II 2

Real-Time Ptex and Vector Displacement

Karl Hillesland

2.1 Introduction

A fundamental texture authoring problem is that it's difficult to unwrap a mesh with arbitrary topology onto a continuous 2D rectangular texture domain. Meshes are broken into pieces that are unwrapped into "charts" and packed into a rectangular texture domain as an "atlas" as shown in Figure 2.4(b). Artists spend time setting up UVs to minimize distortion and wasted space in the texture when they should ideally be focusing on the actual painting and modeling.

Another problem is that edges of each chart in the atlas introduce seam artifacts. This seam problem becomes much worse when the texture is a displacement map used for hardware tessellation, as any discrepancy manifests as a crack in the surface.

This chapter describes an implicit texture parametrization system to solve these problems that we call *packed Ptex*. It builds on the Ptex method developed by Disney Animation Studios for production rendering [Burley and Lacewell 08]. Ptex associates a small independent texture map with each face of the mesh. Each texture map has its own mip chain. In the original Ptex method, adjacency information is used for filtering across the edge of one face texture and into the next.

There are two main advantages of Ptex relative to conventional texture atlasing. First, there is no need for explicit UV. Second, there are no seaming issues arising from unwrapping a complete mesh of arbitrary topology onto a single-texture domain. These are the two main advantages of the original Ptex method that we preserve in our adaptation.

The drawbacks of packed Ptex relative to conventional texture atlasing are additional runtime computation, additional texture filtering expense, and changes in asset production. The main change in asset production is that our method currently targets meshes consisting of quads. There can either be additional memory cost or savings relative to conventional texture atlasing methods depending on the particular circumstances.

Although this approach works for many texture types, it works particularly well for vector displacement mapping. The lack of seam issues is particularly valuable for this application, while many of the drawbacks of the approach are irrelevant.

There are two barriers to real-time performance in the original Ptex method. First, it's typically not practical to have an individual texture for each primitive. Second, the indirection required when a filter kernel crosses from one face to another is costly in performance, and precludes the use of any hardware texture filtering. The next section describes the offline process to address these issues. Then we follow up with how to use this at runtime and some details related to displacement mapping. We finish by discussing the tradeoffs of this method as well as some possible alternatives.

2.2 Packed Ptex

To reduce the number of textures required, we pack all face textures and their mip chains into a single texture atlas (Figure 2.1). The atlas is divided into blocks of the same resolution. Within the block, the face textures are packed one after another in rows. Because the atlas width generally is not a multiple of the face-texture width, there will be unused texels at the end of each row. There will be additional empty space at the end of the last row, because it generally will not be filled to capacity.

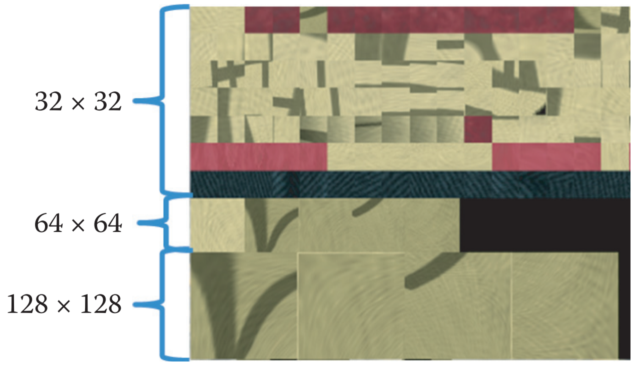


Figure 2.1. This is a portion of a packed Ptex atlas. There are four faces that have 128 × 128 base (level 0) resolution and one with 64 × 64 base resolution. The block of 64 × 64 contains both the one level 0 for the 64 × 64 face texture, and the four level 1 mips from the 128 × 128 face textures.

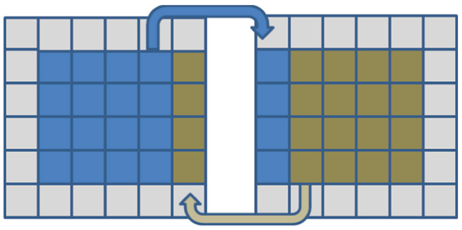


Figure 2.2. Faces that are adjacent in model space are not generally adjacent in texture space. A filter kernel that spills over the edge of a face must pick up texels from the adjacent face, which will generally be somewhere else in the texture atlas. We copy border texels from adjacent face textures to handle this case.

