Progressive High-Quality Rendering for Interactive Information Cartography using WebGL

Daniel Limberger*
Marcel Pursche
Faculty of Digital Engineering
University of Potsdam

Jan Klimke
Jürgen Döllner
Faculty of Digital Engineering
University of Potsdam

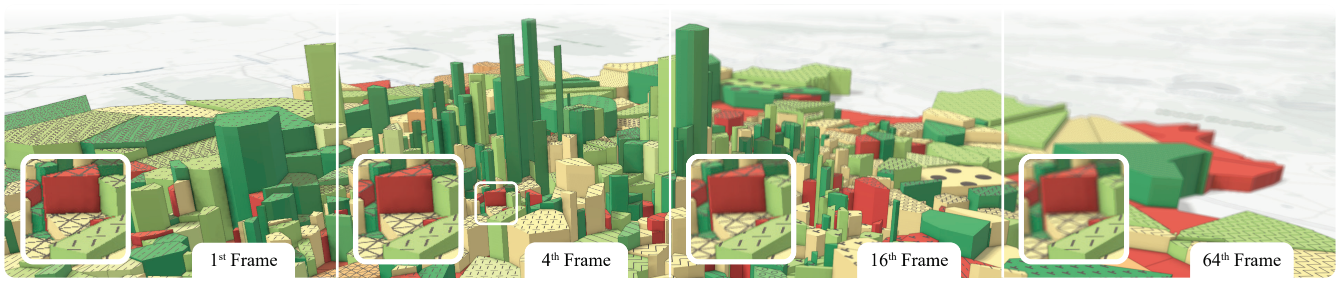


Figure 1: 2.5D Voronoi Map depicting geo-referenced data using progressive rendering (AA, DoF, SSAO, and soft shadows).

ABSTRACT

Information cartography services provided via web-based clients using real-time rendering do not always necessitate a continuous stream of updates in the visual display. This paper shows how progressive rendering by means of *multi-frame sampling* and *frame accumulation* can introduce high-quality visual effects using robust and straightforward implementations. For it, (1) a suitable rendering loop is described, (2) WebGL limitations are discussed, and (3) an adaption of THREE.js featuring progressive anti-aliasing, screen-space ambient occlusion, and depth of field is detailed. Furthermore, sampling strategies are discussed and rendering performance is evaluated, emphasizing the low per-frame costs of this approach.

CCS CONCEPTS

• **Computing methodologies** → **Rendering**; • **Human-centered computing** → Web-based interaction; Treemaps;

KEYWORDS

Progressive Rendering, Interactive Visualization, WebGL, Mobile

ACM Reference format:
Daniel Limberger, Marcel Pursche, Jan Klimke, and Jürgen Döllner. 2017. Progressive High-Quality Rendering for Interactive Information Cartography using WebGL. In *Proceedings of Web3D '17, Brisbane, QLD, Australia, June 05-07, 2017,* 4 pages.

*{daniel.limberger, marcel.pursche, jan.klimke, juergen.doellner}hpi.de

DOI: http://dx.doi.org/10.1145/3055624.3075951

1 INTRODUCTION

Todays interactive information cartography clients, e.g., 2.5D visualization of geo-referenced data or non-spatial software system information (Limberger et al. 2013), are made accessible to users via web applications preferably. For their visual display, real-time rendering techniques optimized for a continuous stream of individual high-quality frames are applied. High-quality, in this context, denotes synthesizing effects such as physical-based shading, *anti-aliasing* (AA), *screen-space ambient occlusion* (SSAO), *depth-of-field* (DoF), or soft shadows. Most of the respective rendering techniques are designed and optimized for single-frame execution and are not feasible when targeting end user platforms with heterogeneous hardware and software settings, especially mobile devices. Their requirements regarding GPU features and performance, memory, and CPU resources makes it hard to implement them robustly in order to create a homogeneous, high-quality rendering result that works on a large number of end user devices and platforms.

