frac{\partial \mathbf{f}}{\partial u} \times \frac{\partial \mathbf{f}}{\partial v}}{\|\frac{\partial \mathbf{f}}{\partial u} \times \frac{\partial \mathbf{f}}{\partial v}\|} \\ &= \frac{\left(-\frac{\partial H}{\partial u}, -\frac{\partial H}{\partial v}, 1\right)}{\sqrt{1 + \frac{\partial H}{\partial u}^2 + \frac{\partial H}{\partial v}^2}}, \end{aligned} \tag{2}$$

The derivative is a linear operator which means by blending the partial derivatives the obtained result is equivalent to blending directly from the height map. In the context of normal mapping our samples $x_i$ represent the partial derivatives to be blended.

In choosing our $\delta$ we attribute significance to samples based on the steepness of their slope. We empirically choose to use sine to the angle $\theta$ between the normal and the $\vec{z}$-axis as our measure. Given Equation (2) it follows that $\cos^2\theta = \frac{1}{1+\|x_i\|^2}$ and from this we obtain $\sin\theta$ from the following equation when $\beta$ equals 1.

$$\delta_N(x_i) = (1-\beta) + \beta \cdot \sqrt{\frac{\|x_i\|^2}{1+\|x_i\|^2}} \tag{3}$$

We use the parameter $\beta \in ]0;1[$ to control the range for $\delta_N : \mathbb{R}^2 \to [1-\beta; 1]$.

*Color.* In the context of color $x_i$ the choice for $\delta$ is more arbitrary. In our case we use luminance to drive the diffusion between hex tiles.

$$\delta_C(x_i) = (1-\beta) + \beta \cdot \left( x_i^T \cdot \begin{bmatrix} 0.299 \\ 0.587 \\ 0.114 \end{bmatrix} \right) \tag{4}$$

Unlike normals it is more common for colors to form individual separated clusters. Particularly, when these are tileable patterns. This makes the omission of histogram–preservation a more significant problem in this case compared to normal mapping. To address this problem we apply a ramp to $\omega_i'$, shaped like an S–curve, just before blending the samples. To perform the described operation we use the following curve

$$g(x) = \begin{cases} \frac{1}{2}(2x)^k & \text{if } x < 0.5 \\ 1 - \frac{1}{2}(2-2x)^k & \text{if } x >= 0.5 \end{cases} \tag{5}$$

which was introduced by Perlin [1989] and is shown in Figure 3. The author introduces a second curve $k = log_{\frac{1}{2}}(1-r)$ which allows the user to choose the exponent $k$ indirectly on a range which is normalized $r \in ]0;1[$. Note that for $r = 0.5$ the curve will have no impact. For $r \in \left]\frac{1}{2};1\right[$ we get an increase in sharpness at the border between adjacent hex tiles. We divide by the sum of the weights to renormalize prior to blending.

## 3. Implementation

Our implementation is based on the one given in Deliot [2019] and is provided here in the Appendix. The main functions are given in Listing 3 and in Listing 4 for normal maps and color respectively. For the parameters $\gamma$ and $\beta$ in Equations (1), (3) and (4) we are using $\gamma = 7$ and $\beta = 0.6$. Though these could be exposed as values to the user we have found these to be good choices in general. For the curve given by Equation (5) we expose the input parameter $r$ to the user which allows them to preserve contrast. Our implementation is given in Listing 8. An alternative option to using Perlin's version is to use the approximate form given in Schlick [1994].

