Rethinking Texture Mapping

Marco Tarini  
Università dell’Insubria, Varese  
ISTI-CNR, Pisa  
marco.tarini@isti.cnr.it

Cem Yuksel  
University of Utah  
cem@cemyuksel.com

Sylvain Lefebvre  
INRIA  
Sylvain.Lefebvre@inria.fr

Figure 1: Examples alternatives to texture mapping: (a) volume-encoded uv-maps [Tarini 2016], (b) octree textures [Lefebvre et al. 2005], (c) Ptex [Burley and Lacewell 2008] (© Walt Disney Animation Studios), (d) brickmaps [Christensen and Batali 2004], (e) polycube-maps [Tarini et al. 2004], (f) giga voxels [Crassin et al. 2009], (g) invisible seams [Ray et al. 2010], (h) perfect spatial hashing [Lefebvre and Hoppe 2006], (i) mesh colors [Yuksel et al. 2010], and (j) tiletrees [Lefebvre and Dachsbacher 2007].

ABSTRACT

The intrinsic problems of standard Texture Mapping, regarding UV-maps and seams, are well-known, but often considered unavoidable. In this course we will discuss various radically different ways to rethink texture mapping that have been proposed over decades, each offering different advantages and trade-offs.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGGRAPH '17 Courses, July 30 - August 03, 2017, Los Angeles, CA, USA  
© 2017 Copyright held by the owner/author(s).  
ACM ISBN 978-1-4503-5014-3/17/07... $15.00  
https://doi.org/http://dx.doi.org/10.1145/3084873.3084911
1 INTRODUCTION

In computer graphics, texture mapping is the fundamental means by which high-frequency signals such as diffuse colors, normals, and other shading parameters are defined over 3D surfaces. The principle is to store surface details in 2D high-resolution texture images, and then define a mapping from the 3D surface to the 2D image, assigning a uv coordinate to each mesh vertex. This approach is ubiquitously adopted by virtually all computer graphics applications and implemented on all available graphics hardware, from high-end to smartphone GPUs.

However, texture mapping has a number of fundamental issues. Creating uv-maps is time consuming and involves extensive manual effort. Distortions and seams introduced by mapping complicate texture authoring, filtering, and procedural synthesis. The final result is optimized for a specific mesh and it does not necessarily work through LoDs. Any change to the geometry or the connectivity implies updating the uv-mapping and textures. As a consequence, texture mapping continues to occupy a substantial portion of artist time, which dominates the cost of AAA video game production.

Since the early days of texturing, there has been a constant research effort to alleviate or even bypass traditional texture mapping limitations (Figure 1). Unfortunately, the ubiquitous adoption of texture mapping implies that it is seldom questioned as the method of choice, and both authoring pipelines and rendering engines have been shaped around its intrinsic limitations, thereby making it harder for alternatives to be adopted. Yet, the industry recently started to recognize the advantages of alternative approaches to texture mapping, in particular the Ptex method [Burley and Lacewell 2008] is becoming increasingly popular.

The objective of this course is to make the audience more familiar with such alternative approaches and their advantages and trade-offs regarding versatility, ease of authoring, storage cost, rendering quality and performance, and implementation difficulty. We believe the course to be both timely and necessary: several advances in GPU technologies has made alternative texturing approaches computationally very efficient and the industry has shown a renewed interest in moving beyond texture mapping. Furthermore, the algorithms and data-structure used by some alternative approaches extend beyond texturing, towards solid modeling, volume rendering, and simulations on surfaces (e.g. dynamic texturing).

2 OVERVIEW

We begin with discussing the limitations and strengths of standard 2D texture mapping, which is ubiquitously used for virtually all computer graphics applications. This approach stores UV coordinates as attributes on the vertices of a mesh and they are interpolated inside faces, mapping the surface over one or multiple rasterized texture image(s). Then we explain the criteria we used for evaluating alternative methods to texture mapping.

Perfecting Traditional UV-maps. We present methods that address specific shortcomings of the standard 2D texture mapping approach by carefully using it in specific ways. This Section covers the following techniques:
**Invisible seams** [Ray et al. 2010] provide a method that aligns the seams on a texel grid in texture space, making them consistent with bilinear filtering, thereby hiding the filtering artifacts near seams that appear with traditional texture mapping. **Seamless toroidal/cylindrical textures** [Tarini 2012] avoid vertex duplication at seams and eliminate filtering artifacts near seams for some specific classes of maps. **Seamless texture atlases** [Purnomo et al. 2004] produce texture atlases that prevent filtering artifacts near seams. They also support down-sampling for mip-mapping and mesh simplification.

**Connectivity-based Representations.** These methods use the inherent parameterization of the model, instead of defining a separate parameterization for mapping. The topology of the mesh model is used directly for defining the texture data on each primitive. These approaches substantially improve the texture authoring process, but they require 3D painting tools. This Section covers the following techniques:

- **Ptex** [Burley and Lacewell 2008] is used and promoted by Disney Animation and Pixar studios. It effectively assigns a separate texture map to each quad-shaped faces of a mesh. This structure eliminates the need for defining uv-mapping, and it is primarily designed for quad meshes. Texture filtering across faces is handled by accessing the mesh topology.

- **Mesh colors** [Yuksel et al. 2010] are closely related to p-tex and similarly used in production [Lambert 2015]. Extending the concept of vertex colors with additional samples inside mesh faces and on edges, mesh colors provide a topological dual of p-tex in terms of color sample placement. This provides better support for triangular meshes and correct handling of extraordinary vertices. It also eliminates the need for accessing the topology information during texture filtering.

- **Mesh color textures** [Yuksel 2016] aim to utilize the existing texture filtering hardware on current GPU for sampling/filtering mesh colors. It achieves this by effectively converting mesh colors to a representation similar to standard 2D textures. As a result, the authoring benefits of mesh colors can be used without any visible overhead at render time (as compared to standard 2D textures).

**Sparse Volumetric Textures.** These techniques associate the texture data directly using the volume embedding the 3D model, bypassing the need of constructing or storing any mapping. A naive implementation of this approach would require a large space, which would be cubic with the resolution of the sampling. Several countermeasures are adopted to avoid this problem using sparse volumetric data structures. This Section covers the following techniques:

- **Adaptive texture maps** [Kraus and Ertl 2002] provide a GPU-based method for locally adjusting the texture resolution depending on the texture content and adaptive texture boundaries.

- **Octree textures** [Benson and Davis 2002; Lefebvre et al. 2005] encode a volumetric texture using an efficient octree hierarchy that is used for texturing surfaces without the need for a uv-map.

- **Brick maps** [Christensen and Batali 2004] were developed for storing precomputed global illumination in arbitrary scenes. They extend the concept of octree textures by introducing voxel blocks with efficient caching that make them suitable for handling extremely large scenes. An implementation of brick maps is included in the RenderMan software.

- **Perfect spatial hashing** [Garcia et al. 2011; Lefebvre and Hoppe 2006] also provides a volumetric encoding of texture values to efficiently store the color values in a hash table that is accessed using 3D positions of the surface. Construction of the hash functions is time consuming but automatic. Cache coherency is an issue.

- **Gigavoxels** [Crassin et al. 2009] are designed for efficiently rendering large volumetric data sets using a sparse 3D structure for texture storage.
Volume-based Parameterizations. These techniques construct a mapping from the 3D model space to a 2D texture space. This Section covers the following techniques:

**TileTrees** [Lefebvre and Dachsbacher 2007] also extend octree textures by storing 2D texture tiles on the surfaces of octree nodes. They can efficiently adapt to the given object shape and require much fewer octree levels for representing high-resolution textures.

**PolyCube-maps** [Tarini et al. 2004] generalize the concept of cube-maps that provides a tighter enclosure for the target surface. They are GPU-friendly and produce a cut-free parameterization over a polycube surface that can be used for texturing.

**Volume-encoded uv-maps** [Tarini 2016] define the uv-mapping as a trilinearly interpolated low-resolution 3D lattice, instead of a per-vertex assignment of uv coordinates. This requires only basic HW support and is almost as efficient as plain texture mapping, and supports different tessellations, including LoDs, but does not bypass the need for uv-map creation.

The course will provide the details of the alternative methods to texture mapping listed above, discuss their similarities and differences, and present their advantages and limitations. Our aim is to provide the knowledge needed for determining the best candidate for replacing texture mapping for any application, which we expect to be different based on the constraints of the application.
Texture Mapping

- Texture mapping is the fundamental means by which high-frequency details (such as color) are defined on surfaces.
Texture Mapping

• Requires defining a mapping from the model space to the texture space.

