Eurographics Symposium on Point-Based Graphics (2005)
M. Pauly, M. Zwicker (Editors)

Conversion of Point-Sampled Models to Textured Meshes

Martin Wicke Sandro Olibet Markus Gross

ETH Zurich

Abstract

We present an algorithm to convert point-sampled objects to textured meshes. The output mesh carries the geometric information present in the input model, while information about color and other surface attributes is separated and stored in textures. The point cloud is triangulated and decimated so it adequately represents the object geometry. Using EWA splatting, we compute textures patches for all triangles in the mesh. In an iterative process, the size of the texture patches is chosen adaptively such that texture information is preserved during the conversion. The texture filtering capabilities of EWA splatting ensure that no texture aliasing occurs. Finally, the texture patches are compiled into a texture atlas. Aside from colors, other surface attributes can be treated similarly. Normal maps can be computed to allow for further simplification of the output mesh while maintaining high visual quality.

1. Introduction

Point-based surfaces are receiving growing attention from the computer graphics community. Objects represented by point-samples are often created by scanning real-world objects [LPC*00, RHHL02]. Avoiding triangulation, various tools were developed to clean [WPK*04] and process [PG01] these models.

Point-sampled surfaces can be rendered in high quality using EWA splatting [ZPBG02], or raytracing [AA03]. The Pointshop3D system [ZPKG02] offers a large set of tools to create and edit point-based objects, including free-form deformation and texturing. Adams et al. [AWD*04] present a system for painting on point-sampled surfaces.

Recently, methods for physically-based simulation of point-sampled objects have been proposed [MKN*04, PKA*05].

However complete, the set of editing and processing tools for point-sampled surfaces pales to insignificance when compared to the vast pool of mesh based tools and methods. Commercial products like 3D Studio Max, Maya or mental ray work on triangle meshes, and do not natively support point-sampled surfaces.

Without appropriate tools for conversion, deciding for point-sampled representation of an object is a one-way street. An object's geometry can be converted to a mesh by triangulation. However, due to the dense sampling of the surface with point-samples, the resulting mesh is of unnecessary size. This effect is aggravated since point samples also carry appearance attributes and are used to represent textures. Thus, a high-frequency texture on a geometrically simple point model will lead to a huge triangulation.

When working with meshes, the appearance of the object is usually separated from the geometry by the use of textures for surface attributes such as colors. Various approaches have been proposed that simplify meshes while separating and storing the texture information present in the original mesh. However, these approaches are designed for meshes and do not consider sampling properties of point-based surfaces, represented with attributes like surfel radii. Varying sampling density, one of the main advantages of point-sampled representations, further complicates the generation of textures for converted objects.

We present an algorithm that converts a point-sampled object to a textured mesh. We obtain an output mesh of reasonable size by triangulating the point cloud and simplifying the resulting mesh. Using the original point cloud, texture patches are computed for each triangle in the output mesh. In an iterative process, the patch size for each triangle is chosen such that texture information present in the point-sampled original is preserved during the conversion. The texture patches are compiled into a texture atlas. Other surface attributes, such as normals, displacements or extended reflectance properties can be treated similarly. The conversion process is guided by a geometry error controlling the simplification and a texture error controlling the texture size.

2. Related Work

Pfister et al. introduced *layered depth cube* (LDC) sampling as a method to convert other representations to point-sampled objects [PZvBG00]. The naïve approach to conversion of point-sampled objects to textured meshes is using triangulation and subsequent texture-preserving simplification (see below). To our best knowledge, no work has been published that specifically deals with converting point-sampled objects to textured meshes.

The first step in our conversion algorithm is the triangulation of the input point cloud. We use the *Cocone* algorithms described in [ACDL02, DG03], which are based on global delaunay triangulation. Other approaches to surface reconstruction use implicit surfaces as an intermediate representation [HDD*92], or use local measures to find a triangulation [BMR*99].

A lot of research has been devoted to the problem of mesh decimation [SZL92, Hop96, GH97]. In this context, methods for texture preservation have also been proposed.

Maruya [Mar95] implemented a texture preserving mesh simplification method. He modified [SZL92] to retain color information, which is then stored in a texture. The color information is interpolated linearly, hence the resulting texture is prone to aliasing. Soucy et al. [SGR96] propose a similar method, but use only nearest-neighbor sampling to compute the output texture. The triangular texture patches are scaled and sheared to half-squares of size 2^n. These half-squares are then used to build the rectangular texture atlas.

