Hybrid Client/Server Rendering with Automatic Proxy Model Generation

Jens Olav Nygaard
and Jon Mikkelsen Hjelmervik

SINTEF Digital
Applied Mathematics
Norway
Email: Jens.O.Nygaard@sintef.no,
and Jon.M.Hjelmervik@sintef.no

***Abstract*—A common problem in remote rendering setups is that of temporarily insufficient bandwidth and latency. For a proper experience of immersiveness, at least some rendering should be presented to the user, and it should appear to be responsive to user input, even in the presence of connection glitches. The all too familiar spinning hourglass symbol, or equivalent, will degrade such an experience. With the advent of Virtual Reality (VR) and Augmented Reality (AR), solutions to this become important even though connectedness in general is improving. We are dealing with a remote rendering of three-dimensional (3D) geometry being pushed to a lesser client, and what we in essence do is to replace a spinning hourglass symbol with an automatically client-generated approximation of the 3D geometry rendered on the client, responding to client-recorded user input. We call this a *proxy model*. Our main result is a system for automatically producing such proxy models on the client, from received images and depth buffers only, for showing on the client when remotely rendered frames do not arrive sufficiently fast.**

Keywords–Client-server; remote rendering; high latency; low bandwidth

I. INTRODUCTION

Hand in hand with increasingly powerful rendering engines comes ever increasing requirements on computational accuracy, power efficiency, data sizes, scaling properties, etc. This is also reinforced by popular cloud-based approaches and wireless usage patterns. An effect of this is that interactivity still is a difficult issue. We consider a client/server model for 3D rendering, addressing latency, bandwidth and scaling problems in a novel way.

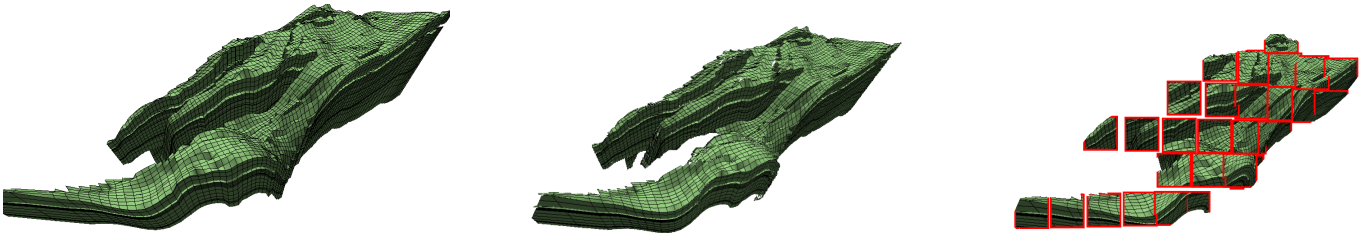
We introduce a *proxy model*, defined as a temporary model to be shown and manipulated locally on a client while waiting for the appropriate image from a connected server. Producing such proxy models can be difficult for many kinds of 3D data, like in our main case in which we have an oil reservoir viewer [1] that renders large *corner point grids*, together with faults, oil wells, and more; see Figure 1 for an example of both a full server-side rendering, and automatically generated proxy models rendered in Google Chrome. Our solution is to pass depth information from the server along with ordinary rendered images. From this, a rudimentary 3D model is built, and with the Red, Green and Blue (RGB) image as a texture, this model can be manipulated and rendered on the client while waiting for the next update from the server. If the client does not change the position or orientation of the model too much, this proxy model rendering integrates seamlessly with the slower stream of server-rendered frames. Even if bandwidth and latency is not a problem, it may be desirable to let a server of limited capacity serve many simultaneous users, hence limiting the effective server time available for each one. Suitable scaling may still be achieved using our solution. This is currently being commercialized as a part of the result of a recently finished European Union (EU) project called CloudFlow [2].