Mapping introduces seams and it is labor intensive.

Separate the model into parts
Map each part to the texture space

and it is labor intensive.
Texture Mapping

- Model editing after texture painting is problematic.

Texture Mapping

- Seams introduce filtering artifacts.
Texture Mapping

- Artifacts are more pronounced at higher mip-map levels.

Low-resolution mip-map level

Texture Mapping

- Carefully painting around seams can hide artifacts, but not completely eliminate them.

High-resolution texture
Texture Mapping

• Seam artifacts still appear in mip-map levels.

Low-resolution mip-map level

Texture Mapping

• Without mip-mapping, seam artifacts can be mostly hidden.
Texture Mapping

Displacement maps

- Seams cause cracks! Eliminating these cracks requires carefully adjusting the mapping, so that the texture filtering results on either side of the seam are identical, which is often impossible with standard 2D textures.

Texture Mapping

- Vertex attributes along seams must be duplicated
Texture Mapping

- Mapping is mesh dependent

Attempts to automatize mapping

- While there are methods for automated mapping, in practice mapping requires substantial manual effort.
Problems of Texture Mapping

- Defining mapping is labor-intensive
  - Cannot be fully automated
  - Time consuming even for experts
  - Beyond the ability of most non-experts
- Local resolution adjustment is problematic
  - Locally changing the resolution requires (partial) remapping
- Model editing after texture painting is problematic
  - Changes to the model may require (partial) remapping
- Seams introduce artifacts
  - Inconsistencies on either side of a seam reveal the seam and cause cracks in displacement mapping

Problems of Texture Mapping (cont.)

- Additional storage cost
  - UV mapping data per vertex
  - Duplicated vertex data along seams
  - Wasted space due to imperfect packing and borders around seams
- Mesh dependency
  - It is not easy to use the same texture on a different tessellation of the same model, which would be particularly useful for LoD.
Course Outline

<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00</td>
<td>Introduction: Limitations of Traditional Texture Mapping</td>
</tr>
<tr>
<td>9:15</td>
<td>Perfecting Traditional UV-maps</td>
</tr>
<tr>
<td></td>
<td><em>Invisible Seams, Seamless Toroidal/Cylindrical Textures, Seamless Texture Atlases</em></td>
</tr>
<tr>
<td>9:30</td>
<td>Connectivity-based Representations</td>
</tr>
<tr>
<td></td>
<td><em>Ptex, mesh colors, mesh color textures</em></td>
</tr>
<tr>
<td>9:45</td>
<td>Sparse Volumetric Textures</td>
</tr>
<tr>
<td></td>
<td><em>Adaptive Texture Maps, Octree/N³ Textures, Brick Maps, Perfect Spatial Hashing, Gigavoxels</em></td>
</tr>
<tr>
<td>10:05</td>
<td>Volume-based Parameterizations</td>
</tr>
<tr>
<td></td>
<td><em>Tiletrees, PolyCube Maps, Volume-encoded UV-Maps</em></td>
</tr>
<tr>
<td>10:20</td>
<td>Conclusion and Questions</td>
</tr>
</tbody>
</table>
Evaluation

- Applicability
  Supported surface representations or model types
- Usability
  Permitted mapping/painting operations
- Quality
  Texture filtering quality
- Performance
  Storage, access, and computation overhead
- Implementation
  Development effort needed

Evaluation: Applicability

- Meshes
- Point Clouds
- Implicit Surfaces
- Shape/Topology Limits
  Any restrictions on the surface shape or mesh topology
- Subdivisions
  Higher resolution tessellations of a mesh
- Tessellation Independence
  Lower resolution tessellations and/or remeshing support
Evaluation: Usability

- Automated Mapping
  *Can a “good” mapping to the texture space be automatically generated?*
- Manual Mapping
  *Manually generating/editing the mapping to the texture space*
- Model Editing after Painting
  *Changing the model topology after mapping/painting*
- Resolution Readjustment
  *Changing the local texture resolution after the texture is (partially) painted*
- Texture Repetition
  *The ability to use the same texture (color) data on multiple parts of the model*
- 2D Image Representation
  *Support for editing the texture using existing 2D image editing/painting tools*

Evaluation: Quality

- Magnification Filtering
  *Bilinear filtering quality*
- Minification Filtering
  *Trilinear filtering (Mip-map) quality*
- Anisotropic Filtering
  *Anisotropic filtering quality*
Evaluation: Performance

- Storage Overhead
  The additional data needed beyond the texture (color) data

- Vertex Data Duplication
  The need to specify multiple mapping for some vertices

- Access Overhead
  Indirections needed for accessing the texture data

- Computation Overhead
  Additional computation needed for accessing/filtering the texture

- Hardware Filtering
  Can existing texture filtering hardware on GPUs be used?

Evaluation: Implementation

- Asset Production
  Development work needed for the asset production tools, such as texture painting and automated mapping

- Rendering
  Implementation work needed for developing texture sampling/filtering
# Standard 2D Textures

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Filtering Quality</th>
<th>Performance</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes: Yes</td>
<td>Magnification Filtering: Yes, with seam artifacts</td>
<td>Vertex Data Duplication: Yes</td>
<td>Asset Production: Huge array of sophisticated tools exist (uv-mapping+texture authoring)</td>
</tr>
<tr>
<td>Point Clouds: Single color per point</td>
<td>Minification Filtering: Yes, with seam artifacts</td>
<td>Storage Overhead: 2D mapping (uv x vert) &amp; wasted text. space</td>
<td>Rendering: GPU support: hard-wired, highly complex &amp; optimized texture fetch mechanism</td>
</tr>
<tr>
<td>Implicit Surfaces: With implicit mapping</td>
<td>Anisotropic Filtering: Yes, with seam artifacts</td>
<td>Access Overhead: None</td>
<td></td>
</tr>
<tr>
<td>Shape/Topology Limits: None</td>
<td></td>
<td>Computation Overhead: None</td>
<td></td>
</tr>
<tr>
<td>Subdivisions: Yes</td>
<td></td>
<td>Hardware Filtering: Yes</td>
<td></td>
</tr>
<tr>
<td>Tessellation Independence: As long as seams are preserved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Usability

|----------------------------|-------------------------------|------------------------------------------|--------------------------------------|------------------------|-----------------------------|

## Filtering Quality

<table>
<thead>
<tr>
<th>Magnification Filtering: Yes, with seam artifacts</th>
<th>Minification Filtering: Yes, with seam artifacts</th>
<th>Anisotropic Filtering: Yes, with seam artifacts</th>
</tr>
</thead>
</table>

## Performance

<table>
<thead>
<tr>
<th>Vertex Data Duplication: Yes</th>
<th>Storage Overhead: 2D mapping (uv x vert) &amp; wasted text. space</th>
<th>Access Overhead: None</th>
</tr>
</thead>
</table>

## Implementation

- **Asset Production**: Huge array of sophisticated tools exist (uv-mapping+texture authoring)
- **Rendering**: GPU support: hard-wired, highly complex & optimized texture fetch mechanism
PERFECTING TRADITIONAL UV-MAPS

Perfecting Traditional UV-Maps

RETHINKING TEXTURE MAPPING
4.1 Invisible seams

**Invisible seams**

Can we fix seams artifacts within the standard pipeline? ➔ Yes!

What is the problem?

Texels do not align across seams

It might seem this is hopeless, but invisible seams is actually a technique that can remove the seams entirely while not changing the standard pipeline at all.

Let’s have a closer look at the issue to understand the core idea.

This is a sphere textured with a typical approach. The rendering uses nearest mode to clearly see the big texels.

If we look closely where the charts meet on the surface, you will see that not only the colors disagree, but the grids are misaligned. This is why even if we tried to match the colors, the seam would still be there (unless, again, if the color is constant!).

Now, this second case shows you a very special mapping, called grid preserving. As you can see the colors mismatch but now the texel grid on both sides do match! This is revealed by the fact that the square boundaries perfectly align across the seam.

This third case shows what happens if you now match the colors on both sides. No seam! It is still there in fact, but because everything matches perfectly now – colors and alignment – the rendering perfectly agrees on both sides, making the seam effectively invisible.
Invisible seams

Given this carefully prepared UV map …

And here is what happens through bilinear filtering. Still no visible seam!
Here at the top left you can see two triangles along the surface. The blue square is an interpolation cell, that is, four texels in between which we’d like to interpolate color.