The incremental nature of user interactions in information visualization (Van Wijk 2005) does often not require strict continuity in rendering due to (1) the lack of dynamic objects and animations, (2) infrequent data changes, and (3) mostly discontinuous changes of the virtual camera (e.g., navigation). Thus, we suggest to apply *multi-frame sampling* (Limberger et al. 2016) which distributes rendering cost over time while providing explicit quality control primarily by the number of subsequent frames that are to be accumulated. Furthermore, it increases the responsiveness of our visualizations, reduces resource requirements, and facilitates the use of straightforward rendering techniques. This paper discusses how multi-frame sampling can be integrated using WebGL (Section 3), describes utilization of THREE.js (Section 4) with respect to AA, DoF, and SSAO, and presents a brief performance evaluation and discussion (Section 5).

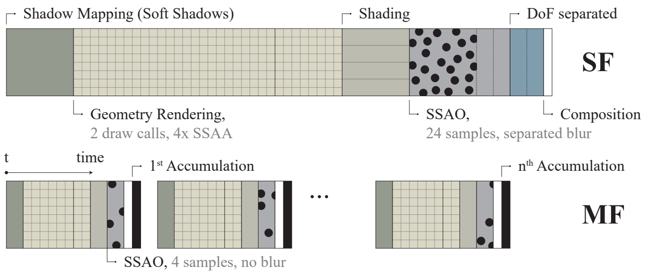


Figure 2: Illustration of a single-frame SF and an equivalent multi-frame MF. For MF a simplified shadow pass and 4 SSAO samples per IF are sufficient; DoF and AA are inherent due to camera and normalized device coordinate (NDC) shifting. Image adapted from (Limberger et al. 2016).

2 RELATED WORK

An integral part of today's hardware-accelerated, real-time rendering is built on a variety of sampling strategies. Sampling can be used to approximate continuous characteristic, e.g., reduce aliasing artifacts caused by insufficient depictions of continuous domains. Although increasing the number of samples increases the resulting image quality, it affects the rendering cost per frame (either time, memory, or both). The multi-frame approach distributes and accumulates (Fuchs et al. 1985; Haeberli and Akeley 1990) samples over a well-defined number of consecutive frames. Recently, more techniques start to make use of previous frames, such as Nvidia's TXAA and subsequently MFAA (Nvidia 2015) that use reprojection to access previous fragment information. In this paper we rely on accumulation solely in order to keep the implementation as simple as possible. If, however, dynamics are essential for the visualization, multi-frame sampling could be enhanced by using reprojection for per-fragment accumulation (instead of plain buffer accumulation). Another approach to account for local changes within the visual display is to only update small viewport regions resulting in fast, seamless convergence as long as regular, fixed sampling kernels are applied. The foundation of sampling characteristics by means of number, distribution, regularity, and completeness of samples as well as temporal convergence constraints w.r.t. finite quality are covered by Limberger and Döllner (2016) and considered in this work. Favoring responsiveness and overlaying high-quality features later seems to gain popularity, as can be seen in Sketchfab and 3DS Max by Autodesk[1]. There are various other domains and web applications which could take advantage in supporting multi-frame sampling as well, such as rendering point clouds (Richter and Döllner 2014) or applications using, e.g., Cesium or THREE.js (Cabello 2010). Similarly, the rendering of embed 3D graphics within the HTML DOM (Jankowski et al. 2013) by means of X3DOM (Behr et al. 2009) or XML3D (Sons et al. 2010) could be enhanced.

3 PROGRESSIVE RENDERING IN WEBGL

Instead of rendering a typical single frame (**SF**) in response to changed inputs (e.g., mapped data, camera), multiple intermediate frames (**IF**) are rendered and accumulated into a multi-frame (**MF**) For it, a multi-frame number n_{MF} is defined, denoting the number of intermediate frames to be rendered for a full multi-frame. Every sampling kernel should be optimized for this exact number in order to provide the best temporal convergence and quality (Fig. 2).

rgb8UI
rgb32F

The progressive rendering control flow should: (1) (re)start rendering of a intermediate frames immediately when any input changes. The renderer should (2) display the accumulated results of every intermediate frame, and (3) stop rendering when n_{MF} frames were rendered, continue displaying the final result. WebGL 1 provides `RGB8` color buffers by default, which limit the accuracy and quality of accumulation; small changes in color get lost in the accumulation of later frames. Thus, a comparatively small n_{MF} should be applied. Although float textures are supported on most systems by extension (`OES_texture_float`) they cannot be used as render target. Instead, the float color buffer (`WEBGL_color_buffer_float`) is required for a render target with floating point precision in order to increase accumulation quality.

