



REGULAR PAPER

Michael Becher · Moritz Heinemann · Thomas Marmann · Guido Reina ·  
Daniel Weiskopf · Thomas Ertl

## Accelerated 2D visualization using adaptive resolution scaling and temporal reconstruction

Received: 23 December 2022 / Revised: 4 March 2023 / Accepted: 10 April 2023 / Published online: 8 July 2023  
© The Author(s) 2023

**Abstract** Data visualization relies on efficient rendering to allow users to interactively explore and understand their data. However, achieving interactive frame rates is often challenging, especially for high-resolution displays or large datasets. In computer graphics, several methods temporally reconstruct full-resolution images from multiple consecutive lower-resolution frames. Besides providing temporal image stability, they amortize the rendering costs over multiple frames and thus improve the minimum frame rate. We present a method that adopts this idea to accelerate 2D information visualization, without requiring any changes to the rendering itself. By exploiting properties of orthographic projection, our method significantly improves rendering performance while minimizing the loss of image quality during camera manipulation. For static scenes, it quickly converges to the full-resolution image. We discuss the characteristics and different modes of our method concerning rendering performance and image quality and the corresponding trade-offs. To improve ease of use, we provide automatic resolution scaling in our method to adapt to user-defined target frame rate. Finally, we present extensive rendering benchmarks to examine real-world performance for examples of parallel coordinates and scatterplot matrix visualizations, and discuss appropriate application scenarios and contraindications for usage.

**Keywords** Information visualization · Rendering · Temporal reconstruction · Resolution scaling

---

This paper is an extended version of our previously published work as reported by Becher et al. (Proceedings of the 15th international symposium on visual information communication and interaction, New York, 2022).

**Supplementary Information** The online version contains supplementary material available at <https://doi.org/10.1007/s12650-023-00925-3>.

---

M. Becher (✉) · M. Heinemann · T. Marmann · G. Reina · D. Weiskopf · T. Ertl  
VISUS, University of Stuttgart, Allmandring 19, 70569 Stuttgart, Baden-Württemberg, Germany  
E-mail: michael.becher@visus.uni-stuttgart.de

M. Heinemann  
E-mail: moritz.heinemann@visus.uni-stuttgart.de

T. Marmann  
E-mail: st141753@stud.uni-stuttgart.de

G. Reina  
E-mail: guido.reina@visus.uni-stuttgart.de

D. Weiskopf  
E-mail: daniel.weiskopf@visus.uni-stuttgart.de

T. Ertl  
E-mail: thomas.ertl@visus.uni-stuttgart.de

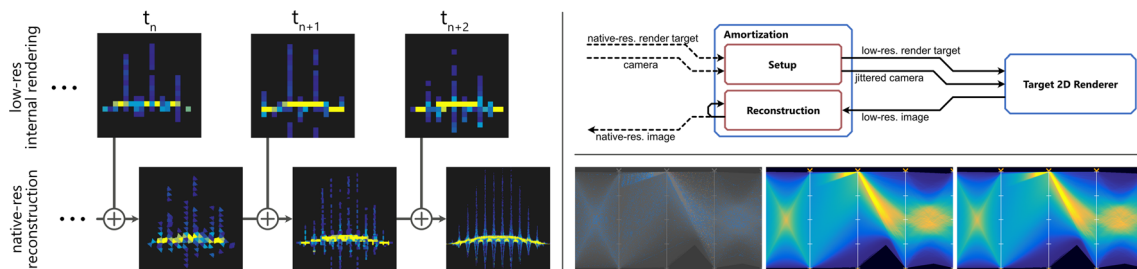
## 1 Introduction

Interactivity has always been an important part of exploring, manipulating, analyzing, and understanding data visualizations. To facilitate gaining insight and make interaction responsive, renderers need to quickly react to user input, such as camera movement or scene changes in the form of data filtering and brushing. In practice, however, rendering times increase for large datasets or display resolutions, to a degree where interactivity is hard to guarantee. For example, we have observed performance problems with our OpenGL 2D information visualization systems when viewing large datasets, especially as displays with resolutions above 1080p are becoming more and more common. In multi-view applications that combine 3D and 2D rendering, we noticed that the 2D visualizations are often the limiting factor for overall performance and responsiveness of interactive brushing and linking. Investigating the problem revealed that even though only simple geometric objects, such as points and lines, are used, the sheer amount of vertices and generated fragments in 2D visualizations becomes a bottleneck for rendering. In short, rendering performance has become an increasingly relevant issue for 2D image synthesis in typical information visualizations.

To improve response times, different approaches can be employed. One obvious solution is to thin out the data shown, rendering only data items that contribute noticeably to the final visual result. However, we are specifically interested in filtering and brushing of all individual data points. Naive sub-sampling is therefore not an option and more advanced filter techniques would require more knowledge about the renderer and the data, which conflicts with our additional goal of having a renderer-agnostic approach. Alternatively, it is possible to pre-render 2D visualizations into a high-resolution texture atlas. Exploration of the dataset is then possible by simply navigating this texture atlas. However, manipulations by filtering, brushing, or zooming in on the visualization might require us to re-render the texture atlas. Interactions therefore have inconsistent response times or latency, which should be avoided for good user experience (Shneiderman and Plaisant 2005).

An approach commonly used in real-time graphics to reliably achieve better performance without sacrificing scene detail is to simply lower the rendering resolution and thereby reduce the load on the fragment stage. To minimize the impact on image quality, more sophisticated techniques amortize the rendering cost over multiple frames by reducing the rendering resolution for individual frames and then reconstructing a native-resolution image by reprojecting samples from previous frames. This approach does not inherently reduce quality for static images, since samples of the low-resolution rendering can be jittered over time to reconstruct all image details. Ideally, visual quality is only impacted during camera movement or for changes within the scene.

For our proposed technique, we adapt this approach to the special case of 2D visualization with orthographic projection. Due to the lack of parallax, we can reconstruct native-resolution images perfectly for large parts of the screen even during camera movement, as shown in Fig. 1. Only scene changes and zooming operations cause visual artifacts from reusing outdated samples or from missing image information. Generally, it is preferable to sacrifice visual quality during movement and changing scenes, due to the inherent loss of detail perception of the human eye in such situations (Ludvig and Miller 1958; Miller 1958), which is compounded by technical limitations of displays (response time, ghosting, smearing, afterglow, etc.). Furthermore, since our approach works purely in image space, it is renderer-agnostic and



**Fig. 1** *Left:* Rendering at lower internal resolution improves responsiveness of interactive visualizations. Our amortized rendering method reconstructs a native-resolution output image each frame from the low-resolution rendering and the reprojected output image of the previous frame. *Upper right:* Our amortized rendering pipeline: An unmodified renderer is given a low-resolution framebuffer and a camera jittered in a specific sampling pattern to sample all native-resolution pixel centers over time. *Lower right:* Native-resolution rendering (right), our result (middle), and per-pixel image differences with CIELAB  $\Delta E^* < 2$  highlighted in blue,  $\Delta E^* < 4$  highlighted in orange, and  $\Delta E^* > 4$  highlighted in red (left)

can be implemented as a generic (black-box) pre-processing and post-processing step for unmodified, existing 2D renderers.

We contribute a progressive rendering method for 2D visualizations that achieves faster response times during interaction and converges to the ground truth image. It can automatically adapt the internal rendering resolution to match a user-defined performance target as close as possible. We specifically make use of the properties of 2D orthographic projection and re-project fragments from the previous frame to perfectly reconstruct large parts of the image even during camera movement. We evaluated the effectiveness of our method and its performance behavior through a number of benchmarks, using both dedicated and integrated GPUs, different resolution scaling strategies, and present and discuss our findings. This paper is an extended version of our work previously presented at the 2022 International Symposium on Visual Information Communication and Interaction (VINCI) (Becher et al. 2022) and takes material from this earlier article. The additions include:

- New modes for scaling the internal rendering resolution in either horizontal or vertical direction to complement the isotropic scaling.
- An algorithm for automatically adapting the internal rendering resolution to match a user-defined performance target.
- Two additional benchmark scenarios with integrated GPUs and device-specific native resolution.
- An updated selection of GPUs for the parameter study. All benchmarks were re-run using the new set of GPUs and including the new scaling modes.

## 2 Related work

Re-using data over several rendered frames has been investigated for both interactive and offline rendering. The same approaches serve both for reducing frame times (Adelson and Hodges 1995) and for improving output quality if performance is not an issue (Yang et al. 2009). To increase performance in video games and other interactive environments, checkerboard rendering is widespread. This method became popular with the appearance of the Playstation 4 because this method was already implemented in the hardware and therefore available to developers with little effort. Rendering at a lower resolution and using several buffered frames to satisfy performance requirements was utilized, for example, in *Rainbow Six Siege* (Mansouri 2016) and the Unreal Engine (Karis 2014). The approach was further accelerated by determining the sample positions with the help of multi-sampling anti-aliasing (MSAA) to benefit from hardware optimization (McFerron and Lake 2018). Compared to our approach, these methods are primarily designed for 3D perspective rendering and stricter requirements for image quality and stability. Thus, they need to rely on different strategies for image reconstruction, such as depth-based and velocity-based sample rejection, and are less aggressive in reducing the internal rendering resolution.