Cignoni et al. [CMSR98] have proposed a general method for computing textures for simplified meshes, without requiring any knowledge on the simplification process. However, the computed textures are not sensitive to the input texture detail. Only simple texture filtering using supersampling is implemented. Texture packing avoids scaling of the triangles, while shearing is still allowed.

Hale [Hal98] proposes a different projection method, avoiding discontinuity artifacts resulting from extreme simplification. His texture packing method can handle arbitrary triangles and thus avoids stretching or shearing of texture patches.

These simplification methods are designed for meshes and do not take into account the sampling properties of the surfaces. Varying sampling densities will lead to texture aliasing and loss of detail.

The remainder of this paper is organized as follows: Section 3 gives an overview of the algorithm, and describes its first two steps. The main part of the algorithm, the texture generation, is treated in Section 4. The properties of the algorithm are discussed in Section 5 before we present results (Section 6) and conclude.

3. Algorithm Overview

Input to our algorithm is a point cloud P, with each sample carrying a number of attributes like color and normals, or more exotic surface properties. Such a surface sample is called a *surfel*. From this input, we create a triangle mesh which represents the object geometry, and one or more texture atlases containing the surface properties.

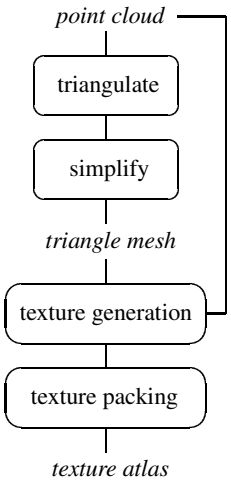

Figure 1: *Algorithm overview. The input geometry is given as a point cloud. First, the point cloud is triangulated and simplified. Then, a texture patch is computed for each of the triangles in the simplified mesh by splatting the input points onto the triangle. Finally, these texture patches are compiled into one texture atlas.*

Our algorithm consists of four stages. In a first step, the point-sampled object is triangulated. This leads to very small triangles. Therefore, the mesh is decimated in a second step. The resulting triangle mesh adequately represents the object's geometry. The core of the algorithm is the generation of a texture for each of the triangles in the mesh. These textures are created by splatting the point samples onto the triangle plane. We use EWA splatting [ZPBG02] for texture filtering in order to avoid sampling artifacts. Finally, a texture atlas is compiled from the individual texture patches. Figure 1 illustrates the process.

The remainder of this section will give more details on the triangulation and simplification steps, while the texture generation is described in Section 4.

3.1. Triangulation

We use the Cocone and Tight Cocone algorithms [ACDL02, DG03] for triangulation. Tight Cocone always generates a watertight triangulation, while the Cocone algorithm can be used to triangulate non-manifold surfaces. The triangulations produced by these algorithms are generally of good guality. In the presence of noise, a less susceptible triangulation method, for example Robust Cocone [DG04], can be used.

Since the sampling density on the point-sampled model depends on both texture and geometry, the resolution of the output mesh also depends on texture detail.

3.2. Simplification

We aim to separate textures and geometry, hence the mesh is simplified in a second step. This ensures that the mesh resolution adequately represents the object geometry, and is not influenced by the object texture. We use Garland and Heckbert's method [GH97] to simplify the mesh.

We only allow pair contractions along edges of the mesh. This eliminates the possibility of connecting previously unconnected parts of the mesh, which would cause problems in the texture generation procedure.

Without changing the texture generation algorithm, the triangulation and simplification steps can be replaced with a different surface reconstruction method that is insensitive to texture information present in the set of samples, for instance using an implicit function as intermediate representation [HDD*92].

4. Texture Generation

The main part of our algorithm deals with the generation of textures for the simplified mesh. We break the problem down to individual triangles, and create a texture patch for each triangle in the mesh. Since many small textures are not practical, we pack them into one or more texture atlases that can be used in rendering or further processing.

The patch rendering algorithm uses iterative refinement to adapt the patch size to the detail present in the original model. An error metric controls the texture size.

4.1. Patch Rendering

We use EWA splatting [ZPBG01] to render the texture patches. In addition to color information, this technique requires that each surface sample either carries a normal and a radius in case of circular splats, or tangent axes for elliptical splats. If these additional attributes are absent, they can be estimated from the neighborhood of the sample, for instance using covariance analysis [PGK02, Pau03].