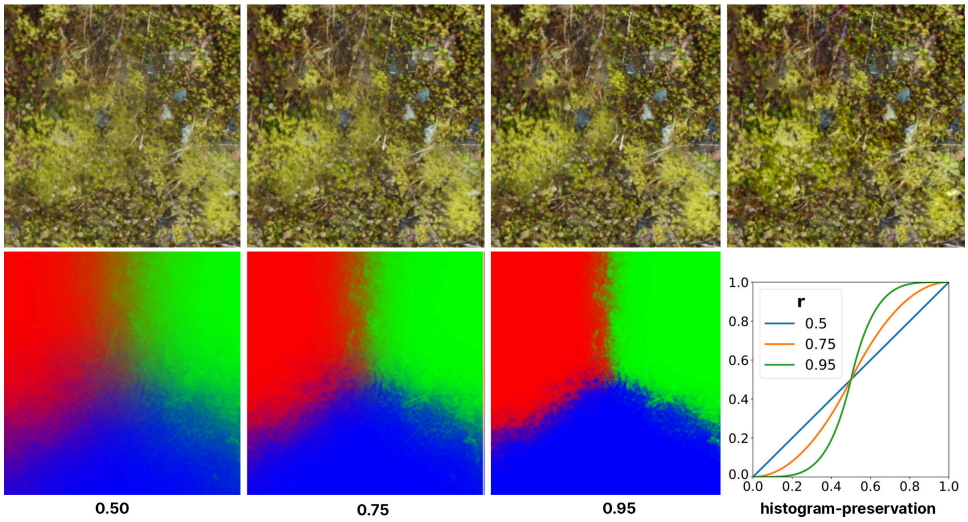


**Figure 3.** On the lower right we see Perlin's S–curve for three different input values $r$. The lower left image shows the mask we produce when the blending weights are modulated by this curve using $r = 0.5$. The corresponding composited image is shown above the mask. Columns two and three show similar results for $r = 0.75$ and $r = 0.95$ respectively and appear less blurry. As ground truth the histogram–preserving result is shown in the upper-right image.

As mentioned in Section 2, for normal mapping, we are blending partial derivatives. We obtain the derivative using the shader function given in Listing 2 in Mikkelsen [2020] but for completion we provide it here as well in Listing 9. When hex–tiling we must use `SampleGrad()` due to the randomized offset applied to the sampling coordinate with respect to each hex tile. We do this in Listing 10 and the returned derivative can be converted to a tangent space normal using `normalize(float3(-deriv.xy, 1.0))`. When using `gFlipVertDeriv` set to `true`, in Listing 9, the second component must also be negated when converting back to a normal. Refer to Mikkelsen [2020] for further details.

To allow for further flexibility we have added support for randomized rotation of

hex tiles. As a note this feature is only possible with normal maps when histogram–preservation is disabled. The reason for this is normals/derivatives represent directional data. Thus when rotating the hex tiles individually we must rotate each sample with respect to its hex tile prior to blending. When histogram–preservation is used the directionality is unknown until blending has already taken place.

## 4. Results

In Figure 4 test results are presented using a selection of different normal map sample textures as the source. In each case the test result is the corresponding synthesized image with twice the width of the source image. A comparison is shown between our method and the original histogram–preserving approach and in our opinion the results are strikingly similar. In practice it is very difficult to tell which is which unless a toggle is done between the two methods.

A similar example is given in Figure 5 where the sample textures are different color patterns. Though our approach, as described in Section 2, is not as ideal for color as it is for normal maps the approach appears to be surprisingly resilient in this case too. As predicted there is a more significant distinction between the two methods compared to normal maps but in our experience it remains difficult to tell which is which until you toggle between them or scrutinize a location that is known to be at the border between two hex tiles such as shown in Figure 3.

When choosing the parameter $r$ this is a compromise between preserving contrast and not making the hexagonal boundaries noticeable. The latter is specific to the chosen source image however in our experience a conservative choice is in the range $r \in [0.65; 0.75]$.

In Figure 6 an example is given of randomized rotation applied to the sampling coordinate during hex–tiling. This is a particularly useful feature when there are recognizeable elements in the sample texture with a distinct orientation. In the middle image where randomized rotation is disabled we see the same pebble appearing at different locations but always with the same alignment. In the top image randomized rotation is applied which alleviates the issue.

A collection of lit materials composited from texture maps for albedo, roughness and normal are shown in Figure 7. On the left side we see a single instance of the material without hex–tiling applied. The corresponding hex–tiled result is shown on the right using a width three times that of the source image. The roughness texture map is sampled using the same approach as we have described for color. For both albedo and roughness we use the parameter $r = 0.75$ and as in Figure 4 the ramp is disabled for normal maps at $r = 0.5$.