4 PROGRESSIVE RENDERING IN THREE.JS

Switching the rendering engine of an existing application can be a time consuming task. It is therefore often more sensible to integrate new techniques into the existing rendering engine. As a proof of concept we integrated multi-frame sampling into a `THREE.js` (Cabello 2010) application which can be used as a blueprint for the integration of multi-frame sampling into other `THREE.js` applications. One goal of the integration is to keep compatibility with `THREE.js`' built-in rendering effects. For this, we implemented specialized `THREE.ShaderMaterials`, reusing shader code for vertex transformation, lighting, texturing etc. of the `THREE.js` built-in materials. The shader code components for the built-in materials, included in `THREE.ShaderChunk`, can be combined to create custom materials.

4.1 Frame Accumulation

We use the `three-effectcomposer`[2] library as a convenience to combine subsequent post processing passes since it automatically provides the result of the previous render pass, the render target, and a full-screen quad. The composer library provides a `ShaderPass` class which makes it straightforward to implement full-screen post processing passes by applying a configured fragment shader.

As the frame accumulation pass requires the rendering result of the previous **IF**, it is necessary to extend the `EffectComposer` (**EC**) to store it. The **EC** uses an input buffer and output buffer in an alternating way for consecutive post processing. We adjusted the **EC**s `render` method by subclassing to ensure that input and output buffer are swapped an even number of times to guarantee the same render target is used for the rendering result in each frame:

```
class EffectComposer extends THREE.EffectComposer {
  constructor(renderer, renderTarget) {
    super(renderer, renderTarget);
    this.ensureFinalSwap = true; }
  render(delta) {
    THREE.EffectComposer.prototype.render.call(this, delta);
    // ensure that final rendering result is in renderTarget1
    if (this.readBuffer === this.renderTarget1 && this.ensureFinalSwap) {
      this.swapBuffers();
      this.copyPass.render(this.renderer, this.writeBuffer,
        this.readBuffer, delta); } } }
```

[1]https://sketchfab.com/developers/viewer and http://www.autodesk.com/3ds-max

[2]https://github.com/hughsk/three-effectcomposer

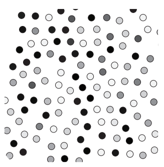
When a **IF** is rendered the `writeBuffer` is swapped with the render target of the previous frame and passed to accumulation as texture. For it, a render loop loads the kernel values to be applied for the current **IF** and updates the blending weight for the accumulation pass using a frame number. The number is reset if the configuration of the virtual camera changed or the rendered scene itself is updated, causing a new multi-frame request. After a configured number of **IF** the rendering loop is paused to reduce GPU and CPU load while a static scene is displayed. The render loop itself is implemented using the `requestAnimationFrame` function supplied by browsers, which also restricts framerates to a maximum of 60 fps:

```
function render() {
    if (camera.needsUpdate || scene.needsUpdate)
        frameId = 0; // reset frame id if camera or scene has changed
    if (frameId < numSampleFrames) { // render on demand
        // load kernel values for current frame
        renderPipeline.setCurrentFrame(frameId);
        renderPipeline.render(); // render scene
        //swap targets to keep current frame for accumulation in next frame
        renderPipeline.swapRenderTargets();
        ++frameId; }
    requestAnimationFrame(render); }
```