The viewing transformation for rendering is set up such that the triangle T lies in the image plane, with its longest side parallel to the screen-space x-axis. We use an orthogonal projection to project the surfels onto the triangle plane. Hidden surface removal is performed using visibility splatting [PZvBG00]. During splatting, those surfels whose projected splat ellipses intersect with the triangle and are not discarded by the hidden surface removal, are added to a set S_T.

The viewport size determines the resulting patch size. Starting from the smallest possible patch size, we iteratively enlarge the viewport until an error function E drops below a user-defined threshold. The set S_T contains all surfels that contribute to the texture. E measures the difference between the color function reconstructed from these irregular point samples, $c_{S_T}(x,y)$, and the piecewise linear function represented by the texture, $t(x,y)$.

$$\begin{aligned} E &= \int_{(x,y)\in T} e\left(c_{S_T}(x,y), t(x,y)\right) \mathrm{d}x\mathrm{d}y \\ &\approx \sum_{s\in S_T} A_s \cdot e\left(c_s, t(p_s)\right) \end{aligned} \tag{1}$$

The integral is approximated numerically by computing a sum over all surfels in S_T, summing up local errors at the projected surfel positions p_s, weighted with the area of the surfel, A_s. The local error $e(\cdot,\cdot)$ is some metric for colors. We use the metric induced by the L^2 norm. Note that in general, $c_{S_T}(p_x, p_y) \neq c_s$ for a surfel s with projected coordinates p_x and p_y. This is because the reconstruction of the color function using EWA splatting is not interpolating. Therefore, there can be pathological cases in which the error never drops below the user threshold, independent of the resolution.

The iteration is terminated once the texture error is below a threshold, or when a maximum resolution is reached. The maximum pixel spacing s is a function of the minimum distance between two surfels in S_T. Given a function that is sampled at the surfel positions, we can safely resample it on a regular grid with spacing no larger than s, independent of the grid orientation.

$$s = \frac{1}{2\sqrt{2}} d_{\min} \tag{2}$$

We can thus exit the refinement loop once the pixel spacing falls below s. Note that this only defines an upper bound on the texture resolution. Even if $d_{\min}$ is small, a small texture can be sufficient if the surfel color is constant. The same holds if small surfels are used to represent a linear color gradient. Most heuristics based on surfel count, surfel size or color variance fail in these cases.

Upon completion of the rendering stage, we extract the triangular texture patch from the framebuffer by rasterizing the triangle. In order to avoid artifacts resulting from linear texture interpolation over the triangle border, we leave a one pixel boundary to all sides. The resulting texture patch is stored and later packed into a texture atlas.

EWA Splatting can be used to interpolate any surfel attribute, such as normals and depths. These data can be used for normal mapping or displacement mapping respectively. The procedure remains largely unchanged, however, a suitable local error function $e(\cdot,\cdot)$ needs to be found for other attributes. For normals, we use $e(n_s, n_t) = 1 - \langle n_s, n_t \rangle$.

4.2. Texture Packing

Once all texture patches are available, we compile them into rectangular textures. We use the texture packing algorithm described in [Hal98] with some minor modifications. It packs triangular patches into bigger rectangular textures. In the following, the modified algorithm is summarized.

The longest edge of the input triangles is always parallel to the x-axis of the texture, with the remaining vertex above

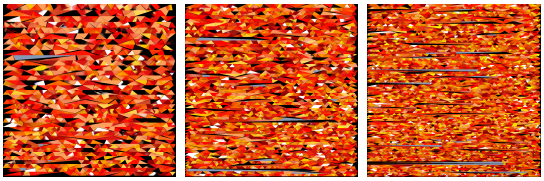

Figure 2: *Result of the texture packing algorithm for the dragon model. The texture was split into several $2^n \times 2^m$ pieces. 89% of the texture are used.*

the longest edge. We will call this edge the *base* of the triangle. The left and right angles adjacent to this edge are called *base angles* α_l and α_r respectively.

The algorithm creates rows consisting of triangles of similar height. To add triangles into a row, the best fitting triangle within the height range of that row is chosen. Every second triangle is mirrored horizontally, such that its third vertex is below the base. The quality of a triangle with respect to a row is determined by comparing the base angles of the new triangle with the free base angle β of the last triangle in the row. The triangle that minimizes $min(|\alpha_l - \beta|, |\alpha_r - \beta|)$ is inserted into this row. It is aligned with the bottom or top border of the current row and mirrored vertically if $|\alpha_l - \beta| > |\alpha_r - \beta|$. Then, the triangle is pushed as far to the left as possible. New rows are started if no suitable triangle can be found or if a row is full.