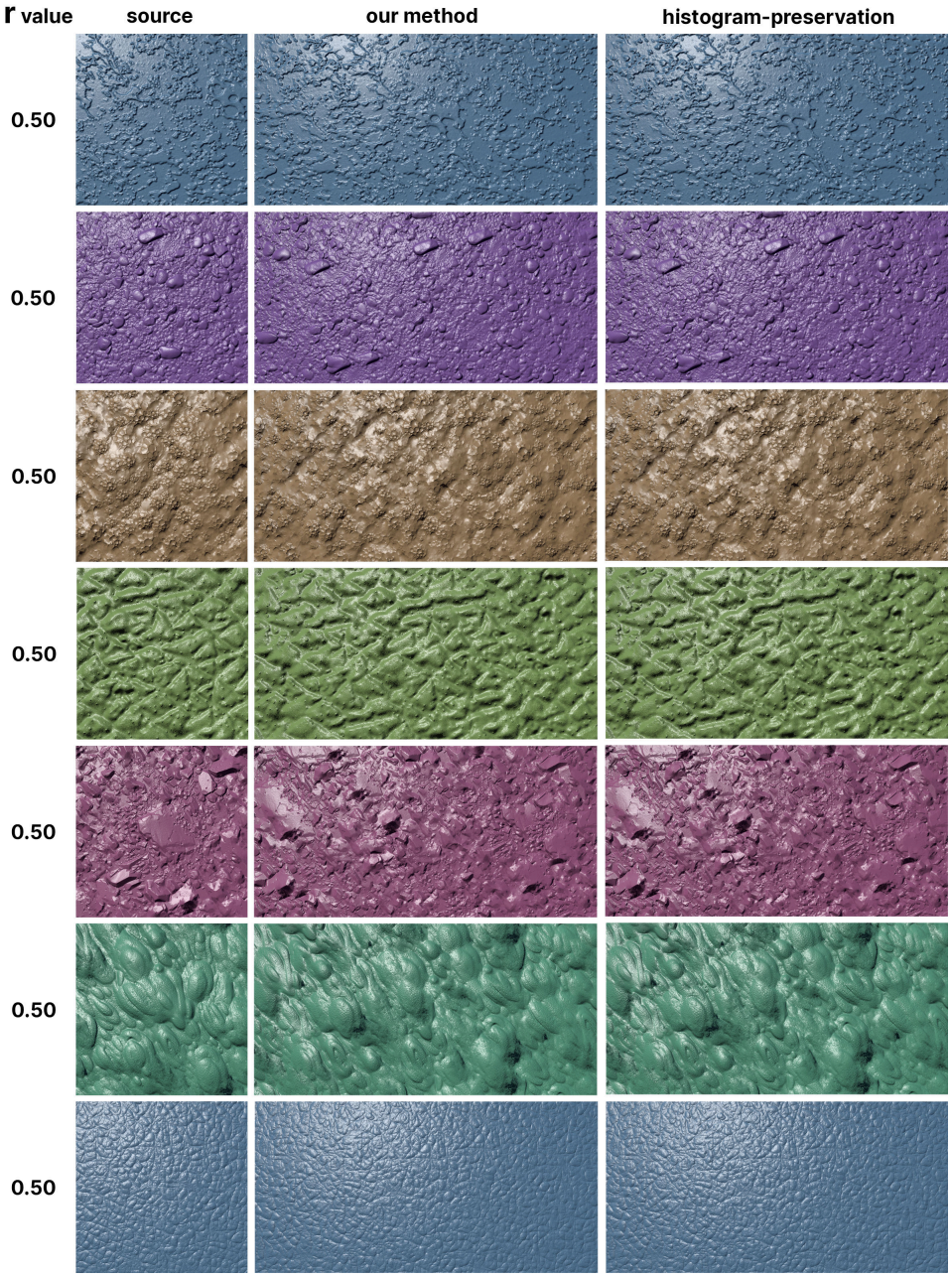


**Figure 4**. This figure presents different example patterns as normal maps. In each case hex–tiling is used to produce a similar image of twice the width of the original source image. We compare our proposed adaptation to the original histogram–preserving method.