Just as in the original Ptex system, each face texture has its own mip chain. We sort the faces by their base (level 0) resolution to create the packing we describe here, and for runtime as described in Section 2.3.2. Since we are including face textures with different base resolutions, a given block will include different mip levels (Figure 2.1).

2.2.1 Borders for Filtering

Texture filtering hardware assumes that neighbors in texture space are also neighbors in model space. Generally, this is not true either for conventional texture atlasing methods nor for Ptex. It's the reason conventional texture atlasing methods often come with seam artifacts.

For our method, we copy texels from the border of a neighboring face to solve this problem (Figure 2.2). That way, there will be data available when the texture filter crosses the edge of the face texture. The padding on each side will be equal to at least half the filter width. This is a common solution to the problem, particularly for situations like tile-based textures for terrain. However, the memory overhead for this solution is generally much higher for Ptex than for conventional texture atlasing methods. This is one disadvantage in using Ptex; anisotropic filtering quickly becomes too expensive in terms of memory cost.

2.2.2 Texture Compression

Current GPUs have hardware support for texture compression. The compression relies on coherency within 4 × 4 texel blocks. For this reason, it is best not to have a 4 × 4 block span face textures. We have already discussed adding a single-texel border to support hardware bilinear filtering. To get good results with compression, we add an additional border to get to a multiple of 4 × 4. Generally, this means two-texel borders for compressed textures.

2.3 Runtime Implementation

In the original Ptex system, texture lookups were done by first finding which face you are in and then finding where you are within the face. The real-time version essentially starts with the same steps, but with an additional mapping into the texture atlas space. This section walks through each of these steps in detail. For trilinear filtered lookups, the basic outline is the following:

1. Select the texture level of detail (LOD) (Section 2.3.1).
2. Compute the location within atlas for each LOD level and perform a hardware, bilinear lookup for each (Section 2.3.2).
3. Lerp in the shader for a final trilinear value.

For nonfiltered lookups, the sequence is easier; all that's required is to find the location in the atlas and do a single lookup. We will discuss the first two steps in detail.

2.3.1 Texture LOD Selection

The first step in a trilinear filtered, packed Ptex lookup is to determine which resolution of face texture is desired. In conventional hardware trilinear filtering, this is done for you automatically by the GPU. However, hardware trilinear filtering assumes the derivative of texture space with respect to screen space is continuous everywhere. This is not the case for a texture atlas in general, although it's often "good enough" for conventional texture atlasing with some tweaking. However, tiled textures like real-time Ptex often require manual texture LOD selection. The code for this is given in Listing 2.1.

2.3.2 Packed Ptex Lookup

Once we know which resolution we want, we clamp it to the maximum resolution available for that face (i.e., mip level 0). Table 2.1 demonstrates how to look up the maximum resolution for a face texture without having to resort to any per-face information. The method uses a sorted ordering according to face texture resolution and prefix sums.

The next step is to find the location of the resolution block within the atlas. This is possible by lookup into a table indexed by resolution.

The sorted ordering and prefix sum are used again to find the index of the face within the block. In general, not all faces will have a representation in the resolution block, as some face-texture base resolutions will be higher than others. Again, Table 2.1 describes the procedure.

We can find the face texture origin within the block using the index of the face within the resolution block. If the texture width is W and the face texture

```
float ComputeLOD( float2 vUV, float nMipLevels )
{
    float2 vDx = ddx(vUV);
    float2 vDy = ddy(vUV);

    // Compute du and dv magnitude across quad
    float2 vDCoords;
    vDCoords = vDx * vDx;
    vDCoords += vDy * vDy;

    // Standard mip mapping uses max here
    float fMaxTexCoordDelta = max(vDCoords.x, vDCoords.y);
    float fMipLevelPower;

    if (fMaxTexCoordDelta == 0)
        fMipLevelPower = nMipLevels - 1;
    else
    {
        // 0.5 is for the square root
        fMipLevelPower = 0.5 * log2(1.0 / fMaxTexCoordDelta);
    }

    float mipLevel = clamp(fMipLevelPower, 0, nMipLevels - 1);
    return nMipLevels - 1 - mipLevel;
}
```

Listing 2.1. Texture LOD Selection. Allowing for nonsquare textures simply requires a scale by aspect ratio on one of the directions.

width including borders is w, then the number of faces in a row is $n = \lfloor W/w \rfloor$. Using i as the index within the block, we can compute the row as $\lfloor i/n \rfloor$ and the column as $i \,\%\, n$.