A pseudocode version of the algorithm is given in Appendix A. Figure 2 shows the result of this texture packing algorithm. Typically, 85% to 90% of the texture space is used. Texture usage tends to be better if more and smaller texture patches are available.

5. Discussion

The proposed method produces high-quality meshes and textures. As long as the geometry error during the simplification steps is kept reasonable, the converted objects are visually indistinguishable from the point-sampled originals. Only under large magnifications, differences become visible.

Older approaches have used nearest neighbor interpolation [SGR96] or linear interpolation [Mar95] to compute texel values. No texture filtering is performed. Both [Mar95] and [SGR96] determine patch sizes using the number of vertices projected onto a triangle. This heuristic can result in undersampling in regions of varying sampling density. If a surface is densely sampled in a uniform color, or densely sampled to represent a linear color gradient, determining the patch size based on the number of vertices leads to large textures where only little information is present on the surface. Our adaptive refinement works around these problems and guarantuees an adequate sampling in all cases.

The resulting texture atlas is tightly packed and does not introduce distortion artifacts due to scaling or shearing of patches. However, it is not well suited for manual editing,

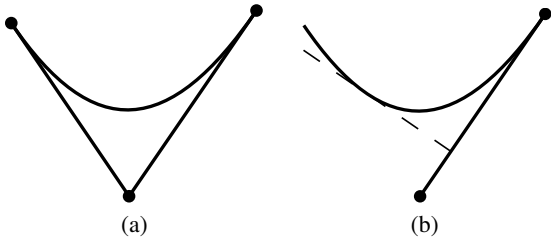

Figure 3: *Cross-section of a saddle point. (a) Original geometry and simplified mesh around a saddle point. (b) No texture is available for the lower part of the Triangle, the texture in the upper part is distorted.*

e.g. in a image manipulation program, since adjacent triangles do not have neighboring texture patches in the atlas.

It is also not possible to use standard mip-mapping for texture simplification without introducing severe artifacts. Custom tailored mip-mapping can be performed by not only dumping the texture patches at the computed optimal resolution, but also at half and quarter resolutions. The texture packing only needs to be carried out once, the smaller resolution patches are then assembled in the same pattern as the original resolution.

The triangulation of the input model is by far the most time-consuming task. Since the texture generation does not assume anything about how the mesh was acquired, it is possible to substitute any surface reconstruction algorithm for the triangulation and simplification steps. An adaptive version of [HDD*92] would be a suitable candidate.

5.1. Limitations

Due to the different interpolation schemes used for textures and point samples, bilinear interpolation and EWA splatting, respectively, the reconstructed color functions look slightly different.

Note that a crucial prerequisite to the texture generation is that the mesh adequately represents the geometry of the input point cloud. Under extreme simplification, textures become distorted and the resulting mesh can even contain triangles that cannot be fully textured using the method described herein. In these cases, the orthogonal projection of the object surfels does not entirely cover the triangle area. These triangles typically lie around points of negative Gaussian curvature (see Figure 3).

If the surface deviates from the mesh at the mesh edges, the orthogonal projections either ignore or repeat parts of the surface. Figure 4 (a) illustrates the problem. This can cause discontinuity artifacts when the mesh is highly simplified.

[Hal98] shows how the projection can be adapted to gracefully handle these cases. He interpolates the vertex normals over the area of each triangle in order to find a projection normal for each point on the triangle. Applying the interpolated normals projection to our approach, each surfel has to be splatted using its own projection normal. This

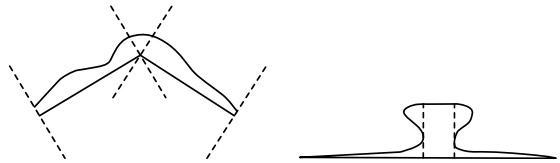

Figure 4: *(a) Using orthogonal projection of adjacent faces ignores or repeats parts of the original surface. (b) A mushroom-shaped geometric detail is discarded by the simplification algorithm. The resulting texture might show discontinuities along the contours of the mushroom.*

method also greatly alleviates distortion artifacts, however, finding the correct projection normal for a surfel is a non-trivial optimization problem.