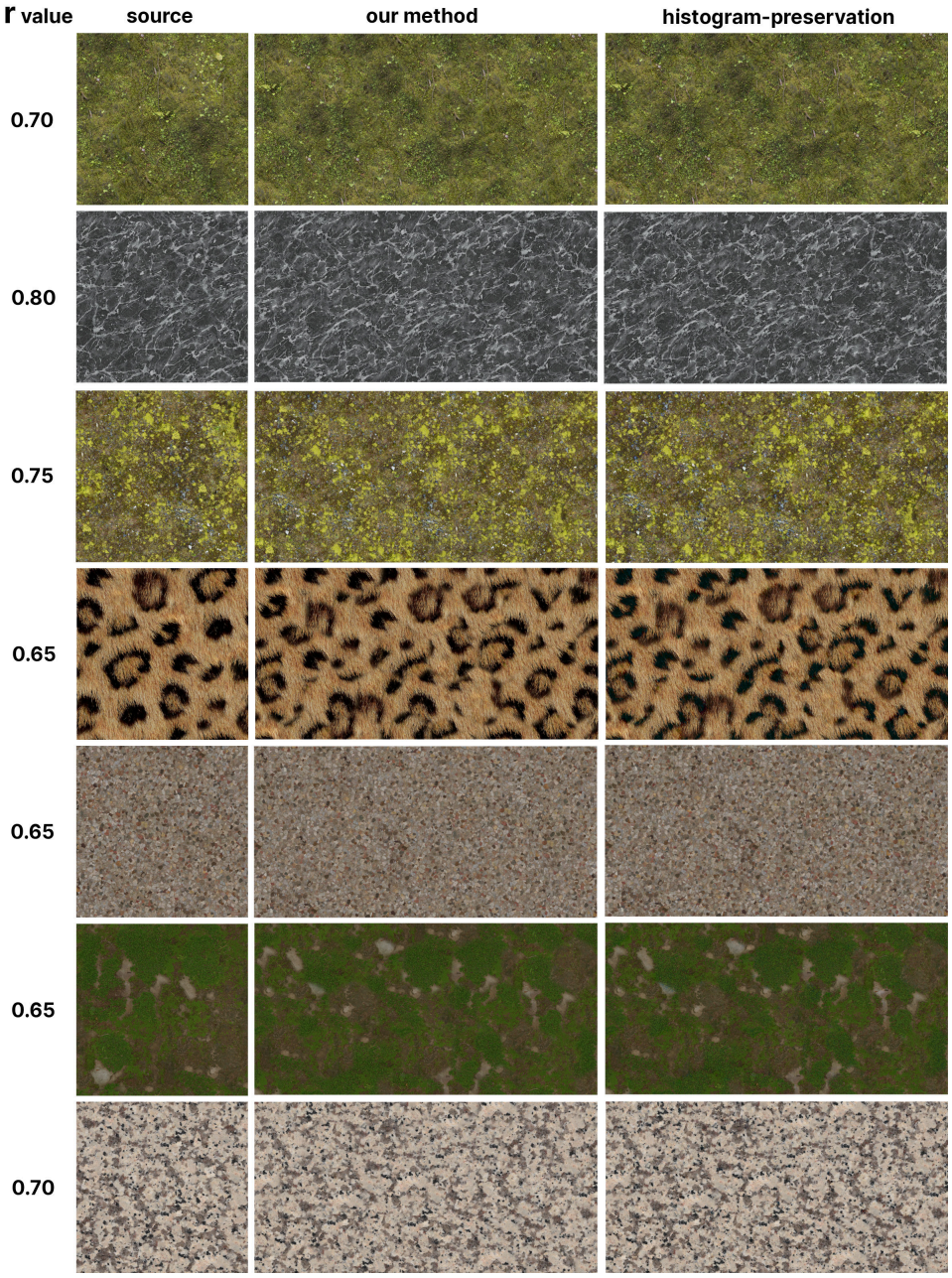


**Figure 5**. This figure presents different example patterns in color. In each case hex–tiling is used to produce a similar image of twice the width of the original source image. We compare our proposed adaptation to the original histogram–preserving method.