Another form of decoupled rendering (Xiao et al. 2018) achieves “coarse pixel shading,” where the shading rate is lower than the visibility sampling rate. Various filters are then used to determine which color a pixel should actually have with the aid of the visibility sample. This approach accelerates rendering even further compared to checkerboard rendering, with similar image quality. Recently, an orthogonal approach has been explored: using lower-resolution images that are then up-scaled to the target framebuffer resolution using machine learning (Xiao et al. 2020; Liu 2020). For a more detailed overview of temporal coherence methods, we direct the reader to the survey of Scherzer et al. (2011) and to the recent survey on temporal anti-aliasing by Yang et al. (2020).

Beyond distributing rendering cost across frames in image space, Limberger et al. (2018) describe progressive, multi-frame sampling techniques for various graphical effects, including depth of field and shadow mapping. Compared to a purely image-space-based approach, this has to be adapted to and integrated with the rendering implementation.

Progressive rendering has also been explored in visualization. Heinrich et al. (2011) propose a splatting algorithm to handle rendering of large data sets for parallel coordinates and scatterplot matrix visualizations. Their algorithm provides higher refresh rates at the cost of images quality, which is progressively refined over time. Frey et al. (2014) present a model for spatiotemporal error that is used to control rendering parameters to dynamically balance image quality and response time of a progressive volume renderer. In comparison with our method, the error-based control requires at least some level of interaction with the rendering implementation. However, dynamically adjusting quality to meet a performance target is an idea

we also would like to address in future work. Another approach for improving response time is frameless rendering (Bishop et al. 1994), which completely decouples the displayed image from the samples rendered per frame and has been applied to volume visualization by Petkov and Kaufman (2016).

### 3 Amortization algorithm and implementation

Our method aims to improve response times for interactive 2D visualizations by reducing internal rendering resolution, while still reconstructing a native-resolution output image each frame. For the reconstruction, the low-resolution rendering is combined with the native-resolution output of the previous frame, which is reprojected to compensate for camera movement between frames. By jittering the camera position in each frame, all pixel positions of the native resolution are progressively sampled by the low-resolution rendering and the image converges to the original quality over time. The sampling strategy that controls the camera jitter needs to be carefully designed as it can greatly impact how fast image quality convergences.

Our method aims to improve dynamic scenarios including user interaction instead of focusing on the static case, where caching the rendering result would be a straightforward solution. Furthermore, our goal is to provide a renderer-agnostic method that should be straightforward to be adopted to any visualization framework. Overall, our amortization pipeline, as also shown in Fig. 1, consists of three steps:

1. Prepare a low-resolution framebuffer and calculate camera jitter (Subsect. 3.1).
2. Execute the original target renderer.
3. Reconstruct the native-resolution image by reprojecting the last image and integrating the new samples from the current low-resolution rendering (Subsect. 3.2).

#### 3.1 Setup and sampling

To set up amortization, the down-scaled internal rendering resolution is calculated by dividing the width and height of the current native display resolution by the user-defined amortization level  $a$ . The internal framebuffer  $U$  is then resized to the calculated resolution, if necessary, and set as the render target for the target renderer. The number of newly rendered pixels per frame is thus reduced to  $1/a^2$ . Further, this means that every pixel of the low-resolution image represents an  $a \times a$  area of native-resolution pixels.

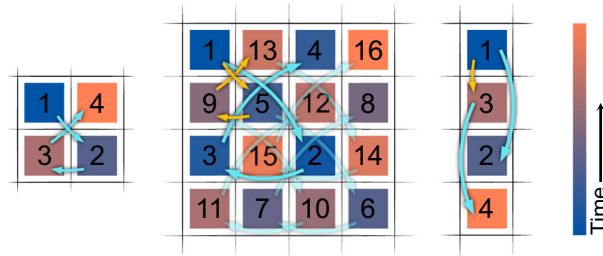
Since now only a single pixel of each  $a \times a$  region within the native-resolution image is rendered each frame, the screen positions sampled by the low-resolution render target need to change in each frame, to sample all pixel positions of the native-resolution image over time. This is accomplished by moving the camera in a pattern of sub-pixel offsets (w.r.t. to  $U$ ) within the image plane, so that pixel centers of  $U$  align with the pixel centers of each block of  $a \times a$  pixels of the native-resolution image. Camera offsets for each point of that pattern can be calculated for normalized device coordinates:

$$\mathbf{o}_{ij} = \begin{pmatrix} (2i - a + 1)/w \\ (2j - a + 1)/h \end{pmatrix}, \quad (1)$$

with  $a$  being the level of amortization and indices  $i$  and  $j$  being grid positions within the  $a \times a$  area (therefore,  $0 \leq i, j \leq a - 1$ ), and  $w$  and  $h$  the width and height of the native-resolution image. If the native display resolution is not dividable by  $a$ , we assume a resolution of the next multiple of  $a$ . In that case, extra pixels are rendered when jittering the camera, but these can be ignored during reconstruction since they are outside of the visible screen area.

To achieve a good approximation of the final image with as few frames as possible, it is important to distribute the samples within the  $a \times a$  block equally over time and to maximize the distance between consecutive samples both within each block and across blocks. We propose a pattern that subdivides each block into four quadrants and takes turns in picking a sample location from each quadrant. Quadrants are recursively subdivided to pick samples within the quadrants in each turn. Furthermore, we prefer to move diagonally to maximize distances when iterating quadrants. This pattern is illustrated for  $a = 2$  and  $a = 4$  in Fig. 2 (left and middle). If  $a$  is not a power of two, we use the next power of two to generate the pattern and cut out an  $a \times a$  area.

So far, we only considered scaling down image resolution evenly in horizontal and vertical direction. Alternatively, it possible to choose independent scaling factors for width and height and still retain a direct



**Fig. 2** Left: Our basic sampling pattern for  $a = 2$  moves diagonally twice and horizontal once. Middle: For  $a > 2$ , we subdivide into quadrants and use a recursive scheme. At the lowest level, we take a sample from each quadrant (teal arrows), again moving diagonally twice and horizontal once. At higher recursion levels, we move the start position of the large pattern on a smaller scale following the same pattern (yellow arrows). With this recursion scheme, we can extend the sampling pattern to arbitrary power-of-two sizes. Right: When scaling the resolution only in vertical or horizontal direction, we sample a 1D pattern using the same scheme but collapsed on the 1D domain

mapping between pixels of the native-resolution display image and the low-resolution rendering. While we are still restricted to integer factors in each direction, this allows for a less aggressive reduction of image information generated with each newly rendered frame. Furthermore, for specific scene content, it can be beneficially to prioritize image information along either horizontal or vertical direction. For example, parallel coordinates often feature a large amount of lines oriented along the horizontal direction and, therefore, image quality benefits from horizontal scaling (see Fig. 13).

To investigate how scaling along horizontal and vertical directions effects performance and quality, we also consider the two modes *Horizontal* and *Vertical*, to complement the *Isotropic* mode described above. These map either a  $1 \times a$  or  $a \times 1$  region within the native-resolution image to a single low-resolution pixel. The 1D sampling pattern for distributing samples within blocks is computed analogously to the 2D blocks. It recursively subdivides the blocks into half segments and takes turns in picking a sample location from each segment. The sampling pattern for  $1 \times 4$  is illustrated in Fig. 2 (right).

### 3.2 Reconstruction

For the reconstruction, we use two textures  $A$  and  $B$  at native resolution:  $A$  contains the reconstructed image of the previous frame and  $B$  stores the current reconstruction. In addition, there are two position textures  $A_p$  and  $B_p$  that hold the world-space positions of corresponding samples of  $A$  and  $B$ , respectively. After each reconstruction,  $A$  and  $B$  (and  $A_p$  and  $B_p$ ) are swapped for the next frame (i.e., serving as ring buffers for ping-pong rendering).

To obtain the pixels of the native-resolution image in  $B$  we need to distinguish two cases. First, in each  $a \times a$  block of pixels, one pixel center aligns with a pixel of the low-resolution image  $U$ . These pixels can be directly copied from  $U$  to  $B$ . For all other pixels in  $B$ , we look up the best approximation, i.e., the nearest sample within  $A$  and  $U$ . The nearest sample in  $A$  is found using the reprojection  $\Psi$  for each pixel position from the current frame to the last frame according the camera changes and reading the corresponding position from  $A_p$ . Further, we can project the pixel position for each pixel in  $B$  to world space to obtain the world-space distance

$$d_A = \text{dist}(A_p(\Psi(x, y)), \Omega(x, y)) , \quad (2)$$

where  $\Omega$  is the corresponding projection from the image space of  $B$  to world space. For orthographic 2D rendering, the reprojection is:

$$\tilde{p} = M_t \cdot \left( \frac{2x}{w} - 1, \frac{2y}{h} - 1, 0, 1 \right)^T , \quad (3)$$

$$\Psi(x, y) = \begin{pmatrix} (0.5 \cdot \tilde{p}_1 + 0.5) \cdot w \\ (0.5 \cdot \tilde{p}_2 + 0.5) \cdot h \end{pmatrix} , \quad (4)$$

with width  $w$  and height  $h$  of the native-resolution image and the reprojection matrix  $M_t = MVP_{t-1} \cdot MVP_t^{-1}$ , computed from the model-view-projection matrices at time  $t$  and  $t - 1$ , respectively. The reprojection function first transforms pixel coordinates to clip coordinates and then applies the

inverse model-view-projection transform of the current frame to compute the world coordinates. By then applying the model-view-projection of the previous frame and transforming the resulting clip coordinates back to pixel coordinates, we can find out where that part of the scene was visible in the previously rendered image. Note that, depending on the camera changes and pixel positions, this reprojection  $\Psi$  might not always yield usable pixel positions (e.g., it might be outside the framebuffer); in this case, we assume a distance of infinity to avoid using this position.

