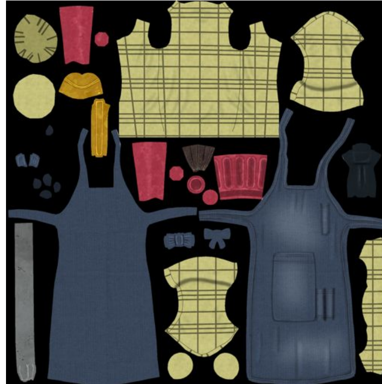
A Space-efficient and Hardware-friendly Implementation of Ptex

Sujeong Kim*
University of North Carolina at Chapel Hill
Department of Computer Science

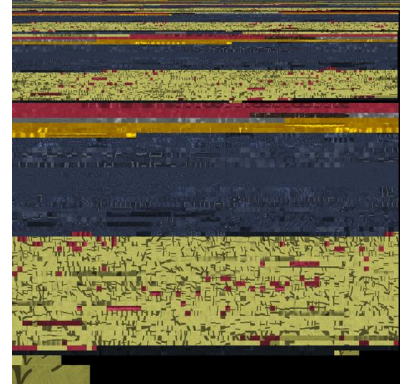
Karl Hillesland†
Advanced Micro Devices, Inc.

Justin Hensley‡
Advanced Micro Devices, Inc.

(a) Rendered asset with texture data



(b) Texture data stored as atlas



(c) Texture data stored as packed Ptex data

Figure 1: *1(a) shows a single Ptex asset rendered in our real-time graphics engine. 1(b) shows the texture data stored as a texture atlas, and 1(c) shows the same base texture data stored as a packed Ptex texture. Only 63% of the texels in the texture atlas contain actual color data, while 93% of texels in the packed Ptex texture contain data. Additionally, the packed Ptex texture contains all the mip-map levels instead of just a single level. The packed Ptex texture needs only 10% extra texels added to accelerate bilinear texture filtering. This rather modest increase in size can be eliminated at the cost of manually performing interpolation in shader.*

Abstract

We introduce a method to pack Ptex per-face texture data that is both space-efficient and hardware-friendly. Recently presented real-time implementations of Ptex have been wasteful with space and required a storage cost many times higher than the size of the original texture data. Our method packs multiple levels of Ptex data together, and requires only around 8% increase in storage for our test textures. Additionally, because of efficient data packing, our method wastes less space than a typical texture atlas, which requires buffer regions to be added between the separate charts within the texture.

CR Categories: I.3.7 [Computer Graphics]: Three-dimensional Graphics and Realism—Texture;

Keywords: real-time graphics, texturing

1 Introduction and Background

Texture mapping is an integral part of real-time graphics applications, and in a typical application such as a video game, each art asset will have one or more UV-sets paired with the texture data. Unfortunately, handling the UV-sets introduces several complications. First, the actual creation of the UV-map is an art unto itself because most automatic generation methods do not create maps with high enough quality. Often, an artist will have to modify the generated UV-maps by hand, which is a labor-intensive task, to get the desired look. Additionally, small local changes to a UV-map can cause drastic, non-local effects on rendered images. For instance, increasing the effective sampling density in one region of the UV-map will necessarily affect the sampling density in the rest of the UV-map. Finally, seams in the UV-map add additional complexity to UV-map creation when using tessellation due to the difficulty in preventing cracks.

A texture atlas is a common way to pack a model's surface data into a single texture; an example is shown in Image 1(b). As can be seen in this image, there are a large number of unused texels - in this texture, only 63% of the texels store actual surface data. Additionally, colors can bleed from one piece to another in the down-sampled mip-map levels when the pieces are too close together in the atlas, and a texture atlas does not give the artist fine control over the creation of the mip-map.

Ptex [Burley and Lacewell 2008], a UV-less per-face texturing method, solves many of the issues with traditional texture mapping, and is becoming widely used in the film industry. The need to assign UV-maps is completely eliminated with Ptex, and resolution can be varied independently for each face.

The basic idea of Ptex is to store a small texture, including its mip-map chain, for each face of the model along with the adjacency information for each face. Using the adjacency information, seamless interpolation can be handled at run time by searching through the neighboring faces when necessary. Because multiple digital content creation applications natively support Ptex, it is an attractive technique to modify for real time use because it can be integrated into existing art pipelines relatively easily.

There are a couple of complicating factors to implementing a real-time version of Ptex. Ptex's native per-face texture data is not very GPU-friendly because the built-in interpolation hardware cannot be used across the edges of the faces. Additionally, directly storing

*e-mail:sujeong@cs.unc.edu
†e-mail:karl.hillesland@amd.com
‡e-mail:justin.hensley@amd.com

the texture blocks in separate textures is impractical due to current graphics APIs, which limit the number of textures that can be bound at any one time. Recently, a real-time implementation of Ptex was presented [McDonald and Burley 2010]. Unfortunately, this method used a wasteful packing strategy that increased the storage cost of the textures many times over their original size.

