High-Quality Non-Planar Projections Using Real-Time Piecewise Perspective Projections

Haik Lorenz and Jürgen Döllner

Hasso-Plattner-Institute, University of Potsdam, Germany
{haik.lorenz, doellner}@hpi.uni-potsdam.de

Abstract. This paper presents an approach to real-time rendering of non-planar projections with a single center and straight projection rays. Its goal is to provide the same optimal and consistent image quality GPUs deliver for perspective projections. It therefor renders the result directly without image resampling. In contrast to most object-space approaches, it does not evaluate non-linear functions on the GPU, but approximates the projection itself by a set of perspective projection pieces. Within each piece, graphics hardware can provide optimal image quality. The result is a coherent and crisp rendering. Procedural textures and stylization effects greatly benefit from our method as they usually rely on screen-space operations. The real-time implementation runs entirely on GPU. It replicates input primitives on demand and renders them into all relevant projection pieces. The method is independent of the input mesh density and is not restricted to static meshes. Thus, it is well suited for interactive applications. We demonstrate an analytic and a freely designed projection based on our method.

Key words: Non-planar projections, geometry shaders, geometry amplification, non-photorealistic rendering

1 Introduction

Automatic depiction of three-dimensional worlds, being real or virtual, requires a camera. What is the construction and inner workings of a real camera becomes the camera model and projection for the virtual camera. From all possible models, the pinhole camera model is the most widely used model. Today's graphics hardware is tailored to the underlying projection types: planar perspective or orthographic projections. Nonetheless, numerous applications in computer graphics require other, non-pinhole projection types:

- Non-planar displays, such as cylindrical or spherical walls, require non-linear projections to compensate for distortions [1].
- Some natural phenomena, such as caustics, reflections, or refractions off curved surfaces, can be described by projections [2].
- Visualizations benefit from adapted projections, such as increased field of view, lens effects, or reduced panorama distortions [3–5]. Such deliberate distortions can provide improved perception of a virtual environment.
- Arts and non-photorealism achieve dramatic effects using irregular projections [6–9].

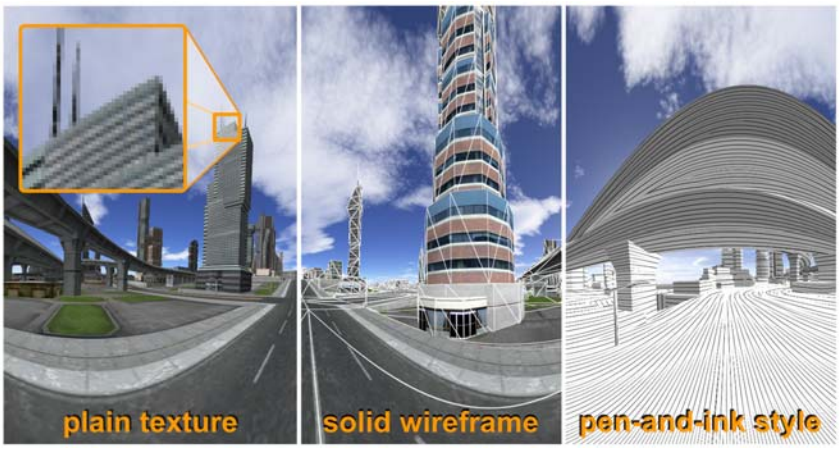

Fig. 1. A 360° cylindrical view of a city rendered using a piecewise perspective projection. The city model contains 35,000 triangles. The projection uses 160 pieces. At a resolution of 1600×1200, an NVidia 8800GTS achieves 55 fps with 16x anisotropic texture filtering and 16x antialiasing. Our technique enables the use of screen-space-dependent rendering effects such as solid wireframe [16] or pen-and-ink style [17].

- Images of particular projection types serve as storage for parts of the plenoptic function [10]. Most commonly, these are cubical, spherical, or paraboloidal images used for rendering reflections or refractions in real-time [11, 12].

Non-pinhole projections have been discussed extensively in literature, resulting in various camera models, e.g., [4, 13, 14]. They cannot be rendered directly with today's graphics hardware. Instead, ray-tracing is commonly used [13]. A large body of work exists on implementing ray-tracing on GPU for various phenomena and scene conditions. Wei et al. [2] and Popov et al. [15] present recent approaches and good surveys of related methods. Mostly, they use the GPU as powerful stream processor instead of as rasterization device and thus rarely benefit from built-in capabilities such as anisotropic texture filtering, perspective-correct interpolation, or screen-space derivatives.

1.1 Real-time Non-pinhole Projections on GPU Without Raytracing

A straightforward and efficient approach is the implementation as *image-based* post-processing effect [18, 19]. The rendering consists of two steps: First, a perspective projection is rendered into an offscreen buffer. Second, this buffer is used as texture for rendering a deformed mesh. The offscreen buffer can contain a cube map to enable 360° views. This approach is capable of rendering projections with a single center and straight projection rays (Single Center Of Projection – SCOP) only. It is image-based since the actual deformation happens after, not during, rendering the scene. Its advantages are easy implementation and good support by graphics hardware. The major drawback is image quality. The resampling in the second step inevitably introduces blurring

artifacts that especially degrade edges, detailed geometry, procedural textures, and stylization effects. Today's hardware capability of antialiasing through multi-sampling does not improve image quality substantially as it applies before resampling.