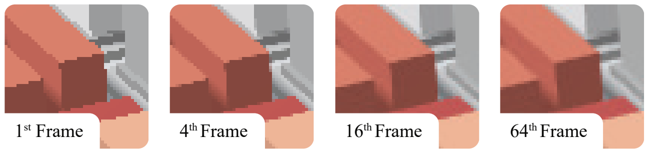
4.2 Progressive Antialiasing

Anti-aliasing as technique massively improves the visual quality of 3D renderings. Our implementation uses the multi-frame anti-aliasing technique proposed by Limberger et al. (2016): per **IF**, a sub-pixel offset, provided via a uniform to the vertex shader, is applied to the vertices' xy-coordinates in NDC. We reuse existing `THREE.js` uniforms, varyings, and local variables, e.g., `projectionMatrix` and `mvPosition` to keep compatibility with built-in effects:

```
    vec4 ndcPosition = projectionMatrix * mvPosition;
    // Add subpixel offset to ndc vertex position.
    ndcPosition.xy += ndcOffset * ndcPosition.w;
    gl_Position = ndcPosition;
```



The subpixel offset is chosen using shuffled poisson-disk sampling. The kernel is precomputed on the CPU depending on the number of **IF** as sampling offsets in $[-0.5, +0.5]$ and transformed into a subpixel offset using the current render target resolution. The technique produces seemingly optimal results with about 16 **IF**. Since the accumulation buffer format is limited to 8 bit per color, additional frames yield only marginal improvements:

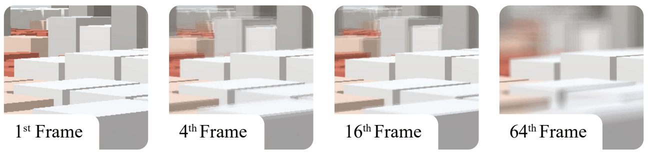


4.3 Progressive Depth-of-Field

There are several approaches to implement DoF effects using several rendering passes and different G-Buffers (Bukowski et al. 2013; Selgrad et al. 2015). These are quite complex to implement since they rely on sophisticated combinations of these buffers. A multi-frame DoF effect can be implemented similar to anti-aliasing: an offset is applied to the vertices' xy-coordinates in view coordinates. The offset is scaled by the vertices z-coordinate and a configurable focal distance. This effectively shifts a vertex within its circle of confusion (**CoC**) gradually covering it with each subsequent frame. The size of the **CoC** is configured by scaling the position offset with a constant factor. We reuse the the same kernel algorithm from the anti-aliasing technique but scale it to $[-1, +1]$ additionally.

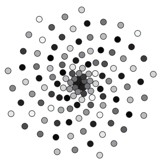
```
uniform vec2 cocPoint;
uniform float focalDistance;
// ...
    mvPosition.xy += cocPoint * (mvPosition.z + focalDistance);
```

The convergence of the effect depends on the configured **CoC** factor and focal distance. The larger the **CoC** the more frames are necessary. For very large **CoC**, the accumulation buffer precision may not be sufficient, resulting in visible visual artifacts.



4.4 Progressive SSAO

We use the Screen Space Ambient Obscurance as proposed by McGuire et al. (2011), but with an optimized kernel for comparable image quality to HBAO (Bavoil et al. 2008) with less samples.



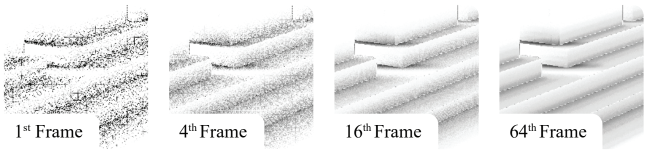
The kernel uses a spiral shaped pattern that is projected to a local tangent plane to produce the sample offsets. We precompute the kernel on the CPU and provision it to the GPU as texture. Compared to the original implementation we sort the samples of the kernel in alternating order to ensure a more steady improvement in image quality for subsequent **IF**. For n frames and m samples per frame the set of samples S_i for frame i are $S_i = \bigcup_{j=0}^{m-1} \{s(jn + i)\}$. Where function $s(n)$ gives the n-th sample of the original kernel. This order ensures that in each **IF** samples with low, medium and high distance from the hemisphere center are taken to account for different scales of occlusion.

```
function ssaoKernel(samplesPerFrame, spiralTurns, numFrames) {
    var numSamples = samplesPerFrame * numFrames;
    function samplePosition(sampleId) {
        var alpha = (sampleId + 0.5) / numSamples,
        angle = alpha * spiralTurns * Math.PI * 2.0;
        return [angle, alpha]; }

    var imageData = new Float32Array(numSamples * 2);
    for (var y = 0; y < numFrames; ++y) {
        for (var x = 0; x < samplesPerFrame; ++x) {
            var sample = samplePosition(x * numFrames + y);
            imageData[2 * (x + y * samplesPerFrame)] = sample[0];
            imageData[2 * (x + y * samplesPerFrame) + 1] = sample[1]; } }

    return imageData; }
```