In Section II, we review some previous and related work, before we consider our contribution in Section III. This is further split into three subsections, in Subsection III-A we consider the server side part of the system, in Subsection III-B, the client side and in Subsection III-C we consider automatic parameter tuning during use of the system. We discuss results in Section IV, before we finally sum up in Subsection V.

II. PREVIOUS WORK

Our approach has some similarities to *Image Based Rendering* (IBR) techniques [3], with the difference that an important IBR problem would be the reconstruction of a depth map from images, while we have access to the full depth map from a rendering pass. Another way to use depth maps similar to what we do, is for "immersive streaming", see, e.g., [4], where focus is on depth map compression, an issue that we also consider. Another work focused on similar streaming and geometry compression, is Teler [5]. Also, [6] contains some of the same ideas as our work, with respect to image-based rendering acceleration. A very early work aiming at the same kind of "inter-frame rendering", is Mark et al. [7]. Here, several frames are warped, and subsequently combined, in order to avoid large unpainted areas caused by occlusion. This corresponds to our use of several proxy models, each from a pair of (rgb, depth)-images, the main difference being that they work on separate pixels, while we use larger, textured "splats". Our method avoids their slightly complicated meshing, discontinuity detection and final image composition stage. Common to many IBR-algorithms is also that of stitching together 3D or 2.5D point clouds. We could do this for our 2.5D maps fetched from the server, but it is unclear if the benefit would outweigh the cost. Our approach differs from many similar ones, in that we use existing depth maps to distill and render temporary geometry, rather than retrieveing the depth maps from images. We observe that these 2.5D height maps are exactly what "3D cameras" (time-of-flight and other range image sensors) produce, but most authors considering these are typically building more complex geometries before

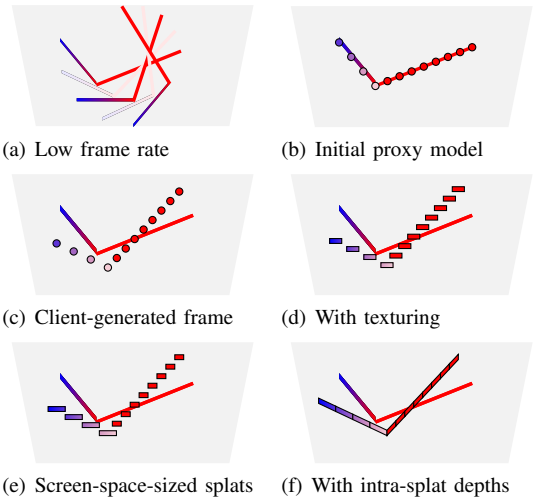
Copyright (c) IARIA, 2017. ISBN: 978-1-61208-598-2



a) b) c)

Figure 1. Oil reservoir viewer showing one server-rendered and two client-rendered images of automatically generated and slightly rotated proxy models. For (b), parameters are chosen to produce the best possible image, and in (c) we want to highlight artifacts and implementational details. See the text for further discussion.

visualization, see [8]. Another approach is that of [9]. They use websockets where we use the http protocol, a more significant difference is that we get away with transferring a lot less data due to our adaptive compression ratio depending on continuous bandwidth and latency measurements. In [10], Pajak et al. considers remote rendering and streaming of frames rendered from a dynamical 3D model, we are quite agnostic to the source of imagery. Their setup requires more powerful clients than ours, due to the use of the Open Graphics Library (OpenGL) vs. Web Graphics Library (WebGL), but they will also have higher fidelity. A special case is provided for in the rendering of stereo images. In this case, the 3D disparity is limited, while the rendering cost is doubled, since two views per frame is needed. In [11], this is exploited to make a solution tailored to such stereo synthesis; performance approaches that of rendering non-stereo, with a minimization of depth disparity artifacts. Occlusion and disocclusion holes are avoided by warping quads rather than pixel and filling in with previous images.



(a) Low frame rate (b) Initial proxy model

(c) Client-generated frame (d) With texturing

(e) Screen-space-sized splats (f) With intra-splat depths

Figure 2. Bird's view of 3D model (solid lines) and proxy model renderings.