Object-space approaches do not suffer from image resampling artifacts and are not limited to SCOP effects as they render the image directly. A simple solution is applying the non-pinhole projection in the vertex shader [20]. Then, a triangle's vertices are projected correctly, but the interior and edges are rasterized incorrectly in a linear fashion. This is acceptable as long as a triangle's size on screen and thus the rasterization error is limited. To ensure this property, interactive environments require dynamic refinement. Approaches include precomputed static levels of detail [21], progressive meshes [22], adaptive mesh refinement [23, 24], render-to-vertex-buffer [25], dynamic mesh refinement [26], or hardware tessellation units [27, 28]. They vary in the distribution of computation between CPU and GPU. The rendered mesh must be free of T-junctions to prevent artifacts due to correct vertex location but incorrect edge rasterization. Even with refinement, the incorrect rasterization greatly amplifies Z-Buffer artifacts, such as inaccurate near plane clipping and interpenetrations of parallel triangles. A solution is emitting correct depth values in the fragment shader. This reduces depth test performance and increases fragment processing overhead due to a disabled early z-test [29].

A more sophisticated solution is using *non-linear rasterization*. Since the rasterizer uses hardwired linear interpolation, Hou et al. [30] and Gascuel et al. [31] replace each primitive by a bounding shape and use ray intersection in a fragment shader to compute the actual fragments and all their attributes under non-pinhole projections within that shape. As a consequence, these methods cannot benefit from high quality screen-space-dependent operations built into modern graphics hardware, such as mipmapping, anisotropic filtering, or screen-space derivatives.

This paper focuses on enabling these high quality rasterization capabilities of current GPUs for non-pinhole projections. Our approach achieves a significantly improved and consistent image quality regardless of the input mesh while maintaining real-time performance (Fig. 1). We enable the rasterization hardware to work under perspective projections exclusively and, hence, obtain optimal image quality. As a result, procedural textures and stylization effects can be used instantly regardless of the actual projection.

1.2 Piecewise Perspective Projection Overview

Piecewise perspective projections use an idea proposed in [30]: approximate a complex projection by a set of simpler projections. We refer to these simpler projections as *projection pieces*. The pieces' projection frusta are connected but disjoint with their union approximating the original projection volume. Hou et al. [30] rely on *triangle cameras*, simple non-pinhole projections, which make this method capable of rendering multi-perspective views but prevent it from exploiting hardware functionality. We restrict the projection pieces to using perspective projections exclusively, which limits our technique to SCOP effects.

Key advantage of our construction compared to other object-space approaches is the absence of non-linearities during rasterization. All non-linear aspects of the non-pinhole projection are encoded into the layout of the piecewise approximation. Consequently, rasterization and all high quality screen-space-dependent operations work with optimal

quality within each piece. Similarly, existing shaders, particularly procedural textures and stylization effects, can be used instantly. The resulting images do not exhibit blurring artifacts and capture the non-pinhole projection regardless of the input mesh and rendering effect. There is no need for refinement. In addition, Z-buffer artifacts are not amplified.

The increase in image quality comes at the cost of increased geometry processing overhead. Each primitive needs to be rendered into each projection piece it is visible in. Hence, it needs to be processed multiple times per frame. Depending on the number of pieces in an approximation, this can result in a substantial overhead.

The remaining paper is organized as follows: the next section provides an in-depth discussion of the idea of piecewise perspective projections, devises a real-time implementation using geometry shaders, and gives an implementation outline for future GPUs. Sec. 3 provides two example applications. Sec. 4 provides experimental results and compares them to alternative approaches. Sec. 5 concludes.

2 Piecewise Perspective Projection

The key idea of piecewise perspective projections is the approximation of a non-pinhole projection volume by a set of connected but disjoint perspective projection frusta, so-called projection pieces. Each projection piece uses a regular perspective projection clipped to the piece's boundaries for image formation. Thus, the rendering encounters no non-linearities and hardware rasterization generates a correct image at optimal quality for each piece. The combination of all piece images creates an approximation of the desired non-pinhole projection. The number of pieces defines the approximation quality.

Our method can reproduce the same projection effects as the image-based approach of [18]. Their intermediate rendering uses a perspective projection described by a matrix M_{P}. The triangle mesh used for deforming this rendering implicitly defines an affine transformation $M(t)$ from the source area of each triangle t to the screen. An equivalent piecewise perspective projection can be constructed of triangular pieces by clipping a perspective projection $M(t) * M_{\mathrm{P}}$ to the deformed triangle's boundaries for all t.

For implementing this idea, three challenges need to be addressed:

1. Approximation of the non-pinhole projection with projection pieces,
2. Rendering of a primitive in all projection pieces it is visible in, and
3. Clipping of a primitive's rendering to the projection piece's boundaries.