Each face has its own implicit UV parametrization. We adopt a convention with respect to the order of the vertices in the quad. For example, we choose the first index to be (0,0), the next is (1,0) and the last as (0,1). These can be assigned in the hull-shader stage. The pixel shader will receive the interpolated coordinate, which we call the "face UV." We also need the primitive ID, which is also defined in the hull-shader stage.

Max Resolution	Face Count	Prefix Sum
16×16	5	5
32×32	5	10
64×64	3	13

Table 2.1. If faces are sorted by resolution, and you have the prefix sum of face count for each resolution bin, you can look up the resolution for any given face from the index in the sorting. In this example, a face of index 7 would have a maximum resolution of 32×32 because it is greater than 5 and less than 10. If we want the index of that face within that bin, it is $7 - 5 = 2$.

```
float2 ComputeUV(
    uint faceID, // From SV_PrimitiveID
    float2 faceUV, // Position within the face
    uint nLog2, // Log2 of the resolution we want
    int texWidth, // Atlas texture width
    int resOffset, // Prefix sum for this resolution
    int rowOffset, // Start of resolution block in atlas
    int borderSize ) // Texel thickness of border on each face
{
    // Here we assume a square aspect ratio.
    // A non-square aspect would simply scale the height
    // relative to width accordingly.
    float faceWidth = 1 << nLog2;
    float faceHeight = faceWidth;
    float borderedFaceWidth = faceWidth + 2*borderSize;
    float borderedFaceHeight = borderedFaceWidth;

    int nFacesEachRow = (int)texWidth / (int)borderedFaceWidth;
    int iFaceWithinBlock = faceID - resOffset;

    float2 faceOrigin = float2(
        (iFaceWithinBlock % nFacesEachRow) * borderedFaceWidth,
        (iFaceWithinBlock / nFacesEachRow) * borderedFaceHeight
        + rowOffset );


    // Take face UV into account.
    // Still in texel units, but generally not
    // an integer value for bilinear filtering purposes.
    float2 uv = float2(faceWidth, faceHeight) * faceUV;
    uv += float2(nBorderSize, nBorderSize);
    uv += faceOrigin;

    // Finally scale by texture width and height to get
    // value in [0,1].
    return float2(uv) / float2(texWidth, texHeight);
}
```

Listing 2.2. Go from face UV to atlas UV.

Scale and offsets are applied to get the face UV range of [0,1] mapped into the atlas UV, including an offset to get to the right resolution block and another to put the face texture origin (0,0) inside the face border. Listing 2.2 details the process of computing a UV within the packed Ptex atlas.

The last steps are to do the bilinear filtered lookup for each LOD we need, and the final trilinear lerp between them.

2.3.3 Resolution Discrepancies

There are discrepancies in resolution that translate to discontinuities when approaching a polygon edge from either side. This is illustrated in Figure 2.3. This can happen when the gradient used for mip selection changes as the edge of a polygon is crossed. However, this is not particular to packed Ptex and is fur-

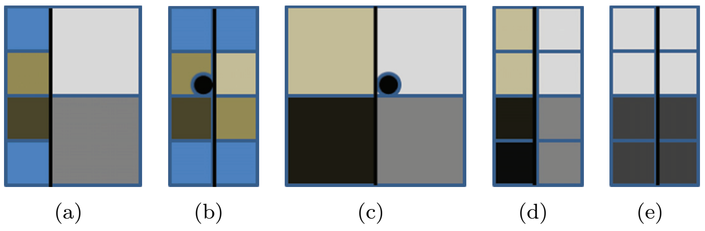


Figure 2.3. (a) Resolution discrepency (b) Bilinear lookup into border from view of left face. (c) Bilinear lookup into border from view of right face. (d) Changing the border on the left face to resolve resolution discrepancy by downsampling. (e) The solution for displacement when values must match exactly is to replicate at the lower resolution.

ther mitigated by the final lerp of trilinear filtering. In cases when we have used packed Ptex with trilinear filtering we have not seen any problems yet; therefore we have not pursued more sophisticated solutions.

The second cause for resolution discrepancy is capping to different resolutions due to different maximum (mip level 0) resolutions. The problem of max resolution discrepancy is mitigated by effectively clamping borders to the lower resolution when padding (Figure 2.3).

2.4 Adding Displacement

Displacement mapping adds geometric detail to a coarser mesh. Each polygon of the coarse mesh is tessellated further at runtime, and the vertices are displaced according to values stored in a displacement texture map. Displacement mapping provides a method of geometric LOD. The model can be rendered without the displacement for the lowest poly model, and different tessellations can be applied for higher quality models.