We find the nearest sample within  $U$  by minimizing the distance  $d_U$  over all pixels  $(s, t)$  in  $U$ :

$$d_U = \min_{s,t} \text{dist}(\Omega'(s, t), \Omega(x, y)) , \quad (5)$$

where  $\Omega'$  maps from the image space of  $U$  to world space. Note that the position  $(s, t)$  matches exactly a pixel position within the image space of  $B$  and we, therefore, can directly calculate the nearest sample by checking the 4 possible neighbors, i.e., by direct search.

Finally, we need to compare the distances  $d_A$  and  $d_U$  and choose the nearest options, unless we can directly copy the sample from  $U$  that corresponds to the current  $a \times a$  block. Therefore, we can define the computation of a pixel in  $B$  at the location  $(x, y)$  as:

$$B(x, y) = \begin{cases} U(x/a, y/a) & \text{if } d_U = 0 \\ U(s, t) & \text{if } d_U \neq 0 \wedge d_U \leq d_A , \\ A(\Psi(x, y)) & \text{else} \end{cases} , \quad (6)$$

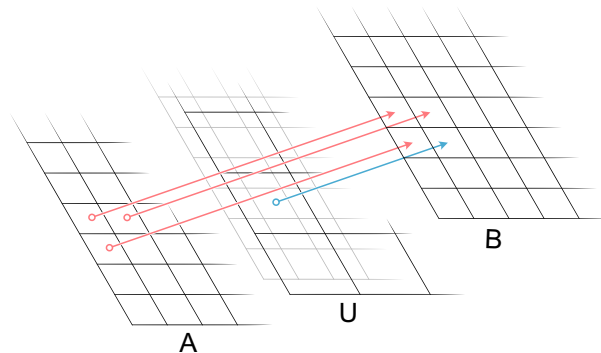
where  $x$  and  $y$  represent the pixel coordinates in the native-resolution image and  $s$  and  $t$  denote the pixel coordinates in the low-resolution image  $U$  with minimal value for  $d_U$ . An example of a simple case is shown in Fig. 3.

After the reconstruction is completed, the resulting native-resolution image is displayed on the screen and the textures are swapped internally to prepare for the next frame. As the samples within  $U$  always match exactly one pixel within  $B$ , our algorithm converges to the ground truth image at the latest after  $a^2$  consecutive frames with unchanged camera and data.

### 3.3 Adaptive resolution scaling

For many visualization techniques, rendering performance can vary greatly depending on many factors, including dataset size, the actual view on the dataset, the current GPU, or display resolution. Using a fixed amortization level is therefore often not ideal and will either not offer enough acceleration or unnecessarily decrease image quality. Having users constantly adjust the level manually would interfere with the actual data exploration task. Instead, the amortization level should be adjusted automatically based on the current rendering performance.

A similar feature is commonly employed in video games that use dynamic resolution scaling to meet a given performance target (Binks 2011; Epic Games 2023). While games usually target 30, 60, or even more frames per second (fps) to offer highly reactive player feedback and to allow for vertical synchronization



**Fig. 3** Reconstruction example for amortization level  $a = 2$ : the sample at the position corresponding to the current frame in the  $a \times a$  pattern is fetched from the low-resolution render target  $U$ . The remaining pixels of the pattern are fetched from  $A$ , assuming  $d_A < d_U$  in this example

with displays to avoid screen tearing, visualization use cases often have lower requirements for the refresh rate. Refresh rates below 30 fps or even 20 fps are often still acceptable and considered as interactive for scientific and information visualization. Our approach for automatically setting the amortization level aims to minimize the distance between current rendering performance and a user-defined performance target given in fps.

Our algorithm for adaptive auto-scaling is based on the idea of predicting frame time changes for increasing or decreasing the amortization level  $a$ , using a prediction factor  $p$ . We define the reduction factor for the frame time when increasing the amortization level as:

$$p(a) = \frac{\text{frametime}(a+1)}{\text{frametime}(a)}. \quad (7)$$

In practice, however, the frame time for  $a+1$  is unknown, unless we render an additional frame with this amortization level. Therefore, we only update  $p$  after changing  $a$ , using the frame times before and after the change. And we assume that this value for  $p$  is a reasonable approximation for the next update of  $a$ , i.e., it can be used to predict approximate frame times for  $a+1$  and  $a-1$ .

We determine in each frame the amortization level for the next frame, employing the user-defined target frame time  $t_{\text{tgt}}$  and the frame time average over the last  $m'$  frames,  $t_{\text{avg}}$ . We set  $m'$  to bound the useful size of the look-back window. It is the minimum of the number of frames since the last change of  $a$  and a user-defined value  $m$ , which we empirically set to  $m = 100$  as default. If the current average frame time is above the target frame time, we check if the predicted frame time for  $a+1$  is closer to the target frame time and update  $a$  if it is. Likewise, if  $t_{\text{avg}}$  is below  $t_{\text{tgt}}$ , we calculate the predicted frame time for  $a-1$  and update  $a$  accordingly. Let  $t_{\text{exp}}$  be the expected frame time for an increased or decreased  $a$ :

$$t_{\text{exp}} = \begin{cases} t_{\text{avg}} \cdot p & \text{if } t_{\text{avg}} > t_{\text{tgt}} \\ t_{\text{avg}}/p & \text{else} \end{cases}. \quad (8)$$

In order to avoid jumping back and forth between amortization levels with frame times that are similarly close to the target frame time, we use a modified target framerate  $t'_{\text{tgt}}$  that is biased toward keeping the current amortization level. We therefore add some target leeway if  $t_{\text{avg}}$  and  $t_{\text{exp}}$  are on opposite sides of  $t_{\text{tgt}}$ :

$$t'_{\text{tgt}} = \begin{cases} t_{\text{tgt}} + \lambda \cdot (t_{\text{avg}} - t_{\text{exp}}) & \text{if } t_{\text{avg}} > t_{\text{tgt}} \wedge t_{\text{exp}} < t_{\text{tgt}} \\ t_{\text{tgt}} - \lambda \cdot (t_{\text{avg}} - t_{\text{exp}}) & \text{if } t_{\text{avg}} < t_{\text{tgt}} \wedge t_{\text{exp}} > t_{\text{tgt}} \\ t_{\text{tgt}} & \text{else} \end{cases}, \quad (9)$$

with a bias factor  $\lambda$  that we empirically set to 0.1. The amortization level is then changed for the next frame if this decreases the expected distance from the target framerate:

$$a \leftarrow \begin{cases} a+1 & \text{if } t_{\text{avg}} > t_{\text{tgt}} \wedge |t_{\text{exp}} - t'_{\text{tgt}}| < |t_{\text{avg}} - t'_{\text{tgt}}| \\ a-1 & \text{if } t_{\text{avg}} < t_{\text{tgt}} \wedge |t_{\text{exp}} - t'_{\text{tgt}}| < |t_{\text{avg}} - t'_{\text{tgt}}| \\ a & \text{else} \end{cases}. \quad (10)$$

If  $a$  was changed, we store the current  $t_{\text{avg}}$  as  $t_{\text{avg}_{\text{last}}}$  and the value of  $a$  before the change as  $a_{\text{last}}$ . For a cooldown period of  $k$  frames, we do not update  $a$  again. With the cooldown period, we make sure to compute a new stable value for  $t_{\text{avg}}$  again, as the frame time average only considers frames rendered after the last update of the amortization level. At the same time, we want to minimize the time needed to converge to a stable amortization level. Thus, the length of the cooldown period has to be balanced between computing reliable average frame times and low latency between updates of the amortization level. A value of  $k = 10$  was sufficiently reliable in our experiments. At the end of the cooldown phase, we also update the prediction value  $p$ :

$$p' = \begin{cases} \frac{t_{\text{avg}}}{t_{\text{avg}_{\text{last}}}} & \text{if } a > a_{\text{last}} \\ \frac{t_{\text{avg}_{\text{last}}}}{t_{\text{avg}}} & \text{else} \end{cases}, \quad (11)$$

$$p = \max(0.25, \min(p', 0.95)). \quad (12)$$

The algorithm is based on the assumption that a higher amortization value will always lower the frame times. Therefore, we enforce the premise of at least 5% lower frame times for the next higher amortization

level and clamp  $p'$  to 0.95. At the same time, we assume a maximum acceleration by a factor of 4, as the number of pixels is reduced to  $1/4$  (isotropic mode) between amortization levels, and clamp  $p'$  to 0.25. Clamping  $p'$  also stabilizes the algorithm for edge cases, including measurement errors for frame times and outliers for achieved acceleration (see Fig. 6b for  $a = 6$  at a resolution of  $1280 \times 720$ ).

### 3.4 Render primitive interaction

A major side effect of our amortization method is that in the target renderers, operations that rely on absolute pixel size of the rendering resolution will be affected, and in the worst case, the resulting image will differ. One such case is the thickness of line primitives or the size of points rasterized in OpenGL or DirectX. To preserve the original image, line thickness or point size would have to be set in the sub-pixel range, which is not guaranteed to be supported in the used graphics APIs. To counteract this, the size of employed render primitives cannot be expressed in absolute pixel sizes, but should instead be given in world space for example.

