Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

# Tile-Based Texture Mapping on Graphics Hardware

Li-Yi Wei

NVIDIA Corporation

**Abstract**

*Texture mapping has been a fundamental feature for commodity graphics hardware. However, a key challenge for texture mapping is how to store and manage large textures on graphics processors. In this paper, we present a tile-based texture mapping algorithm by which we only have to physically store a small set of texture tiles instead of a large texture. Our algorithm generates an arbitrarily large and non-periodic virtual texture map from the small set of stored texture tiles. Because we only have to store a small set of tiles, it minimizes the storage requirement to a small constant, regardless of the size of the virtual texture. In addition, the tiles are generated and packed into a single texture map, so that the hardware filtering of this packed texture map corresponds directly to the filtering of the virtual texture. We implement our algorithm as a fragment program, and demonstrate performance on latest graphics processors.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture: Graphics Processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Texture;

**Keywords:** Texture Mapping, Graphics Hardware, Texture Synthesis

## 1. Introduction

Texture mapping [Hec86] is a technique to represent surface details in computer rendered images without adding geometric complexity. Texture mapping has been a standard feature for recent consumer-level graphics hardware. The cost of texture mapping on graphics hardware includes the memory for storing textures, as well the bandwidth for transferring and accessing these textures. For applications that use large amounts of textures, the storage or bandwidth requirements may prohibit the graphics hardware from achieving real-time performance.

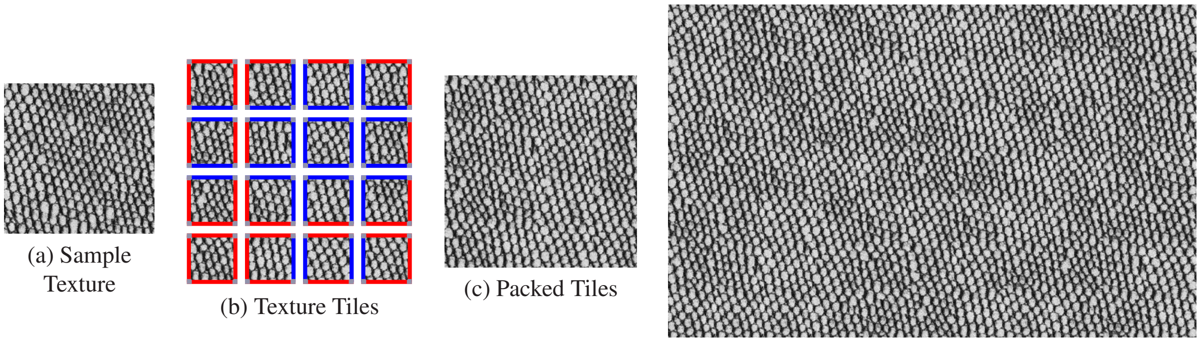
One possible solution to address these problems is texture compression [BAC96]. However, existing texture compression techniques are designed for general images and may achieve sub-optimal compression ratio for textures that contain repetitive patterns. For this kind of textures, texture synthesis algorithms [Wei02] usually provide better compression ratio. Unfortunately, previous texture synthesis algorithms are often too slow or too complex for real time applications on graphics hardware.

We address this problem by implementing a large virtual texture as a stochastic tiling of a small set of texture image tiles [CSHD03]. Our technique functions transparently as a traditional texture mipmap and provides the illusion of the availability of a large, non-periodic virtual texture while consuming only a small amount of texture memory. We implement our technique as a fragment program and demonstrate performance on current generation programmable graphics hardware.

## 2. Previous Work

A variety of techniques have been proposed to reduce the texture storage requirements. We classify these techniques into three categories : Texture Compression, Texture Synthesis, and Texture Tiling.

**Texture Compression** : One way to reduce texture memory consumption is to compress the textures. [BAC96] presented a texture compression scheme based on Vector Quantization (VQ). This technique allows fast texture decoding while achieving a compression ratio up to 35:1, but may require the entire texture VQ codebook to be stored on-chip. Most current graphics hardware vendors adopt a variation of VQ compression, termed S3TC [S3 98], that does not re-



(a) Sample Texture (b) Texture Tiles (c) Packed Tiles (d) Output Virtual Texture

**Figure 1:** Overview of our system. Given a sample texture (a), we first construct a set of texture tiles [CSHD03] (b) so that adjacent tiles have continuous pattern across shared edges. We pack the tiles into a single texture map (c) and store it on a graphics processor. This texture map can then be sampled and filtered via a fragment program to support an arbitrarily large virtual texture (d) without physical storage.

quire the on-chip storage of entire codebooks. Instead, the textures are compressed by encoding each 4 by 4 pixel tile into a 64 bit data chunk. Unfortunately, the compression ratio of S3TC is only about 6:1. [Fen03] developed a compression scheme that could achieve very low bit rates around 4 or 2 bits per texel. However, the compressed data size is still linearly proportional to the original texture size.

[KE02] proposed a technique to compress textures adaptively into blocks of different resolutions. Unlike Vector Quantization where the compressed data is linearly proportional to the original data size, their technique employees adaptive block size and may achieve higher compression ratio for sparse textures. However, the technique as presented does not fully support texture filtering and mipmapping.

A related idea to texture compression is texture caching on graphics hardware [HG97, TMJ98]. These techniques can help reduce texture bandwidth, but not texture memory consumption.

**Texture Synthesis** : Texture compression techniques as presented above are designed for general texture images. However, for a specific class of textures that are composed of repeating elements, texture synthesis algorithms can offer higher compression rates. Techniques such as [EF01, Wei02, ZG02, KSE*03] only require a small texture sample to generate arbitrarily large results, so in some sense the compression ratio can approach infinity for really large textures. Unfortunately, these techniques are often too slow or too complex for graphics hardware implementation. Instead, they would pre-compute large textures on CPUs and consume a comparably large amount of texture storage on graphics chips.

[PFH00, SCA02] proposed a variant of these approaches by synthesizing texture coordinates rather than texture pixels on mesh surfaces. Although the result mesh can be rendered using the original small texture sample, these approaches require the application programs to pre-compute texture coordinates. In addition the computation is only valid for a specific combination of texture and mesh model.