In classical displacement mapping, there is just a single scalar per texel. However, we have pursued vector displacement, which uses a 3D vector to specify displacement. This technique is much more expressive, but at greater memory cost on a per texel basis.

Authoring displacement maps in a conventional texture atlas without cracks can be quite difficult. If a shirt, for example, is unwrapped onto a texture, the edges where the charts meet on the model must match in value at every location. This is why games typically only apply displacement maps to flat objects like terrain, and why even Pixar's RenderMan, which is considered well engineered for displacement mapping, still performs a messy procedural crack-fill step during rendering [Apodaca and Gritz 99]. By contrast, you can export Ptex vector displacement maps from Autodesk Mudbox, and apply them without the need for manual fixup or runtime crack patching.

For displacement maps, we treat the borders and corners a little differently than described in Section 2.2.1. First of all, we do not need an extra border for filtering, as we use point sampling. However, adjacent faces must have identical values along their shared border to avoid cracks when using hardware tessellation. So instead of copying in a border from an adjacent face, we change the original borders by averaging them as shown in Figure 2.3.

Corners of a face texture correspond to a vertex in the model. Similar to how we handle borders, we walk the mesh around the vertex, gathering all corner values and average them. This value is then written back to all corners that share this vertex so that they are consistent and do not produce cracks. Note that it's necessary that this value is exactly the same. If you recompute this average for each quad, remember you are using floating-point math, and therefore must accumulate in the same order for each quad.

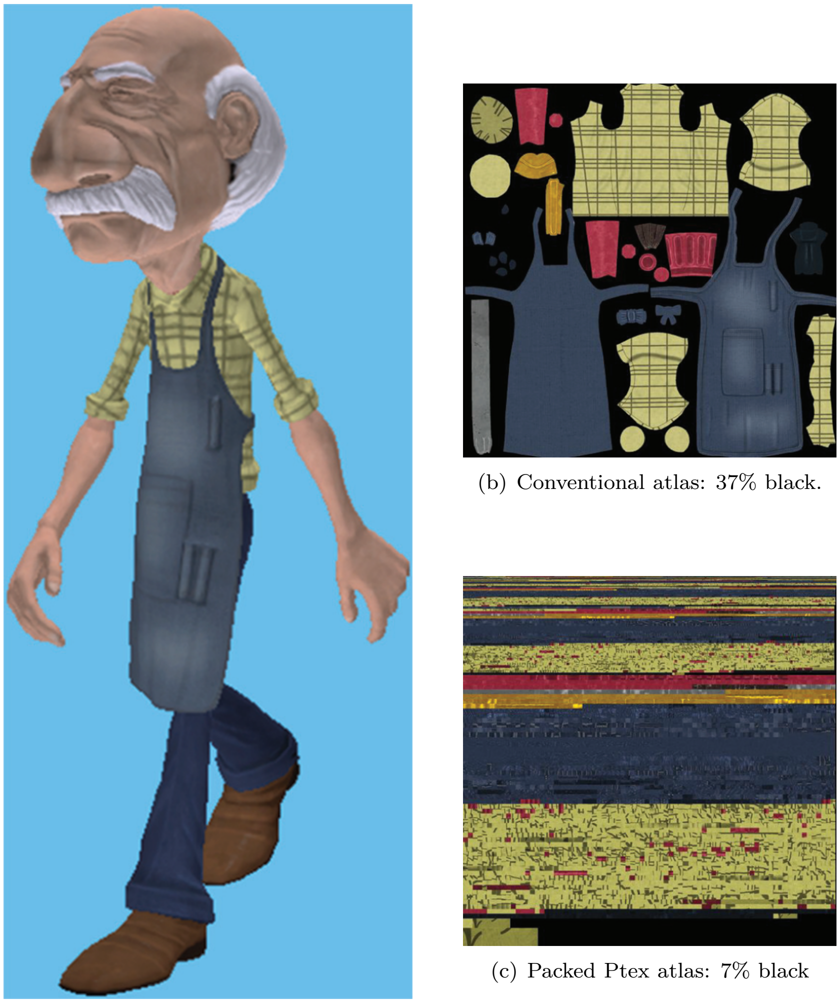
Displacement mapping is done in object space in the domain shader. In our implementation, we point sample from the highest resolution displacement map regardless of tessellation level. Because we are not filtering, the filter-related issues of packed Ptex are not relevant, and there is both less compute and bandwidth cost than for the typical texture map application in a pixel shader.