III. The Auto Proxy Algorithm

Since the depth buffer is a height map seen from the observer, it does not contain information about occluded scene elements. Our approach assumes that small transformations of this height map still will give good approximations of the scene. In Figure 2 below, a sequence of three server-rendered images (thick lines) is shown, together with intermediate client-rendered proxy models with different features that will be discussed in Section III-B.

A. The server side

The server renders the 3D model into a framebuffer, and in the process generates a depth image that we also send to the client. Since this adds to the data being sent, it must be kept to a minimum. We have found that reducing the spatial resolution of the depth image (for instance by a factor of $1/16$) only degrades the proxy model imperceptibly. We also encode each depth value in the range $[0, 1]$ as a 16 bit fixed point number, and the bundling of the depth image with the RGB image then imposes just a small data overhead. Further compression may bring this down even more, but the cost of compression/decompression must also be considered. One proposed solution is to be found in [12], which promises to be fast both for the compression and decompression stages.

B. The client side

When the client receives an RGB and depth image, together with view transformations, it builds a proxy model from this. This model can then be transformed and rendered directly, or it can be combined with other proxy models the client already has in store, see Section III-B2.

As indicated by Figure 2 (b), the received height map does not allow us more than concluding where a set of points belong on the 3D model, i.e., we have little topologic information. Since the information from the depth map can only contain the foremost point along any ray from the observer, it is said to be in 2.5D, as opposed to 3D. The simplest thing the client can do, is just to transform and render the set of 3D points with color sampled from the server-rendered image, as is illustrated in Figure 2 (b).

Rendering a 3D point for each depth fragment available may tax a thin client, and it may be necessary to use a smaller number of primitives and instead render each with a larger number of pixels on the client side, a *splat*. A set of such splats for a given depth image we call a *proxy model*. By rendering splats, we get fewer "false connections" than if we connect 3D points and reconstruct topology, but we risk getting more "holes" in our models. We can render each splat as a fixed geometry, e.g., a disk or rectangle, with the correspondig color

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-598-2

from the image, see Figure 2 (c), where we have introduced a transformation local to the client. We will briefly describe some improvements to this. Note that other possibilities include building and maintaining a 3D occupancy mesh, computing a distance field from which iso-surfaces can be extracted, etc.

1) Texturing: Each splat produces many client pixels, necessitating an "intra-splat" fragment texturing for which we need a local texture coordinate transform. A first approximation is for the client to assume that the corresponding part of the server's model is planar in a region around the given point. If this is the case, a local 2D texture transformation will provide a good approximation to the intra-splat texturing to be performed on the client. Let P_c and P_s be projection matrices on the client and server, respectively, and M_c and M_s corresponding view matrices. For the splat centered in $\mathbf{v}_{i,j} = (x_j, y_i, z_{i,j})$, to be centered on the client's canvas at screen coordinate $\mathbf{p}_{i,j} = P_c M_c M_s^{-1} P_s^{-1} \mathbf{v}_{i,j} = U\mathbf{v}_{i,j}$, the texture coordinate transformation to be used is,

$$T = \begin{pmatrix} \frac{1}{n_x} & 0 \\ 0 & \frac{1}{n_y} \end{pmatrix} \begin{pmatrix} \mathbf{s}_x & \mathbf{s}_y \end{pmatrix}^{-1} \begin{pmatrix} \frac{w}{n_x} & 0 \\ 0 & \frac{h}{n_y} \end{pmatrix} = A \begin{pmatrix} \mathbf{s}_x & \mathbf{s}_y \end{pmatrix}^{-1} B,$$