Procedural texturing algorithms [Per02, Har01] allow textures to be computed efficiently on GPUs, but can only reproduce a limited class of textures such as marble, wood, and clouds. [MGW01, WWT*03] presented variations of texture mapping on graphics hardware, but they did not address issues with large textures.

**Texture Tiling** : An alternative approach is to use texture synthesis to pre-compute a small set of tiles and use these tiles to generate a large, non-periodic texture at run time. [LN03] procedurally combines texture tiles via indirection to generate large virtual textures. The technique requires only a small amount of texture memory, but as the authors discussed, texture filtering and mipmapping are inherently incorrect with indirection. [CSHD03] introduced Wang Tiles as texturing primitive and discussed methods to construct these tiles, but did not address the implementation issues on graphics hardware.

**Computation on GPUs** : With the advance of recent programmable graphics hardwares (GPU), many techniques have been proposed to map general computation kernels on GPUs to solve different rendering problems [BFGS03, KW03, Har03]. Our approach also utilizes the programmability of GPUs, but we present a new algorithm to resolve an existing hardware problem, texture storage and bandwidth, rather than directly porting an existing rendering algorithm to run on GPUs.

## 3. Overview

Our goal is to design a new algorithm that combines the advantages of previous approaches. The algorithm should be general enough to handle textures with repeating patterns. It should be simple, efficient, and consumes minimum amount of texture memory for graphics hardware implementation. It should support texture filtering and mipmapping. It should also be non-intrusive to application programs and does not require modifications on input geometry or texture coordinates.

We attempt to achieve these goals by using Wang Tiles [CSHD03] to generate large virtual textures directly on GPUs. Wang Tiles are square tiles in which each tile edge is assigned a color. A valid tiling requires all shared edges between tiles to have matching colors. When a set of Wang Tiles are filled with texture patterns that are continuous across matching tile edges, a valid tiling from such a set can produce an arbitrarily large texture without pattern discontinuity. [CSHD03] discussed techniques to construct such tile sets from sample images, and presented a sequential algorithm for non-periodic tiling.

We present a new texture mapping algorithm based on Wang Tiles. Our algorithm assembles these tiles on the fly to build a large virtual texture rather than physical storage. As a preprocess step, we pack these tiles into a single texture map, and our packing scheme ensures correct mipmap filtering when a texel is fetched from this tile pack. During run time, for each texture request (s, t), we first determine which input tile it lands at based on the position of (s, t) within the output texture. We then compute the relative offset of (s, t) within that input tile, and fetch the corresponding texel from the input pack. Since out packing scheme supports correct mipmap filtering, the fetched input texel will appear to be the same as a texel fetched from a large, physically stored texture map. An overview of our system is illustrated in Figure 1.

Our contributions are as follows. We present a new tiling technique to allow random access at arbitrary tiles. We propose a new method to pack these tiles into texture memory to support mipmap filtering, and demonstrate an efficient implementation of our algorithm on graphics hardware.

## 4. Tile-Based Texture Mapping

The goal of texture mapping is to assign a texel Tex(s, t) for each screen pixel with texture coordinate (s, t). Current graphic hardware stores the entire texture in the memory hierarchy, and implements Tex(s, t) by fetching and filtering relevant texels. This approach works well for small textures that can fit into on-chip texture cache or off-chip texture memory, but has difficulty storing and accessing large textures that cannot fit into available texture memory.

We present an alternative approach to Tex(s, t), TileTex(s, t), that does not require large texture storage. TileTex(s, t) provides the same interface as traditional Tex(s, t), supporting filtering and mipmapping for large textures. Internally, TileTex(s, t) represents a large texture as a small set of Wang Tiles, and assembles the tiles into a large virtual texture on the fly for answering texture

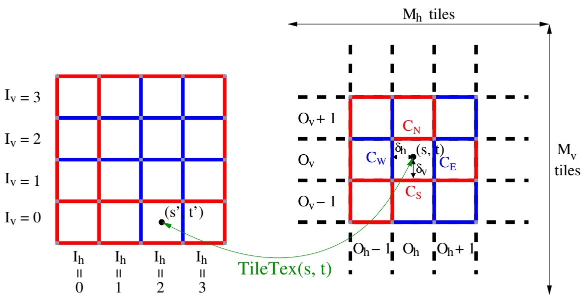



**Figure 2:** Graphical illustration of symbols used in our algorithm.

**Inputs**

**S**: the sample texture
$T_h \times T_v$ : the individual tile size in pixels
$K_h, K_v$ : number of horizontal and vertical edge colors
$M_h \times M_v$ : output texture size in # of tiles

**Preprocessing**

Construct the Wang tiles from **S** // *Section 4.1*
Pack the tiles into TilePack // *Section 4.4*
$(P_h, P_v) \leftarrow (K_v \times K_v, K_h \times K_h)$ // *TilePack size in # of tiles*
**if** need CornerPack
  Build a dual corner packing CornerPack // *Appendix B*
  // *CornerPack has size* $P_v \times P_h$

**Runtime** $u_f \leftarrow$ TileTex(s, t)

$(O_h, O_v) \leftarrow$ floor((s, t) $\times (M_h, M_v)$) $\%(M_h, M_v)$
$(C_S, C_E, C_N, C_W) \leftarrow$ EdgeHash$(O_h, O_v)$ // *Equation 2*
// *The above 2 steps compute TileMap(s, t), as described in Section 4.2.*
$(I_h, I_v) \leftarrow$ TileIndex$(C_S, C_E, C_N, C_W)$ // *Equation 7*
$(\delta_h, \delta_v) \leftarrow$ fraction((s, t) $\times (M_h, M_v)$)
$u_e \leftarrow \mathrm{Tex}(\frac{I_h+\delta_h}{P_h}, \frac{I_v+\delta_v}{P_v})$ // *texel fetch from TilePack*
**if** no CornerPack
  **return** $u_e$
**else**
  $(Q_h, Q_v) \leftarrow \mathrm{floor}\left((s, t) \times (M_h, M_v) - (\frac{1}{2}, \frac{1}{2})\right) \%(M_h, M_v)$
  $(C_B, C_R, C_T, C_L) \leftarrow$ CornerHash$(Q_h, Q_v)$ // *Equation 12*
  $(J_h, J_v) \leftarrow$ CornerIndex$(C_B, C_R, C_T, C_L)$ // *Equation 10*
  $(\varepsilon_h, \varepsilon_v) \leftarrow \mathrm{fraction}\left((s, t) \times (M_h, M_v) - (\frac{1}{2}, \frac{1}{2})\right) + (\frac{1}{2}, \frac{1}{2})$
  $u_c \leftarrow \mathrm{Tex}(\frac{J_h+\varepsilon_h}{P_v}, \frac{J_v+\varepsilon_v}{P_h})$ // *texel fetch from CornerPack*
  **return** Interpolate$(u_e, u_c)$

**Table 1:** Summary of our algorithm. The graphical illustration of some symbols is shown in Figure 2, and the Cg code implementation of **Runtime** is shown in Appendix C.

queries.

requests. Note that our algorithm is implemented entirely on the texture space and does not require applications to modify the input geometry or texture coordinates.