Another class of artifacts is introduced by surface patches with depth complexity greater than one. Figure 4 (b) shows an example. If a small part of the geometry is entirely discarded by the simplification, the rendering will result in discontinuities along the contours. A remedy to this class of artifacts is to use a parameterization of the original, as done in [Mar95, SGR96]. It might be possible to modify a method like [Mar95] to use EWA splatting for texture filtering. However, surfels have finite extent and cannot be attributed to only one triangle. A method like this requires triangulation, lacking the flexibility to change the mesh generation method.

6. Results

The conversion algorithm was implemented as a Pointshop3D plugin. If desired, the intermediate result after each conversion step can be inspected and the step can be repeated with different parameters to manually fine-tune the conversion. However, the only parameters necessary for the conversion are an error bound for the mesh simplification and a error threshold for the texture generation.

Figures 5 and 6 show painted point-sampled models and corresponding meshes generated using our algorithm.

As can be seen in Figure 5 (c), fine detail present in the original model texture is preserved during the conversion. Since the patch size is computed for each triangle individually, regions with low texture resolution in the input point cloud only take up little space in the output texture. In this example, the texture patch for the triangle containing the bee is 5228 pixels large. The patch of an adjacent, blue triangle which is of similar size occupies only 6 pixels.

Figure 6 (c) and (d) show surface sampling and triangles for a part of the bunny model.

Figure 7 shows the effect of simplification on the result. It is hard to visually distinguish between the original point cloud and the output of our conversion algorithm using reasonable simplification parameters. Only after further simplification of the mesh, simplification artifacts become visible.

Model	#Points	#Tris	Textures	Times [s]
Igea	134345	11340	2.3MB/93%	258/15/35
Bunny	349989	10000	3.8MB/86%	511/28/104
Dragon	553619	30000	12.1MB/89%	907/37/444

Table 1: *Statistics for converted models. The data shown are: number of points in the input model, number of triangles after simplification, number and size of (color) textures, times for triangulation/simplification/texture generation (in seconds).*

Table 1 lists statistics for some models. Shown are the number of points, the number of triangles in the output mesh, the size and fill rate of the (uncompressed) output texture, and the conversion times.

7. Conclusion

We have presented an algorithm to convert point-sampled models to textured meshes. The generated textures adaptively capture the surface detail present in the point-sampled object. Using EWA splatting, texture aliasing can be avoided entirely. The resulting textured meshes can then be used for further processing with mesh-based tools. Point-based editing and mesh editing are no longer entirely separate pipelines. Artwork in the form of point-sampled models can be used in mesh-based programs.

Future research will focus on adapting the interpolated normals projection to splatting in order to improve the quality of texture generation for highly simplified models.

Acknowledgements

We would like to thank Tamal Dey and Joachim Giessen for allowing us to use their implementation of the Cocone and Tight Cocone algorithms.

References

[AA03] ADAMSON A., ALEXA M.: Ray Tracing Point Set Surfaces. In *Proceedings of Shape Modeling Intl.* (2003), pp. 272–279.

[ACDL02] AMENTA N., CHOI S., DEY T., LEEKHA N.: A Simple Algorithm for Homeomorphic Surface Reconstruction. *Intl. J. Comp. Geom. and Appl. 12* (2002), 125–141.

[AWD*04] ADAMS B., WICKE M., DUTRÉ P., GROSS M., PAULY M., TESCHNER M.: Interactive 3D Painting on Point-Sampled Objects. In *Proceedings of Eurographics Symp. on Point-Based Graphics* (2004), pp. 57–66.

[BMR*99] BERNARDINI F., MITTLEMAN J., RUSHMEIER H., SILVA C., TAUBIN G.: The Ball-Pivoting Algorithm for Surface Reconstruction. *IEEE Trans. Vis. and Comp. Graphics 5*, 4 (1999), 349–359.

[CMSR98] CIGNONI P., MONTANI C., SCOPIGNO R., ROCCHINI C.: A General Method for Preserving Attribute Values on Simplified Meshes. In *Proceedings of Visualization '98* (1998), pp. 59–66.

[DG03] DEY T., GOSWAMI S.: Tight Cocone: A Water-Tight Surface Reconstructor. *J. Computing, Inf. Sci. and Engin. 30* (2003), 302–307.

[DG04] DEY T., GOSWAMI S.: Provable Surface Reconstruction from Noisy Samples. In *Proceedings of Symp. on Comp. Geometry 2004* (2004), pp. 330–339.

[GH97] GARLAND M., HECKBERT P.: Surface Simplification Using Quadric Error Metrics. In *Proceedings of SIGGRAPH '97* (1997), pp. 209–216.