2.5 Performance Costs

To give a general idea of the cost difference between packed Ptex and conventional texturing, we measured the difference between a packed-Ptex and a conventionally textured version of the character shown in Figure 2.4(a). The AO, specular, albedo, normal and displacement maps are packed Ptex. GPU render time is 3.6 ms on an AMD Radeon HD 7970. If we change the AO, specular, albedo and normal maps to conventional texture lookups (all but displacement) we find the time goes down by an average of 0.6 ms.

The main cost is the search for maximum resolution in this implementation, for which there are plenty of opportunities for optimization we have not yet explored. We could, for example, move the computation as far up as the hull constant shader. There is also a cost due to reduced texture cache efficiency, as packed Ptex will generally not have as good locality of reference relative to conventional texturing. The entire UV computation was repeated for each texture, which should also not generally be necessary in practice.

Given the difficulty in authoring a valid displacement map for a model like the character in Figure 2.4(a) we measured packed Ptex displacement mapping against no displacement mapping at all. This model is made from 5,504 quads and tessellated up to 99,072 triangles. The cost of vector displacement with packed Ptex on this model is 0.14 ms. This is with 16-bit floats for each component, which is on the high end of what should normally be necessary in practice.



(b) Conventional atlas: 37% black.

(c) Packed Ptex atlas: 7% black

(a)

Figure 2.4. All textures for the model in (a) are in the packed Ptex format: Albedo, AO, specular, normal, and vector displacement: (b) an example of a conventional texture map and (c) an equivalent packed Ptex texture map.

We have a second model, shown in Figure 2.5, that uses packed Ptex only for vector displacement. It has a total of 86,954 quads in the base mesh and is animated with both skinning and blend shapes. When tessellated up to 1.6 million triangles, the cost of displacement lookup is 2.7 ms out of a total of 14.2 ms with our current implementation.

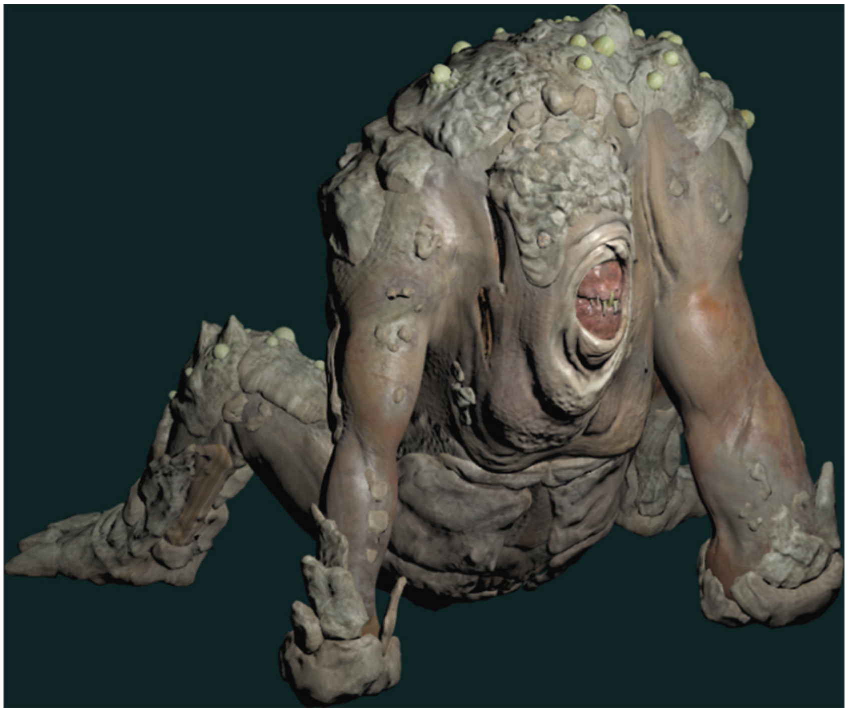


Figure 2.5. Model with vector displacement.

2.6 Memory Costs

Conventional texture atlases are often difficult to author without wasting texture space between the charts. In Figure 2.4(b) we see a typical example where 37% of the space is wasted. Ptex, by contrast, is built completely from rectangular pieces and is therefore much easier to pack into a rectangular domain. We make no special effort to find the optimal resolution to pack the face textures into, and yet the waste in our experience has been only around 7% (Figure 2.4(c)).