We now describe our tile-based texture mapping algorithm TileTex(s, t).

### 4.1. Texture Tile Construction from Input Sample

The inputs of our algorithm consist of a sample texture **S**, the individual tile size $T_h \times T_v$, the number of horizontal colors $K_h$ for south (S) and north (N) edges, and the number of vertical colors $K_v$ for east (E) and west (W) edges. These input parameters have the same meanings as in [CSHD03] and we directly adopt their approaches to build the Wang Tile set.

However, there remains an important difference between [CSHD03] and our approach in required number of tiles. [CSHD03] uses a sequential tiling method and requires only $K_h \times K_v \times 2$ tiles to support all possible NW color combinations for non-periodically tiling. Since our approach supports non-sequential, random tile access, we require ${K_h}^2 \times {K_v}^2$ tiles to support all possible edge color combinations. This is a disadvantage of our approach compared to [CSHD03]. But we haven't found it to be a major issue since in practice $K_h$ and $K_v$ are usually small (2 is sufficient for non-periodic tiling). In addition, a tile set with complete color combinations has further advantages such as the ease to support continuous packing for texture filtering, as we shall see later.

### 4.2. Texture Tile Mapping for Random Access

Once we build the tile set, we can tile them to generate arbitrarily large textures. To ensure continuous texture pattern, the tiling needs to be done so that all shared tile edges have identical colors. One method to achieve such a tiling is by putting the tiles sequentially from north to south and west to east, as shown in [CSHD03]. However, sequential tiling has a slight disadvantage that if only the SE tile is accessed, we still need to compute the entire tiling to make sure we end up with the same tile configuration at the SE corner. This can cause efficiency issues for graphics hardware implementation.

One possible solution is to pre-compute the tile mapping and store the results in an index texture map. However, the index texture can still be too large for a large output texture composed of small tiles.

Our approach is to compute the tile mapping on the fly rather than storing a pre-computed index map. For rendering consistency, we require our mapping computation to be *random-accessible*, meaning that the computation of each tile can be performed independently while ensuring all shared tile edges have identical colors. This avoids the sequential dependency problem and results in much faster computation. Our tile mapping computation, TileMap(s, t), is as follows:

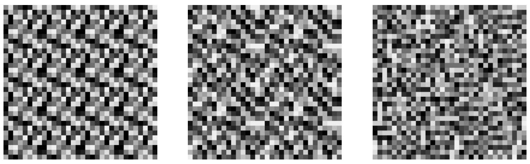
- For an input texture coordinate (s, t), compute which output tile it lands on. Assume the output texture contains $M_h$ horizontal tiles and $M_v$ tiles. Then the output tile index $(O_h, O_v)$ can be computed by:

$$(O_h, O_v) = \text{floor}((\text{s, t}) \times (M_h, M_v))\ \%(M_h, M_v) \quad (1)$$

  where % is modular operator. We use % for toroidal boundary handling but it can easily be extended for other boundary wrapping modes.
- Use $(O_h, O_v)$ to hash the 4 edge colors $C_S$, $C_E$, $C_N$, $C_W$, of the output tile, as follows.

$$\begin{array}{rcl} C_S & = & \text{hash}[\text{hash}[O_h] + O_v]\ \%K_h \\ C_E & = & \text{hash}[(O_h+1)\ \%M_h + \text{hash}[2\times O_v]]\ \%K_v \\ C_N & = & \text{hash}[\text{hash}[O_h] + (O_v+1)\ \%M_v]\ \%K_h \\ C_W & = & \text{hash}[O_h + \text{hash}[2\times O_v]]\ \%K_v \end{array} \quad (2)$$

  where hash[] is a 1D hash function which we currently implement via a permutation table as described in [Per02]. We have found it sufficient to use a hash[] table with $\max(M_h, M_v)$ entries. Since $M_h$ and $M_v$ are often small the caching and storage of hash[] is usually not an issue.



(a) 8 entries (a) 16 entries (a) 32 entries

**Figure 3:** Effect of hash table size on Equation 2. The image sizes are 32 × 32, and each pixel color encodes a unique combination of $C_S$, $C_E$, $C_N$, and $C_W$.

  Our mapping computation is inspired by a discrete chaos map called "Cat-Map" [XGS00]. In Cat-Map, a pair of integers $(x, y)$ is mapped to a new location $(x', y')$ by

$$\begin{array}{rcl} x' & = & x+y \\ y' & = & x+2\times y \end{array} \quad (3)$$

  Cat-Map helps to break diagonal symmetry and increase randomness in tile mapping computation. In the equations above, $C_S$ and $C_W$ are adopted directly from Cat-Map, and $C_N$ and $C_E$ are derived from $C_S$ and $C_W$ to preserve edge color consistency. For example, consider tile $(O_h, O_v)$ and the tile on the E side of it $(O_h+1, O_v)$. The computation above ensures that $C_E$ of $(O_h, O_v)$ is the same as $C_W$ of $(O_h+1, O_v)$. Similar arguments can be applied to the computation of $C_S$ and $C_N$.
- Select the input tile that has the corresponding 4 edge colors. Since our tile set has complete color combinations, this is always possible.

In summary, given a texel request at texture coordinate (s, t), the above steps compute TileMap(s, t), specifying which input tile (s, t) lands on. The input tile is designated by the edge colors $C_S$, $C_E$, $C_N$, and $C_W$.

