Vol. [VOL], No. [ISS]: 1–5

Cylindrical and toroidal parameterizations without vertex seams

Marco Tarini
VCG - ISTI - CNR (Pisa, Italy) *and* Università dell'Insubria (Varese, Italy)

Abstract.
A simple rendering method is presented to avoid vertex seams in cylindrical and toroidal uv-mappings used for texture mapping (a vertex seam is a vertex duplication of a polygonal mesh with different texture coordinates assigned to the two geometrically coinciding copies). As a result, the method leads for simpler, leaner, replication-free data structures. Is also allows for an higher degree of proceduralism in generation of texture coordinates.

The method is general, trivial to implement (exhaustive pseudocode is provided), very cheap on resources (with a virtually null impact on performance) and leverages only basic mechanisms widely available in most GPU implementations. An open-source implementation is made available.

1. Motivations

Recall that a *"vertex seam"*, in a polygonal mesh data structure, is a duplicated vertex with different attribute values assigned to the two (geometrically coinciding) copies of the vertex: it is used to implement a discontinuity in the values encoded as vertex attributes occurring in an geometrically C0 continuous surface. Vertex seams recur in *UV-mappings* used for standard texture mapping. An *UV-mapping* not requiring any vertex seam is sometimes termed "seamless" (not to be confused with seamless *textures*, a term often used to indicate that the signal encoded in the texture is made continuous across seams,

1086-7651/06 $0.50 per page

e.g. by replicating a few texel values).

Seamless UV-mappings offer crucial advantages. Lack of vertex duplication means simpler data structures, with no xyz-data replicas[1], and avoiding the need to keep the two copies consistent (e.g. in an animation, a shape smoothing operation, etc). Lack of vertex duplication also allows for unconstrained triangle stripping, re-meshings, mesh simplification, LOD-ding, GPU-based vertex arrays, etc. Importantly, this kind of mapping is also needed in all procedural approaches where texture coordinates are produced on the fly (for example environment map approaches).

The vast majority of the UV-mapping schemas used in practice need vertex seams. For example, all atlas based UV-mappings do, with the only exception of [Tarini et al. 04]. The few other examples of vertex-seam free UV-mappings are: trivial single-chart UV-mappings (usable to texture objects with disk-topology, e.g. a flag waving in the wind); and, UV-mapping defined on a cube-map, thanks to hardwired support for per-fragment gnomic projection and face selection (usable for example to texture a round object, but designed and more commonly employed for environment map techniques). The presented method adds the case of Cylindrical and Toroidal UV-mappings to the list.

1.1. Cylindrical and Toroidal UV-mappings

In a cylindrical UV-mapping, a surface S, topologically equivalent to a cylindrical side area, is unwrapped into a 2D rectangular texture T, with the right-end side of T logically connected with its left-end side. This kind of UV-mappings recur for example in garments, solids of revolution, environment maps.

All graphic cards offer a basic hardwired mechanism, embedded in texture fetch operations, which consists in using only the fractional part of the texture coordinate to index a texel, ignoring the integer part. This implements an equivalence relationship $\simeq$ between texture coordinate values:

$$(k_0 + u) \simeq (k_1 + u) \qquad \forall k_0, k_1 \in \mathbb{N}, u \in [0..1) \tag{1}$$

This mechanism allows to define polygons spanning across the implicit connection between two opposite sides of the texture. However, it is common knowledge that this does not avoid the need of duplicated vertices (see Fig. 3 for examples of the artifacts arising unless vertex seams are introduced).

The reason is that the attribute interpolation of texture coordinates in the rasterizer is unaware of equivalence (1): for example, interpolation between

[1]Data structures for meshes can be designed to deal with jumps in texture coordinates in ways other than vertex duplication: texture-coordinates can be indexed per face separately from vertex-position data, or, they can be embedded as per-wedge attribute (thus replicating UV positions). Non of these alternatives bypasses the problems discussed here.

$u_a = 0.9$ and $u_b = 0.1$ produces values $0.8, 0.7 \cdots 0.2$, thus taking the "long route" which span most of the texture, instead of the "shorter route" across the cut (Fig. 1). To get the expected result, $u'_b = 1.1$ must be used in place of u_b for that primitive: the vertex must be duplicated and u'_b assigned to the second copy. This cannot be avoided by any assignment of texture coordinates to vertices, not even disallowing triangles spanning across the cut.

When both pairs of opposite texture sides are logically connected to one another, we have a toroidal map, which presents the same problem in both U and V texture directions.

2. Method description

The presented method can address one or both texture coordinates, resulting in cylindrical or toroidal maps respectively, or actually any other set of per-vertex attributes over which equivalence (1) must be enforced.