Approximation An artifact-free approximation is only possible for SCOP effects, as other projections lead to overlapping piece frusta. In general, each projection piece p uses an individual projection matrix $M_{\mathrm{P}}(p)$, such that neighboring pieces produce matching renderings at their shared boundary. Matrix computation depends on the particular projection and happens once in a preprocessing step. Two typical approaches are exemplified in Section 3.

Rendering A simple implementation renders the whole model for each projection piece with the respective projection matrix and clipping in effect. Rendering should use additional culling to account for a projection piece's small screen area but does not require any changes to shaders or meshes to deal with a non-pinhole projection.

Clipping Clipping can rely on viewport clipping if all pieces are rectangular or on user clip planes for convex pieces. Since both possibilities are performed by the rasterization hardware, no explicit clipping needs to be implemented in a shader. The resulting piece images are non-overlapping and thus they can be rendered directly to the framebuffer without the need for a dedicated composition step.

Due to additional culling and a large number of draw calls, this straight-forward implementation suffers from increased CPU load. At the same time, GPU efficiency is reduced as each draw call renders to a small screen portion only. Thus, rendering becomes CPU-bound and real-time performance is limited to projection approximations with very small piece counts.

2.1 Real-time Implementation

Our real-time implementation reverses the rendering approach to make it GPU-friendly. Instead of determining all primitives for a projection piece through culling, we determine all relevant, i.e., covered, pieces for a primitive. We can then render the final image in a single draw call by replicating each primitive to all its relevant pieces. Projection matrices are stored in buffer textures for shader access. User clip planes cannot be updated by a shader. The alternative is to define a standard clip space with fixed clip planes and provide a transformation from camera space to clip space for each projection piece.

Since a primitive can fill the whole screen, the maximum replication count is the projection piece count. Hence, a straight forward replication using geometry shaders, which are currently limited to at most 128 output vertices, is not possible. In [26] a solution to a similar problem has been described. They use a fixed three-pass scheme for per-primitive view-dependent tessellation on GPU and achieve arbitrary and unbounded geometry amplification without CPU intervention. Core of their scheme is a continuously updated intermediate mesh of barycentric subtriangles. This transforms the geometry shader's output limit from a mesh size limit to a per-frame growth limit. Since we only require replicated, not tessellated, primitives, we replace their intermediate mesh with a primitive index and accompanying replication numbers. Both are stored in separate ping-pong buffers. The primitive index works similar to a traditional vertex index. The accompanying replication number consecutively numbers all a single primitive's occurences in the index. Together, they enable indexed access to a primitive's vertex attributes in the vertex shader (e.g., by passing all 3 positions for a triangle at once) but also allow for distinguishing replications of a single primitive in the geometry shader.

In the following, we provide a brief description of the rendering process (Fig. 2). For details, refer to [26]. In the first pass, all original primitives are processed by a geometry shader to determine the number of covered projection pieces. This information is stored in a buffer using transform feedback. The second pass takes the previous frame's primitive index and produces a new primitive index and matching replication numbers, such that each primitive is replicated according to the counts calculated in pass 1. Pass 3 finally renders all replicated primitives. It uses the primitive index to fetch all a primitive's vertex attributes from vertex buffers and the replication number

to select the projection piece with projection and clip matrix. Additional vertex, geometry and fragment processing can implement any effect as if no piecewise perspective projection was in effect. Thus, existing shaders are easily incorporated.

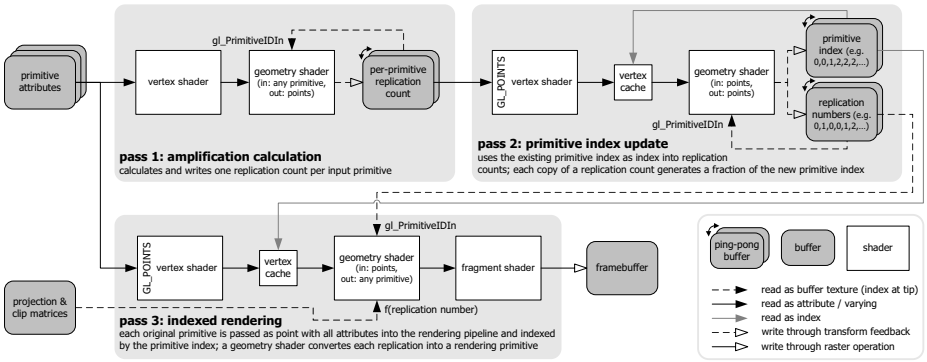

Fig. 2. Primitive replication algorithm overview and data flow. Pass 3 can implement any rendering effect.

This scheme applies to arbitrary "primitive soups" since no connectivity information or topological restriction is assumed. Key for rendering is the determination of covered projection pieces (pass 1) and their enumeration (pass 3). Both depend on the desired non-pinhole projection. Two aspects need to be considered: First, while rendering primitives to irrelevant pieces does not influence image quality, it degrades performance since additional replications are created and processed only to become clipped again. Thus, the determination is not required to be exact. Nevertheless, poor estimation reduces performance. Second, given a primitive and its replication number, the target projection piece needs to be identified in $O(1)$ time in a shader (function $f(\text{replicationnumber})$ in Fig. 2). The mapping can be supported by additional information generated in pass 1. In Sec. 3, we present two approaches.