### 4.3. Texture Tile Access and Filtering

After knowing which input tile $I(C_S, C_E, C_N, C_W)$ the texture request (s, t) locates on, we need to figure out the relative position of (s, t) inside the tile to fetch correct texels. This relative position $(\delta_h, \delta_v)$ can be computed by

$$(\delta_h, \delta_v) = \text{fraction}((\text{s, t}) \times (M_h, M_v)) \qquad (4)$$

where fraction() takes the fractional part of a floating point value. For nearest-neighbor texture sampling, the final texel value TileTex(s, t) will come from the texel nearest $(\delta_h, \delta_v)$ within the corresponding input tile. For other sampling modes such as bilinear or anisotropic filtering, the issue will be more complicated. If $(\delta_h, \delta_v)$ is sufficiently far away from the tile border, then it is adequate to perform the desired filtering operations within that tile. However, if $(\delta_h, \delta_v)$ is close to the border, then correct filtering will involve texels from adjacent tiles in the output texture. Since the output texture is virtual, this filtering operation cannot be directly performed on graphics hardware.

There are several possible way to support filtering when $(\delta_h, \delta_v)$ is close to the border. One possible solution is to set the texture filtering mode to be point sampling, and use a fragment shader to perform texture filtering. This is the most flexible method since we can access any texels across tile boundaries, but it can be slow since we need to request many texture samples (8 samples for trilinear mipmap filtering and possibly more for anisotropic filtering). Usually it is much more efficient to let the texture unit perform texture filtering since texture reads are relatively slow compared to arithmetic operations inside a fragment shader.

However, since hardware texture filtering is performed in the stored texture map, we have to find a way to store the set of Wang Tiles to support correct filtering. There are several possible ways to store the tiles. One solution is to store each tile as an individual texture. Since most hardware has a limited number of bind-able textures we can run out of texture IDs quickly. This approach also does not handle the tile boundary condition correctly. Another solution is to pack all the tiles into one single texture map. Assume the input packing has $P_h$ horizontal tiles and $P_v$ vertical tiles and the desired input tile is located as the $(I_h, I_v)^{th}$ tile in this packing, then we can fetch the correct texel for (s, t) with offset $(\delta_h, \delta_v)$ as follows:

$$\text{TileTex(s, t)} = \text{Tex}\left(\frac{I_h+\delta_h}{P_h}, \frac{I_v+\delta_v}{P_v}\right) \qquad (5)$$

The challenge of this scheme is to make sure that texture filtering across output tile boundaries can be performed as filtering in the input tile pack. To achieve this goal, we present a packing scheme that leverages existing hardware texture filtering and mipmapping capability, and describe how to efficiently compute $(I_h, I_v)$ from tile edge colors ($C_S$, $C_E$, $C_N$, $C_W$) in this packing scheme.

### 4.4. Texture Tile Packing and Corner Handling

In packing texture tiles into a single texture map we would like to achieve three goals. First, to utilize minimum amount of texture memory, each tile should appear only once in the packed texture. Second, to avoid filtering artifacts across texture tile borders, adjacent tiles in the packed texture should have matching edge colors. Third, the packing scheme should support efficient indexing. That is, given an edge color description of the input tile $I(C_S, C_E, C_N, C_W)$, the packing scheme should efficiently compute the horizontal and vertical indices $(I_h, I_v)$ of $I(C_S, C_E, C_N, C_W)$ inside the packed texture.

We present a new tile packing scheme, TilePack, that attempts to satisfy the above requirements. For clarify, we first describe how to pack tiles in 1D. We then describe how to extend it to 2D.

Assuming we have a set of Wang Tiles that has $K_v$ vertical edge colors but only 1 horizontal edge color. Our goal is to pack these tiles in a horizontal row so that adjacent tiles have matching vertical edge color (including the left-most and right-most tiles since the packed texture needs to be tileable). We encode all the vertical edge colors into integers in the range $[0..K_v - 1]$. Given a tile with west and east edge colors $(e_1, e_2)$, we use the following mapping to determine the horizontal position of that tile inside the packed 1D texture: (The thought process for deriving this formula can be found in Appendix A.)

$$\text{TileIndex}_{1D}(e_1, e_2) = \begin{cases} 0, & e_1 = e_2 = 0 \\ {e_1}^2 + 2 \times e_2 - 1, & e_1 > e_2 > 0 \\ 2 \times e_1 + {e_2}^2, & e_2 > e_1 \geq 0 \\ (e_1+1)^2 - 2, & e_1 = e_2 > 0 \\ (e_1+1)^2 - 1, & e_1 > e_2 = 0 \end{cases} \qquad (6)$$

$\text{TileIndex}_{1D}$ satisfies the three requirements listed above. It packs the $K_v \times K_v$ tiles into a horizontal 1D texture map without using any tile more than once. It ensures all shared tile edges have identical colors. In addition, it is efficient to compute, involving only multiplications, additions, and subtractions. An example of 1D packing with $K_v$ = 3 is shown in Figure 4.

We can implement 2D TileIndex by $\text{TileIndex}_{1D}$ on horizontal and vertical colors orthogonally, as shown in Equation 7. Given a tile with edge colors ($C_S$, $C_E$, $C_N$, $C_W$), the horizontal index $I_h$ of the tile is computed by $\text{TileIndex}_{1D}(C_W, C_E)$, while the vertical index $I_v$ of the tile is computed by $\text{TileIndex}_{1D}(C_S, C_N)$. Since $\text{TileIndex}_{1D}$ satisfies the three requirements in 1D, and $(I_h, I_v)$ are computed orthogonally, it can be easily seen that TileIndex meets the

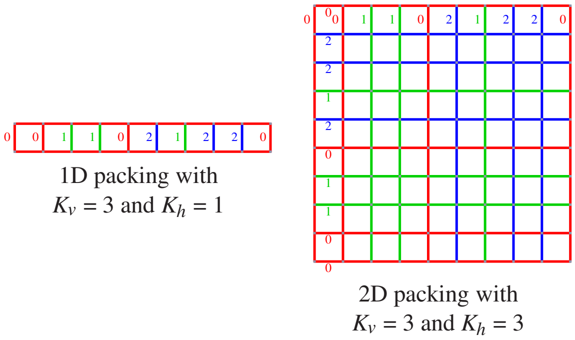



**Figure 4:** Tile packing examples. The illustration on the left shows a 1D tile packing with 3 vertical colors, and the one on the right shows a 2D tile packing with 3 vertical and 3 horizontal colors.