Both renderers we show in this paper feature rendering modes that construct line and point geometry from triangles, as this generally avoids a range of issues with these graphic primitive types, such as limited point size or using a line width other than 1.0 in an OpenGL core profile. With triangle-based rendering, size and thickness can be varied and set to match desired pixel sizes in native resolution, instead of rendering resolution. Since the vertices of triangles can be freely set in sub-pixel range, no inherent loss of quality occurs with our method. However, vertex processing of the triangle-based geometry is more costly compared to line and point primitives. Still, this additional cost is easily offset by being able to use higher amortization levels without compromising visual quality. Additionally, we strongly recommend rendering user interface (UI) elements either as an additional layer after the amortization or also using resolution-independent primitives and shading for UI rendering. Otherwise, the amortization can result in rendering artifacts.

## 4 Results and evaluation

To evaluate the performance of our method, in particular regarding acceleration, we carried out a parameter study. Our chosen parameters cover a wide range of rendering loads and hardware to understand how each parameter affects performance. As parameters we use GPU, target renderer, dataset, resolution, amortization mode, and amortization level. The parameter values are listed in Table 1. We systematically ran all possible combinations of parameter values. In each configuration, we performed four full reconstruction cycles, so each run rendered  $4 \times a \times a$  frames. For native rendering without amortization, we measured timings for 25 frames. The rendering times for each frame were recorded and averaged for comparison. We provide tables with all frame times in the supplemental material. In this way, we follow the model of scalability in visualization according to Richer et al. (2022), with a systematic sampling of the effort function (i.e., the compute times or derived fps values and acceleration factors) depending on the parameters from Table 1.

All benchmarks were performed on a PC running Windows 10 64 bit and configured with a Ryzen 9 5900X CPU, 64 GB of RAM, and an SSD. We set the GPUs to a stable power state to avoid measurement bias from thermal throttling during our benchmarks. Note that this also prevents GPUs from using a dynamic boost clock, which can noticeably lower absolute frame times. We implemented our technique within the visualization framework MegaMol. (Gralka et al. 2019), using the C++ programming language. Generally, a rendering module within MegaMol is provided with a render target and a camera, both of which can be

**Table 1** List of investigated benchmark parameters. In total, we tested 4,620 different configurations to cover all possible combinations including the baseline performance with amortization turned off. The raw data can be found in the supplemental material

GPU	Renderer	Dataset	Resolution	Amort. Mode	Amort. Level
Intel ARC A770	<i>Parallel Coordinates</i>	<i>Concrete Beam</i>	$853 \times 480$	<i>Isotropic</i>	2
AMD Radeon RX 7900 XT	<i>Scatterplot Matrix</i>	<i>Droplets</i>	$1280 \times 720$	<i>Horizontal</i>	3
AMD Radeon RX 6900 XT		<i>Iris</i>	$1920 \times 1080$	<i>Vertical</i>	4
AMD Radeon Vega VII			$2560 \times 1440$	<i>Off</i>	5
Nvidia GeForce RTX 4090			$3840 \times 2160$		6
Nvidia GeForce RTX 3090 Ti					7
Nvidia GeForce GTX 2070 Super					8



modified by other modules before the renderer is executed, and the render target provides access to the rendering result for further processing afterward. Our technique was implemented as a module that we can simply insert in front of any 2D renderer into existing projects (see also Fig. 1).

For datasets, we chose three examples of various sizes to cover a meaningful range of rendering costs:

- *Iris*: 5 dimensions, 150 data points
- *Concrete Beam*: 22 dimensions, 160,105 data points
- *Droplets*: 17 dimensions, 1,043,168 data points

*Iris* is taken from the UCI Machine Learning Repository (Dua and Graff 2017), *Concrete Beam* is from an FEM structural mechanics simulation of a beam (containing nodes and stress tensors) (Kelleter et al. 2020), and *Droplets* are extracted physical quantities per droplet of a multiphase jet simulation as used by Heinemann et al. (2021). We picked *Parallel Coordinates Plot* and *Scatterplot Matrix* renders as typical representatives of 2D visualizations. In both cases, blending is used to render density plots. Based on the number of data points  $n$  and dimensions  $d$ , the number of rendered line primitives for *Parallel Coordinates Plot* is  $n \cdot (d - 1)$  and the number of rendered point primitives for *Scatterplot Matrix* is  $n \cdot d \cdot (d - 1)/2$ . An important measure is the achieved acceleration defined by the term  $\alpha = t_b/t_a$ , with  $t_a$  being the amortized frame time and  $t_b$  the non-amortized frame time of the otherwise identical configuration.

Please note that absolute frame times changed compared to the previous publication (Becher et al. 2022), especially on AMD GPUs. As we are focusing on the speedup for individual cards and do not aim to compare absolute performance, we did not further investigate this issue, except for double checking our benchmark setup for errors. We suspect that driver updates might be the cause for the decline in performance for our special rendering use case.

## 4.1 Performance

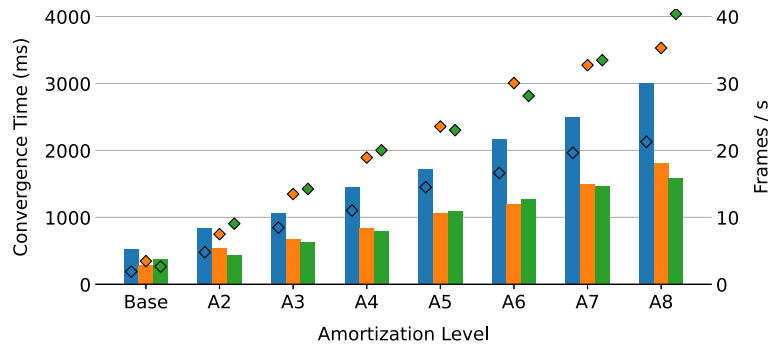
An important trade-off of our method is between response time (affecting interactivity), which is largely determined by the time it takes to render a single frame, and convergence time, which refers to the time it takes to update every visible pixel. Each amortization level further improves the response time by reducing the amount of rendered pixels per frame but needs  $a^2$  frames to converge with the *Isotropic* amortization mode or  $a$  frames with the *Horizontal* and *Vertical* modes. For the renderers we used for this paper, we found convergence times of up to two seconds to be still acceptable, especially considering image quality improves rapidly during the first half (see Fig. 10).

### 4.1.1 Systematic evaluation

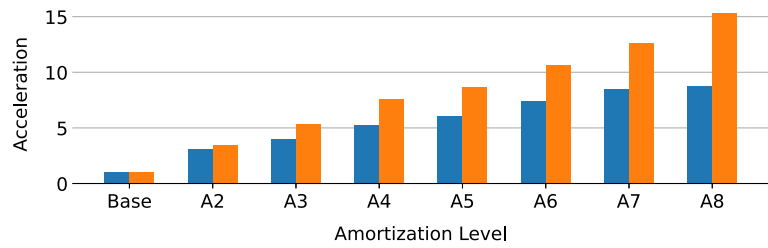
Figure 4 compares the convergence time (bars) to the frame rate (scattered diamonds) for increasing amortization level, three different GPUs, and using our largest dataset and display resolution. Rendering at native resolution, no GPU achieves frame rates above 5 fps (Base), which would be the minimum requirement for interactive applications. With our method, frame rates can be significantly improved and reach above 30 fps for two of the shown GPUs. All GPUs consistently improve frame rates across amortization levels, but convergence times are also rising increasingly with each amortization level. Results for the AMD Radeon Vega VII show how convergence time increases approximately quadratically as frame rate improvements decline with rising amortization levels.

Figure 5 compares the achieved acceleration with increasing amortization level for two different dataset sizes. Here, our middle-sized and our largest dataset are rendered using the *Parallel Coordinates* renderer at a resolution of  $3840 \times 2160$  on the Nvidia GeForce RTX 3090 Ti. Frame times range from 62.6 ms (Base) to 7.1 ms (A8) for the *Concrete Beam* dataset and from 378.3 ms (Base) to 24.8 ms (A8) for the *Droplets* dataset. The achieved acceleration is generally lower for the smaller dataset, but improves consistently for both datasets. However, for the smaller dataset, the increase in acceleration begins to decline for higher amortization levels and converges toward a potential maximum acceleration value. We observed this behavior in all configurations, but often only at amortization levels beyond  $a = 8$ , which we considered to be less relevant and did not cover in our benchmarks, as the convergence time becomes impractical at this point in most cases. Here, this suggests that rendering the larger dataset is limited by the amount of generated fragments and confirms our initial assumption about large datasets and high resolutions being potentially fragment-bound in 2D information visualizations.