2.2 Single-Pass Implementation With a Configurable Tessellator Unit

The next generation of GPUs will include a configurable tessellator unit [28]. In conjuction with two new shader stages, hull and domain shaders, it becomes possible to generate up to 8192 triangles for a single input primitive. This unit is expected to provide much better performance regarding geometry amplification than geometry shaders. In this section, we outline an implementation of piecewise perspective projections based on the tessellator unit in Direct3D 11 (Fig. 3). As there is no supporting hardware available so far, the implementation has been tested using the Microsoft reference rasterizer. Thus, it should be considered a proof-of-concept and a starting point for optimizations.

Basic implementation idea is the mapping of all computations found in Fig. 2 to the new pipeline stages. Amplification calculation is handled by the hull shader, which

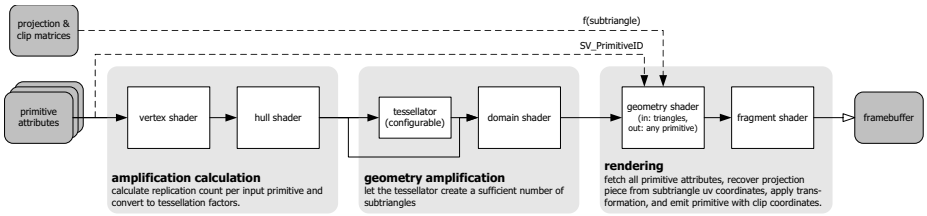

Fig. 3. Single-pass piecewise perspective projections based on Direct3D 11. The three passes from Fig. 2 clearly map to Direct3D 11's extended graphics pipeline. This figure uses the same notation as Fig. 2.

determines the number of covered projection pieces. It then selects corresponding tessellation factors for each primitive, such that the tessellator creates the required number of subtriangles. Since Direct3D 11 does not guarantee specific tessellation rules so far, we use a table that contains tessellation factors for each possible subtriangle count. This table is filled in advance. The domain shader computes vertices of these subtriangles. Since we use the tessellator to replicate the input primitive, not to create a contiguous mesh, the domain shader only passes the produced uv-coordinates to the geometry shader. The geometry shader loads the primitive's attributes from a buffer texture using the primitive id and determines the projection and clip matrices using the uv-coordinates. The geometry shader cannot rely on a replication number (pass 3 in Fig. 2) as it is not provided by the tessellator. After transformation the replicated triangle is emitted. The pixel shader, again, is not influenced by our method and can implement any rendering style without taking into account the non-pinhole projection.

A pitfall in this implementation is the limitation to 8192 replications per primitive. If the projection approximation uses more than 8192 pieces, a primitive could exceed this limitation if it fills the whole screen. As a workaround, the geometry shader can emit multiple replications per subtriangle to retain the single-pass scheme.

Compared to the implementation from Sec. 2.1, the tessellator-based implementation has various advantages: It can cull primitives early by specifying negative tessellation factors. It does not require intermediate buffers and overflow control. Finally, it does not require ping-pong buffers and thus has no interframe dependencies. As a result, this implementation is much better suited for integration in existing graphics frameworks and for use in multi-GPU environments.

3 Applications

Our real-time implementation involves two application-dependent parts: projection piece definition and projection piece coverage determination/enumeration. We demonstrate the use for two typical applications: a horizontal cylindrical projection, which can be described analytically, and a texture-based view deformation, which improves the camera texture technique of [20].

3.1 Cylindrical Projection

A horizontal cylindrical projection uses a perspective projection in the vertical direction but a non-planar projection horizontally. Thus, it suffices to limit the horizontal edge length to control an approximation's quality. The piecewise perspective projection then splits the curved projection volume into narrow rectangular slices. Figure 4 sketches this setting.

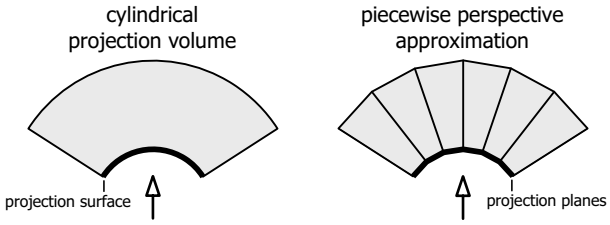

Fig. 4. Top-down view of the cylindrical projection volume and its approximation with perspective projections.

Projection piece coverage determination and enumeration for rendering is rather simple as a single primitive normally covers a consecutive range of projection pieces. It suffices to find the leftmost and rightmost point of the primitive's projection and render it to all pieces in between. Wrap-arounds require special care. A special case occurs when cylinder axis and primitive intersect. In that case, the primitive is potentially visible in all projection pieces. Finally, pass 1 outputs both start piece index (which can be to the right of the end index) and piece count. Pass 3 uses a primitive's replication number plus the start index modulo n – the number of projection pieces – as target projection piece.