We chose to omit the depth-aware bilateral filter of the original algorithm as the noise is drastically reduced within a few **IF** and it produces better overall results:



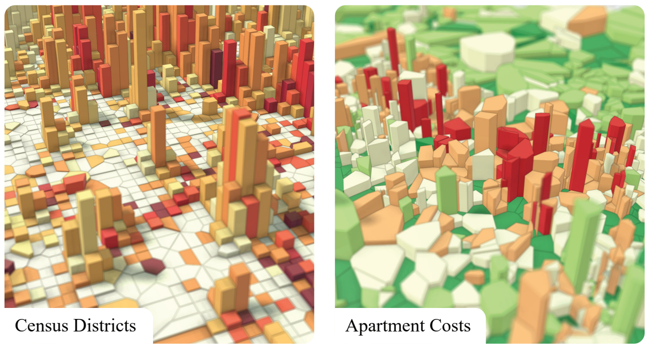


Figure 3: Left: Census districts (1 sq. km. in average) depicting average income (color) with respect to vacancy (height). Right: Voronoi Map that correlates living space (height) of available apartments (cells) to purchase cost (color).

5 RESULTS

Performance. For performance evaluation we tested the implementation with three datasets of different sizes as scenarios in our information cartography framework (Fig. 3). For each scenario, we used height and color mapping for visualization to assure the same real time mapping shaders to be used. The scenarios differ mainly in the number of objects to be included (small - 96 objects with 8100 vertices, medium - 6693 objects with 240984 vertices, and large - 22840 objects with 823716 vertices). We tested the implementation on two resolutions (1920x1080 and 1280x720) and with and without multi-sampling effects (SSAO, DOF, and Antialiasing - 32 intermediate frames) using the Chrome browser (version 56) on a Mac operating system. The hardware was set up as follows: Intel Core i7, 2.6 GHz, 16 GB RAM, GPU GT 650M with 1GB RAM. The selected GPU hardware marks an expected mean target hardware to be addressed. Result figures can be found in Table 1.

Scenario	**IF**		**MF**	**IF**		**MF**
	1080p	1080p*	1080p	720p	720p*	720p
small	28.7	16.8	688.2	18.0	16.8	1075.0
medium	65.9	37.3	1851.0	42.1	17.0	1265.0
large	77.4	45.2	2163.0	67.5	48.1	1695.0

*rendering without AA, DoF, SSAO etc.

Table 1: Frame times in ms for 1080p and 720p resolutions. IF denotes an intermediate frame within a mullti-frame (MF).

Discussion. The measurements show that the multi-frame sampling effects affect the performance for frame rendering by approximately factor 2. The reason for this difference is the general rendering and shading overhead that is required by the single rendering techniques and not significantly influenced by the progressive accumulation of intermediate framebuffers. Since intermediate frames, especially the first ones of a multi-frame may contain visible artifact, the single rendering techniques do not need to be configured using high quality settings. In this way, they do not influence the rendering performance of the overall system as much as they would if multi-frame sampling would not be applied. The additional measurements have shown, that the rendering is accumulated in linear time depending on the overall frame rate by the number of selected intermediate frames. Table 1 shows, that the finally rendered image is ready after about 2100 ms for very large scenes with 32 intermediate frames on medium range hardware.