where A maps the "client's splat region" (in $[0,1]^2$) to the corresponding texture area, $\begin{pmatrix} \mathbf{s}_x & \mathbf{s}_y \end{pmatrix}^{-1}$ maps the "client's screen space splat area" to $[0,1]^2$, and B provides a scaling factor to fill the *viewport* of size $w \times h$ with $n_x \times n_y$ splats laid out in a grid, when $M_c = M_s$. We obtain $\mathbf{s}_x$ and $\mathbf{s}_y$ by evaluating proxy model positions followed by perspective division and transformation into window coordinates,

$$\mathbf{p} = U \begin{pmatrix} x \\ y \\ d \\ 1 \end{pmatrix}, \mathbf{p}_{x+\Delta x} = U \begin{pmatrix} x + \Delta x \\ y \\ d_{\Delta x} \\ 1 \end{pmatrix} \text{ and } \mathbf{p}_{y+\Delta y} = U \begin{pmatrix} x \\ y + \Delta y \\ d_{\Delta y} \\ 1 \end{pmatrix},$$

where $d = 2D(x,y) - 1$, $d_{\Delta x} = 2D(x+\Delta x, y) - 1$, and $d_{\Delta y} = 2D(x, y+\Delta y) - 1$ are depths sampled from the received depth buffer D and transformed to $[-1, 1]$. With a w-component set to one, we have in effect done a perspective division, so that we are in clip space and multiplication with U is appropriate. Note that Δx and Δy are not unique, we use

$$\Delta x = n_x / \text{width}(D) \quad \text{and} \quad \Delta y = n_y / \text{height}(D),$$

but something larger may also be used. These are used for looking up depth image samples, and the calculation of $\mathbf{s}_x$ and $\mathbf{s}_y$ is really a gradient approximation, so we should not make them too large either.

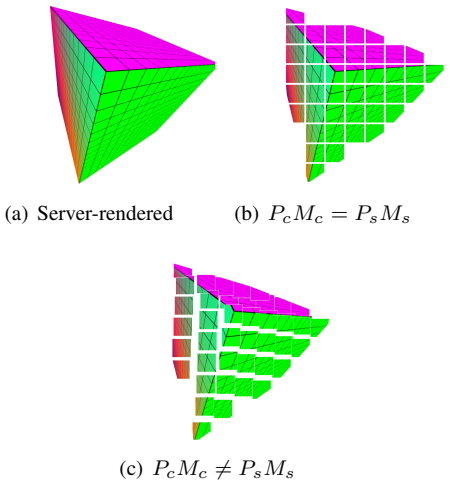
This leaves us $\mathbf{p}$, $\mathbf{p}_{x+\Delta x}$ and $\mathbf{p}_{y+\Delta y}$ in clip coordinates, and from this we get corresponding window coordinates $\mathbf{s}$, $\mathbf{s}_{x+\Delta x}$ and $\mathbf{s}_{y+\Delta y}$, from which we finally get splat spanning vectors

$$\mathbf{s}_\delta = \mathbf{s}_{x+\delta} - \mathbf{s} = \frac{L}{2\delta} \left(\frac{\mathbf{p}_{x+\delta}.xy}{\mathbf{p}_{x+\delta}.w} - \frac{\mathbf{p}.xy}{\mathbf{p}.w} \right),$$

with $\delta = \Delta x$ or $\delta = \Delta y$, L is viewport size, either w or h, and we have used the "shader notation" for vector components. The client renders a large `glPoint` for each splat, with texture coordinates $(s,t)'$, and each fragment then looks up the server-rendered image at position

$$\begin{pmatrix} s \\ t \end{pmatrix} + T \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} s \\ t \end{pmatrix} + T \left(\texttt{glPointCoord} - \tfrac{1}{2} \right)$$

where (u,v) are "intra-splat texture coordinates". When the assumption that the geometry is locally planar does not hold, e.g., if the splats are very large, or they originate from a curved or non-smooth part, this may look slightly odd, see Figure 3. The figure shows splats for which depths are sampled on different planar regions at an angle to one another, causing contortions when large splats cover more than one such planar region.