Now, the center drawing is the texture space. This particular mapping is grid preserving. What does this mean? It mean that I can take this green triangle on the left and translate it next to the right one. When doing this, we note two important things: First, the edges align perfectly, and second the translation is an integer vector, here (+5,+1).

This is why the interpolation cells actually perfectly align on both sides of the seam. Note that in general, a grid preserving mapping also allows (and requires) 90 degree rotations.

Because the interpolation cells match, it is enough to make sure the colors are the same. On one side the colors are inside the triangle, so we can expect these have been painted along the surface. On the right side they are outside of the triangle, so we can simply duplicate the color there.

After doing this, you can now see the interpolation cells. Note the dashed line, which is the edge of the triangles. The colors match perfectly along it, this is why no seam will be visible.
Pipeline

1- Compute grid preserving parameterization [difficult]

2- Regroup triangles in charts [easy]

3- Propagate color constraints [some difficulties]

So what invisible seams does is to first compute a grid preserving parameterization. This is a difficult problem, directly related to quadrangulation, but there is now quite a state of the art with good methods out there. Nevertheless, this is not something easy to implement, especially in a robust manner. Once this is done, the approach regroups triangles into charts, and then propagates colors so that interpolation across boundaries match exactly – as I have just described.
Invisible seams

Example of seamless UV map

Here is how it looks. On the left the rendering, on the right the automatically generated grid preserving mapping. You can see that the charts are not painting-friendly, which is one limitation.
There are additional benefits. MIP-mapping is supported, as well as multi-resolution, which comes from the fact that the grids align.
To conclude, invisible seams is a way to make rendering seams truly invisible, that supports MIP-mapping and multi-resolution and is backward compatible with whatever supports texture mapping. Unfortunately it is quite hard to implement, manual construction of grid preserving mapping is impossible without computational support, and as a consequence painting directly in the texture becomes quite difficult.

---

### Invisible Seams

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Filtering Quality</th>
<th>Performance</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes</td>
<td>Magnification Filtering Yes ★</td>
<td>Vertex Data Duplication Yes</td>
<td>Asset Production</td>
</tr>
<tr>
<td>Point Clouds</td>
<td>Minification Filtering Yes ★</td>
<td>Storage Overhead 2D mapping + wasted space</td>
<td>Automated mapping</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td>Anisotropic Filtering Yes but seams (could be fixed)</td>
<td>Access Overhead None ★</td>
<td>Rendering</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td>Depends on parameterization robustness</td>
<td>Computation Overhead None ★</td>
<td>Standard pipeline ★</td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td>If seams are preserved</td>
<td>Hardware Filtering Yes ★</td>
<td></td>
</tr>
<tr>
<td>Subdivisions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automated Mapping</td>
<td>Limited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual Mapping</td>
<td>Nearly impossible</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model Editing after Painting</td>
<td>Problematic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolution Readjustment</td>
<td>Problematic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texture Repetition</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D Image Representation</td>
<td>Poor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

To conclude, invisible seams is a way to make rendering seams truly invisible, that supports MIP-mapping and multi-resolution and is backward compatible with whatever supports texture mapping. Unfortunately it is quite hard to implement, manual construction of grid preserving mapping is impossible without computational support, and as a consequence painting directly in the texture becomes quite difficult.
4.2 Seamless Toroidal/Cylindrical Textures

**Seamless Toroidal/Cylindrical Textures**

[Tarini 2012]
Task: UV map this!
Task: UV map this!

Easy: cylindrical maps
Wrapping Textures

Q: can a triangle span from a to b?

• A: sure!
  • $a.u = 0.9$
  • $b.u = 0.1 = 1.1$
  • relay on wrap texture fetch mechanism

Oldest trick in the book!
Seams: invisible
Built in support in any GPU
Easy HW implementation.
Effortless.
The most common way of having a seam in a texture… which almost doesn’t count as a seam!
The perfect UV param (when it applies, which is not too often)
Wrapping Textures

Q: can a triangle span from a to b?

• A: sure!
• Q: so, can you avoid duplicating vertices in ------>
• A: sadly, no …but this can be helped!

*Almost* doesn’t count.
You still have to duplicate vertices:
complicate data structures.
Hinders procedurality: e.g. you cannot create U coords on the fly.

But, there is a simple little known technique to do this.
See: Cylindrical and toroidal parameterizations without vertex seams M Tarini
Journal of Graphics Tools 16 (3), 144-150
It is a short paper, just use it if it fits your needs.
Take home message

Contrary to common belief,
you don’t really need duplicate vertices in a cylindrical / toroidal map

How: Cylindrical and toroidal parameterizations
without vertex seams
M Tarini
JGT 2012

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

Let’s see how this work
Wrapping Textures

This triangle here needs RED VERTEX to be at 1.1
But this other triangle B here needs the *same* vertex to be at 0.1
Wrapping Textures

[Animation on slide – doesn't read well on printed slide]
If you used always used U coords in 0..1, any triangle spanning across the left-right seam would get wrong interpolated U values, like here
Wrapping Textures
Wrapping Textures

\[ u_1, \quad u_2 = \left[ u_1 + 0.5 \right] - 0.5 \]

[Animation on slide – doesn’t read well on printed slide]
But what if you used this other interval: U inside [-0.5, +0.5].
(it is easy to go from the prev interval to this, with this formula)
Now that triangle now works, thanks to the wrapping built in in the fetch mechanism.
But, now, a triangle spanning the original mid point of the texture gets the wrong interpolated position.
So this triangle requires $t$
Wrapping Textures
Wrapping Text

\[ u_1, u_2 = \lfloor u_1 + 0.5 \rfloor - 0.5 \]

{ vertices } →

\[ u_1 \text{ OR } u_2 ? \]
Wrapping Textures

\[ u_1, \quad u_2 = \left[ u_1 + 0.5 \right] - 0.5 \]

So this triangle needs the original cords \( u_1 \)
but this requires the modified coord $u_2$. Can we make all happy?
We could use the geometry shader to choose which way to go for each triangle, but the Geom Shader is often very costly. We don’t need it!

So here’s the trick.

In the vertex shader (first green arrow), you compute $u_2$ from $u_1$.
Both gets interpolated by the rasterizer (vertical arrow).

Last problem: in the fragment shader, you get two (potentially different) values $u_1$ and $u_2$.
We know one is right. The other might be wrong (see the two slides above)

How do you pick the right one?
This is seemingly impossible for the fragment shader to tell…
But: the answer is: it is always the one ($u_1$ or $u_2$) which is travelling less fast in screen space.
The triangle “spanning the texture in the wrong direction” is always the one “taking the longest route”.
(this assumes a triangle is never larger than HALF the entire texture space, which is more than reasonable).

So, you want to pick the other one.
You want to pick the $u_1$ or $u_2$ value… which is associated to the smallest (in module) SCREEN SPACE DERIVATIVE.
Recall that

$$u_1, u_2 = \lfloor u_1 + 0.5 \rfloor - 0.5$$
Screen space derivatives!

DEMO

Screen space derivatives!