In the following example, the method is applied on the horizontal coordinate u_A of a cylindrical map. In a toroidal map, all steps (except the final texture access) are performed separately on both texture coordinates u_A and v_A.

The method takes place in the vertex and fragment processor:

```
In the vertex processor:
Input attribute: $u_A$
Output varyings: $u_1$ and $u_2$
1. $u_1 \leftarrow frac(u_A)$
2. $u_2 \leftarrow frac(u_A + 0.5) - 0.5$
```

```
In the fragment processor:
Input varyings: $u_1^i$ and $u_2^i$
Temp values: $u^T$
1. if $d(u_1^i) \leqslant d(u_2^i)$ then $u^T \leftarrow u_1^i$
                          else $u^T \leftarrow u_2^i$
2. use $u^T$ as coordinate
   to perform texture access
```

where $frac(u) = u - \lfloor u \rfloor$, $d(u) = |d_x(u)| + |d_y(u)|$, and $d_x(u)$,$d_y(u)$ are the horizontal and vertical screen-space derivatives of u. Recall that fragment shader languages (e.g. OpenGL GLSL and DirectX HLSL) give access to these derivatives for any given quantity available per fragment[2].

In case that k repetitions of the the texture around the cylinder are needed, it suffices to use $k \cdot u^T$ in place of u^T in the final texture access.

Bilinear texel interpolation is applied correctly even on the seam line by taking advantage of built-in support for repeating textures on texture fetch.

MIP-map level is also determined correctly, as long as the inequality in line 1 of fragment program is resolved consistently for equalities. To ensure this robustly despite numerical errors, in the implementation one inequality side is safely biased by a small positive value.

[2]In GLSL function d is directly available as `fwidth`

2.1. Why it works (sketch)

Refer to Fig. 2. At vertices (before interpolation), $u_1 \in [0, 1)$ and $u_2 \in [-0.5, 0.5)$, but $u_1 \simeq u_2$. For most primitives, this also holds for interpolated values: $u_1^i \simeq u_2^i$. In these cases (A and C in Fig. 2) either value can be safely used as u^T (and their derivatives are virtually equal). However, if hypothetically u_1^i was always used, the problem would arise for primitives interpolating between values ε ($\simeq 1+\varepsilon$) and $1-\varepsilon$ ($\simeq -\varepsilon$) (case D); conversely, if u_2^i was always used, the problem would arise when interpolating between values $-0.5+\varepsilon$ ($\simeq 0.5+\varepsilon$) and $0.5-\varepsilon$ ($\simeq -0.5-\varepsilon$) (case B). By picking either u_1^i or u_2^i , the problem can be avoided for all primitives (as long as they are smaller than half the texture size). The one to pick is always the one presenting the smaller screen-space derivative. This corresponds to favoring the alternative resulting in the span of the smaller distance in U direction.

2.2. Application side

During rendering, an application simply needs to send (or to procedurally create) univoche texture coordinates as vertex attributes. The intended semantic is that, inside each triangle, the interpolated coordinate will span the texture space either forward or backward, depending of which direction is shorter in parametric space, keeping equivalence (1) in account. This assumes that no triangle spans more than half of the cylindrical/toroidal texture space (this is reasonable in most applicative contexts).

The method directly applies to coordinates for textures of any dimensionality and to rendering of any primitive (n-gons, triangles, lines, strips, etc).

From a mesh data structure point of view, this allows storage of texture coordinates as vertex attributes and still avoid vertex duplications (a caveat here is that the pre-computation of tangent vectors, if needed, must take equivalence (1) in account, selecting the shortest route in each face).

3. Discussion

Thanks to the simple method presented, cylindrical and toroidal UV-mappings become schemas which do not require vertex seams, thus unlocking all practical advantages discussed in Sec. 1. A most important advantage is the higher degree of proceduralism in generation of texture coordinates. The method requires to customize solely the behavior vertex and the fragment shaders, and its implementation is straightforward using any common shading language (e.g. OpenGL, OpenGL ES or DirectX). It has a negligible impact on rendering performances, respect to using plain texture mapping; the main cost is that an extra interpolant is used in the rasterizer.

A few examples of the application of this technique are shown in Fig. 3, and a full working implementation is available at the linked web site.

3.1. Available alternatives

In principle, the problem solved here could be dealt on a per-primitive basis, rather than with per-fragment tests: just before rasterization, the "shorter route" would have to be selected for the purpose of computing interpolated texture coordinates for that primitive only. This could be implemented for example in the *geometry shader*, which has usually an impact on performance if activated. Else, the operation could be performed directly by the rasterizer: Direct3D offers this as one ad-hoc mechanism, termed "Texture Wrapping" [MSDN 11] (with limitations: e.g. results are undefined for coordinates outside [0::1) and it cannot be employed with the tessellation engine). However, OpenGL has no equivalent. In general, it can be argued that the opportunity to use cylindrical/toroidal mappings is not common enough to justify a hard-wired adaptation of the rasterizer (which is the core part of the graphic engine and should be kept as simple and generic as possible), given that the same functionality can be fully reproduced, as shown here, by relying solely on the present customizability of fragment and vertex shaders.