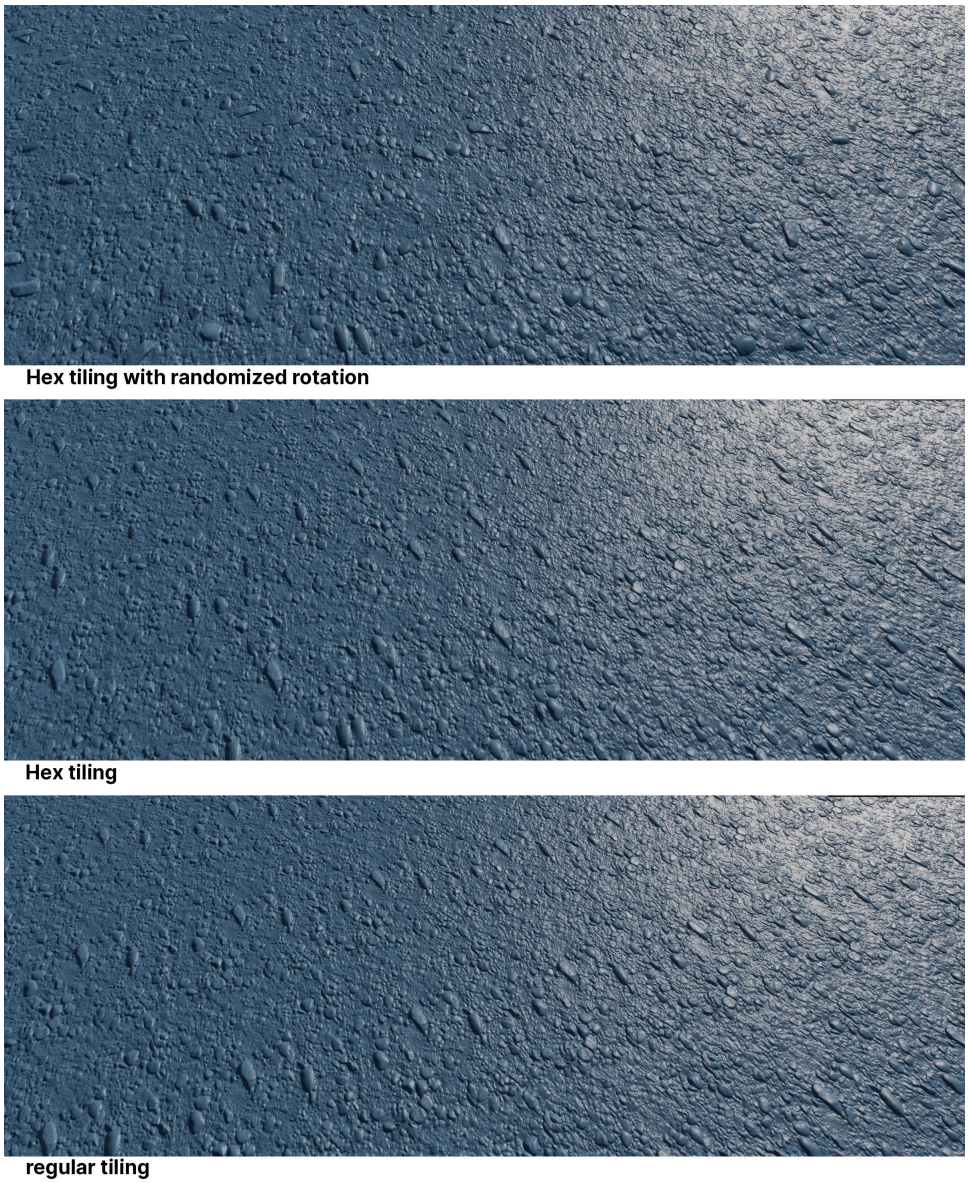


**Figure 6**. Conventional texture tiling is shown in the bottom image and the sense of repetition is evident. In the middle image hex–tiling is used which improves the result. However, some sense of repetition is still visible because a few distinct pebbles are consistently pointing in the same direction. We are able to solve this using randomized rotation as shown in the top image.