three requirements in 2D. An example of 2D packing with $K_h = 3$ and $K_v = 3$ is shown in Figure 4.

$$(I_h, I_v) = \text{TileIndex}(C_S, C_E, C_N, C_W) = \\ (\text{TileIndex}_{1D}(C_W, C_E), \text{TileIndex}_{1D}(C_S, C_N)) \qquad (7)$$

One of the primary advantages of our tile packing scheme is that it allows us to perform texture filtering directly via the hardware texturing unit. Since adjacent tiles in the packing share matching edge colors, there will be no pattern discontinuity when performing bilinear filtering across tile edges. However, bilinear filtering can be theoretically incorrect across tile corners. One solution to this problem is to simply ignore it. We have found that this corner artifact noticeable only when there is a distinctive object or pattern placed near a tile corner. For textures with uniform repetitive patterns, this is usually not an issue.

Another solution is to add an additional tile packing with all possible corner combinations. This solution would allow correct filtering across tile corners, but consumes extra amount of texture memory and shader computations. The derivation of this corner packing is analogous to the tile packing above, and we refer the reader to Appendix B for more details.

## 5. Results and Discussion

We have implemented our algorithm as a Cg fragment program [MGAK03] as shown in Appendix C. Our approach supports correct mipmap filtering across tile boundaries, as demonstrated in Figure 5. The performance of our implementation measured on a 350 MHz Geforce FX 5600 graphics processor [NVI03] is 2.7 million trilinearly-filtered texture samples per second without any hand optimized assembly code. This is slower than standard texture mapping which achieved at least 60 million texture samples per second on the same graphics card. However, the speed and storage requirement of our algorithm remain roughly constant regardless of the size of the output virtual texture. In comparison, the speed and storage requirement of traditional texture mapping degrade with increasing texture sizes, and most graphics chips impose an upper limit on the available texture size.[†]

At small texture sizes, our approach is slower than traditional texture mapping due to the run-time computation of TileMap(s, t) (Section 4.2), which would consume 8 accesses to the hash[] texture per request. For speedup, we have provided an alternative implementation that precomputes the TileMap(s, t) into a separate texture map. This approach has a better performance with more than 20 million trilinearly-filtered samples per second, with the cost of an extra memory for storing the TileMap(s, t). A better solution would be to utilize a built-in hash generator in the hardware, but unfortunately this feature is not yet available in current generation graphics chips. With such a hardware hash generator, we believe our first implementation would achieve comparable performance with respect to traditional texture mapping at small texture sizes.

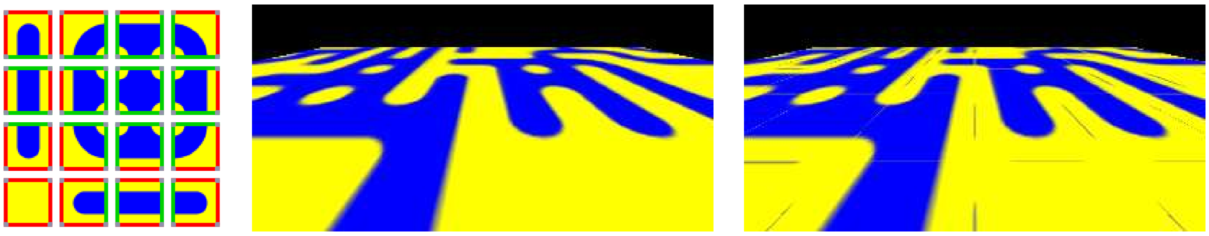
Another limitation of our approach is that our tile filtering scheme is theoretically incorrect at lower mipmap levels where the filtering can access texels from more than 2 tiles. In practice, this is usually not an issue for textures with repeating patterns since the patterns reduce to homogeneous void at lower resolutions anyway. In addition this is an inherent limitation for any tile-based texturing algorithms unless you are willing to precompute and store lower mipmap levels for the output virtual texture.

Our approach can be implemented as part of the driver and exposed as an extension of standard API calls. For example, the **Preprocessing** part in Table 1 can be implemented as a special case of glTexImage2D, and the **Runtime** part can be implemented by letting the driver replacing any texture access code to a tile-based texture map with our fragment program. The result would be a transparent driver implementation that allows application programmers to access our tile-based texture map just like a traditional texture map.

## 6. Conclusions and Future Work

We present a tile-based texture mapping algorithm that allows the creation of a large, non-periodic, virtual texture map from a small set of pre-computed tiles. We propose a novel packing scheme that allows correct texture filtering across tile boundaries, as well a run time algorithm for sampling individual texels on the virtual texture map that can be directly implemented as a fragment program on current generation graphics processors.

[†] The maximum texture size is 4K × 4K pixels on a Geforce FX 5600 GPU.



(a) Tiles (b) Result with correct packing (c) Result with incorrect packing

**Figure 5:** Filtering artifact example. When the input tile set (a) is correctly packed into a single texture map via our method, the output texture, when rendered with bilinear mipmap filtering, exhibits no border artifact as shown in (b). However, if the input tiles are not properly packed, the rendering result may exhibit noticeable boundary artifacts as shown in (c).

There are several possible directions for future work. Our hash operation in Equation 2 relies on an input hash table, which consumes extra memory and additional texture fetches. This limitation can be overcome by replacing the hash[] table in Equation 2 with a well behaved hash algorithm such as MD5 implemented either as a fragment program or as part of the graphics hardware. However, since most hash functions require bitwise operations, a fragment program implementation would require integer and bitwise arithmetic beyond the floating-point-only instructions in current generation graphics chips. Another potential future work is to extend our approach to 3D textures. This would involve more shader program computations to sample 3D tiles, but could provide even higher compression ratio compared to 2D textures.

**Acknowledgements**

I would like to thank Wei-Chao Chen for his insights and discussions on an early draft of this paper, the anonymous reviewers for their invaluable suggestions, and my NVIDIA colleagues for their support and encouragement.

**References**

[BAC96] BEERS A. C., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. In *Proceedings of SIGGRAPH 1996* (1996), pp. 373–378.

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖDER P.: Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Transactions on Graphics 22*, 3 (July 2003), 917–924.