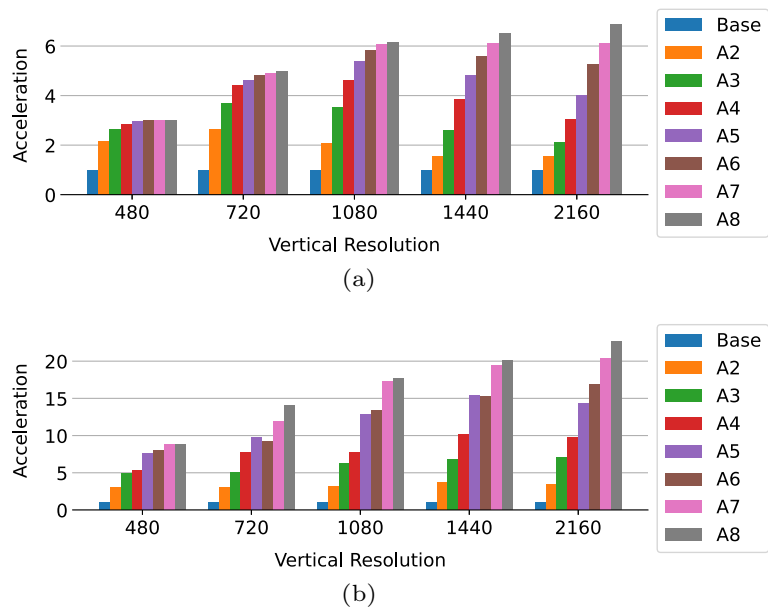
Figure 6 compares the achieved acceleration for different native display resolutions and amortization levels on the Nvidia GeForce RTX 2070 Super for two different scenarios: (a) the *Scatterplot Matrix*



**Fig. 4** Comparison of frame rates (scattered diamonds, in fps) and convergence times (solid bars, in milliseconds) for rendering the *Droplets* dataset with *Parallel Coordinates* at a resolution of  $3840 \times 2160$  using the *Isotropic* amortization mode for different amortization levels (A2–A8) and with three different GPUs: Radeon VegaVII ■, Radeon RX 6900 XT ■, GeForce RTX 3090 Ti ■



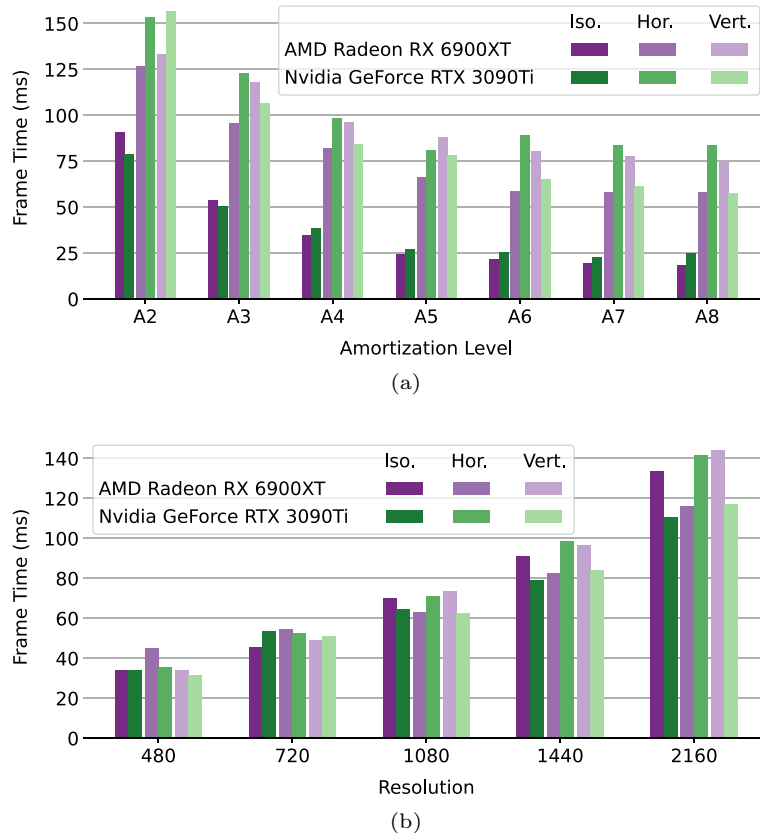
**Fig. 5** Comparing accelerations of *Concrete Beam* ■ and *Droplets* ■ datasets using the *Isotropic* amortization mode with the *Parallel Coordinates* renderer at a resolution of  $3840 \times 2160$  on the GeForce RTX 3090 Ti shows that the larger dataset achieves better acceleration



**Fig. 6** Acceleration for different amortization levels using the *Isotropic* amortization mode at different resolutions on a GeForce RTX 2070 Super for two configurations: **a** *Scatterplot Matrix* with *Concrete Beam* and **b** *Parallel Coordinates* with *Droplets*. For each resolution, bar charts are grouped and sorted left to right by increasing amortization level

renderer with our medium dataset (*Concrete Beam*) and (b) the *Parallel Coordinates* renderer with our largest dataset (*Droplets*). In both scenarios, rendering without our method performs poorly at any resolution—with frame times between 167.2 ms (6.0 fps) to 49.9 ms (20.0 fps) and 760.3 ms (1.3 fps) to 151.1 ms (6.6 fps) for (a) and (b), respectively. In (a), acceleration does not improve much beyond amortization level  $a = 4$  at the lower display resolutions. This is expected as for increasingly lower rendering resolution, performance will eventually no longer be fragment-bound and additional effects with impact on performance such as overdraw can become more pronounced. With display resolutions of  $1920 \times 1080$  and above, the use of higher amortization levels becomes more favorable and acceleration increases noticeably with each additional level. For a resolution of  $3840 \times 2160$ , absolute performance improves from 6.0 fps to 41.1 fps. Acceleration across different resolutions in (b) behaves similar to (a), but acceleration scales much better with amortization levels even at low resolutions. Generally, this configuration achieves some of the highest accelerations recorded during our benchmarks. For a resolution of  $3840 \times 2160$ , absolute performance improves from 1.3 fps to 29.7 fps. We can also see some outliers at  $a = 6$ , where the acceleration is, for some resolutions, worse than at the previous level.

Figure 7a compares the achieved frame times for the three different amortization modes and across amortization levels on both the AMD Radeon RX 6900XT and the Nvidia GeForce RTX 3090 Ti. The *Parallel Coordinates* renderer is used to render our largest dataset. As expected, the *Isotropic* amortization modes shows the best performance, as it renders fewer pixels per frame, reduced by a factor of  $a$  compared to the other two modes. We additionally expected that the *Horizontal* mode would perform better than *Vertical*, due to the horizontal layout of the parallel coordinates plot. While this is the case for the Radeon RX 6900XT, the *Vertical* mode performs better for the GeForce RTX 3090 Ti. We observed the same behavior on the other GPUs, with AMD GPUs generally favoring the *Horizontal* mode, whereas Nvidia



**Fig. 7** **a** Frame times (in ms) for different amortization modes and levels using the *Parallel Coordinates* renderer, the *Droplets* dataset, and a resolution of  $2560 \times 1440$  on two different GPUs. Note that the *Horizontal* mode performs better on the AMD card, whereas the NVIDIA card renders faster in *Vertical* mode. **b** Similar to **a**, but with constant amortization level  $a = 2$  for the *Isotropic* mode and  $a = 4$  for the *Horizontal* and *Vertical* mode, shown for different resolutions. This setup renders the same amount of pixels per frame

GPUs and the Intel GPU favor the *Vertical* mode. Further, the achieved performance of the *Horizontal* and *Vertical* modes does not strictly scale with the number of rendered pixels. For example, looking at amortization level  $a = 5$ , *Horizontal* and *Vertical* modes are roughly 3 times slower than the *Isotropic* mode, while rendering 5 times as many pixels per frame.

To further investigate this, Fig. 7b shows a comparison between *Isotropic* mode with  $a = 2$  and *Horizontal* and *Vertical* modes with  $a = 4$ . These configurations render the same number of pixels in each frame ( $1/4$  of the original resolution). Here, we observe all three modes competing in roughly the same order of magnitude. For higher resolutions, the Radeon RX 6900XT performs best using the *Horizontal* mode, whereas for the GeForce RTX 3090 Ti, the *Isotropic* and *Vertical* are closely matched, with the *Isotropic* mode showing slightly better performance for the two highest resolutions.

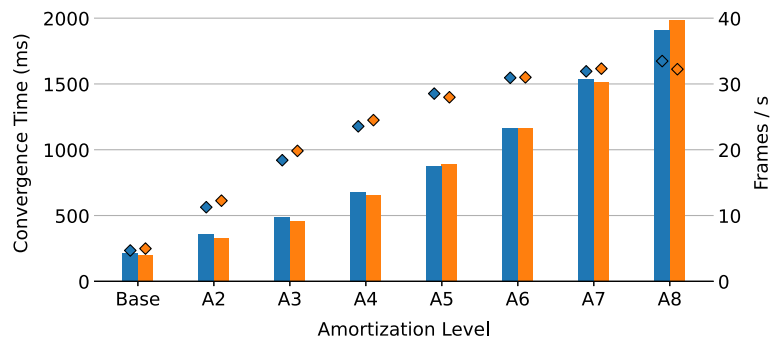
#### 4.1.2 Integrated graphics

In addition to performing a systematic performance evaluation using a variety of desktop configurations with dedicated graphics cards, we are also interested in the effectiveness and efficiency of our technique on portable systems using integrated graphics processing units (iGPUs). From our experience in collaborating with researchers from other fields, the end users of visualizations do not necessarily have access to high-end GPUs and are instead often using laptops with integrated graphics. These iGPUs offer less computing performance and especially mobile systems also have to adhere to strict power and thermal budgets. We chose two mobile computers with different iGPUs to evaluate the performance of our method: (1) An HP EliteBook 845 G7 configured with an AMD Ryzen 7 4750U that uses AMD Radeon RX Vega 7 graphics, 32 GB of shared dual-channel RAM, and an SSD. (2) A Microsoft Surface Pro 8 configured with an Intel i7 1185G7 that uses Intel Iris Xe Graphics G7 with 96EUs, 16 GB of shared dual-channel RAM, and an SSD. Note that the performance of the same iGPU can vary strongly based on the device (and its cooling design), which is why we identify the specific devices here. We benchmarked both devices using their native display resolution to closely match a real-world usage scenario where the visualizations would be displayed on the internal screen. The AMD Radeon RX Vega 7 graphics rendered at a resolution of  $1920 \times 1080$ , whereas the Intel Iris Xe Graphics G7 96EUs rendered at a resolution of  $2880 \times 1920$ . Therefore, the results shown here are neither intended for the comparison of both iGPUs nor for comparison with the desktop hardware.