The projection matrix M of a piece p can be described by a series of transformations:

$$M(p) = M_{\mathrm{tx}}(p) * M_{\mathrm{sx}} * M_{\mathrm{P}} * M_{\mathrm{ry}}(p) \tag{1}$$

M_{ry} rotates the center axis of a projection piece about the y axis onto the negative z axis. M_{P} is a perspective projection matrix with a horizontal field of view $\varphi_{\mathrm{p}} = \varphi_{\mathrm{c}}/n$, where φ_{c} denotes the cylindrical field of view. M_{sx} scales the standard postprojective space to fit the piece's width on the screen. M_{tx} finally moves the piece's projection from the screen center to its actual location on screen.

Clipping requires a standard clip space to enable fixed clip planes. The following transformation leads to such a space:

$$M_{\mathrm{clip}}(p) = M_{\mathrm{s}}(M_{\mathrm{P}_{11}}; M_{\mathrm{P}_{22}}; 1) * M_{\mathrm{ry}}(p) \tag{2}$$

M_{s} is a scaling operation that uses the first ($M_{\mathrm{P}_{11}}$) and second ($M_{\mathrm{P}_{22}}$) value from the projection matrix's diagonal. The complete transformation effectively transforms into the normalized space used for perspective division. The corresponding four clip

planes define an infinite pyramid with the tip being located in the origin and the opening pointing down the negative z axis with an opening angle of 90° both vertically and horizontally.

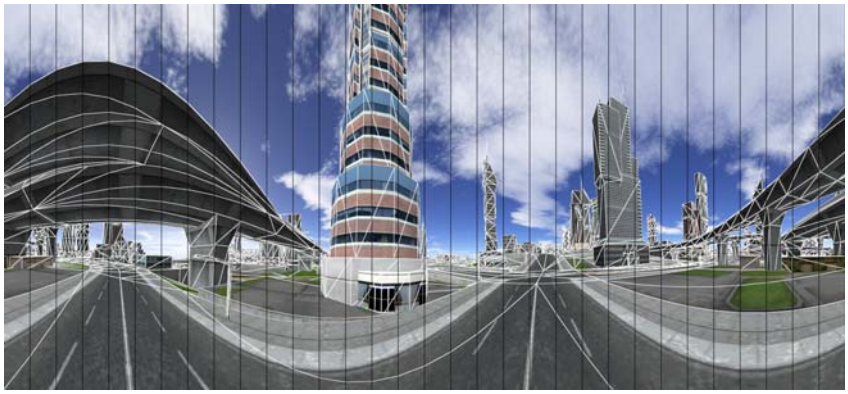

Fig. 5. Rendering of a piecewise perspective 360° cylindrical projection with an overlayed thick white wireframe. Piece borders are marked with thin black lines.

Fig. 5 depicts a sample image with highlighted piece boundaries and primitive edges. For clarity, it uses only 32 pieces with a width of 50 pixels. Experiments show, that pieces of width 10-20 pixels provide a good approximation. The average replication count in that case is less than 2, while the maximum replication is the total piece count n.

3.2 Texture-based View Deformation

View deformation [19] uses one or more standard perspective views (e.g., a cube map) and distorts them to create the final image. The distortion is either analytical, such as a paraboloid mapping, or freely defined, such as camera textures [20]. Both approaches use a rectangular two-dimensional grid in the perspective view(s) and map it to a deformed mesh on the screen. The construction of a piecewise perspective projection follows the description in Section 2. In the following, we provide details for an improved implementation of camera textures. They encode the distortion as offset vectors in a 2D texture (Fig. 6). A point's deformed projection is found by using its perspective projection for texture lookup and adding the resulting offset vector to that perspective projection. In contrast to the original implementation, ours is independent of mesh density. Regardless of a primitive's projected size, all details of the distortion are captured in the primitive's interior.

Piecewise perspective projections require splitting the rectangular grid into triangles. Even though it is possible to specify a projective mapping from a two-dimensional rectangle to an arbitrary quadrangle, it is not possible to guarantee a matching mapping for shared edges. This property would require a bilinear transformation [32] current rasterizers cannot deal with. Splitting the rectangle into two triangles leads to two affine

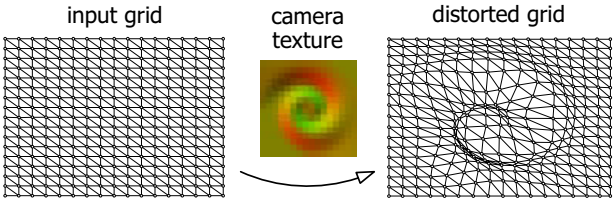

Fig. 6. Projection plane splitting and subsequent distortion using a 16×16 pixel camera texture. The model is rendered directly into the distorted grid.

transformations and a continuous approximation. Nevertheless, considering pairs of triangluar projection pieces as one cell is benefical regarding coverage determination and enumeration. It enables operating on a simple rectangular grid in pass 1. View deformation is irrelevant to pass 1 as it does not change visibility. Pass 2 replicates primitives for cells. Pass 3 finally emits each primitive twice – once for each triangle in a cell – with the respective transformation matrices in effect.