The greater memory overhead for packed Ptex is in the use of borders. Fundamentally, the border cost goes up proportional to the square root of the area. Here we give some formulas and concrete numbers to give an idea of what the overhead is. Each face of square resolution r and border size n_B wastes $(2n_B + r)^2 - r^2$ texels. Table 2.2 shows example costs as a percentage of waste due to borders in packed Ptex. Two items are worth mentioning here. First, we can see more concretely how high the per-face resolution should be to keep memory overhead down. Second, we also see why borders beyond a couple texels, as would be required for anisotropic filtering, is too expensive in memory cost.

Resolution	Border Size 1	2	4
4×4	56/56%	75/75%	89/89%
8×8	36/41%	56/62%	75/80%
16×16	21/24%	36/41%	56/62%
32×32	11/13%	21/23%	36/40%
64×64	6.0/6.4%	11/12%	21/23%
128×128	3.1/3.2%	6.0/6.2%	11/12%

Table 2.2. This table shows memory overhead for borders. The first percentage in each pair is for a single resolution, and the second is for mip chains down to 4×4. These values should be weighed against the waste inherent in a conventional texture atlas, such as the 37% illustrated in Figure 2.4(b).

2.7 Alternatives and Future Work

One way to avoid having a separate texture per face is to put each per-face texture in its own texture array slice [McDonald and Burley 11, McDonald 12]. This simplifies the texture addressing to some extent. However, there are limitations in the number of texture array slices, and resolutions cannot be mixed within a single texture array. Therefore, what would be a single texture in the conventional or packed Ptex approach would be split into multiple textures, one for each resolution, with further splitting as required for texture array limits. The amount of texture data used in the shader does not increase, excepting perhaps due to alignment or other per-texture costs, but the amount of conditional reads is significantly higher.

Rather than computing per-face texture information, we could store it in a resource indexed by face ID, and possibly by mip level [McDonald and Burley 11, McDonald 12].

Ptex takes the extreme approach of assigning an individual texture map to each primitive. The paper by B. Purnomo, et al. describes similar solutions to what is described here, but they group multiple primitives into rectangular patches in texture space for packing and handling seams [Purnomo et al. 04]. This reduces the overhead for borders, which would make larger filter kernels feasible. A next step might be to integrate some of the ideas from that paper.

2.8 Conclusion

Packed Ptex enables the main advantages of the original Ptex method while enabling real-time use. Authoring effort is saved first by eliminating the need for explicit UV assignment and second by naturally avoiding seaming issues that normally arise when trying to unwrap a 3D surface into at 2D rectangular domain. It does, however, require modeling with quads in its current implementation.

Packed Ptex also incurs higher runtime cost than conventional texture mapping. Memory costs can actually be lower relative to conventional texturing, depending primarily on the per-face texture resolution, filter kernel width, and the savings relative to the waste inherent with conventional texture atlases. Although packed Ptex can be applied to many different texture types, the most promising is probably displacement mapping, where the relative overhead is lower and the benefit of seamlessness is greatest.

2.9 Acknowledgments

The techniques described here were developed in collaboration with Sujeong Kim and Justin Hensley. Tim Heath, Abe Wiley, Exigent and Zoic helped on the art side. This work was done under the management and support of Jason Yang and David Hoff. Sujeong Kim, Takahiro Harada and Christopher Oat all provided valuable feedback in writing this article.

Bibliography

[Apodaca and Gritz 99] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*, First edition. San Francisco: Morgan Kaufmann Publishers Inc., 1999.

[Burley and Lacewell 08] Brent Burley and Dylan Lacewell. “Ptex: Per-Face Texture Mapping for Production Rendering.” In *Proceedings of the Nineteenth Eurographics conference on Rendering*, pp. 1155–1164. Aire-la-Ville, Switzerland: Eurographics Association, 2008.

[McDonald 12] John McDonald. “Practical Ptex for Games.” *Game Developer Magazine* 19:1 (2012), 39–44.

[McDonald and Burley 11] John McDonald, Jr and Brent Burley. “Per-face Texture Mapping for Real-Time Rendering.” In *ACM SIGGRAPH 2011 Talks*, article no. 10. New York: ACM, 2011.

[Purnomo et al. 04] Budirijanto Purnomo, Jonathan D. Cohen, and Subodh Kumar. “Seamless Texture Atlases.” In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pp. 65–74. New York: ACM, 2004.