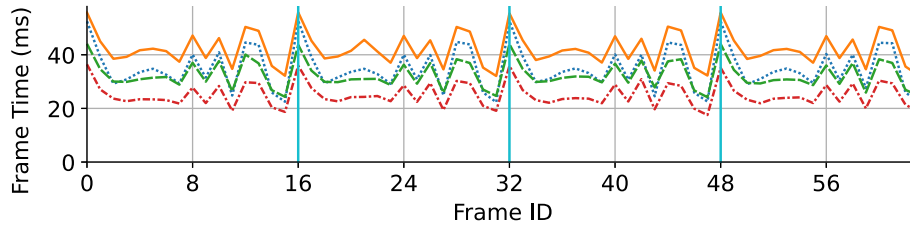
Figure 8 compares the convergence time (bars) to the frame rate (scattered diamonds) for increasing amortization level and the *Isotropic* amortization mode, using the *Concrete Beam* dataset and *Parallel Coordinates* renderer. Both devices achieve only barely interactive frame rates (around 5 fps) when rendering at native resolution. Performance can be significantly improved with our method up to amortization level  $a = 6$ , with both devices reaching 30 fps. However, both integrated GPUs do not improve much beyond this level and instead converge toward a maximum acceleration.

#### 4.1.3 Limitations

The reconstruction stage of our method adds a fixed resolution-dependent overhead to rendering (less than 1 ms for all configurations we benchmarked). If the original native-resolution rendering already performs



**Fig. 8** Comparison of frame rates (scattered diamonds, in fps) and convergence times (solid bars, in milliseconds) for rendering the *Concrete Beam* dataset with *Parallel Coordinates* using the *Isotropic* amortization mode for different amortization levels (A2–A8) and with two different mobile systems: Intel Iris Xe Graphics G7 96EUs ■ with a native resolution of  $2880 \times 1920$  and AMD Radeon RX Vega 7 ■ with a native resolution of  $1920 \times 1080$



**Fig. 9** Frame times for selected GPUs (Radeon RX 6900 XT ■ (dotted), Radeon Vega VII ■ (solid), GeForce RTX 2070 Super ■ (dashed), RTX 3090Ti ■ (dash-dotted)) of a specific scenario that causes inconsistent frame times (*Scatterplot Matrix, Concrete Beam*,  $2560 \times 1440$ , amortization mode *Isotropic*,  $a = 4$ ). The frame times form a repeating pattern with a period of  $a \times a$  frames, here marked with cyan lines. We can observe similar patterns for all GPUs and measurement runs

very well, it is possible that performance gains from our method are smaller than the additional costs for reconstruction and total frame times can even increase. Note that this implies no additional acceleration is needed in the first place. We observed such scenarios for particularly small datasets, e.g., for the smallest dataset (*Iris*) in our benchmark.

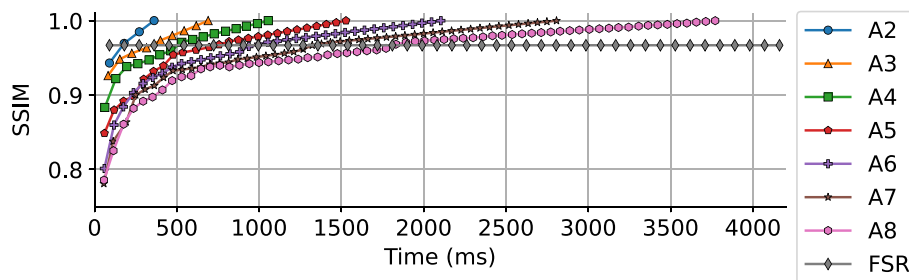
Another drawback is that we cannot guarantee constant frame times for the jittered low-resolution rendering. This happens when some image regions are significantly more costly to render than others, e.g., due to heavy overdraw caused by clustered geometry. While such structures are distributed spatially in the native-resolution image, they are also distributed temporally with our amortized method. An example of this can be seen in Fig. 9.

## 4.2 Image quality

### 4.2.1 Static image quality and convergence

Static image quality here refers to the converged image after rendering  $a^2$  low-resolution frames without moving the camera or changing the visualization. Since sampling positions are moved for each frame to exactly correspond to the center of a native display resolution pixel, the resulting image is virtually identical to the non-amortized ground truth image. The only exception are a few scattered pixels. This is due to floating-point inaccuracies in the coordinate transformations. Figure 1 shows a comparison between a native-resolution rendering and our technique using amortization level 2 and the per-pixel color difference in CIELAB  $\Delta E^*$  between them.

To better evaluate image quality of static scenes, structural similarity index measure (SSIM) (Wang et al. 2004) is used to quantify differences between the ground truth image and our reconstructed images. Figure 10 shows SSIM of each frame starting without any information from previous frames until convergence after  $a^2$  frames have been rendered. Higher amortization levels initially have lower SSIM and require more time to converge to the ground truth image, as due to lower internal rendering resolution fewer pixels are updated per frame. All levels, however, converge toward the ground truth image with an SSIM of more than 0.9999 after  $a^2$  frames.



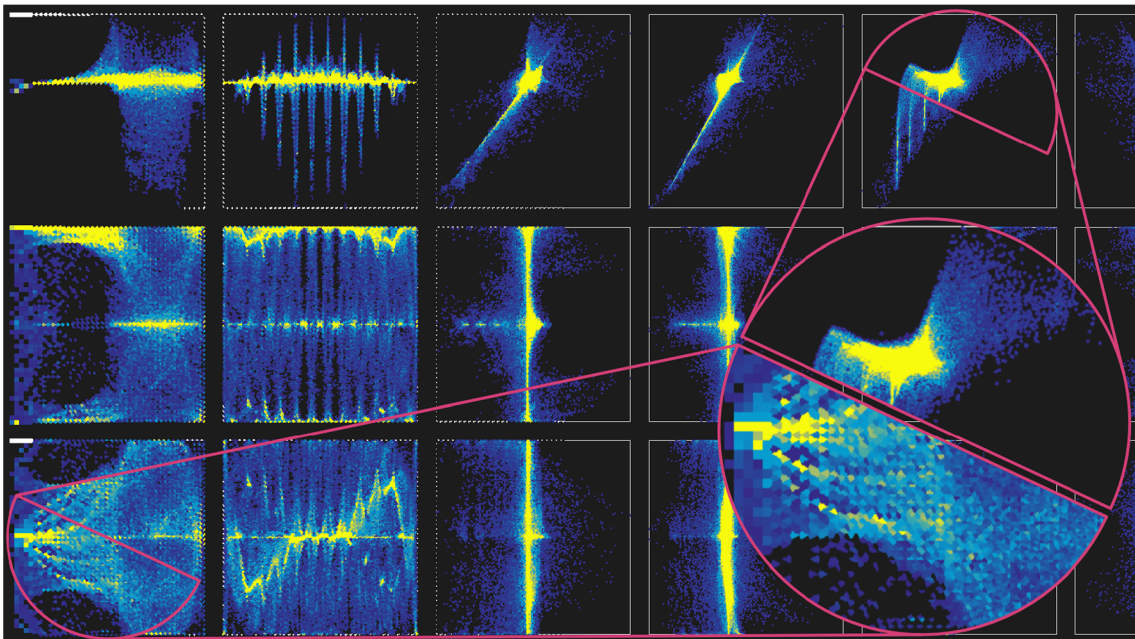
**Fig. 10** Comparison of SSIM over time after a complete refresh of the image for different amortization levels as well as for upscaling with AMD FidelityFX™ Super Resolution (Advanced Micro Devices 2022)