DEMO
# Seamless Toroidal/Cylindrical Textures

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Filtering Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Polygonal Meshes</strong></td>
<td>Magnification Filtering</td>
</tr>
<tr>
<td>Quads &amp; triangles</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Point Clouds</strong></td>
<td>Minification Filtering</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Implicit Surfaces</strong></td>
<td>Anisotropic Filtering</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Shape/Topology Limits</strong></td>
<td></td>
</tr>
<tr>
<td><em>ONLY TORUS/CYLINDER TOPOLOGY</em></td>
<td></td>
</tr>
<tr>
<td><strong>Subdivisions</strong></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td><strong>Tessellation Independence</strong></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td><strong>Usability</strong></td>
<td></td>
</tr>
<tr>
<td>Automated Mapping</td>
<td>Magnification Filtering</td>
</tr>
<tr>
<td>Limited, usually easy</td>
<td>Yes</td>
</tr>
<tr>
<td>Manual Mapping</td>
<td>Minification Filtering</td>
</tr>
<tr>
<td>Not much customizability</td>
<td>Yes</td>
</tr>
<tr>
<td>Model Editing after Painting</td>
<td>Anisotropic Filtering</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Resolution Readjustment</td>
<td></td>
</tr>
<tr>
<td>Problematic</td>
<td></td>
</tr>
<tr>
<td>Texture Repetition</td>
<td></td>
</tr>
<tr>
<td>Yes (multiple rounds around cyl possible)</td>
<td></td>
</tr>
<tr>
<td>2D Image Representation</td>
<td></td>
</tr>
<tr>
<td>Yes, complete</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Filtering Quality</th>
<th>Performance</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnification Filtering</td>
<td>Vertex Data Duplication</td>
<td>Asset Production</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Automated mapping</td>
</tr>
<tr>
<td>Minification Filtering</td>
<td>Storage Overhead</td>
<td>Rendering</td>
</tr>
<tr>
<td>Yes</td>
<td>2D mapping (uv)</td>
<td>Simple UV manipulation</td>
</tr>
<tr>
<td>Anisotropic Filtering</td>
<td>Access Overhead</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Computation Overhead</td>
<td>Hardware Filtering</td>
<td></td>
</tr>
<tr>
<td>Extremely small (1 extra interpolant)</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

## SIGGRAPH ’17 Courses, July 30 - August 03, 2017, Los Angeles, CA, USA
4.3 Seamless Texture Atlases

[Seamless Texture Atlases]

Seamless Texture Atlases

• Begins with quadrangulation
  – Split the input mesh into sets of polygons that are flattened onto square-shaped regions on the uv-map

Seamless Texture Atlases
[Purnomo 2004]
Seamless Texture Atlases

- The texels for quad regions (charts) are packed into a texture
- The mip-map levels are stored within the same texture

- A lookup table stores locations of each chart for each mip-map level

Seamless Texture Atlases

- Alternatively, hardware mip-map storage can be used

- A lookup table stores locations of each chart
Seamless Texture Atlases

• Half a texel boundary needed around each chart
• Texture coordinates must be scaled according to the texel size of the mip-map level.

Seamless Texture Atlases

• Hardware bilinear filtering
• Indirection using a lookup table
• Requires quadrangulation and automated mapping per chart
## Seamless Texture Atlases

### Applicability

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes</td>
<td>Yes</td>
</tr>
<tr>
<td>Point Clouds</td>
<td>No</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td>No</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td>None</td>
</tr>
<tr>
<td>Subdivisions</td>
<td>Yes</td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td>No</td>
</tr>
</tbody>
</table>

### Filtering Quality

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnification Filtering</td>
<td>Yes</td>
</tr>
<tr>
<td>Minification Filtering</td>
<td>Yes, with custom mip-map construction</td>
</tr>
<tr>
<td>Anisotropic Filtering</td>
<td>Yes, with seam artifacts</td>
</tr>
</tbody>
</table>

### Usability

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated Mapping</td>
<td>Limited</td>
</tr>
<tr>
<td>Manual Mapping</td>
<td>Extra difficult</td>
</tr>
<tr>
<td>Model Editing after Painting</td>
<td>Problematic</td>
</tr>
<tr>
<td>Resolution Readjustment</td>
<td>Per patch only</td>
</tr>
<tr>
<td>Texture Repetition</td>
<td>Per patch only</td>
</tr>
<tr>
<td>2D Image Representation</td>
<td>Poor</td>
</tr>
</tbody>
</table>

### Performance

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Data Duplication</td>
<td>Yes</td>
</tr>
<tr>
<td>Storage Overhead</td>
<td>2D mapping &amp; indirection per chart</td>
</tr>
<tr>
<td>Access Overhead</td>
<td>1 indirection</td>
</tr>
<tr>
<td>Computation Overhead</td>
<td>Indirection</td>
</tr>
<tr>
<td>Hardware Filtering</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Implementation

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asset Production</td>
<td>Quadrangulation, automated mapping, and 3D painting</td>
</tr>
<tr>
<td>Rendering</td>
<td>Simple UV manipulation &amp; indirection</td>
</tr>
</tbody>
</table>
Connectivity-based Representations

RETHINKING TEXTURE MAPPING

Ptex – Per-Face Textures

• [Burley and Lacewell 2008]
5.1 Ptex

Ptex – Per-Face Textures

• [Burley and Lacewell 2008]

Images © Walt Disney Animation Studios

- Separate 2D texture per face
- No need for UV-mapping!
- Filtering near edges requires colors of neighboring faces
- Stores an adjacency list per face
Ptex – Per-Face Textures

- Adjacency data per face
  - 4 neighboring face IDs
  - 4 edge indices

- Example:

<table>
<thead>
<tr>
<th></th>
<th>Adjacent Faces</th>
<th>Adjacent Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>face 0</td>
<td>2,1,-1,-1</td>
<td>2,3,x,x</td>
</tr>
<tr>
<td>face 1</td>
<td>3,-1,-1,0</td>
<td>2,x,x,1</td>
</tr>
<tr>
<td>face 2</td>
<td>-1,3,0,-1</td>
<td>x,3,0,x</td>
</tr>
<tr>
<td>face 3</td>
<td>-1,-1,1,2</td>
<td>x,x,0,1</td>
</tr>
</tbody>
</table>

Ptex – Per-Face Textures

- Naturally supports quads
- Triangle texels are packed as quads

Triangle texels

odd texels are flipped around one edge

packed as quad
Ptex – Per-Face Textures

- Custom texture filtering (for filtering across edges)
- Multiple indirect lookups using the adjacency list
- Problems with extraordinary vertices

Ptex – Per-Face Textures

### Applicability

<table>
<thead>
<tr>
<th>Polygonal Meshes</th>
<th>Quads &amp; triangles (packed as quads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point Clouds</td>
<td>No</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td>No</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td>Problems with extraordinary vertices</td>
</tr>
<tr>
<td>Subdivisions</td>
<td>Yes</td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td>No</td>
</tr>
</tbody>
</table>

### Usability

<table>
<thead>
<tr>
<th>Automated Mapping</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Mapping</td>
<td>N/A</td>
</tr>
<tr>
<td>Model Editing after Painting</td>
<td>Yes</td>
</tr>
<tr>
<td>Resolution Readjustment</td>
<td>Yes</td>
</tr>
<tr>
<td>Texture Repetition</td>
<td>Only for identical topology</td>
</tr>
<tr>
<td>2D Image Representation</td>
<td>Poor</td>
</tr>
</tbody>
</table>

### Filtering Quality

<table>
<thead>
<tr>
<th>Magnification Filtering</th>
<th>Yes (seams only near extraordinary vertices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minification Filtering</td>
<td>Yes, up to single color per quad</td>
</tr>
<tr>
<td>Anisotropic Filtering</td>
<td>Possible with custom filtering</td>
</tr>
</tbody>
</table>

### Performance

<table>
<thead>
<tr>
<th>Vertex Data Duplication</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage Overhead</td>
<td>Neighborhood data &amp; face resolution</td>
</tr>
<tr>
<td>Access Overhead</td>
<td>Indirections</td>
</tr>
<tr>
<td>Computation Overhead</td>
<td>Custom filtering</td>
</tr>
<tr>
<td>Hardware Filtering</td>
<td>No</td>
</tr>
</tbody>
</table>

### Implementation

<table>
<thead>
<tr>
<th>Asset Production</th>
<th>3D painting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering</td>
<td>Custom filtering</td>
</tr>
</tbody>
</table>
5.2 Mesh Colors

Mesh Colors

- [Yuksel et al. 2008] [Yuksel et al. 2010]

Mesh Colors

- Extends vertex colors by adding edge colors and face colors.
- No need for UV-mapping!
Mesh Colors

• Conceptually similar to Ptex

Ptex
All texels are inside the face.

Mesh Colors
Texels on the edges and vertices are shared.

Mesh Colors

• Conceptually similar to Ptex

Ptex
All texels are inside the face.