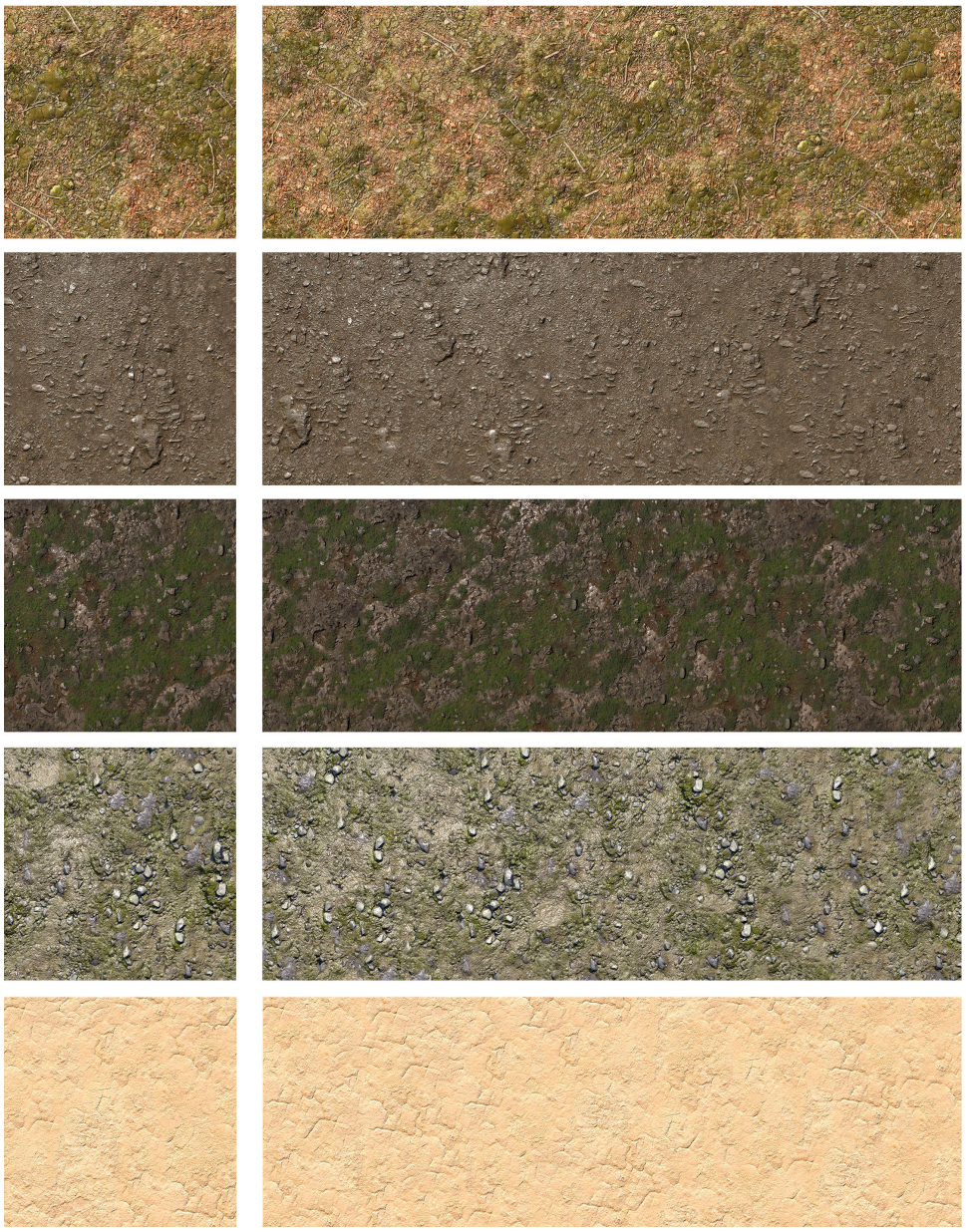

**Figure 7**. This figure presents different materials of terrain using texture maps for albedo, roughness and normal. In each case hex–tiling is used to produce a similar image which is three times the width of the original source image.

## 5. Conclusion

We have introduced an adaptation of the hex–tiling method, by Heitz and Neyret, which allows a shader artist or engineer to adopt the technique within minutes into any shader. The original method, although impressive, requires a substantial amount of engineering work to achieve an integration which is practical to users.

Furthermore, we have shown our approach achieves similar results to the histogram–preserving method though with subjectively minor trade–off in terms of quality when applied to color.

Given how useful hex–tiling is we hope our work will help pave the way for a wider adoption of this technique.

## References

BURLEY, B. 2019. On histogram-preserving blending for randomized texture tiling. *Journal of Computer Graphics Techniques (JCGT) 8*, 4 (November), 31–53. URL: http://jcgt.org/published/0008/04/02/. 3

DELIOT, T., AND HEITZ, E. 2019. *GPU Zen 2: Procedural Stochastic Textures by Tiling and Blending*. Black Cat Publishing Inc. 5

HEITZ, E., AND NEYRET, F. 2018. High-performance by-example noise using a histogram-preserving blending operator. *Proc. ACM Comput. Graph. Interact. Tech. 1*, 2 (aug). URL: https://doi.org/10.1145/3233304, doi:10.1145/3233304. 2

MIKKELSEN, M. S. 2020. Surface gradient–based bump mapping framework. *Journal of Computer Graphics Techniques (JCGT) 9*, 4 (October), 60–91. URL: http://jcgt.org/published/0009/03/04/. 3, 5

PERLIN, K., AND HOFFERT, E. M. 1989. Hypertexture. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1989, Boston, MA, USA, July 31 - August 4, 1989*, ACM, J. J. Thomas, Ed., 253–262. URL: https://doi.org/10.1145/74333.74359, doi:10.1145/74333.74359. 4

PERLIN, K. 1985. An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3, 287–296. doi:10.1145/325165.325247. 2

PERLIN, K. 2002. Noise hardware. in real-time shading. In *ACM SIGGRAPH 2002 Course Notes*, Association for Computing Machinery, New York, NY, USA, SIGGRAPH '02. URL: https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf. 2

SCHLICK, C. 1994. Fast alternatives to perlin's bias and gain functions. In *Graphics Gems IV*, P. S. Heckbert, Ed. Morgan Kaufmann, 401–403. 5

TRICARD, T., EFREMOV, S., ZANNI, C., NEYRET, F., MARTÍNEZ, J., AND LEFEBVRE, S. 2019. Procedural phasor noise. *ACM Trans. Graph. 38*, 4 (jul). URL: https://doi.org/10.1145/3306346.3322990, doi:10.1145/3306346.3322990. 2

WORLEY, S. 1996. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, Association for Computing

Machinery, New York, NY, USA, SIGGRAPH '96, 291–294. URL: https://doi.org/10.1145/237170.237267, doi:10.1145/237170.237267. 2

## Index of Supplemental Materials

A standalone demo is provided at https://github.com/mmikk/hextile-demo.

## Author Contact Information

Morten S. Mikkelsen
Unity Technologies SF
mikkelsen7@gmail.com
http://mmikkelsen3d.blogspot.com/p/3d-graphics-papers.html

---

Morten S. Mikkelsen, Practical Real-Time Hex-Tiling, *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 1, 1–1, 2022
????

| | | | |
|---|---|---|---|
| Received: | ????-??-?? | | |
| Recommended: | ????-??-??7 | Corresponding Editor: | Angelo Pesce |
| Published: | ????-??-?? | Editor-in-Chief: | Marc Olano |