#### 4.2.2 Dynamic image quality

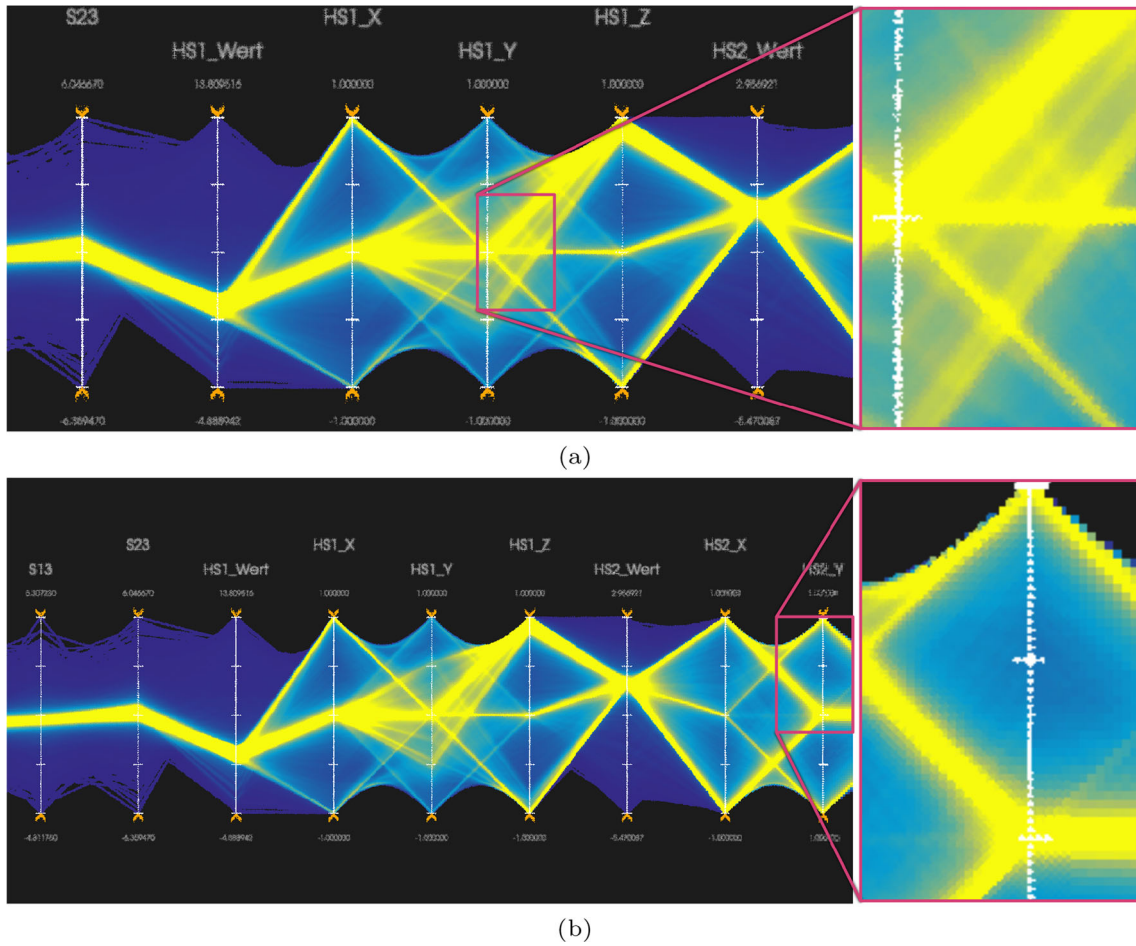
Image quality for dynamic content depends on what kind of change occurred. We differentiate between three scenarios: (1) camera translation, i.e., camera movement along the  $x$ - $y$  plane, (2) camera zoom, i.e., resizing of the view frustum, (3) scene changes, e.g., updating the shown data or transfer function.

For camera translation, we can re-project known parts of the image that remain within the view frustum without loss of quality, comparable to the static image quality. Note that this requires camera translation to be a multiple of the display pixel size in world space, which is usually the case as mouse input is given in pixels and also to keep the cursor position relative to the scene fixed during interaction. However, areas that were not visible in the prior frames can become visible at the edge of the screen. When an image region first enters the view, only samples from the low-resolution rendering are available, giving them a blocky appearance. With each consecutive frame, these regions are progressively refined and converge to the original image, as is shown in Fig. 11. Note that the overall shape of the image is mostly retained, allowing the user some level of orientation even with initially lower quality.

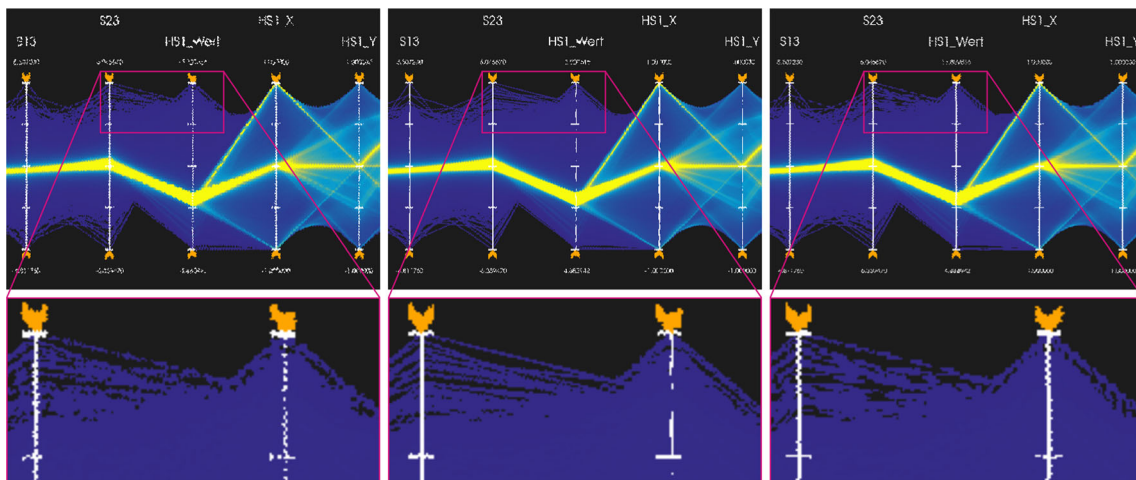
Zooming usually leads to quality loss, as it changes the sample density in world space with respect to the previous image. Samples from the previous image are now either too dense (when zooming out) or too sparse (when zooming in), and the 1:1 correspondence between textures  $A$  and  $B$  (see Subject. 3.2) is lost. The transformation  $M$  already accounts for zooming and for each pixel in the native-resolution output image, our algorithm simply chooses the nearest available sample. Thus, image quality degrades during zooming as an error in sampling positions and sampled area is introduced, as can be seen in Fig. 12. Correcting these errors, especially w.r.t. minification using, e.g., Mipmapping, is left for future work. Additionally, when zooming out, it is likely that samples are required at the border of the screen that were not previously visible. As during camera translations, these regions can only use samples from the low-resolution rendering and start off with a blocky appearance that converges to the original quality over time. Figure 13 shows a comparison between the *Isotropic*, *Horizontal*, and *Vertical* amortization modes. All modes use an amortization factor of  $a = 4$ . Note that the *Isotropic* amortization therefore updates fewer pixels per frames and generally will perform better in comparison (see Fig. 7). Due to the nature of the rendered line primitives for parallel coordinates, the *Horizontal* amortization modes preserves higher image detail for individual lines, whereas both *Isotropic* and *Vertical* modes break up lines vertically with visible gaps.



**Fig. 11** During camera translation to the left, new image regions enter the view frustum (left border). These can only be reconstructed from samples of the low-resolution rendering and appear blocky. Regions that have already been visible for a number of frames (left border to center) started to converge to the ground truth image even during persisting translational movement. Regions visible for at least  $a^2$  frames (right half) are reconstructed perfectly due to the reprojection



**Fig. 12** Image quality degrades when zooming, i.e., resizing the camera frustum, as pixels no longer cover the same area of the visualization in the current and the previous frame. **a** When zooming in, the image becomes noisy as reprojected samples generally cannot perfectly match up with the current pixel grid. **b** When zooming out, new regions come into view for which only limited image information is available, which results in a blocky appearance



**Fig. 13** Comparison of image quality for different amortization modes at amortization level  $a = 4$  during zooming in, i.e., resizing the camera frustum. As the line primitives rendered for parallel coordinates are mostly horizontal structures, the *Horizontal* amortization mode (middle) preserves more detail for individual lines, compared to the *Isotropic* (left) and *Vertical* (right) amortization modes, which both produce disconnected lines in the parallel coordinates. The vertical coordinate axes exhibit the opposite effect

Changes within the scene are naturally only reflected in newly rendered low-resolution images and not the reconstruction result of frames rendered prior to the change. Completely updating the output image requires a full cycle of  $a^2$  frames. Visually, this is similar to an animated transition between scene states. Alternative strategies, such as triggering a complete refresh of the image upon becoming aware of a scene change, could be explored in future work.

### 4.3 Comparison with image-space upscaling

To further assess our results, we compared our method to alternative baseline approaches for renderer-agnostic acceleration of rendering by downscaling resolution. To that end, we compared our amortized rendering to simple bi-linear upscaling from a low-resolution rendering output, as well as to the purely image-space-based upscaling provided by AMD FidelityFX™ Super Resolution (FSR) (Advanced Micro Devices 2022). Similar to our method, FSR improves rendering performance by reducing internal rendering resolution. However, unlike our method, it has no temporal component and instead uses edge-adaptive spatial upsampling and optional contrast-adaptive sharpening to reconstruct a native-resolution image from the low-resolution rendering along. To directly compare the results with our 2D renderers, we ported the available FSR Vulkan source code to OpenGL as a module for MegaMol. In our test scene (Nvidia GeForce RTX 3080, *Scatterplot Matrix, Droplets*,  $2560 \times 1440$ ), using FSR with its *Performance* preset, which scales resolution by a factor of 2 and is therefore comparable to our amortization level 2, results in an image with an SSIM of 0.9668 (see Fig. 10), whereas bi-linear upscaling results in an SSIM of 0.9626. In comparison, amortization level  $a = 2$  of our methods starts off with slightly lower quality after a complete image refresh but exceeds both FSR *Performance* quality and bi-linear upscaling once a second frame has been rendered. Higher amortization levels require longer convergence time to exceed FSR *Performance* quality, e.g., approximately 500 ms for  $a = 4$ .

Apart from image quality, we also compared response time improvements between the different approaches. FSR *Performance* achieves an average frame time of 90 ms. As expected, amortization level 2 exactly matches this value, whereas higher levels further improve average frame time, e.g., to 62 ms for  $a = 4$ .