6 CONCLUSION AND FUTURE WORK

This paper describes how multi-frame sampling can be used to incorporate high-quality effects in web-based rendering engines. In addition to these effects, massive lighting can be used for highlighting of blocks with local light impact, and glossy screen space reflections and transparency become feasible. A drawback of our THREE.js integration is that it might introduce unnecessary copying passes for an uneven number of post processing passes. A complete rewrite of the `EffectComposer` with multi-frame sampling in mind could address this but might break compatibility with existing effects that use the library. Additionally, the use of `THREE.ShaderMaterials` for the integration of multi-frame sampling effects is not optimal because of the way these materials are implemented in `THREE.js`. If the same shader is used in a scene with different uniform configurations the shader material needs to be instanced multiple times which results in multiple `WebGL` shader and program objects. This might introduce uncessary shader compilations and context switches. This could be addressed by refactoring `THREE.js` to allow the reuse of `WebGL` program objects over multiple material instances. With the increasing support for WebGL 2 new image formats become available for the use as render targets. As the precision of the accumulation buffer has a influences the quality, render targets with higher bit depth could help to improve the image quality.

REFERENCES

Louis Bavoil, Miguel Sainz, and Rouslan Dimitrov. 2008. Image-space Horizon-based Ambient Occlusion. In *ACM SIGGRAPH 2008 Talks*. 22:1–22:1.

Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. 2009. X3DOM: A DOM-based HTML5/X3D Integration Model. In *Proc. ACM Web3D*. 127–135.

Mike Bukowski, Padraic Hennessy, Brian Osman, and Morgan McGuire. 2013. The Skylanders SWAP Force Depth-of-Field Shader. In *GPU Pro 4*. 175–184.

Ricardo Cabello. 2010. Three. js. *URL: https://github. com/mrdoob/three. js* (2010).

Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. 1985. Fast Spheres, Shadows, Textures, Transparencies, and Imgage Enhancements in Pixel-planes. *Proc. ACM SIGGRAPH Comput. Graph.* (1985), 111–120.

Paul Haeberli and Kurt Akeley. 1990. The Accumulation Buffer: Hardware Support for High-quality Rendering. In *Proc. ACM SIGGRAPH*. 309–318.

Jacek Jankowski, Sandy Ressler, Kristian Sons, Yvonne Jung, Johannes Behr, and Philipp Slusallek. 2013. Declarative Integration of Interactive 3D Graphics into the World-Wide Web: Principles, Current Approaches, and Research Agenda. In *Proc. ACM Web3D*.

Daniel Limberger and Jürgen Döllner. 2016. Real-time Rendering of High-quality Effects Using Multi-frame Sampling. In *ACM SIGGRAPH 2016 Posters*.

Daniel Limberger, Karsten Tausche, Johannes Linke, and Jürgen Döllner. 2016. Progressive Rendering Using Multi-frame Sampling. In *GPU Pro 7*.

Daniel Limberger, Benjamin Wasty, Jonas Trümper, and Jürgen Döllner. 2013. Interactive Software Maps for Web-based Source Code Analysis. In *Proc. ACM Web3D*. 91–98.

Morgan McGuire, Brian Osman, Michael Bukowski, and Padraic Hennessy. 2011. The Alchemy Screen-space Ambient Obscurance Algorithm. In *Proc. ACM HPG*.

Nvidia. 2015. Multi-frame Sampled Anti-Aliasing (MFAA). http://www.geforce.com/hardware/technology/mfaa/technology. (2015).

Rico Richter and Jürgen Döllner. 2014. Concepts and techniques for integration, analysis and visualization of massive 3D point clouds. *Computers, Environment and Urban Systems* 45 (2014), 114–124.

Kai Selgrad, Christian Reintges, Dominik Penk, Pascal Wagner, and Marc Stamminger. 2015. Real-time Depth of Field Using Multi-layer Filtering. In *Proc. ACM I3D*. 121–127.

Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. 2010. XML3D: Interactive 3D Graphics for the Web. In *Proc. ACM Web3D*. 175–184.

Jarke J Van Wijk. 2005. The value of visualization. In *Proc. IEEE VIS*. 79–86.