## Appendix: source code

```
// Output: weights associated with each hex tile and integer centers
void TriangleGrid(out float w1, out float w2, out float w3,
          out int2 vertex1, out int2 vertex2, out int2 vertex3,
          float2 st)
{
  // Scaling of the input
  st *= 2 * sqrt(3);

  // Skew input space into simplex triangle grid
  const float2x2 gridToSkewedGrid =
    float2x2(1.0, -0.57735027, 0.0, 1.15470054);
  float2 skewedCoord = mul(gridToSkewedGrid, st);

  int2 baseId = int2( floor ( skewedCoord ));
  float3 temp = float3( frac( skewedCoord ), 0);
  temp.z = 1.0 - temp.x - temp.y;

  float s = step(0.0, -temp.z);
  float s2 = 2*s-1;

  w1 = -temp.z*s2;
  w2 = s - temp.y*s2;
  w3 = s - temp.x*s2;

  vertex1 = baseId + int2(s,s);
  vertex2 = baseId + int2(s,1-s);
  vertex3 = baseId + int2(1-s,s);
}
```

**Listing 1**. Snaps the sampling coordinate st to a triangle grid.

```
float2 hash( float2 p)
{
  float2 r = mul(float2x2(127.1, 311.7, 269.5, 183.3), p);

  return frac( sin( r )*43758.5453 );
}
```

**Listing 2**. Produces an arbitrary offset from a hex tile center.

```
// Input: nmap is a normal map
// Input: r increase contrast when r>0.5
// Output: deriv is a derivative dHduv wrt units in pixels
// Output: weights shows the weight of each hex tile
void bumphex2derivNMap(out float2 deriv, out float3 weights,
             Texture2D nmap, SamplerState samp, float2 st,
             float rotStrength, float r=0.5)
{
  float2 dSTdx = ddx(st), dSTdy = ddy(st);

  // Get triangle info
  float w1, w2, w3;
  int2 vertex1, vertex2, vertex3;
  TriangleGrid(w1, w2, w3, vertex1, vertex2, vertex3, st);

  float2x2 rot1 = LoadRot2x2(vertex1, rotStrength);
  float2x2 rot2 = LoadRot2x2(vertex2, rotStrength);
  float2x2 rot3 = LoadRot2x2(vertex3, rotStrength);

  float2 cen1 = MakeCenST(vertex1);
  float2 cen2 = MakeCenST(vertex2);
  float2 cen3 = MakeCenST(vertex3);

  float2 st1 = mul(st - cen1, rot1) + cen1 + hash(vertex1);
  float2 st2 = mul(st - cen2, rot2) + cen2 + hash(vertex2);
  float2 st3 = mul(st - cen3, rot3) + cen3 + hash(vertex3);

  // Fetch input
  float2 d1 = sampleDeriv(nmap, samp, st1,
              mul(dSTdx, rot1), mul(dSTdy, rot1));
  float2 d2 = sampleDeriv(nmap, samp, st2,
              mul(dSTdx, rot2), mul(dSTdy, rot2));
  float2 d3 = sampleDeriv(nmap, samp, st3,
              mul(dSTdx, rot3), mul(dSTdy, rot3));

  d1 = mul(rot1, d1); d2 = mul(rot2, d2); d3 = mul(rot3, d3);

  // produce sine to the angle between the conceptual normal
  // in tangent space and the Z-axis
  float3 D = float3( dot(d1,d1), dot(d2,d2), dot(d3,d3));
  float3 Dw = sqrt(D/(1.0+D));

  Dw = lerp(1.0, Dw, g_fallOffContrast);  // 0.6
  float3 W = Dw*pow(float3(w1, w2, w3), g_exp); // 7
  W /= (W.x+W.y+W.z);
  if(r!=0.5) W = Gain3(W, r);

  deriv = W.x * d1 + W.y * d2 + W.z * d3;
  weights = ProduceHexWeights(W.xyz, vertex1, vertex2, vertex3);
}
```

**Listing 3**. Hex tile sampling a normal map with hex rotation.

```
// Input: tex is a texture with color
// Input: r increase contrast when r>0.5
// Output: color is the blended result
// Output: weights shows the weight of each hex tile
void hex2colTex(out float4 color, out float3 weights,
        Texture2D tex, SamplerState samp, float2 st,
        float rotStrength, float r=0.5)
{
  float2 dSTdx = ddx(st), dSTdy = ddy(st);

  // Get triangle info
  float w1, w2, w3;
  int2 vertex1, vertex2, vertex3;
  TriangleGrid(w1, w2, w3, vertex1, vertex2, vertex3, st);

  float2x2 rot1 = LoadRot2x2(vertex1, rotStrength);
  float2x2 rot2 = LoadRot2x2(vertex2, rotStrength);
  float2x2 rot3 = LoadRot2x2(vertex3, rotStrength);

  float2 cen1 = MakeCenST(vertex1);
  float2 cen2 = MakeCenST(vertex2);
  float2 cen3 = MakeCenST(vertex3);

  float2 st1 = mul(st - cen1, rot1) + cen1 + hash(vertex1);
  float2 st2 = mul(st - cen2, rot2) + cen2 + hash(vertex2);
  float2 st3 = mul(st - cen3, rot3) + cen3 + hash(vertex3);

  // Fetch input
  float4 c1 = tex.SampleGrad(samp, st1,
                mul(dSTdx, rot1), mul(dSTdy, rot1));
  float4 c2 = tex.SampleGrad(samp, st2,
                mul(dSTdx, rot2), mul(dSTdy, rot2));
  float4 c3 = tex.SampleGrad(samp, st3,
                mul(dSTdx, rot3), mul(dSTdy, rot3));

  // use luminance as weight
  float3 Lw = float3(0.299, 0.587, 0.114);
  float3 Dw = float3(dot(c1.xyz,Lw),dot(c2.xyz,Lw),dot(c3.xyz,Lw));

  Dw = lerp(1.0, Dw, g_fallOffContrast);  // 0.6
  float3 W = Dw*pow(float3(w1, w2, w3), g_exp); // 7
  W /= (W.x+W.y+W.z);
  if(r!=0.5) W = Gain3(W, r);

  color = W.x * c1 + W.y * c2 + W.z * c3;
  weights = ProduceHexWeights(W.xyz, vertex1, vertex2, vertex3);
}
```