[CSHD03] COHEN M. F., SHADE J., HILLER S., DEUSSEN O.: Wang tiles for image and texture generation. *ACM Transactions on Graphics 22*, 3 (July 2003), 287–294.

[EF01] EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. In *Proceedings of ACM SIGGRAPH 2001* (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 341–346.

[Fen03] FENNEY S.: Texture compression using low-frequency signal modulation. In *Graphics Hardware 2003* (July 2003), pp. 84–91.

[Har01] HART J. C.: Perlin noise pixel shaders. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001), pp. 87–94.

[Har03] HARRIS M.: General-purpose computation on gpus, 2003. http://www.gpgpu.org/.

[Hec86] HECKBERT P. S.: Survey of texture mapping. *IEEE Computer Graphics & Applications 6*, 11 (November 1986), 56–67.

[HG97] HAKURA Z. S., GUPTA A.: The design and analysis of a cache architecture for texture mapping. *24th International Symposium on Computer Architecture* (1997).

[KE02] KRAUS M., ERTL T.: Adaptive Texture Maps. In *Proc. SIGGRAPH/EG Graphics Hardware Workshop '02* (2002), pp. 7–15.

[KSE*03] KWATRA V., SCHÖDL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics 22*, 3 (July 2003), 277–286.

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics 22*, 3 (July 2003), 908–916.

[LN03] LEFEBVRE S., NEYRET F.: Pattern based procedural textures. In *Proceedings of SIGGRAPH 2003 Symposium on Interactive 3D Graphics* (2003).

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics 22*, 3 (July 2003), 896–907.

[MGW01] MALZBENDER T., GELB D., WOLTERS H.: Polynomial texture maps. In *Proceedings of ACM SIGGRAPH 2001* (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 519–528.

[NVI03] NVIDIA CORPORATION: GeForce FX Graphics Chip Series, 2003. http://www.nvidia.com/.

[Per02] PERLIN K.: Improving noise. *ACM Transactions on Graphics 21*, 3 (July 2002), 681–682.

[PFH00] PRAUN E., FINKELSTEIN A., HOPPE H.: Lapped textures. In *Proceedings of ACM SIGGRAPH 2000* (July

2000), Computer Graphics Proceedings, Annual Conference Series, pp. 465–470.

[S3 98] S3 CORPORATION: S3TC Texture Compression Standard, 1998.

[SCA02] SOLER C., CANI M.-P., ANGELIDIS A.: Hierarchical pattern mapping. *ACM Transactions on Graphics 21*, 3 (July 2002), 673–680.

[TMJ98] TANNER C. C., MIGDAL C. J., JONES M. T.: The clipmap: A virtual mipmap. In *Proceedings of SIGGRAPH 98* (July 1998), pp. 151–158.

[Wei02] WEI L.-Y.: *Texture Synthesis by Fixed Neighborhood Searching*. PhD thesis, Stanford University, 2002.

[WWT*03] WANG L., WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: View-dependent displacement mapping. *ACM Transactions on Graphics 22*, 3 (July 2003), 334–339.

[XGS00] XU Y.-Q., GUO B., SHUM H.: *Chaos Mosaic: Fast and Memory Efficient Texture Synthesis*. Tech. Rep. MSR-TR-2000-32, Microsoft Research, 2000.

[ZG02] ZELINKA S., GARLAND M.: Towards real-time texture synthesis with the jump map. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering* (June 2002), pp. 99–104.

## Appendix A: Derivation of Equation 6

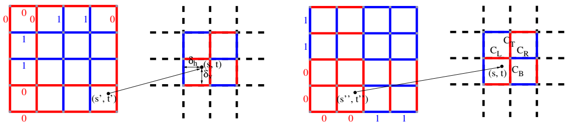
Assume we have $N+1$ tile edges labeled from 0 to $N$. Our goal is to connect them in a tour so that each pair of edge combination appears exactly once. (This is another way to state the requirements listed in Section 4.4.) Mathematically, we need to find the Euler circuit for a directed, complete graph whose nodes are labeled from 0 to $N$. Denote the Euler circuit of nodes 0 to $N$ as $[0 \rightleftharpoons N]$. Then, we can construct $[0 \rightleftharpoons N]$ from $[0 \rightleftharpoons (N-1)]$ via induction as follows.

$$\begin{aligned}&[0 \rightleftharpoons N] = \\ &[0 \rightleftharpoons (N-1)] \to N \to 1 \to N \to 2 \cdots N \to (N-1) \to N \to 0\end{aligned} \tag{8}$$

Let $F_{N+1}$ denote TileIndex$_{1D}$ with edges 0 to $N$. Essentially, $F_{N+1}(e_1, e_2)$ indicates the position of edge pair $e_1 \to e_2$ within $[0 \rightleftharpoons N]$. It can be derived by a similar induction process, as shown below.

$$F_{N+1}(e_1,e_2) = \begin{cases} F_N(e_1,e_2) & e_1 \leq N-1, e_2 \leq N-1 \\ {e_1}^2 + 2 \times e_2 - 1 & e_1 = N, 1 \leq e_2 \leq N-1 \\ 2 \times e_1 + {e_2}^2 & e_2 = N, 0 \leq e_1 \leq N-1 \\ (e_1+1)^2 - 2 & e_1 = e_2 = N \\ (e_1+1)^2 - 1 & e_1 = N, e_2 = 0 \end{cases} \tag{9}$$

Finally, Equation 6 can be derived by unrolling Equation 9 through successively smaller values of $N$. You can also verify the correctness of Equation 6 by plugging it into Equation 9.



(a) Original Tile Packing (b) Dual Corner Packing

**Figure 6:** Dual packing examples.

## Appendix B: Corner Handling

We derive the corner packing, CornerPack, by observing that CornerPack can be considered as the "dual" of TilePack. Since TilePack contains tiles with all possible edge color combinations via Equation 7, we can use the the same mechanism to pack corners with all possible edge color combinations. Let $C_B, C_R, C_T, C_L$ be the bottom, right, top, and left edge colors around a corner. We can pack the corners as follows.

$$\begin{aligned}(J_h, J_v) = \text{CornerIndex}(C_B, C_R, C_T, C_L) = \\ (\text{TileIndex}_{1D}(C_L, C_R), \text{TileIndex}_{1D}(C_B, C_T))\end{aligned} \tag{10}$$

The computation is very similar to Equation 7. The major difference is that in Equation 7, the vertical/horizontal tile locations are determined by horizontal/vertical tile edge colors, while in Equation 10 the vertical/horizontal tile locations are determined by vertical/horizontal corner edge colors. As a corollary, the height/width of CornerPack is the same as the width/height of TilePack. CornerPack also inherits all the desirable properties from TilePack: (1) each corner appears only once in CornerPack, (2) adjacent corners in CornerPack have matching edge colors, and (3) the computation is as efficient as TilePack. An example of this dual corner packing is shown in Figure 6.