References

[MSDN 11] MSDN. "Texture Wrapping (Direct3D 9).", 2011. Available online (http://msdn.microsoft.com/en-us/library/bb206256).

[Tarini et al. 04] Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. "PolyCube-Maps." *ACM Trans. Graph.* 23:3 (2004), 853–860.

Web Information:

http://vcg.isti.cnr.it/~tarini/no-seams

Marco Tarini, Isti CNR (Pisa, Italy), (marco.tarini@isti.cnr.it)

Received [DATE]; accepted [DATE].

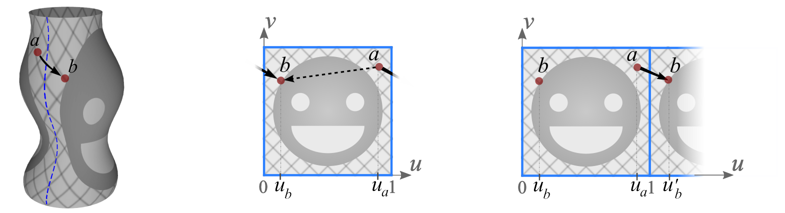

Figure 1. In a cylindircal map, coordinates are interpolated differently (dotted arrow, middle) than it is intended (bold arrows). Texture repetitions can be employed to fix this (right), but only if the two equivalent but different coordinates u_b and u'_b are assigned to the same vertex b, causing the seam (refer to Sec. 1.1).

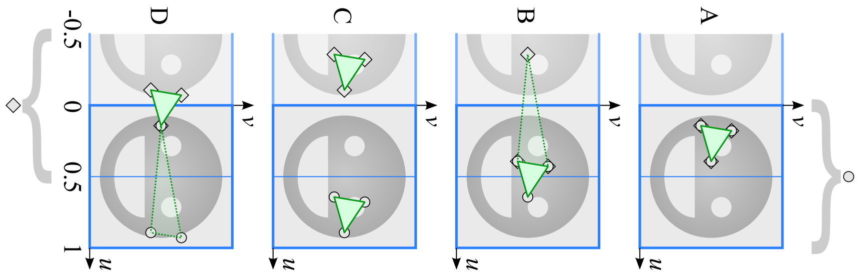

Figure 2. Visual explanation of the method in a cylindrical UV-mapping (refer to Sec. 2.1). Four examples of a triangle primitive are labeled A to D. Circles: u_1 positions; diamonds: u_2 positions (sometimes they coincide); green triangles: area covered by interpolated coordinates–the dotted ones are discarded.

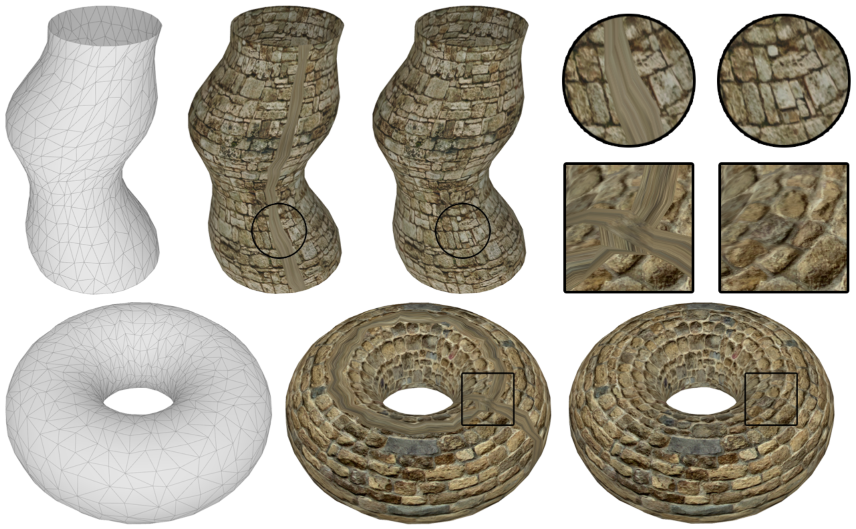

Figure 3. Examples. Top a cylindrical UV-mapping; bottom, a toroidal one. If no vertex seams are introduced, artifacts are bound to arise (middle), but the presented method fixes the problem (right). Round and squared insets: close-ups.