We introduce a packing strategy for Ptex data that is both space-efficient and hardware-friendly. Using our method, we found that Ptex ends up being more efficient than a typical texture atlas, while remaining efficient to render in real time. For our test textures, we found that our packing increases the size of the texture by approximately 8%. An additional 10% was added for the borders to enable hardware bilinear interpolation. This compares with a texture atlas that increased the texture by approximately 58%. While our method does not require DirectX 11-class hardware, it does easily enable an art pipeline to bring subdivision-based art assets to real-time applications on hardware that supports domain and hull shaders.

2 Implementation

To pack the Ptex data, faces are sorted by maximum resolution of their textures. To enable hardware bilinear filtering, a border of one pixel is added to each block. The border data is then copied from the neighboring blocks using the adjacency information provided by Ptex. For a given resolution, the per-face texture blocks are packed in row-major order, starting from the left edge of the packed texture for each resolution. Each mipmap level of a face's texture is stored with its respective resolution, not with its respective face. However, the ordering within each resolution slice is still according to maximum resolution sorting.

Given this face ordering, we can generate a small table that tells us the starting index for faces that contain the corresponding maximum resolution or higher. We can also generate a table indexed by resolution that tells us the starting row in the packed texture for each resolution. These two tables, called nRes and nRow respectively, are used in the pixel shader to clamp to available resolutions and compute texture coordinates within the packed texture. Because we need to compute the LOD per pixel, this step must be done in the pixel shader, and cannot be done in the domain shader. Code Listing 1 shows how to use nRes and nRow for the simple case of square face texture resolutions and nearest-mip selection. It is trival to extend the method to non-square textures and trilinear filtering.

Listing 1: *HLSL to sample packed data*

```
// UV is parameterization within face [0,1]
int log2 = ComputeMipLevel(UV);

// Clamp to highest resolution available.
while(log2>= 0 && nRes[log2] > faceID)
  --log2;

// Assuming square face textures for illustration
int size = (1 << log2) + 2;

// Index of face within this resolution.
// The subtraction accounts for faces that
// Do not have mips at this resolution.
int iFaceInRes = faceID - nRes[log2];

// Integer division to get number of faces
// that fit in each row of the packed texture.
int nFacesPerRow = textureWidth / size;

// Building texel index of face within slice.
int2 iUV = int2(
  (iFaceInRes % nFacesPerRow),
  (iFaceInRes / nFacesPerRow));

// Offset for border.
iUV += int2(1,1)

// Add row offset for the selected resolution.
iUV += int2(0, nRow[log2]);

// Scale to face size.
UV = UV * float2(size)*float2(size, size);

// Offset by location in packed texture.
UV = UV + float2(iUV);

// Scale to [0,1] for hardware bilinear filtering.
tPTex.Sample(sampler,
             UV / float2(textureWidth, textureHeight))
```

3 Texture Compression

Texture compression [Iourcha et al. 1999] is vital for real-time applications such as games. There are several ways to handle texture compression with our packed format. The simplest method is to directly compress the packed textures using standard compressors such as DXT1 or DXT5. Assuming that the Ptex per-face textures are created with power-of-two sizes, the added border pixels will interfere with the blocking that the texture-compression algorithm uses and will give non-ideal results. This is because the packed texture data can place unrelated chunks of data adjacent to each other, which will cause color-bleeding in the compressed data. This issue can happen in texture atlases as well if there is not enough buffer space between the pieces of the texture, but it is much less of an issue. One possible solution to this problem is to add a slightly larger border so the packed per-face textures are always a multiple of 4x4 in size. This allows compression on complete blocks without inadvertently pulling in data from nearby per-face textures. This increases the size of the packed data, but in our example texture in Image 1(b), the added data would still be less than the wasted space incurred by the texture atlas. Another possibility is to use slightly smaller, non-power-of-two blocks when originally creating the per-face textures. Once the buffer data is added to enable hardware-accelerated bilinear filtering, per-face textures will naturally match the compressor block size.

4 Future Work

One disadvantage of method for packing is that hardware trilinear texture filtering cannot be enabled. We are currently exploring methods that would allow us to enable hardware trilinear filtering. Additionally, we are exploring combining Ptex with something akin to pelting to limit the amount of extra ghost data that needs to be added because the per-face textures from ordinary vertices can be placed next to each other in the packed texture map.

References

BURLEY, B., AND LACEWELL, D. 2008. Ptex: Per-face texture mapping for production rendering. In *Eurographics Symposium on Rendering 2008*, 1155–1164.

IOURCHA, K., NAYAK, K., AND HONG, Z. 1999. System and method for fixed-rate block-based image compression with inferred pixels values. In *US Patent 5,956,431*.

MCDONALD, J., AND BURLEY, B. 2010. Per-face texture mapping for real-time rendering. SIGGRAPH 2010 Talk.