Mesh Colors
Texels on the edges and vertices are shared.
Mesh Colors

- Custom texture filtering
- No indirect lookups and no adjacency list
- No problems with extraordinary vertices

Mesh Colors

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Filtering Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes</td>
<td>Magnification Filtering</td>
</tr>
<tr>
<td>Point Clouds</td>
<td>Minification Filtering</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td>Anisotropic Filtering</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td></td>
</tr>
<tr>
<td>Subdivisions</td>
<td></td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td></td>
</tr>
<tr>
<td>Automated Mapping</td>
<td></td>
</tr>
<tr>
<td>Manual Mapping</td>
<td></td>
</tr>
<tr>
<td>Model Editing after Painting</td>
<td></td>
</tr>
<tr>
<td>Resolution Readjustment</td>
<td></td>
</tr>
<tr>
<td>Texture Repetition</td>
<td></td>
</tr>
<tr>
<td>2D Image Representation</td>
<td></td>
</tr>
</tbody>
</table>

Performance

- Vertex Data Duplication: N/A
- Storage Overhead: face resolution
- Access Overhead: None
- Computation Overhead: Custom filtering
- Hardware Filtering: No

Implementation

- Asset Production: 3D painting
- Rendering: Custom filtering
5.3 Mesh Color Textures

Mesh Color Textures

- [Yuksel 2016]

- Convert mesh colors to 2D textures
- Duplicate vertex and edge colors
- Add interpolated colors along diagonally placed edges

\[
e_3 = e_1 + e_2 - e_0
\]
Mesh Color Textures

- Mipmap levels ($\ell$)

- 4D texture coordinates ($u_s, u_d$)
  - 2D texture coordinate for level $\ell$ is $u_{\ell} = u_s / 2^\ell + u_d$

Mesh Color Textures

- Hardware texture filtering
- Mip-map levels are stored in separate textures
- 2 texture calls for trilinear filtering
- Minimal computation overhead
  - Pick mip-map level
  - Convert 4D texture coordinate to 2D
  - Lerp results
### Mesh Color Textures

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Filtering Quality</th>
<th>Performance</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes</td>
<td>Magnification Filtering</td>
<td>Magnification Filtering</td>
<td>Asset Production</td>
</tr>
<tr>
<td>Quads &amp; triangles</td>
<td>Minification Filtering</td>
<td>Minification Filtering</td>
<td>Rendering</td>
</tr>
<tr>
<td>Point Clouds</td>
<td>Anisotropic Filtering</td>
<td>Vertex Data Duplication</td>
<td>4D mapping &amp; wasted space</td>
</tr>
<tr>
<td>Single color per point</td>
<td>None</td>
<td>Storage Overhead</td>
<td>UV calculation</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td></td>
<td>Access Overhead</td>
<td>Hardware Filtering</td>
</tr>
<tr>
<td>No</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td></td>
<td>Computation Overhead</td>
<td></td>
</tr>
<tr>
<td>None</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subdivisions</td>
<td></td>
<td>Hardware Filtering</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td></td>
<td>Vertex Data Duplication</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usability</td>
<td></td>
<td>Storage Overhead</td>
<td></td>
</tr>
<tr>
<td>Automated Mapping</td>
<td></td>
<td>Access Overhead</td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td></td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Manual Mapping</td>
<td></td>
<td>Computation Overhead</td>
<td></td>
</tr>
<tr>
<td>No need</td>
<td></td>
<td>4D mapping &amp; wasted space</td>
<td></td>
</tr>
<tr>
<td>Model Editing after Painting</td>
<td></td>
<td>Hardware Filtering</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td>UV calculation</td>
<td></td>
</tr>
<tr>
<td>Resolution Readjustment</td>
<td></td>
<td>Vertex Data Duplication</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texture Repetition</td>
<td></td>
<td>Storage Overhead</td>
<td></td>
</tr>
<tr>
<td>Only for identical topology</td>
<td></td>
<td>Access Overhead</td>
<td></td>
</tr>
<tr>
<td>2D Image Representation</td>
<td></td>
<td>Computation Overhead</td>
<td></td>
</tr>
<tr>
<td>Poor</td>
<td></td>
<td>Hardware Filtering</td>
<td></td>
</tr>
</tbody>
</table>

- **Applicability:**
  - Polygonal Meshes: Quads & triangles
  - Point Clouds: Single color per point
  - Implicit Surfaces: No
  - Shape/Topology Limits: None
  - Subdivisions: Yes
  - Tessellation Independence: No

- **Filtering Quality:**
  - Magnification Filtering: Yes
  - Minification Filtering: Yes
  - Anisotropic Filtering: Yes (seam artifacts on current hardware)

- **Performance:**
  - Vertex Data Duplication: Yes
  - Storage Overhead: 4D mapping & wasted space
  - Access Overhead: None
  - Computation Overhead: UV calculation
  - Hardware Filtering: Yes

- **Implementation:**
  - Asset Production: 3D painting
  - Rendering: Simple UV calculation

 SIGGRAPH '17 Courses, July 30 - August 03, 2017, Los Angeles, CA, USA
We will now discuss a set of methods that share a common idea: defining texture information in a volume surrounding the surface.
Volume textures

Voxel size grows to the cube of resolution \(\Rightarrow\) exploit sparsity!

3D Texturing: store colors in voxels

Here you can see on the left a textured surface, and on the right the set of voxels which encode the colors. Of course only few voxels are actually useful: those that intersect the surface. Thus storing the full volume would be a bad idea: a vast percentage of memory would be wasted on voxels which are never accessed.
Sparse Volume Methods

Adaptive texture maps
[Kraus and Ertl 2002]

Octree textures, \(N^3\)-Trees, Brickmaps, and Gigavoxels
[Benson and Davis 2002; Christensen and Batali 2004; Lefebvre et al. 2005; Lefohn et al. 2006; Crassin et al. 2009]

Spatial Hashing

Several schemes have been proposed to encode such sparse textures for computer graphics applications. We will discuss adaptive texture maps, octree textures and its variants, as well as spatial hashing.
One of the first techniques to try to skip over never accessed data are the Adaptive Texture Maps of Kraus and Ertl.
The idea is to introduce an indirection in the texture. The indirection table is a regular grid that covers the texture space. Each cell can be either empty, or it contains the coordinates of a tile. During texture lookup, if the grid cell is empty a background color is returned – but usually these cells are never accessed. If the grid cell is not empty, then the lookup coordinate is transformed into a coordinate in the packed tile map, where the actual texture lookup is performed.

There are several benefits: there is less memory waste, and it is also possible to scale up or down each individual tile, for instance lowering the resolution in regions having smooth color variations. It is even possible to do instancing, by reusing a tile in several entries of the indirection map.

Of course this idea extends to 3D as well.
Adaptive texture maps are a simple but efficient way to skip empty space in textures, and provides additional flexibility such as local resolution adjustment. However, even though they require a single indirection, this requires a custom shader to correctly perform interpolation and MIP-mapping. Also, in volumes, the single indirection does not allow a tight fit around the surface, meaning that a lot of empty space is still stored.

Adaptive texture maps

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Filtering Quality</th>
<th>Performance</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes</td>
<td>Magnification Filtering Yes</td>
<td>Vertex Data Duplication Yes</td>
<td>Asset Production</td>
</tr>
<tr>
<td>Point Clouds</td>
<td>Minification Filtering Yes (up to tile size)</td>
<td>Storage Overhead 2D mapping (uv) + indirection map</td>
<td>Standard UVs + automated tile construction</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td>Anisotropic Filtering Yes, with seam artifacts</td>
<td>Access Overhead Single indirection</td>
<td>Rendering</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td>None</td>
<td>Computation Overhead Indirection, tile map construction</td>
<td>Single indirection</td>
</tr>
<tr>
<td>Subdivisions</td>
<td>If seams are preserved</td>
<td>Hardware Filtering Yes</td>
<td></td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td>Limited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usability</td>
<td>Manual Mapping Often needed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automated Mapping</td>
<td>Limited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model Editing after Painting</td>
<td>Problematic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolution Readjustment</td>
<td>Yes (local increase in image space)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Texture Repetition</td>
<td>Yes (+ instancing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D Image Representation</td>
<td>Poor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Octree Textures

- Spatial Hierarchies

Octree texture focus exactly on this issue, by generalizing to multiple lookups. Here you can see on the left a colored 3D model, and on the right the octree that encodes the color content. You can see how the structure densify around the surface until it reaches the desired voxel resolution.

Colors are stored in the octree leaves, but also in internal nodes for MIP-mapping.
Octree Textures

- Lookup from surface point

Here is how the lookup is performed. The root node is in red, and the lookup point is the white circle. The coordinates of the lookup point are first expressed as coordinates within the root node. On the right, you can see the 3D texture that stores the actual tree nodes, which we call the node pool.

Each node is a small 2x2x2 brick, that stores coordinates to its child nodes within the node pool, or colors. Having the coordinate of the lookup point within the root, we read the coordinates of its child node. We then compute the coordinates of the lookup point within the child done, and access it in the node pool. This gives us a new child node coordinate, that we access similarly. This time a color is retrieved: we reached the bottom of the octree.