The computations involved in accessing CornerPack is also very similar to the access of TilePack shown above, with some minor differences due to the fact that we are locating the nearest corner rather than the containing tile. The steps are as follows.

- Given a texel request (s, t), we first compute which corner $(Q_h, Q_v)$ is the nearest.

$$(Q_h, Q_v) = \text{floor}\Big((\text{s, t}) \times (M_h, M_v) \text{ - } (\tfrac{1}{2}, \tfrac{1}{2})\Big)\ \%(M_h, M_v) \tag{11}$$

  Note the similarity of the computation of $(Q_h, Q_v)$ with that of $(O_h, O_v)$ in Equation 1. The only difference is the shift by $(\frac{1}{2}, \frac{1}{2})$ which enables floor() to locate the nearest corner for (s, t).
- Use $(Q_h, Q_v)$ to hash the colors of the 4 edges at bottom, right, top, and left of the corner.

$$\begin{aligned} C_B &= \text{hash}[(Q_h+1)\ \%M_h + \text{hash}[2 \times Q_v]]\ \%K_v \\ C_R &= \text{hash}[\text{hash}[(Q_h+1)\ \%M_h] + (Q_v+1)\ \%M_v]\ \%K_h \\ C_T &= \text{hash}[(Q_h+1)\ \%M_h + \text{hash}[2 \times ((Q_v+1)\ \%M_v)]]\ \%K_v \\ C_L &= \text{hash}[\text{hash}[Q_h] + (Q_v+1)\ \%M_v]\ \%K_h \end{aligned} \tag{12}$$

This corner edge color computation is consistent with the tile edge computation in Equation 2 and is derived from it by proper changing of variables. For example, if (s, t) locates near the NE corner of a tile, then $C_B$ computed in Equation 12 will be the same as $C_E$ computed in Equation 2.

- For corner $(C_B, C_R, C_T, C_L)$, use Equation 10 to locate its index $(J_h, J_v)$ in CornerPack. Observe that $(J_h, J_v)$ also indicates the location of the tile in CornerPack which has $(C_B, C_R, C_T, C_L)$ as the NE corner.
- Derive the relative offset of (s, t) with tile $(J_h, J_v)$.

$$(\varepsilon_h, \varepsilon_v) = \text{fraction}\left((s, t) \times (M_h, M_v) - (\tfrac{1}{2}, \tfrac{1}{2})\right) + (\tfrac{1}{2}, \tfrac{1}{2}) \quad (13)$$

Note the similarity of this computation with the computation of $(\delta_h, \delta_v)$ in Equation 4. The only difference is the shift by $(\frac{1}{2}, \frac{1}{2})$ which enables us to compute the correct offset of (s, t) relative to the SW corner of tile $(J_h, J_v)$.

- Fetch the desired texel from CornerPack.

$$\text{Tex}\left(\frac{J_h+\varepsilon_h}{P_v}, \frac{J_v+\varepsilon_v}{P_h}\right) \quad (14)$$

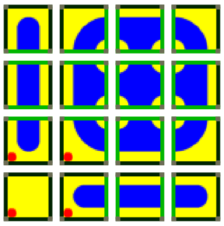
This value fetched from CornerPack is then interpolated with the texel fetched from TilePack to obtain the final texel value. The interpolation depends on how close is (s, t) to a tile corner.

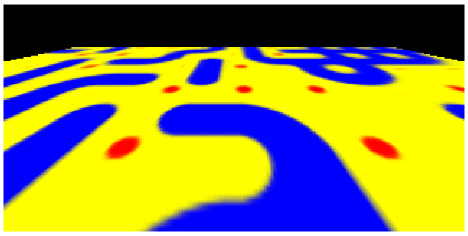
## Appendix C: The Cg Program of Our Algorithm

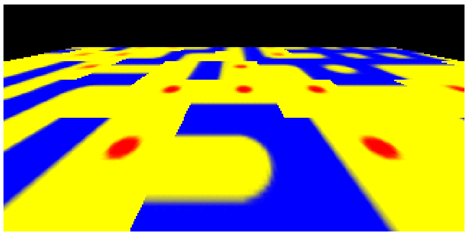
Some math symbols shown below are program constants as defined in Section 4. For simplicity, we have skipped the code for corner handling.