(a) Server-rendered (b) $P_c M_c = P_s M_s$

(c) $P_c M_c \neq P_s M_s$

Figure 3. Notice the corner and edges, parameters are chosen to display effects of different planes meeting at edges.

When the geometry is not planar, T produces the wrong result for parts of a splat. Two ways to remedy this could be to choose either more and smaller splats, or introduce more complex texture transformations. Note that using a more sophisticated texture coordinate transform may amount to performing the same work as for more and simpler transforms. The latter may be regarded as exactly a better "global" texture transform implementation.

2) Other splat considerations: **Splat sizing** Each splat should be rendered into a number of client pixels according to the new splats' 3D position. To achieve this, we use the vectors $\mathbf{s}_{\Delta x}$ and $\mathbf{s}_{\Delta y}$ from the previous section, and in addition, we scale the splats up a bit so that they overlap. Hence, we can render rectangular window-aligned splats with less risk of getting uncovered areas when interactively rotating and scaling the model on the client. In most cases this removes the problem visualized in Figure 3.

Splat depth fragments For larger splats, it makes sense to also compute and use depth fragments on the client. The "intra-splat" depth values can easily be fetched from the depth image, just as the texture is looked up for color. Not all clients support this WebGL extension.

Splat set replacement algorithms We mainly concern ourselves with proxy models defined as sets of splats, and it makes sense to keep a set of such models on the client, then we may combine them to cover larger ranges of client-side transformations. One can imagine a plethora of *splat set replacement algorithms.* We have tested three approaches that all retain a constant number of proxy models. The first is to replace the one with a viewing direction differing the most from the newly received model. The second simply replaces

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-598-2

the oldest one in store. The third replaces a proxy model k if the replacement results in the following objective function being reduced,

$$\text{coverage} = \sum_{i=1}^{n} \sum_{j \neq i}^{n} (\angle(\mathbf{camdir}_i, \mathbf{camdir}_j)^2, \qquad (1)$$

where n is the number of proxy models available, **camdir** is the direction in which the camera was looking when a particular model was generated, and the model to be replaced is the one that minimizes (1) after replacement.

C. Auto-tuning

It is important that the process of generating and sending proxy model data from the server itself does not slow down transmission. There are mainly three sources of delay for server-rendered images to the client; high latency, low bandwidth, and slow server-rendering. In the first case, it seems prudent to have a good proxy model on the client, which can be used for longer time and for a wider range of client-side transformations. In the two other cases, it is important for the proxy model generation/transmission to be cheap/fast, both in order to get the proxy model to the client and keep from delaying the server-image more than necessary. These demands are not always compatible.

We have adopted an adaptive specification of proxy model data from the server involving a more light-weight image (lossy Joint Photographic Experts Group (JPEG) compression with adaptive quality control) while interaction is ongoing. Also, reduced resolution of the depth buffer sent from the server is used. Another possibility is to let the client dynamically set the number of splats, number of proxy models, etc.

IV. Results and discussion

We have tested the proposed algorithms through the Tinia framework [13], which is a programming framework for setting up and managing client/server based interactive visualization applications. As client, we use Google Chrome, code is written in standard Javascript/WebGL. The auto-proxy implementation is invisible to the application, so all existing Tinia-applications will have the feature available. The algorithm is minimally intrusive in that only the depth buffer will have to be added to the rendering output of the application. We have tested several smaller test cases, but also on a larger oil reservoir viewer.

The reservoir viewer utilizes several OpenGL Shading Language (GLSL) shaders to render reservoir cells, boundaries, tubular wells, etc., and visualizes a 3D model that is not trivial to reduce in complexity. It is typically also very large, so rendering it on a thin client is prohibitive. With the automatic proxy model, we obtain interactive frame rates with limited connection from a lightweight client. For a comparison of a server-rendered image and a client-rendered proxy model that is slightly rotated on the client, see again Figure 1. In Figure 1 (a), the full server-rendered image is shown, and in (b) and (c), a slightly (about 10 degrees) rotated proxy model is rendered on the client. In (b), parameters are chosen for best possible results, while in (c), we want to highlight effects, so it uses a small number of non-overlapping large splats. One can easily spot areas not well covered, and areas where the texture coordinate transform T does not produce optimal results.