This is a very simple process to implement. The main drawback is that it requires several dependent lookups, log(N) of them with N the texture resolution.
Generalizations

Brickmaps, $N^3$ trees
  - Split by more than 2 at each level
  - Memory-access tradeoff
  - Optimized for off-line rendering (cache)

Gigavoxels
  - Stores opacity for volume rendering
  - Produces / loads content during traversal

This idea has been generalized to subdivide the node into more than 2x2x2. This allows a tradeoff between memory and access depth. Indeed the octree provides the tighter fit, but reaching the bottom of a 4096 cube tree require 12 lookups. Instead, using 8x8x8 subdivisions will require only 4 lookups – but will use more memory.

Another extension of this idea is to store density in space for volume rendering. This has been done by Gigavoxel, which also introduces an efficient stackless traversal and proposes to generate volume data on-the-fly, only when needed for rendering.
To recap, octrees and their generalizations have the following advantages:

- They require no uv and are very simple to build – nowadays this an be done directly on the GPU.
- They afford for simple local resolution adjustment: just subdivide more locally!

However they have one major drawback, which is that they require a rather expensive access with multiple dependent texture lookups. Note that this will run at hundreds of frame per seconds on a modern GPU, but that remains significantly more expensive that regular texture mapping. Another issue is that filtering, in particular interpolation, requires more accesses, as the shader as to implement it with up to eight lookups.

Volume approaches also suffer from a thin sheet limitation: it is not possible to give different colors to surfaces in close proximity ; even though some works suggest using the normal to distinguish between different directional colors.

Finally, like all volume techniques, this requires authoring to be done with a 3D texture painting tool.
6.3 Perfect Spatial Hashing

How to reduce the overhead?

Overhead in space and time

- **Space**: pointers and nodes
- **Time**: chain of indirection pointers

The main issues with the octree textures we have seen before is that they are hierarchies: Storing the structure of the hierarchy requires additional memory, and accessing it requires extra time.
Instead, spatial hashing stores only the used pixels (or voxels) in a compact hash table. The location of a pixel within the table is given by a hash function applied to its coordinates.

In addition it is possible to find minimal perfect hashes. A minimal hash is one where the hash table is full. In other words there are no unused slots.

One important point is that the hash function itself needs to be encoded and this may take up a significant amount of memory. So a key question is how this does compare to the pointers of a hierarchy. Well it turns out that the overhead can be much smaller, as small as 1.44 bits per entry.
The idea of a perfect spatial hash is to encode the hash function using a small auxiliary table, called the offset table. This is inspired by previous work for hashing dictionaries of strings.

The hash function can be very simple. It starts by accessing the offset table. The access is done using warp around addressing, so many pixels will be mapped to the same location.

The retrieved offset is then added the coordinates of the pixel, and the offset address is used to access the hash table again with modulo addressing.

The challenge of course is to create an offset table so that the resulting hash function is perfect.

As it turns out, it is possible to create a simple hash function that is well suited for GPU evaluation. It contains no branch instructions and enables efficient SIMD execution. It requires only four instructions in most cases, and, on average, only 4 bits per entry. This is more than the theoretical lower bound of 1.44 bits but is still a much smaller overhead than with hierarchies.
This process I just described in 2D is easily extended to 3D. Runtime access has the exact same cost as GPUs perform vector arithmetic.
Filtering can also get tricky, and the most efficient approach is to encode blocks of texture data as opposed to single values. This results in higher memory usage however.

Perfect Spatial Hashing: Filtering

- Bilinear / trilinear interpolation
- MIP-mapping

Filtering can also get tricky, and the most efficient approach is to encode blocks of texture data as opposed to single values. This results in higher memory usage however.
MIP-mapping is supported by flattening the pyramid in a single sparse texture, using the level identifier to compute the new coordinates.
Interpolation can be done in the shader with multiple accesses. This is a bit unfortunate, because now we have up to 16 accesses in 3D (eight times the two required for nearest mode access).

- In 3D, slower by a factor of 3 to 7
Instead, one possible approach is to use blocking. The idea is to store into the hash small texture blocks, instead of single colors. The boundary color is duplicated between neighboring blocks. The advantage is that native trilinear interpolation can be used, and we are back to two lookups. However, this incurs a large memory overhead.

Btw, this approach can also be used with octrees.
Perfect Spatial Hashing: 3D Texture

Here you can see a result. In this example the dragon is textured with a sparse 1k cubed volume.

Using a blocked hash for native trilinear interpolation, the texture fits in 77 MB, including MIP-mapping – this is shown on the left. On the right, without the blocking the storage is down to 17MB.
Perfect spatial hashing solves one of the key issues of hierarchies, the multiple dependent accesses, while preserving a tight storage. It requires only one indirection for nearest mode lookups, but unfortunately things become less elegant with interpolation, which either incurs an access overhead, or a memory overhead.

The construction process of perfect spatial hashing is easy to implement, however it can take a long time as it is based on a stochastic exploration.

<table>
<thead>
<tr>
<th>Perfect Spatial Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Applicability</strong></td>
</tr>
<tr>
<td>Polygonal Meshes</td>
</tr>
<tr>
<td>Point Clouds</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
</tr>
<tr>
<td>Subdivisions</td>
</tr>
<tr>
<td>Tessellation Independence</td>
</tr>
<tr>
<td><strong>Usability</strong></td>
</tr>
<tr>
<td>Automated Mapping</td>
</tr>
<tr>
<td>Manual Mapping</td>
</tr>
<tr>
<td>Model Editing after Painting</td>
</tr>
<tr>
<td>Resolution Readjustment</td>
</tr>
<tr>
<td>Texture Repetition</td>
</tr>
<tr>
<td>2D Image Representation</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
</tr>
<tr>
<td>Rendering</td>
</tr>
<tr>
<td>Without blocking: custom shader</td>
</tr>
</tbody>
</table>

Perfect spatial hashing solves one of the key issues of hierarchies, the multiple dependent accesses, while preserving a tight storage. It requires only one indirection for nearest mode lookups, but unfortunately things become less elegant with interpolation, which either incurs an access overhead, or a memory overhead. The construction process of perfect spatial hashing is easy to implement, however it can take a long time as it is based on a stochastic exploration.
We will now discuss a set of methods that share a common idea: defining texture information in a volume surrounding the surface.
The main motivation of the tiletree is that, when texturing a surface with a volume, some additional samples are stored. For instance, in this figure, the piece of surface is near horizontal, and clearly only the top (pink) samples would be enough to define a color field – of course the bottom ones would be fine as well, but the point is that we do not need both.

TileTrees: Motivation

- Volume textures are great but:
  - Interpolation requires 8 lookups
  - Interpolation requires 8 samples
TileTrees: Key Idea

- Position 2D square tiles around the surface
- Project onto tiles at rendering time
  - Volume for positioning: simplicity + versatility
  - 2D square tiles: efficiency + hardware support

(animated)
So what the tile tree does is to use a volume hierarchy – a shallow octree – to position 2D tiles around the surface. Then the tiles are looked up from the surface, by a simple projection.
This is illustrated on the right, for a side view. The arrows are showing how the surface projects onto the tiles.
Here you can see for volume cells, and this red interval is a 2D tile, and this is another, and here you can see which part is actually used, while this other is not.
TileTrees: Implicit Parameterization

- Within each tree leaf:
  - Surface projected onto tiles
  - Implicit local parameterization using normal

Now, instead of storing an actual mapping, the idea is simply use the normal—so obviously this requires normal to be defined along the surface. The mapping is thus implicit, and the projection is performed along the axis that is most aligned with the normal. This can be +/- X, +/- Y or +/- Z. During construction, the algorithm ensures that all required tiles are there.
TileTrees: Implicit Parameterization

Quite powerful:

This approach is surprisingly powerful, here you can see a number of configurations that are supported.
TileTrees: Implicit Parameterization

When does it break?

- Incorrect!
- Low coverage!

What if it *does* break?

