Interactive Rendering to View-Dependent Texture-Atlases

Matthias Trapp & Jürgen Döllner†

Hasso-Plattner-Institute, University of Potsdam, Germany

Abstract

The image-based representation of geometry is a well known concept in computer graphics. Due to z-buffering, the derivation of such representations using render-to-texture delivers only information of the closest fragments with respect to the virtual camera. Often, transparency-based visualization techniques, e.g., ghosted views, also require information of occluded fragments. These can be captured using multi-pass rendering techniques such as depth-peeling or stencil-routed A-buffers on a per-fragment basis. This paper presents an additional rendering technique that enables the derivation of image-based representations on a per-object level within a single rendering pass. We use a dynamic 3D texture atlas that is parameterized on a per-frame basis. Prior to rasterization, the primitives are transformed to their respective position within the texture atlas, using vertex-displacement in screen space.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1 Introduction

The concept of image-based representation of 3D shapes [ST90] has numerous applications in computer graphics. Despite static and dynamic imposters [DSSD99], they are the basis for advanced rendering effects performed in post-processing (e.g., edge detection, screen-space ambient occlusion, deferred shading). For the purpose of image-based occlusion management [ET08] (ghosted views), object highlighting, or the enhancement of depth perception (halos), it is necessary to efficiently generate such representations for all, or a subset of scene objects of complex 3D virtual environment.

An application that uses render-to-texture (RTT) capabilities of current rendering hardware to derive these representations on a per-object basis encounters two main problems: (1) only fragments with the most minimal depth value (with respect to the virtual camera) are captured; (2) either the complete 3D scene or a single scene object can be captured occlusion-free during a single off-screen rendering pass.

The first problem can be efficiently solved using depth-peeling [LHLW09] or stencil-routed A-buffer [MB07] approaches. These techniques operate at fragment level and usually require multiple rendering passes. The second problem can be compensated using multiple rendering passes in combination with multiple render-targets (textures). However, such an approach results in multiple, sparsely populated texture layers, which require additional management and, if at high viewport resolution, yielding to high GPU memory consumptions.

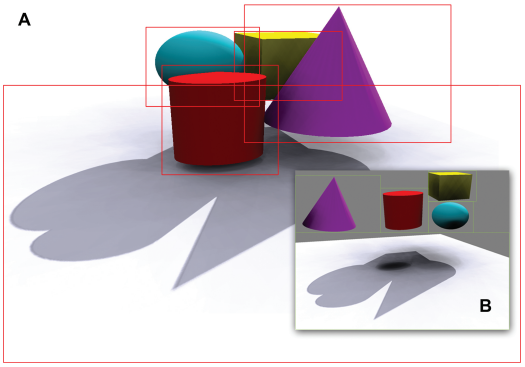

Figure 1: *The five scene objects (A) are rendered occlusion-free into a view-dependent texture atlas (B). Its parameterization is computed per-frame, based on the projected boundary volume of each object (red lines).*

† {matthias.trapp, juergen.doellner}@hpi.uni-potsdam.de

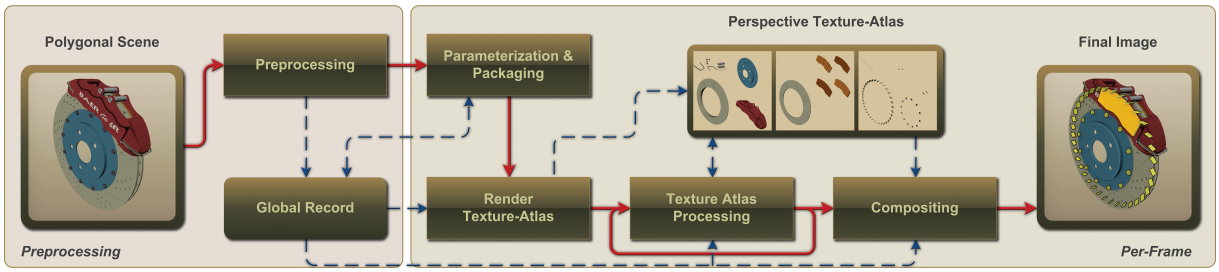

Figure 2: *Control flow (red lines) and data flow (blue) for the generation and rendering of view-dependent textures-atlases.*

For applications that require only a single layer of an object, this paper presents render-to-texture atlas (RTTA): an interactive and scalable rendering technique that enables dynamic generation of occlusion-free, image-based representations for multiple scene objects using graphics hardware (Figure 1). It extends RTT by using a single 3D texture-atlas as render target for all scenes objects. In contrast to the original concept of texture-atlases [Wlo05], it computes the texture-atlas parameterization and packaging per rendering frame, with respect to the projected boundary approximation (e.g., axis-aligned bounding box) of each object. During off-screen rendering it uses screen-space vertex displacement (SSVD) to transform each object into its respective atlas region. This is done using an additional, object-dependent 2D transformation that is applied to each object prior to rasterization. Our approach enables the usage of optimized (batched) scene geometry, which reduces state changes during rendering. Further, it can be easily integrated into existing rendering frameworks and systems. To summarize, our research has the following contributions to the reader:

1. We present a concept for view-dependent parameterization and generation of a texture atlas containing image-base representations of projected scene objects.
2. We describe the concept of screen-space vertex displacement and its application for generating view-dependent texture-atlases within a single rendering pass.
3. We briefly describe a hardware accelerated rendering technique that implements this concept and discuss its performance and limitations.

2 Render-To-Texture Atlas

Our concept mainly consists of two phases that are performed per frame (Figure 2): the view-depended computation of the texture-atlas parameterization and subsequently, the rendering of the scene geometry into the texture atlas.

2.1 Preliminaries

As texture atlas $TA = (T_w, T_h, T_d) \in \mathbb{N}^3$, we denote an a number of T_d layers of 2D textures, each with a fixed width T_w and height T_h. This data structure can be effectively represented on graphics hardware using 3D textures or 2D texture arrays. We assume that the orientation and projection transformation of the virtual camera can be described as a matrix **VPM**, and that the scene is rendered to a viewport given by $VP = (x, y, w, h) \in \mathbb{N}^4$.

At runtime, our concept requires global information about the objects of a 3D scene. Such record can be computed off-line for static meshes or dynamically for animated scenes. For each object, a record R_{ID} of the following structure is stored in a global record set $\mathcal{R}$:

$$R_{ID} = (B_{world}, B_{viewport}, B_{atlas}, l, \mathbf{T}) \quad R_{ID} \in \mathcal{R}$$

To identify an object at run-time, an unique object identifier $ID \in \mathbb{N}$ is required. This identifier must be encoded as a per-vertex attribute to allow geometry batching and arbitrary scene partitions for rendering. To approximate the area a 3D object occupies in a 2D texture atlas, its 3D boundary representation B_{world} is computed in world-space coordinates. Our approach uses 3D axis-aligned bounding boxes (AABB) as boundary representation. The 2D rectangular boundary $B_{viewport} \in VP$ denotes the clipped on-screen area of B_{world} and $B_{atlas} \in TA$ the occupied area within the texture atlas. An affine 2D transformation matrix $\mathbf{T}$ describes the transformation of $B_{viewport}$ into B_{atlas} in normalized device coordinates (NDC). Further, $l = 0, \ldots, T_d$ denotes the texture layer each object is rasterized in.

2.2 Texture-Atlas Parameterization & Packaging

Prior to RTTA, the texture-atlas parameterization needs to be determined, i.e., the mapping of a 3D boundary representation (B_{world}) into a 2D texture domain (B_{atlas}) for all records R_{ID}. Algorithm 1 shows the pseudo code for computing this mapping. In the first step, the boundary representation B_{world} is conservatively culled against the view frustum defined by **VPM**. On success, it is then projected into normalized device coordinates ($B_{projected}$) and clipped against the area $[-1,-1] \times [1,1]$. The resulting 2D boundary ($B_{clipped}$) is then scaled ($B_{viewport}$) with respect to the viewport VP. To enable artifact-free convolution filtering during texture-atlas post-processing and compositing (Figure 2), we can add a border of $b \in \mathbb{N}$ pixels. Fragments in this border area can be identified later, e.g, by using a specific alpha value. Next, texture-atlas packaging computes B_{atlas} for each $B_{viewport}$, starting a maximal texture-atlas resolution of $T_{w_{max}}$ width and $T_{h_{max}}$ height. After completion, it also delivers the resolution and the required number of texture layers T_d. Our current implementation uses a rectangular atlas packaging approach [IC01], which has a run-time complex-