V. Conclusion and future work

The most attractive feature of the method described is the automated generation of the proxy model. Problems with compression and simplification of existing geometries is bypassed altogether. The automatic proxy model implementation is invisible to the application. There are several directions in which we would like to follow up and improve this concept, a couple of these are,

- **Proxy model replacement algorithms** Obtaining good results with a minimal set of proxy models.
- **Depth compression** We would like to ivestigate other approaches than just truncation, see, e.g., [12].
- **Deferred shading** With a normal map more advanced shading could be done on the client. Such a map could also be constructed from the depth map.

References

[1] SINTEF, "Cloudviz — direct visualization in the cloud," Project website: https://www.sintef.no/projectweb/heterogeneous-computing-expired/projects/cloudviz/, January 2010, website, retrieved: 2017-10-18.

[2] CloudFlow: Computational cloud services and workflows for agile engineering. Seventh Framework Programme (FP7) under grant agreement number 609100. Website, retrieved: 2017-10-18. [Online]. Available: http://eu-cloudflow.eu/ (2017)

[3] M. M. Oliveira, "Image-based modeling and rendering techniques: A survey." RITA, vol. 9, no. 2, 2002, pp. 37–66.

[4] P. Verlani and P. J. Narayanan, "Proxy-based compression of $2\frac{1}{2}$-d structure of dynamic events for tele-immersive systems," in Proceedings of 3D Data Processing, Visualization and Transmission (3DPVT), Atlanta, GA, USA, June 18-20 2008.

[5] E. Teler, "Streaming of complex 3d scenes for remote walkthroughs," Master's thesis, School of Computer Science and Engineering, The Hebrew University of Jerusalem, December 2001.

[6] I. Yoon and U. Neumann, "Web-Based Remote Renderingwith IBRAC (Image-Based Rendering Acceleration and Compression)," Computer Graphics Forum, 2000, pp. 321–330.

[7] W. R. Mark, L. McMillan, and G. Bishop, "Post-rendering 3d warping," in Proceedings of the 1997 Symposium on Interactive 3D Graphics, ser. I3D '97. New York, NY, USA: ACM, 1997, pp. 7–ff. [Online]. Available: http://doi.acm.org/10.1145/253284.253292

[8] A. Kolb, E. Barth, R. Koch, and R. Larsen, "Time-of-flight sensors in computer graphics (state-of-the-art report)," in Proceedings of Eurographics 2009 - State of the Art Reports. The Eurographics Association, 2009, pp. 119–134, retrieved 2017-10-18. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?5801

[9] C. Altenhofen, A. Dietrich, A. Stork, and D. Fellner, "Rixels: Towards secure interactive 3d graphics in engineering clouds," Transactions on Internet Research (TIR), vol. 12, no. 1, Jan. 2016, pp. 31–38.

[10] D. Pajak, R. Herzog, E. Eisemann, K. Myszkowski, and H.-P. Seidel, "Scalable remote rendering with depth and motion-flow augmented streaming," Computer Graphics Forum, vol. 30, no. 2, 2011, pp. 415–424. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2011.01871.x

[11] P. Didyk, T. Ritschel, E. Eisemann, K. Myszkowski, and H.-P. Seidel, "Adaptive image-space stereo view synthesis," in Vision, Modeling and Visualization Workshop, Siegen, Germany, 2010, pp. 299–306.

[12] P. Lindstrom, "Fixed-rate compressed floating-point arrays," IEEE Trans. Vis. Comput. Graph., vol. 20, no. 12, 2014, pp. 2674–2683. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2014.2346458

[13] C. Dyken et al., "A framework for opengl client-server rendering," 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, 2012, pp. 729–734.

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-598-2