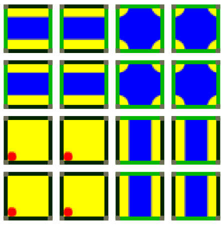
```
struct FragmentInput
{
  float4 tex : TEX0;
  float4 col : COL0;
};

struct FragmentOutput
{
  float4 col : COL;
};

float2 mod(const float2 a, const float2 b)
{
  return floor(frac(a/b)*b);
}

float EdgeOrdering(const float x, const float y)
{
  float result;
  if(x < y) result = (2*x + y*y);
  else if(x == y)
    if(x > 0) result = ((x+1)*(x+1) - 2);
    else result = 0;
  else
    if(y > 0) result = (x*x + 2*y - 1);
    else result = ((x+1)*(x+1) - 1);
  return result;
}

float2 TileLocation(const float4 e)
{
  float2 result;
  result.x = EdgeOrdering(e.w, e.y);
  result.y = EdgeOrdering(e.x, e.z);
  return result;
}

float4 Hash(uniform samplerRECT hashTexture,
    const float4 input)
{
  return texRECT(hashTexture,
      frac(input.xy/HASH_SIZE) * HASH_SIZE);
}

FragmentOutput fragment(const FragmentInput input,
    uniform sampler2D tilesTexture,
    uniform sampler2D cornersTexture,
    uniform samplerRECT hashTexture)
{
  FragmentOutput output;
  float2 mappingScale = float2(M_h, M_v);
  float2 mappingAddress = input.tex.xy * mappingScale;
  float4 numColors = float4(K_h, K_v, K_h, K_v);
  float2 thisVirtualTile = mod(mappingAddress, mappingScale);
  float2 nextVirtualTile = thisVirtualTile.xy + float2(1, 1);
  nextVirtualTile = frac(nextVirtualTile/mappingScale)*mappingScale;
  float4 edgeColors;
  edgeColors.x = Hash(hashTexture,
      Hash(hashTexture, thisVirtualTile.x) + thisVirtualTile.y);
  edgeColors.y = Hash(hashTexture,
      nextVirtualTile.x + Hash(hashTexture, 2*thisVirtualTile.y));
  edgeColors.z = Hash(hashTexture,
      Hash(hashTexture, thisVirtualTile.x) + nextVirtualTile.y);
  edgeColors.w = Hash(hashTexture,
      thisVirtualTile.x + Hash(hashTexture, 2*thisVirtualTile.y));
  edgeColors = frac(edgeColors/numColors)*numColors;
  float2 inputTile = TileLocation(edgeColors);
  float2 tileScale = float2(P_h, P_v);
  float2 tileScaledTex = input.tex.xy * float2(M_h/P_h, M_v/P_v);
  output.col = tex2D(tilesTexture,
      (inputTile.xy + frac(mappingAddress))/tileScale,
      ddx(tileScaledTex), ddy(tileScaledTex));
  return output;
}
```

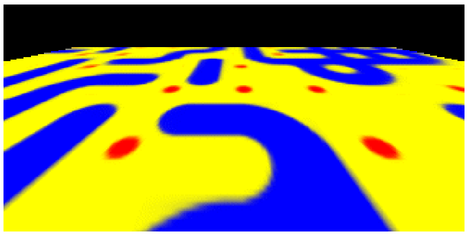
Notice the use of ddx and ddy in the last tex2D() call. This is necessary to avoid spuriously large texture coordinate derivatives near tile boundaries which can cause low resolution mipmap levels to be accessed and produce over-blurry results.


(a) Tile packing


(b) Result with only tile packing


(c) Result with only corner packing


(d) Corner packing


(e) Result with proper blending from (b) and (c)

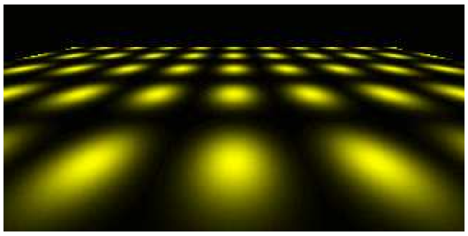
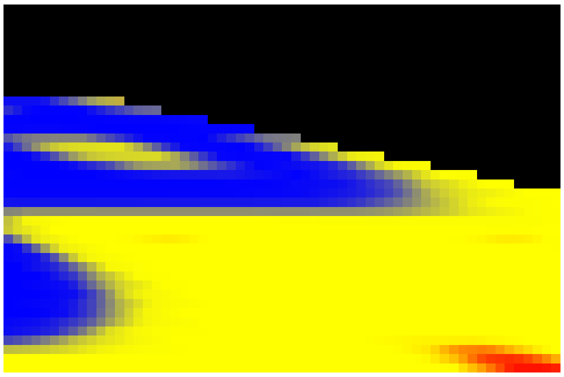
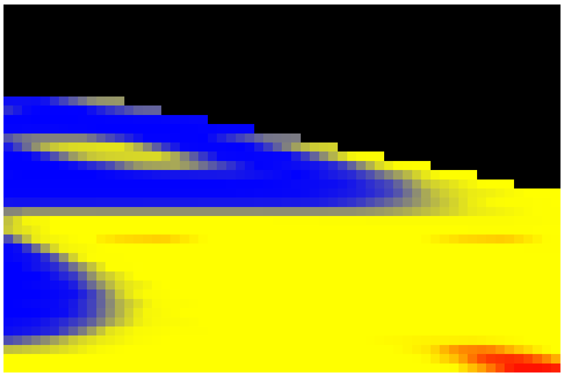
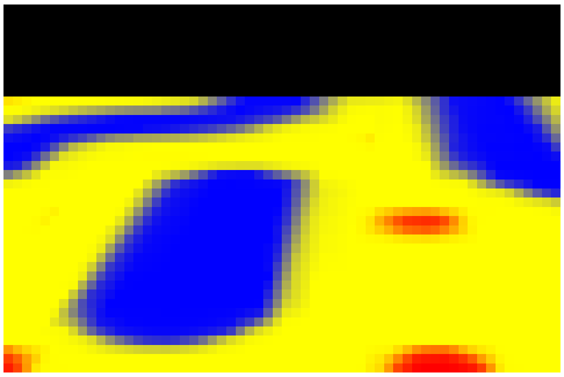

(f) Blending weight

**Plate 1:** Corner artifact example. The input tile set (a) contains 4 corner features shown as red dots. The results shown in (e) and (b) are rendering with and without corner handling, respectively. Note that result (b) contains noticeable errors around the upper right region of the polygon, while in result (e) the errors are much diminished. For reference, the result produced from only corner packing is shown in (c), and the blending weight is shown in (f).


Detailed view of the upper-right region of (e)


Detailed view of the upper-right region of (b)


Detailed view of the upper-middle region of (e)

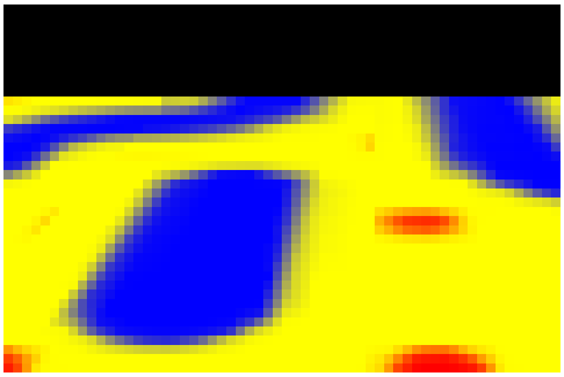

Detailed view of the upper-middle region of (b)

**Plate 2:** Detailed views of corner artifacts. Images on the right have more severe ghosting red lines, compared to images on the left.