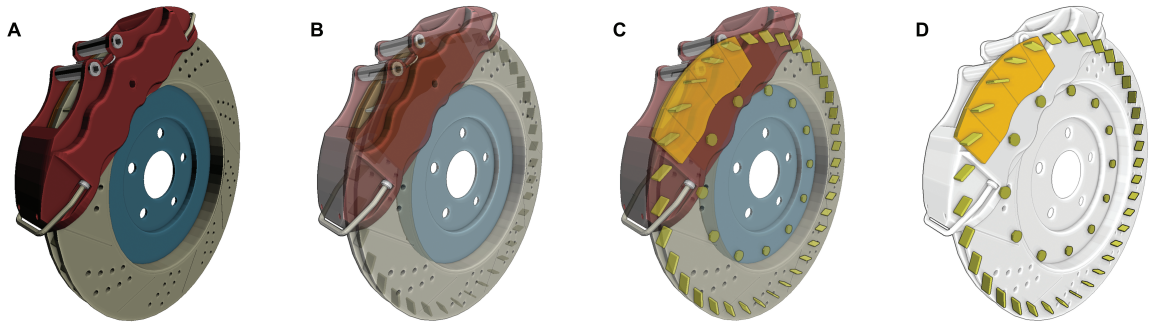

Figure 3: *Compositing variants derived from a view-dependent texture-atlas containing color and depth values per pixel. A: reconstruction of the scene by rendering depth-sprites; B: transparent rendering by ignoring the fragments depth (requires depth sorting); C: ghosted-view visualization showing a brake pad and screws highlighted; D: ghosted-view visualization that uses screen-space ambient occlusion and edge-enhancement during compositing for important scene objects.*

Algorithm 1 Texture-Atlas Parameterization & Packaging

```
for all R_ID ∈ ℛ do
  if viewFrustumCulling(B_world, VPM) then
    B_projected = project(B_world, VPM)
    B_clipped = clip(B_projected)
    B_viewport = scale(B_clipped, VP)
    B_viewport = addBorder(B_viewport, b)
    R_ID ← B_viewport
  end if
end for
TA ← atlasPackaging(ℛ, T_wmax, T_hmax)
for all R_ID ∈ ℛ do
  T = computeTransform(B_viewport, B_atlas, TA, VP)
  R_ID ← T
end for
```

ity of $O(n) = n \log n$. Finally, based on the packing results, the transformation $\mathbf{T}$ is computed, which basically consists of a translation and viewport scaling.

2.3 Screen-Space Vertex Displacement

This type of vertex displacement was introduced for rendering *camera textures* [SBGS06]. It can be applied in image- or object-space by transforming vertices using a translation vector stored in a 2D texture. For our purposes, we extend this concept to arbitrary affine 2D transformations in order to transform a rendered primitive into its respective atlas region prior to rasterization. Every vertex $V = (x, y, z, w)$ is transformed into its designated texture-atlas area B_{atlas} by displacing it parallel to view plane using $\mathbf{T}$. The new vertex position can be obtained by: $V' = \mathbf{T} \cdot V$. Before that, V is transformed from clipping coordinates (CS) into NDC via division by the homogeneous vector component w. Because NDC's cannot generally be interpolated linearly by the rasterizer, V' is transformed back to CS, after the transformation was applied.

3 Real-time Implementation

Our prototypical implementation is based on OpenGL [SA09] in combination with GLSL [Kes09]. It requires the encoding of the global data record $\mathcal{R}$ into a suitable GPU data structure to perform SSVD using the geometry shader stage. Therefore, the transformation matrix $\mathbf{T}$, the target layer l, and the atlas region B_{atlas} of each record R_{ID} are stored successively in a single texture-buffer object, denoted as *record buffer*. At runtime, the *ID* is used to index this buffer. The buffer is encoded per-frame and then shared between RTTA, successive texture atlas post-processing, and compositing steps (refer to Figure 3 and Section 3.2).

3.1 Vertex Displacement & Layered Rendering

The rendering setup for RTTA is similar to standard RTT applications. First, the framebuffer objects and render textures are set up according to TA, then the viewing and projection transformation **VPM** is applied and viewport is set to $(0, 0, T_w, T_h)$. After binding the record buffer (`recordBuffer`) the shader program (Figure 4) is enabled before rendering the scene geometry. The shader performs SSVD for each vertex and assigns the respective layer of the texture atlas to each output primitive. To avoid overdrawing of other atlas areas, four user clip-planes are set up according to the view frustum while the clip coordinates of the vertex are left untransformed.

3.2 Compositing from Texture Atlases

After RTTA is performed, an application-specific processing of the texture-atlas contents, e.g., edge-detection, color quantization, or similar, can be applied. The final compositing is performed on per-object level. Figure 3 shows results of per-object compositing using frame-buffer blending. This is done by generating and rendering 2D sprites for each object using the point-sprite expansion functionality of the geometry shader. Therefore, $n = |\mathcal{R}|$ point primitives with their respective object *ID* are rendered, that are converted into screen-aligned quads. Given the viewport setting VP, the four corner points are set according to B_{atlas} and are then transformed to $B_{viewport}$ using the inverse transformation matrix $\mathbf{T}^{-1}$.

```
uniform samplerBuffer recordBuffer; // global data
in int ID[3]; // per-vertex attribute: object ID
// fetch transformation and layer for object ID
void fetchRecord(inout mat4 T, inout int  layer);
...
void main(void) {
  mat4 T; int layer; fetchRecord(T, layer);
  gl_Layer = layer; // set texture-target layer
  for(int i = 0; i < 3; i++) // set every vertex
  {
    vec4 v = gl_ProjectionMatrix * gl_PositionIn[i];
    // screen-space vertex displacement
    gl_Position = (T *(v/ v.w))*v.w;
    gl_ClipVertex = gl_PositionIn[i];
    // set additional attributes...
    EmitVertex();
  }//endfor
  EndPrimitive();
  return;
}
```