However it does break in some cases, such as this double bump. So what happens in such a case?
This is why there is a hierarchy. In such a case the cell is subdivided to allow for additional, smaller tiles to be positioned.
Another case for subdivision is when the tile is not well covered: this would be wasteful.
So, here is the result for a torus. On the left a visualization of the tiletree in space; on the right the texture storing all the tiles.
**TileTrees: Lookup**

For a surface point $p$ with normal $n$

1. Lookup the tree at $p$
2. Select leaf-face with $n$
3. Project $p$ onto face
4. Access tile map

The lookup process is fairly simple: first locate the cell enclosing the point, then select a face using the normal, project the point onto it and finally access the tile map using standard texture accesses. It gets more involved to properly define filtering across tile boundaries, but that can be done. The final shader is about 30 lines, with 10 texture lookups when the hierarchy has three levels.
Here is a more complex result, and the same dragon as before.

As you can see, this uses 11.3 MB versus 17 MB for the perfect hash without blocking. This means that in this case it provides a more compact representation with less overhead for the filtered access.
To recap, the tiletree is another technique that requires no UV. The lookup is fairly simple, and builds upon 2D filtering in the tilemap, but requires several accesses. One drawback is that the construction process is more difficult to implement than an octree or a hash map. It computes faster than a hash but still slower than an octree. The access overhead, even if smaller especially with interpolation, remains much more expensive than a standard texture map. Also keep in mind the comparison is subtle because there is a memory-access tradeoff.

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Filtering Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes</td>
<td>Magnification Filtering</td>
</tr>
<tr>
<td>Point Clouds</td>
<td>Minification Filtering</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td>Anisotropic Filtering</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td></td>
</tr>
<tr>
<td>Subdivisions</td>
<td></td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td></td>
</tr>
<tr>
<td>Usability</td>
<td></td>
</tr>
<tr>
<td>Automated Mapping</td>
<td>Vertex Data Duplication</td>
</tr>
<tr>
<td>Manual Mapping</td>
<td>Storage Overhead</td>
</tr>
<tr>
<td>Model Editing after Painting</td>
<td>Access Overhead</td>
</tr>
<tr>
<td>Resolution Readjustment</td>
<td>Computation Overhead</td>
</tr>
<tr>
<td>Texture Repetition</td>
<td>Hardware Filtering</td>
</tr>
<tr>
<td>2D Image Representation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Asset Production</td>
</tr>
<tr>
<td></td>
<td>Rendering</td>
</tr>
</tbody>
</table>

Rethinking Texture Mapping

Marco Tarini, Cem Yuksel, and Sylvain Lefebvre
7.2 PolyCube Maps

Polycube-Maps

- [Tarini et-al 2014]

The Goal

- Texture mapping
  - seamless
  - hardware supported
  - low distortion
  - general object
Texture Atlas
(multi-chart approach to parameterization)

3D mesh + 2D texture image = textured bunny

Images courtesy of Lévy, Sylvain, Ray and Maillot, SIGGRAPH 02

Cube-Maps

- Typically used for environment mapping

Images from Bubble demonstration program, nVidia
**Abusing Cube-Maps**

- What if we store surface color in a cube-map?

object space

mesh

with per-vertex

3D texture coord. \((u,v,w)\)

texture space

cube-map

textured apple

SIGGRAPH ’17 Courses, July 30 - August 03, 2017, Los Angeles, CA, USA
Rethinking Texture Mapping

Marco Tarini, Cem Yuksel, and Sylvain Lefebvre

**Texture Atlas**
- seams
- a triangle cannot span multiple charts
- mesh dependency
- mipmapming difficult
- chart packing: wasted texels
- artifacts at boundaries
- no defined neighbors for boundary texels

**Cube-Map**
- seamless
- a triangle can span multiple faces
- mesh independent
- mipmapming ok
- no packing: no wasted texels
- no boundaries, no artifacts
- texel neighbors always defined
- **texture atlas**
  - seams
  - a triangle *cannot* span multiple charts
  - mesh dependency
  - mipmapping difficult
  - chart packing: wasted texels
  - artifacts at boundaries
  - no defined neighbors for boundary texels

- **cube-map**
  - seamless
  - a triangle *can* span multiple faces
  - mesh independent
  - mipmapping ok
  - no packing: no wasted texels
  - no boundaries, no artifacts
  - texel neighbors always defined
**Rethinking Texture Mapping**

Marco Tarini, Cem Yuksel, and Sylvain Lefebvre

### Cube-Map
- Seamless
- A triangle can span multiple faces
- Mesh independent
- Mipmapping ok
- No packing: no wasted texels
- No boundaries, no artifacts
- Texel neighbors always defined

### Texture Atlas
- Seams
- A triangle cannot span multiple charts
- Mesh dependency
- Mipmapping difficult
- Chart packing: wasted texels
- Artifacts at boundaries
- No defined neighbors for boundary texels

---

- **works** (is general)
- **does not** (~spheres only!)
What to Keep from CubeMaps

- Texture defined in 3D *BUT* stored in 2D
- Per-fragment
- Hardware implemented

Going Beyond Apples

- Cube-Maps work well *only* for sphere-like objects
Going Beyond Apples

• But for more general objects?

[Diagram: world space (arbitrary meshes) vs. texture space (cubic texture domain) with a question mark in the middle.]

...huge distortions, incompatible topology
Going Beyond Apples

- But for more general objects?

Introducing Polycubes

Po-ly-cube: n. (Geom.) A solid composed by multiple unit cubes attached face to face
Choosing a Polycube

- Polycube should *roughly* resemble the mesh

Partition of Texture Space
partition of the parameter space

back to 3D

case 3  case 4-a  case 4-b  case 5  case 6-a  case 6-b

dual cells

case 4-a  case 5  case 4-b  case 3
Partition of the Parameter Space

dual cells

case 4-a case 5

PolyCube-Maps in a Nutshell

object

object with interpolated texture coord

not necessarily on the polycube surface: project

stored in texture RAM as:

a fragment with interpolated texture coord

the mesh (with per-vertex text. coord)

the polycube

final texel value for the fragment

a packed texture image

plus a tiny structure to store polycube layout
Examples of poly-cubic parameterizations

object

world space

polycube-map

texture space

Examples of poly-cubic parameterizations

object

world space

polycube-map

texture space
Examples of poly-cubic parameterizations

object
world space

polycube-map
texture space

Examples of poly-cubic parameterizations

object
world space

polycube-map
texture space
An example application: same texture for different LOD

Geometry 3 + Geometry 2 + Geometry 1 = the same polycube-map texture

How to build a Polycube-Map (for a given mesh)

- Not automatic, to this point
  - get a suitable polycube
  - warp it around the mesh
  - project mesh over it
  - unwarp
Global optimization

Discussion

Pros

- truly seamless texture mapping
- no patch boundaries
- no color bleeding
- very low distortion
- nearly optimal texture packing
- bilinear filtering possible
- mipmapping possible
- mesh independency

Cons

- long fragment program
  - ~60 GPU instruction long
  - could be improved, with little branching support
- 3 t-coords per vertex
  - instead of 2

Limits

- cannot handle arbitrary shape/topology complexity
  - e.g. a tree
# Polycube-Maps

<table>
<thead>
<tr>
<th><strong>Applicability</strong></th>
<th><strong>Filtering Quality</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes</td>
<td>Magnification Filtering</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Point Clouds</td>
<td>Minification Filtering</td>
</tr>
<tr>
<td>Single color per point</td>
<td>Yes, up to cube face size</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td>Anisotropic Filtering</td>
</tr>
<tr>
<td>No</td>
<td>No (lacks tangent directions)</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td>Topology must be reproduced with low res polycube</td>
</tr>
<tr>
<td>Subdivisions</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Usability</strong></th>
<th><strong>Performance</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated Mapping</td>
<td>Vertex Data Duplication</td>
</tr>
<tr>
<td>Limited</td>
<td>None</td>
</tr>
<tr>
<td>Manual Mapping</td>
<td>Storage Overhead</td>
</tr>
<tr>
<td>Extra task: polycube.construction</td>
<td>133% storage for texture coords (u,v,w)</td>
</tr>
<tr>
<td>Task removed: cut identification</td>
<td>Access Overhead</td>
</tr>
<tr>
<td>1 indirection</td>
<td>Computation Overhead</td>
</tr>
<tr>
<td>Limited (5 cases, fits in a ~30 line fragment prg)</td>
<td>Hardware Filtering</td>
</tr>
<tr>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Implementation</strong></th>
<th><strong>2D Image Representation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Asset Production</td>
<td>Poor</td>
</tr>
<tr>
<td>3D painting &amp; polycube generation</td>
<td></td>
</tr>
<tr>
<td>Rendering</td>
<td></td>
</tr>
<tr>
<td>Custom texture access</td>
<td></td>
</tr>
</tbody>
</table>