## 5 Conclusion and future work

We have introduced a method that accelerates GPU rendering of 2D information visualizations by rendering at a lower internal resolution and temporally reprojecting the pixels of the last output image to the current frame to reconstruct a native-resolution image. By thus amortizing the rendering cost of a native-resolution image over multiple internal frames, frame times and the responsiveness of the visualizations are greatly improved. Due to the orthographic projection used in 2D rendering, we can perfectly reconstruct the native-resolution image during translational camera movement, with the exception of the screen border. During zooming and when the scene contents change, e.g., due to data selection or filtering, image noise and ghosting artifacts cannot be completely avoided, but image quality quickly converges to the native-resolution image as soon as the scene becomes static again. At the same time, such interactions become fluid for use cases that are hardly usable without acceleration.

Our method offers different amortization modes and levels with continuously decreasing internal rendering resolution, which generally improves performance with each level. Additionally, we include an approach for adaptive resolution scaling that matches a given performance target as close as possible by automatically adjusting the amortization level. To validate our performance expectations, we performed benchmarks with two different visualizations, three different dataset sizes, seven different GPUs, and various settings for native display resolution, amortization level and amortization modes. We performed additional benchmarks on two mobile devices with integrated graphics and fixed display resolutions to cover a wider range of real-world configurations. Our benchmarks demonstrate that our method improves rendering performance significantly. The benchmarks also show that the acceleration for high amortization levels does not always scale as well as the convergence time does, thus requiring careful balancing between interactive response time and convergence time in some scenarios. Finally, we also identified and discussed performance limitations for a few specific scenarios.

In the future, we plan to investigate continuous resolution scaling instead of using fixed levels. Another possible extension is the integration of more sophisticated reconstruction schemes, such as DLSS



(Liu 2020). Temporal supersampling, by retaining multiple, slightly offset sets of samples within one pixel, could be added to further improve static image quality. In the context of image quality, a user study could complement the SSIM analysis and help to evaluate the perceived quality for dynamic images. For a wider application of our technique, a generalization for additional projection spaces, e.g., 3D orthographic projection, while maintaining the advantages of our approach, could be investigated. Finally, we hope to spark further investigations into the adaptation of computer graphics concepts to make 2D information visualizations of large datasets feasible within interactive systems.

**Funding** Open Access funding enabled and organized by Projekt DEAL. Funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1244 – Project ID 279064222, SFB-TRR 75 – Project ID 84292822, and SFB-TRR 161 – Project ID 251654672.

**Availability of data and materials** Complete benchmark results are included in supplemental material. The datasets used in the benchmarks are not publicly available with the exception of *Iris*.

#### Declarations

**Conflict of interest** The authors have no conflicts of interest to disclose.

**Code availability** Code is publicly available at <https://github.com/UniStuttgart-VISUS/megamol>.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

#### References

- Adelson SJ, Hodges LF (1995) Generating exact ray-traced animation frames by reprojection. *IEEE Comput Graph Appl* 15(3):43–52. <https://doi.org/10.1109/38.376612>
- Advanced micro devices: FidelityFX super resolution 1.0 (FSR). <https://github.com/GPUOpen-Effects/FidelityFX-FSR>. Accessed 27 Apr 2022
- Becher M, Heinemann M, Marmann T, Reina G, Weiskopf D, Ertl T (2022) Accelerating GPU rendering of 2D visualizations using resolution scaling and temporal reconstruction. In: Proceedings of the 15th international symposium on visual information communication and interaction. VINCI '22. Association for Computing Machinery, New York, NY. <https://doi.org/10.1145/3554944.3554947>
- Binks D (2011) Dynamic resolution rendering article. Intel Corporation. <https://www.intel.com/content/www/us/en/developer/articles/technical/dynamic-resolution-rendering-article.html>. Accessed 27 Apr 2022
- Bishop G, Fuchs H, McMillan L, Zagier EJS (1994) Frameless rendering: double buffering considered harmful. In: Proceedings of the 21st annual conference on computer graphics and interactive techniques. SIGGRAPH '94, pp. 175–176. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/192161.192195>
- Dua D, Graff C (2017) UCI machine learning repository. <http://archive.ics.uci.edu/ml>. Accessed 27 Apr 2017
- Epic Games: an overview of the dynamic resolution system used in Unreal Engine 4. <https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/DynamicResolution/>. Accessed 4 Mar 2023
- Frey S, Sadlo F, Ma K-L, Ertl T (2014) Interactive progressive visualization with space-time error control. *IEEE Trans Visual Comput Graph* 20(12):2397–2406. <https://doi.org/10.1109/tvcg.2014.2346319>
- Gralka P, Becher M, Braun M, Frieß F, Müller C, Rau T, Schatz K, Schulz C, Krone M, Reina G, Ertl T (2019) MegaMol: a comprehensive prototyping framework for visualizations. *Eur Phys J Special Top* 227(14):1817–1829. <https://doi.org/10.1140/epjst/e2019-800167-5>
- Heinemann M, Frey S, Tkachev G, Straub A, Sadlo F, Ertl T (2021) Visual analysis of droplet dynamics in large-scale multiphase spray simulations. *J Vis* 24(5):943–961. <https://doi.org/10.1007/s12650-021-00750-6>
- Heinrich J, Bachthaler S, Weiskopf D (2011) Progressive splatting of continuous scatterplots and parallel coordinates. *Comput Graph Forum* 30(3):653–662. <https://doi.org/10.1111/j.1467-8659.2011.01914.x>
- Karis B (2014) High-quality temporal supersampling. [http://advances.realtimerendering.com/s2014/#\\_HIGH-QUALITY\\_TEMPORAL\\_SUPERSAMPLING](http://advances.realtimerendering.com/s2014/#_HIGH-QUALITY_TEMPORAL_SUPERSAMPLING). Accessed 21 Apr 2022
- Kelleter C, Burghardt T, Binz H, Blandini L, Sobek W (2020) Adaptive concrete beams equipped with integrated fluidic actuators. *Front Built Environ* 6:1–13. <https://doi.org/10.3389/fbuil.2020.00091>

- Limberger D, Tausche K, Linke J, Döllner J (2018) Progressive rendering using multi-frame sampling. In: Engel W (ed.) GPU Pro 360: Guide to rendering, pp. 537–553. Taylor & Francis, CRC Press, Boca Raton, FL (2018). Chap. 32. <https://doi.org/10.1201/9781351261524>
- Liu E (2020) DLSS 2.0—image reconstruction for real-time rendering with deep learning. <http://behindthepixels.io/assets/files/DLSS2.0.pdf>. Accessed 21 Apr 2022
- Ludvig E, Miller JW (1958) Study of visual acuity during the ocular pursuit of moving test objects. I. Introduction. *J Opt Soc Am* 48(11):799–802. <https://doi.org/10.1364/josa.48.000799>
- Mansouri JE (2016) Rendering ‘Rainbow Six Siege’. <https://www.gdcvault.com/play/1022990/Rendering-Rainbow-Six-Siege>. Accessed 21 Apr 2022
- McFerron T, Lake A (2018) Checkerboard rendering for real-time upscaling on intel integrated graphics. Intel Corporation. <https://software.intel.com/sites/default/files/managed/c0/8e/checkerboard-rendering-for-real-time-upscaling-on-intel-integrated-graphics.pdf>. Accessed 27 Apr 2022
- Miller JW (1958) Study of visual acuity during the ocular pursuit of moving test objects II. Effects of direction of movement, relative movement, and illumination. *J Opt Soc Am* 48(11), 803–808 (1958). <https://doi.org/10.1364/josa.48.000803>
- Petkov K, Kaufman AE (2016) Frameless volume visualization. *IEEE Trans Visual Comput Graph* 22(2):1076–1087. <https://doi.org/10.1109/tvcg.2015.2440262>
- Richer G, Pister A, Abdelaal M, Fekete J-D, Sedlmair M, Weiskopf D (2022) Scalability in visualization. *IEEE Trans Vis Comput Graph*. <https://doi.org/10.1109/TVCG.2022.3231230>
- Scherzer D, Yang L, Mattausch O, Nehab D, Sander PV, Wimmer M, Eisemann E (2011) A survey on temporal coherence methods in real-time rendering. In: John, N., Wyvill, B. (eds.) Eurographics 2011 - State of the Art Reports, pp. 101–126. The Eurographics Association (2011). <https://doi.org/10.2312/EG2011/STARS/101-126>
- Shneiderman B, Plaisant C (2005) Designing the user interface: strategies for effective human-computer interaction, 4th edn. Pearson/Addison Wesley, Boston, MA
- Wang Z, Bovik AC, Sheikh HR, Simoncelli EP (2004) Image quality assessment: from error visibility to structural similarity. *IEEE Trans Image Process* 13(4):600–612. <https://doi.org/10.1109/tip.2003.819861>
- Xiao L, Nouri S, Chapman M, Fix A, Lanman D, Kaplanyan A (2020) Neural supersampling for real-time rendering. *ACM Trans Graph* 39(4):142:1–142:12. <https://doi.org/10.1145/3386569.3392376>
- Xiao K, Liktov G, Vaidyanathan K (2018) Coarse pixel shading with temporal supersampling. In: Proceedings of the ACM SIGGRAPH symposium on interactive 3D graphics and games, pp. 1:1–1:7. Association for Computing Machinery, New York, NY. <https://doi.org/10.1145/3190834.3190850>
- Yang L, Nehab D, Sander PV, Sitthi-amorn P, Lawrence J, Hoppe H (2009) Amortized supersampling. *ACM Trans Graph* 28(5):1–12. <https://doi.org/10.1145/1618452.1618481>
- Yang L, Liu S, Salvi M (2020) A survey of temporal antialiasing techniques. *Comput Graph Forum* 39(2):607–621. <https://doi.org/10.1111/cgf.14018>