Figure 4: *GLSL geometry shader (excerpt) that implements screen-space vertex displacement and layered rendering.*

4 Results & Discussion

4.1 Performance Evaluation

The performance tests are conducted using a NVIDIA GeForce GTX 285 GPU with 2048 MB video RAM on a Intel Xeon CPU with 2.33 GHz and 3 GB of main memory. Table 1 shows the results of our comparative evaluation. The tests are performed at a viewport and texture atlas resolution of 1024^2 pixels without view-frustum culling. The performance mainly depends on the number of scene objects and is bound by the performance of the geometry-shader stage.

4.2 Problems & Limitations

One conceptual problem of our implementation concerns the usage of 2D rectangular boundary approximations for representing $B_{viewport}$ and B_{atlas}. Due to perspective projection of the boundary, this can lead to an under-utilization of the texture atlas, especially for objects with non-convex shapes. For capturing objects that cover the entire screen, the utilization can be improved by choosing T_w and T_h as multiples of the viewport size.

A further problem represents the dynamic allocation of texture memory if adapting the texture-atlas size on a per-frame basis. We observe driver stalls during the removal of layers from the texture array. Currently, we compensate this problem using lazy-updates of the texture array at rendering idle time, or if the number of layers have not changed during a number of passes. Another hardware limitation represents the maximal number of texture layers T_d, as well as its resolution T_w and T_h. Thus, the maximal number of objects that can be captured within a single pass depends on this resolution and the texture atlas utilization.

5 Conclusions & Future Work

This paper presents the concept of view-dependent textures-atlases. It enables the generation and management of occlusion-free, image-based representations for multiple overlapping objects in complex 3D scenes. We further describe a real-time, hardware accelerated implementation that generates these textures-atlases within a single rendering pass and demonstrate its applications for interactive ghosted views. For future work, we plan to research the usage of more complex types of boundary representations to achieve a better texture-atlas utilization. We further evaluate possibilities for atlas-packaging strategies that operate entirely on GPU.

Table 1: *Comparative performance evaluation for different test scenes between plain rendering(STD), standard render-to-texture (RTT), and render-to-texture atlas (RTTA) (in frames-per-second).*

#Vertex	#Face	#Obj	STD	RTT	RTTA
2,191	4,236	5	2066	1015	940
32,081	21,246	21	1066	328	239
56,654	34,596	580	65	14	19
1,040,503	346,835	269	39	21	35

Acknowledgments

This work has been funded by the German Federal Ministry of Education and Research (BMBF) as part of the InnoProfile research group "3D Geoinformation"'.

References

[DSSD99] DÉCORET X., SCHAUFLER G., SILLION F., DORSEY J.: Multi-layered Impostors for Accelerated Rendering. In *Computer Graphics Forum (Proc. of Eurographics '99)* (Sep 1999).

[ET08] ELMQVIST N., TSIGAS P.: A Taxonomy of 3D Occlusion Management for Visualization. *IEEE Transactions on Visualization and Computer Graphics 14*, 5 (2008), 1095–1109.

[IC01] IGARASHI T., COSGROVE D.: Adaptive Unwrapping for Interactive Texture Painting. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (New York, NY, USA, 2001), ACM, pp. 209–216.

[Kes09] KESSENICH J.: *The OpenGL Shading Language Language Version: 1.50 Document Revision: 9*. The Khronos Group Inc., July 2009.

[LHLW09] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Efficient Depth Peeling via Bucket Sort. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 51–57.

[MB07] MYERS K., BAVOIL L.: Stencil Routed A-Buffer. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches* (New York, NY, USA, 2007), ACM, p. 21.

[SA09] SEGAL M., AKELEY K.: *The OpenGL Graphics System: A Specification (Version 3.2 (Core Profile))*. The Khronos Group Inc., July 2009.

[SBGS06] SPINDLER M., BUBKE M., GERMER T., STROTHOTTE T.: Camera Textures. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia* (New York, NY, USA, 2006), ACM, pp. 295–302.

[ST90] SAITO T., TAKAHASHI T.: Comprehensible Rendering of 3-D Shapes. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 197–206.

[Wlo05] WLOKA M.: *ShaderX3*. Charles River Media, 2005, ch. Improved Batching Via Texture Atlases, pp. 155–167.