A simple solution for determining the coverage of a primitive is using its bounding box in the undistorted projection plane. All cells intersected by the bounding box are considered as being covered. Thus, the output of pass 1 is the position of the lower left cell c_{ll} and width w and height h of the bounding box in cell units. For efficient storage, all four values use 16-bit integers and are packed into two 32-bit integers. Pass 3 can map the replication number r to a cell at position $(c_{ll}.x + r \bmod w\,;\ c_{ll}.y + \lfloor r/w \rfloor)$. This two-dimensional index can be used for lookup in a texture containing the affine transformation matrices for both projection pieces in this cell. Since the bounding box coverage determination is very conservative, we added culling to pass 3 to discard invisible primitives before rasterization setup.

The derivation of affine transformation matrices can be found in [32]. During rendering, it is applied subsequent to the original model-view-projection matrix. Clipping uses an approach similar to the cylindrical projection. Here, only three clip planes are in effect which form a triangular pyramid. To clip both pieces of one cell to the same clip planes, one piece's clip coordinates are rotated about the z axis by 180°.

Fig. 7 shows a 64×64 camera texture (similar to Fig. 6) applied to a view of a city model. A thin black wireframe indicates the triangular projection cells, thick white lines highlight primitive edges. Our implementation of texture-based view deformation allows for animating the deformation effect, as this only involves updating the matrices.

4 Results

We compare piecewise perspective projections with image-based implementations for both applications presented in Section 3. The implementations use native OpenGL 2.0 with relevant extensions. All measurements have been taken on a desktop PC running Windows XP with an AMD Athlon 64 X2 4200+ processor, 2GB RAM, and an NVidia GeForce 8800 GTS with 640 MB RAM. The tests use a path through the textured city

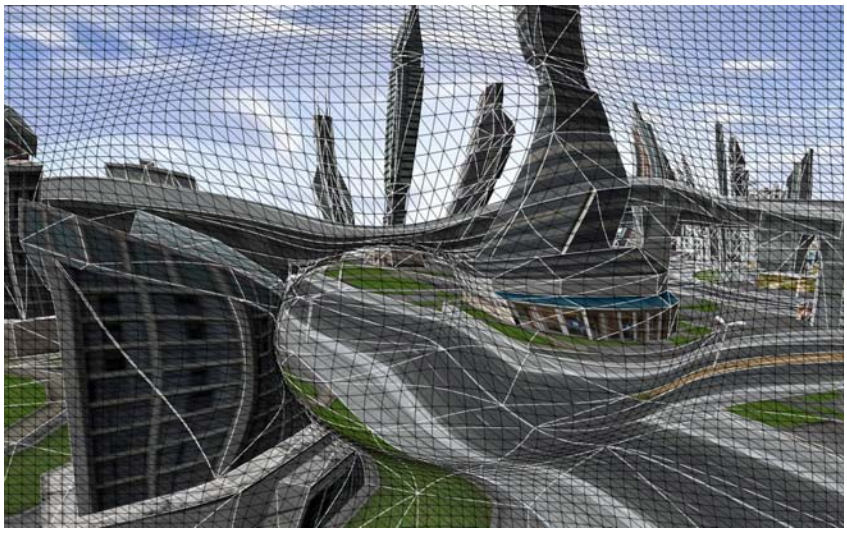

Fig. 7. Rendering using the camera texture shown in Fig. 6 at a resolution of 64×64. Thin black lines indicate projection pieces. Thick white lines highlight primitive edges.

model data shown in Figures 5 and 7. It consists of 35,000 triangles in 14 state groups. The viewport resolution is 1600×1200. In contrast to [26], no latency hiding has been used since it showed no improvements. Besides the frame rate, we provide the number of triangles used for rendering (av. tri. count), their replication ratio to the original triangle count (av. repl. ratio), and the total size of all vertex buffers used for rendering (Vbuf.). High quality (HQ) measurements use 16x anisotropic texture filtering and 16xQ antialiasing.

Table 1. Rendering statistics. Our piecewise perspective projection (PPP) outperforms the image-based implementation (IB) for the 360° cylindrical camera. For view deformation, IB provides higher frame rates. In terms of image quality, PPP is always superior (Fig. 8).

	360° cylindrical camera (160 pieces)				view deformation (9,322 pieces)			
Impl	Fps	Av. tri. count	Av. repl. ratio	Vbuf. (kB)	Fps	Av. tri. count	Av. repl. ratio	Vbuf. (kB)
IB	41.7	21,151	0.61	1,081	206.2	34,596	1	1,081
PPP	84.7	67,675	1.96	2,672	22.1	351,954	10.17	7,661
IB HQ	33.8	21,151	0.61	1,081	95.8	34,596	1	1,081
PPP HQ	54.8	67,675	1.96	2,672	20.9	351,954	10.17	7,661

The image-based implementation of the 360° cylindrical camera uses a dynamic 2048×2048 cubemap that is created in a single pass through layered rendering. It implements frustum and backface culling in the geometry shader [29] which explains

the replication count less than 1. The piecewise perspective projection uses strips of 10 pixels, i.e., 160 pieces, for approximation. On average, each triangle is visible in only two strips. The increased memory footprint results from the intermediate mesh, which requires 16 bytes per rendered primitive. In total, our method outperforms the image-based approach while providing higher image quality (Fig. 8). Even for smaller cubemaps, the image-based approach does not overtake ours, but image quality further degrades.