**Listing 4**. Hex tile sampling a texture with hex rotation.

```
float2 MakeCenST(int2 Vertex)
{
  float2x2 invSkewMat = float2x2(1.0, 0.5, 0.0, 1.0/1.15470054);

  return mul(invSkewMat, Vertex) / (2 * sqrt(3));
}
```

**Listing 5**. Remaps an integer hex tile center to st space.

```
float2x2 LoadRot2x2(int2 idx, float rotStrength)
{
  float angle = abs(idx.x*idx.y) + abs(idx.x+idx.y) + M_PI;

  // remap to +/-pi
  angle = fmod(angle, 2*M_PI);
  if(angle<0) angle += 2*M_PI;
  if(angle>M_PI) angle -= 2*M_PI;

  angle *= rotStrength;

  float cs = cos(angle), si = sin(angle);

  return float2x2(cs, -si, si, cs);
}
```

**Listing 6**. Produce a 2x2 rotation matrix from integer hex tile center.

```
float3 ProduceHexWeights(float3 W,
             int2 vertex1, int2 vertex2, int2 vertex3)
{
  float3 res = 0.0;

  int v1 = (vertex1.x-vertex1.y)%3;
  if(v1<0) v1+=3;

  int vh = v1<2 ? (v1+1) : 0;
  int vl = v1>0 ? (v1-1) : 2;
  int v2 = vertex1.x<vertex3.x ? vl : vh;
  int v3 = vertex1.x<vertex3.x ? vh : vl;

  res.x = v3==0 ? W.z : (v2==0 ? W.y : W.x);
  res.y = v3==1 ? W.z : (v2==1 ? W.y : W.x);
  res.z = v3==2 ? W.z : (v2==2 ? W.y : W.x);

  return res;
}
```

**Listing 7**. swizzle the weights to produce a consistent hex color for visualization purposes.

```
float3 Gain3(float3 x, float r)
{
  // increase contrast when r>0.5 and
  // reduce contrast if less
  float k = log(1-r) / log(0.5);

  float3 s = 2*step(0.5, x);
  float3 m = 2*(1 - s);

  float3 res = 0.5*s + 0.25*m * pow(max(0.0, s + x*m), k);

  return res.xyz / (res.x+res.y+res.z);
}
```

**Listing 8**. Apply an S-curve shaped ramp to the signal and normalizes the weights.

```
// Input: vM is tangent space normal in [-1;1].
// Output: convert vM to a derivative.
float2 TspaceNormalToDerivative(float3 vM)
{
    const float scale = 1.0/128.0;

    // Ensure vM delivers a positive third component using abs() and
    // constrain vM.z so the range of the derivative is [-128; 128].
    const float3 vMa = abs(vM);
    const float z_ma = max(vMa.z, scale*max(vMa.x, vMa.y));

    // Set to match positive vertical texture coordinate axis.
    const bool gFlipVertDeriv = false;
    const float s = gFlipVertDeriv ? -1.0 : 1.0;
    return -float2(vM.x, s*vM.y)/z_ma;
}
```

**Listing 9**. Conversion of tangent-space normal $\vec{m}$ to a derivative $\vec{d}$.

```
float2 sampleDeriv(Texture2D nmap, SamplerState samp, float2 st,
           float2 dSTdx, float2 dSTdy)
{
  // sample
  float3 vM = 2.0*nmap.SampleGrad(samp, st, dSTdx, dSTdy)-1.0;
  return TspaceNormalToDerivative(vM);
}
```

**Listing 10**. Samples a normal map with gradients and returns a derivative dHduv.