[Hal98] HALE J. G.: *Texture Re-Mapping for Decimated Polygonal Meshes*. Master's thesis, Edinburgh University, 1998.

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. In *Proceedings of SIGGRAPH '92* (1992), pp. 71–78.

[Hop96] HOPPE H.: Progressive Meshes. In *Proceedings of SIGGRAPH '96* (1996), pp. 99–108.

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Proceedings of SIGGRAPH '00* (2000), pp. 131–144.

[Mar95] MARUYA M.: Texture Map Generation from Object-Surface Data. In *Proceedings of Eurographics '95* (1995), pp. 397–405.

[MKN*04] MUELLER M., KEISER R., NEALEN A., PAULY M., GROSS M., ALEXA M.: Point-Based Animation of Elastic, Plastic and Malting Objects. In *Proceedings of SIGGRAPH/Eurographics Symposium on Comp. Animation* (2004), pp. 141–151.

[Pau03] PAULY M.: *Point Primitives for Interactive Modeling and Processing of 3D Geometry*. PhD thesis, ETH Zurich, 2003.

[PG01] PAULY M., GROSS M.: Spectral Processing of Point-Sampled Geometry. In *Proceedings of SIGGRAPH '01* (2001), pp. 379–378.

[PGK02] PAULY M., GROSS M., KOBBELT L.: Efficient Simplification of Point-Sampled Surfaces. In *Proceedings of IEEE Visualization '02* (2002), pp. 163–170.

[PKA*05] PAULY M., KEISER R., ADAMS B., DUTRÉ P., GROSS M., GUIBAS L.: Meshless Animation of Fracturing Solids. In *Proceedings of SIGGRAPH '05* (2005). to appear.

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: Surface Elements as Rendering Primitives. In *Proceedings of SIGGRAPH '00* (2000), pp. 335–342.

[RHHL02] RUSINKIEWICZ S., HALL-HOLT O., LEVOY M.: Real-time 3d model acquisition. In *Proceedings of SIGGRAPH '02* (2002), pp. 438–446.

[SGR96] SOUCY M., GODIN G., RIOUX M.: A Texture-Mapping Approach for the Compression of Colored 3D Triangulations. *The Visual Computer 12*, 10 (1996), 503–514.

[SZL92] SCHRÖDER W., ZARGE J. A., LORENSEN W.: Decimation of Triangle Meshes. In *Proceedings of SIGGRAPH '92* (1992), pp. 65–70.

[WPK*04] WEYRICH T., PAULY M., KEISER R., HEINZLE S., SCANDELLA S., GROSS M.: Post-Processing of Scanned 3D Surface Data. In *Proceedings of Eurographics Symp. on Point-Based Graphics* (2004), pp. 85–94.

[ZPBG01] ZWICKER M., PFISTER H.-P., BAAR J. V., GROSS M.: Surface Splatting. In *Proceedings of SIGGRAPH '01* (2001), pp. 371–378.

[ZPBG02] ZWICKER M., PFISTER H.-P., BAAR J. V., GROSS M.: EWA Splatting. *IEEE Trans. Comp. Graphics and Visualization 8*, 3 (2002), 223–238.

[ZPKG02] ZWICKER M., PAULY M., KNOLL O., GROSS M.: Pointshop 3D: An Interactive System for Point-Based Surface Editing. In *Proceedings of SIGGRAPH '02* (2002), pp. 322–329.

Appendix A: Pseudocode of Texture Packing Algorithm

This pseudocode version of the texture packing algorithm expects a set of triangle patches as input and writes them into a rectangular texture.

```
Procedure packTextures
T: set of triangular patches
y_min ← 0
while not empty(T) do
    h ← max_{t∈T} height(t)
    T_h ← {t ∈ T : height(t) > h − Δh}
    β ← 90°
    mirrorH ← false
    while not empty(T_h) do
        t ← arg min_{t∈T_h} min(|α_l − β|, |α_r − β|)
        mirrorV ← |α_l − β| > |α_r − β|
        if mirrorV then
            mirrorVertical(t)
        if mirrorH then
            mirrorHorizontal(t)
        if not insert(y_min, t) then
            y_min ← y_min + h
            break
        mirrorH ← not mirrorH
        if mirrorV then
            β ← α_l
        else
            β ← α_r
    end while
end while
```

The function insert$(\cdot,\cdot)$ inserts a triangle at the given y-location. The triangle is inserted on the far right and pushed as far left as possible. If there is not enough space in the texture to accomodate the triangle, the function returns $false$.