For the texture-based view deformation, the image-based technique uses only a 2D texture, no cubemap. Therefor, it achieves higher frame rates than for the cylindrical projection. In contrast, our method needs to render a significantly higher amount of triangles, which translates to a reduced speed. Each input triangle spans on average about 10 of the 9,322 projection pieces. While delivering interactive frame rates, the vertex processing overhead is substantial. The bottleneck here is pass 3. Primitive replication performed in pass 1 and 2 accounts for less than 10% of the total workload. Consequently, a more aggressive coverage determination than the bounding box test could significantly improve performance. In addition, a projection piece size of 20×20 pixels suffices for good approximations, i.e., the camera texture resolution should be adapted to the viewport resolution. In our example, we use a 80×60 camera texture. For a 128×128 camera texture, the replication ratio jumps from 10 to about 21 (i.e., about 725,000 triangles per frame) and the frame rate drops to 8.2 fps.

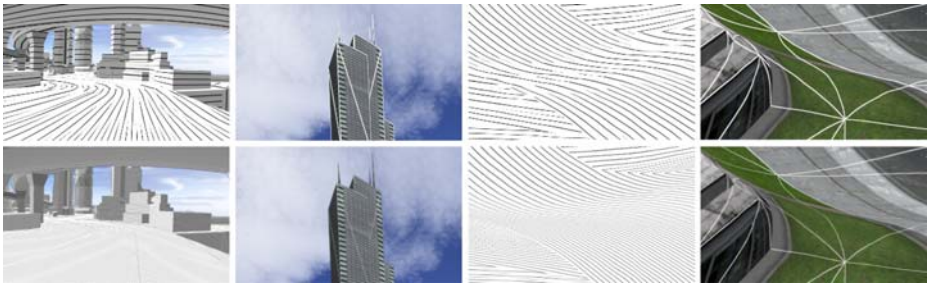

Fig. 8. Comparision of image quality. Closeups of screen shots for PPP (top) and IB (bottom). Left to right: cylindrical camera pen-and-ink style, cylindrical camera solid wireframe, view deformation pen-and-ink style, view deformation solid wireframe.

Figure 8 shows the difference in image quality. The image-based implementation cannot produce a consistent result due to the separate image resampling. Accordingly, stroke width for the solid wireframe and stroke distance for the pen-and-ink style vary across the image. Our piecewise perspective projection, in contrast, delivers consistent results even in highly distorted areas without the effect being "aware" of the non-pinhole projection.

A supplemental demo of the texture-based view deformation is available online [33]. It uses OpenGL 2.1 and requires Windows XP or later and a NVidia GPU 8000 series or better.

5 Conclusions

This paper has presented a novel approach to rendering non-pinhole projections with a single projection center. The piecewise perspective projection technique removes non-linearities from rendering by approximating a projection with a set of perspective projections. The distorted image is formed directly on screen without intermediate rendering steps. As a result, all image quality optimizations provided by modern graphics hardware that assume a perspective projection continue to operate with regular precision. Particularly, antialiasing, procedural textures, and stylization effects profit from our technique. It can be implemented on any graphics hardware but requires Direct3D 10 features for real-time performance. Core is on-demand replication of primitives on the GPU using geometry shaders and transform feedback, such that a primitive is rendered only into projection pieces it actually covers. A demo is available online [33].

The technique's drawback is a high geometry processing overhead. Primitive replication itself is rather efficient. The major bottleneck is vertex processing in pass 3 since a rendered primitive covers at most a single projection piece. In the future, we seek to improve the performance of pass 3 by better coverage determination. A second direction of research is evaluating applicability to other types of projections, such as slit or pushbroom cameras. The rendering scheme might also prove useful for other algorithms, e.g., [30]. Finally, we need to verify and tune our tessellator-based single-pass implementation (Sec. 2.2) as soon as respective GPUs become available.

Acknowledgements

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

References

1. Jo, K., Minamizawa, K., Nii, H., Kawakami, N., Tachi, S.: A GPU-based real-time rendering method for immersive stereoscopic displays. In: ACM SIGGRAPH 2008 posters, ACM (2008) 1
2. Wei, L.Y., Liu, B., Yang, X., Ma, C., Xu, Y.Q., Guo, B.: Nonlinear beam tracing on a GPU. Technical report, Microsoft, MSR-TR-2007-168 (2007)
3. Popescu, V., Aliaga, D.G.: The depth discontinuity occlusion camera. In: SI3D, ACM (2006) 139–143
4. Brosz, J., Samavati, F.F., Sheelagh, M.T.C., Sousa, M.C.: Single camera flexible projection. In: Proc. of NPAR '07, ACM (2007) 33–42
5. Kopf, J., Lischinski, D., Deussen, O., Cohen-Or, D., Cohen, M.: Locally Adapted Projections to Reduce Panorama Distortions. Computer Graphics Forum (Proceedings of EGSR 2009) **28**(4) (2009) to appear
6. Wood, D.N., Finkelstein, A., Hughes, J.F., Thayer, C.E., Salesin, D.H.: Multiperspective panoramas for cel animation. In: Proc. of ACM SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co. (1997) 243–250
7. Agrawala, M., Zorin, D., Munzner, T.: Artistic multiprojection rendering. In: Proc. of the Eurographics Workshop on Rendering Techniques 2000, Springer-Verlag (2000) 125–136