## Applicability
- **Polygonal Meshes**: Yes
- **Point Clouds**: Single color per point
- **Implicit Surfaces**: No
- **Shape/Topology Limits**: Topology must be reproduced with low res polycube
- **Subdivisions**: Yes
- **Tessellation Independence**: Yes

## Usability
- **Automated Mapping**: Limited
- **Manual Mapping**: Extra task: polycube.construction
- **Model Editing after Painting**: No
- **Resolution Readjustment**: Possible: add cubes
- **Texture Repetition**: Possible: see WANG TILES+PCM

## Performance
- **Vertex Data Duplication**: None
- **Storage Overhead**: 133% storage for texture coords (u,v,w)
- **Access Overhead**: 1 indirection
- **Computation Overhead**: Limited (5 cases, fits in a ~30 line fragment prg)
- **Hardware Filtering**: Yes

## Implementation
- **Asset Production**: 3D painting & polycube generation
- **Rendering**: Custom texture access
7.3 Volume-encoded UV-Maps

Volume-Encoded UV-maps

[Tarini 2016]

Box \subseteq \mathbb{R}^3

[f]_{UV-map} \text{ represented as…}

Surface-S

Texture Space $[0,1] \times [0,1]$
Encoding $f$

store in small 3D texture

Evaluating $f$

```c
vec2 texture_coord_for_p( vec3 p )
{
    p ← p · scale + p0;
    return text_fetch_3d( p );
}
```

object space

go to $[0,1]^3$

trilinear HW interpolation

Minimal ALU

Single indirection

Cache coherent

Hardwired GPU
No vertex duplicates
No per-vert UV coords

LOD1 LOD2
LOD3 LOD4
arbitrary remeshing
(here, quads)
Range scans

Scan1 + Scan2 + Scan3 + ... = Combined

Try our demo!
google for: Volume Encoded UV-maps
Encoding $f$

Regular Volumetric Sampling
+ Trilinear Interpolation
= too simple??

U-V coords
On volume. Low freq.

Regular Sampling

Signal
On surface. High freq.
Rethinking Texture Mapping

Marco Tarini, Cem Yuksel, and Sylvain Lefebvre
Volumetric UV-Maps: Cuts
Adding discontinuities

texture_coord_for_p( vec3 p )
{
    p ← p · scale + p0 ;
    p ← p + ⌊ p / k ⌋ ;
    return text_fetch_3d( p ) ;
}

Cuts are Volumetric too!
Using Vol-Encoded UV-maps

text_coord_for_p( vec3 p )
{
    p ← p · scale + p0 ;
    p ← p  + ⌊ p /  k   ⌋ ;
    return text_fetch_3d( p );
}

Final 2D texture access
Rethinking Texture Mapping

Marco Tarini, Cem Yuksel, and Sylvain Lefebvre
Bilinear Interpolation
Rethinking Texture Mapping

Marco Tarini, Cem Yuksel, and Sylvain Lefebvre

Texture Repeat

Construction of Volume Encoded UV-maps
Volume-Encoded UV-maps: construction

Standard UV-map
Volume-Encoded UV-maps: construction

Standard UV-map

Vol-Encoded UV-map
Over Range scans

Scan1 + Scan2 + Scan3 + ... = Combined

Over a point cloud

Combined image of point clouds and 3D models.
UV function: objectives

\[ f : (x, y, z) \rightarrow (u, v) \]

Texture Space \([0,1] \times [0,1]\)

Restriction on \(S\):
- Low Distortion
- No overlap
- Good / Few cuts
- Good Coverage

Near \(S\):
Away from \(S\):
UV function: objectives

\[ f : (x, y, z) \rightarrow (u, v) \]

Restriction on \( S \):
- Near \( S \):
- Away from \( S \): \( \{ \text{we don't care} \)
Orthogonality of $f$ to $S$

Surface $S$

Texture

Orthogonality of $f$ to $S$

Surface $S$

Texture
Orthogonality of $f$ to $S$

Texture

Surface $S$

Construction: Single Patch / Global

$\mathbf{f}: (x, y, z) \rightarrow (u, v)$

On $S$:
- Low Area Distortion
- Low Angle Distortion
- No Local Overlaps
- No Global Overlaps
- Good Coverage
- Good / Few cuts

Near $S$:

$\sim$ constant in normal directions
Construction 1/2: Single Patch

\[ f: (x, y, z) \rightarrow (u, v) \]

On \( S \):
- Low Area Distortion
- Low Angle Distortion
- No Local Overlaps
- No Global Overlaps
- Good Coverage
- Good / Few cuts

Near \( S \):
- \( \sim \) constant in normal directions

Construction 1/2: Single Patch

\[ J_f(p) = (\nabla u(p), \nabla v(p)) \]

linear with the vars!

\[ n \times \nabla u(p) = \nabla v(p) \]
\[ \nabla v(p) \times n = \nabla u(p) \]
Construction 1/2: Single Patch

\[ f: (x, y, z) \rightarrow (u, v) \]

On \( S \):
- Low Area Distortion
- Low Angle Distortion
- No Local Overlaps
- Good Coverage
- Good / Few cuts

Near \( S \):
- \( \sim \) constant in normal directions

\[
J_f(p) = (\nabla u(p), \nabla v(p))
\]

linear with the vars!

\[
n \times \nabla u(p) = \nabla v(p)
\]
\[
\nabla v(p) \times n = \nabla u(p)
\]
Energy for Single Patch Construction

\[ \mathbf{J}_f(p) = (\nabla u(p), \nabla v(p)) \]

linear with the vars!

\[ \mathbf{n} \times \nabla u(p) = \nabla v(p) \]

\[ \nabla v(p) \times \mathbf{n} = \nabla u(p) \]

\[ \nabla u(p) \times \nabla v(p) = \mathbf{n} \cdot \mathbf{k}_a \]
Construction 2/2: **Global**

\[ f : (x, y, z) \rightarrow (u, v) \]

**On S:**
- Low Area Distortion
- Low Angle Distortion
- No Local Overlaps
- No Global Overlaps
- Good Coverage
- Good / Few cuts

**Near S:**
- ~ constant in normal directions

**Siggraph '17 Courses, July 30 - August 03, 2017, Los Angeles, CA, USA**
Construction 2/2: Global
Texture Authoring: business as usual
Texture Authoring: business as usual

VeUV with Tangent Space Normal Mapping
**VeUV with Skinning**

![VeUV with Skinning](image1)

**Direct Texture Painting**

![Direct Texture Painting](image2)
Limitation

- Not one solution for all cases
- Too tiny features $\Rightarrow$ local loss of injectivity

Conclusions

A new **UV-map representation** (volumetric!)
Equivalent to traditional per-vertex one…
…but:
- applicable to most surface representations (not just 2-manifold meshes)
- no vertex replications (in polygonal mesh)
- often $<\ll$ space (but not always)
- independent on the meshing!
- texture + UV-map can be shared by different LoDs
VeUV with Invisible Seams?

- The two techniques can in theory be combined!

**Vol Encoded UV-maps**

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Filtering Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygonal Meshes</td>
<td>Magnification Filtering</td>
</tr>
<tr>
<td>Point Clouds</td>
<td>Yes, with seam artifacts</td>
</tr>
<tr>
<td>Implicit Surfaces</td>
<td>Minification Filtering</td>
</tr>
<tr>
<td>Shape/Topology Limits</td>
<td>Yes, with seam artifacts</td>
</tr>
<tr>
<td>Subdivisions</td>
<td>Anisotropic Filtering</td>
</tr>
<tr>
<td>Tessellation Independence</td>
<td>No</td>
</tr>
</tbody>
</table>

**Usability**

| Automated Mapping                  | Vertex Duplication           |
| Manual Mapping                     | None                         |
| 3D Editing after Painting          | Storage Overhead             |
| Resolution Readjustment            | can much better or much worse|
| Texture Repetition                 | Access Overhead              |
| 2D Image Representation            | 1 indirection (trilinearly interpolated) |

**Performance**

| Asset Production                   | Computation Overhead         |
| Rendering                          | Tiny program in vertex fragment (a pair of instructions) |

**Implementation**

<table>
<thead>
<tr>
<th>good</th>
<th>dubious</th>
<th>bad</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes**

- Limited
- as customizable as standard
- No
- Problematic
- Yes
- Simple indirection
REFERENCES