8. Glassner, A.S.: Digital cubism. IEEE Computer Graphics and Applications **24**(3) (2004) 82–90
9. Glassner, A.S.: Digital cubism, part 2. IEEE Computer Graphics and Applications **24**(4) (2004) 84–95
10. Rademacher, P., Bishop, G.: Multiple-center-of-projection images. In: SIGGRAPH. (1998) 199–206
11. Heidrich, W., Seidel, H.P.: View-independent environment maps. In: HWWS '98: ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, ACM (1998) 39–45
12. Wan, L., Wong, T.T., Leung, C.S.: Isocube: Exploiting the cubemap hardware. IEEE Trans. on Vis. and Comp. Graphics **13**(4) (2007) 720–731
13. Löffelmann, H., Gröller, E.: Ray tracing with extended cameras. Journal of Visualization and Computer Animation **7**(4) (1996) 211–227
14. Yu, J., McMillan, L.: General linear cameras. In: ECCV (2). Volume 3022 of Lecture Notes in Computer Science., Springer (2004) 14–27
15. Popov, S., Gunther, J., Seidel, H.P., Slusallek, P.: Stackless kd-tree traversal for high performance gpu ray tracing. Computer Graphics Forum **26** (2007) 415–424
16. Bærentzen, A., Nielsen, S.L., Gjøl, M., Larsen, B.D., Christensen, N.J.: Single-pass wireframe rendering. In: ACM SIGGRAPH 2006 Sketches, ACM (2006) 149
17. Freudenberg, B., Masuch, M., Strothotte, T.: Walk-through illustrations: Frame-coherent pen-and-ink in game engine. In: Proc. of Eurographics 2001. (2001) 184–191
18. Yang, Y., Chen, J.X., Beheshti, M.: Nonlinear perspective projections and magic lenses: 3d view deformation. IEEE Comput. Graph. Appl. **25**(1) (2005) 76–84
19. Trapp, M., Döllner, J.: A generalization approach for 3d viewing deformations of single-center projections. In: Proc. of GRAPP 2008, INSTICC Press (January 2008) 162–170
20. Spindler, M., Bubke, M., Germer, T., Strothotte, T.: Camera textures. In: Proc. of the 4th GRAPHITE, ACM (2006) 295–302
21. Sander, P.V., Mitchell, J.L.: Progressive Buffers: View-dependent Geometry and Texture for LOD Rendering. In: Symposium on Geometry Processing, Eurographics Association (2005) 129–138
22. Hoppe, H.: Progressive meshes. In: Proc. of SIGGRAPH '96, ACM (1996) 99–108
23. Boubekeur, T., Schlick, C.: A flexible kernel for adaptive mesh refinement on GPU. Computer Graphics Forum **27**(1) (2008) 102–114
24. Tatarinov, A.: Instanced tessellation in DirectX10. In: GDC '08: Game Developers' Conference 2008. (2008)
25. Yu, X., Yu, J., McMillan, L.: Towards multi-perspective rasterization. Vis. Comput. **25**(5-7) (2009) 549–557
26. Lorenz, H., Döllner, J.: Dynamic mesh refinement on GPU using geometry shaders. In: Proc. of the 16th WSCG. (2008)
27. Tatarchuk, N.: Real-time tessellation on GPU. In: Course 28: Advanced Real-Time Rendering in 3D Graphics and Games. ACM SIGGRAPH 2007. (2007)
28. Castaño, I.: Tesselation of displaced subdivision surfaces in DX11. In: XNA Gamefest 2008. (2008)
29. Persson, E.: ATI radeon HD2000 programming guide. Technical report, AMD, Inc. (2007)
30. Hou, X., Wei, L.Y., Shum, H.Y., Guo, B.: Real-time multi-perspective rendering on graphics hardware. In: EUROGRAPHICS Symposium on Rendering, Blackwell Publishing (2006)
31. Gascuel, J.D., Holzschuch, N., Fournier, G., Péroche, B.: Fast non-linear projections using graphics hardware. In: Symposium on Interactive 3D graphics and games SI3D '08, ACM (2008) 107–114
32. Heckbert, P.S.: Fundamentals of texture mapping and image warping. Technical report, University of California at Berkeley, Berkeley, CA, USA (1989)
33. Lorenz, H.: PPP demo. http://www.haik-lorenz.de/geometryshaders.html (2009)