High Performance, Adaptive Texture Streaming and Rendering of Large 3D Cities

Alex Zhang[*] Fraunhofer Singapore

Kan Chen[†] Fraunhofer Singapore

Henry Johan[‡] Nanyang Technological University Fraunhofer IDM@NTU

Marius Erdt[§] Nanyang Technological University Fraunhofer Singapore

Abstract

We propose a high-performance texture streaming system for real-time rendering of large 3D cities with millions of textures. Our main contribution is a texture streaming system that automatically adjusts the streaming workload at runtime based on measured frame latencies, specifically addressing the high memory binding costs of hardware virtual texturing which causes frame rate stuttering. Our system streams textures in parallel with prioritization based on GPU computed mesh perceptibility, and these textures are cached in a sparse partially-resident image at runtime without the need for a texture preprocessing step. In addition, we improve rendering quality by minimizing texture pop-in artifacts using a color blending scheme based on mipmap levels. We evaluate our texture streaming system using three structurally distinct datasets with many textures and compared it to a baseline, a game engine, and our prior method. Results show an 8X improvement in rendering performance and 7X improvement in rendering quality compared to the baseline.

Index Terms: Real-time Rendering, Texture Streaming, Virtual Texturing, 3D Cities

1 Introduction

Many organizations in recent years have introduced a 3D digital twin of their local city to aid in geographic information system (GIS) related activities. These 3D city models recreate real-world urban environments with large quantities of texture images captured from aerial imaging. Although current computer systems can render large-scale 3D cities in real-time, they are limited by GPU memory when rendering millions of textures hundreds of gigabytes in size. Furthermore, rendering performance is heavily impacted when overflowing data is naively paged in and out of GPU memory. Current large-scale rendering methods stream textures to the GPU on demand, adapting texture resolutions to mesh visibility and level-of-detail (LOD) metrics. However, these methods have difficulties maintaining a consistent rendering performance when GPU memory is allocated at runtime, resulting in severe frame rate stuttering in some cases.

In this paper, we present a high-performance texture streaming system for rendering 3D cities with large quantities of textures. We focus on minimizing frame rate stuttering by amortizing texture streaming costs over time, and maximizing texture caching performance for hardware virtual texturing with runtime memory allocation. Our specific contributions are:

- A single-pass, multithreaded, GPU-assisted texture streaming system that identifies and prioritizes textures for streaming based on visual perceptibility, and minimizes texture pop-in artifacts and uneven texture resolutions during rendering. We also propose a parallel, runtime texture caching scheme for hardware virtual texturing.
- An adaptive texture streaming algorithm that automatically adjusts the texture streaming workload based on measured frame latencies. It is robust to support different city dataset structures and computer system performances and is intuitive for users to set parameters.
- A mipmap color blending method to minimize texture pop-in artifacts and preexisting mipmapping artifacts in city datasets. We store original per-mesh colors in a City Colormap, which are then interpolated with sampled colors based on displayed mipmap levels.
- An optional mesh clustering scheme to generate texture atlases by grouping nearby and similarly oriented meshes together. While streaming a 3D city with over five million textures is directly supported, using our proposed scheme greatly improves texture streaming performance.

To evaluate our texture streaming performance and rendering quality, we use three structurally different large city datasets, one containing 40GB of compressed texture data, and another containing 56K large texture atlases with high geometry count. Our results show a significant 8X improvement in rendering performance and 7X improvement in image quality compared to a baseline without using our proposed method. We also compare our adaptive streaming algorithm to that of our previous work [20], as well as the Unity game engine's texture streaming quality to determine the effectiveness of our method. Overall, our technique allows fast texture streaming of large city scenes with many more textures than previously possible, with greatly reduced frame rate stuttering and greatly improved rendering quality.

In our previous texture streaming system [20], we successfully minimized frame rate stuttering, maximized texture streaming rate, and mitigated texture pop-in artifacts. In this extension, we propose a novel texture streaming algorithm that automatically adjusts texture streaming rate based on measured frame latencies, and show that our new algorithms are more robust, more intuitive to set system parameters, and achieve faster mipmap resolution refinement at comparable texture streaming performance. Additionally, we optimize the mesh weight value contributions, provide new texture streaming performance measurements using a rigorous benchmarking system, create entirely new figures and tables, evaluate our new adaptive algorithm in detail under the three city datasets, and compare it to our prior work and the baseline based on measured performance and rendering quality.

2 Related Work

2.1 City rendering in GIS

Virtual globe visualization systems such as Google Maps and Cesium render 3D city models and terrain on the web for GIS purposes. They stream large amounts of texture and geometry as sets of tiles, each adapted to a LOD suitable for real-time rendering on client systems,

[*]E-mail: alex.zhang@fraunhofer.sg
[†]E-mail: chen.kan@fraunhofer.sg
[‡]E-mail: henryjohan@ntu.edu.sg
[§]E-mail: marius.erdt@fraunhofer.sg

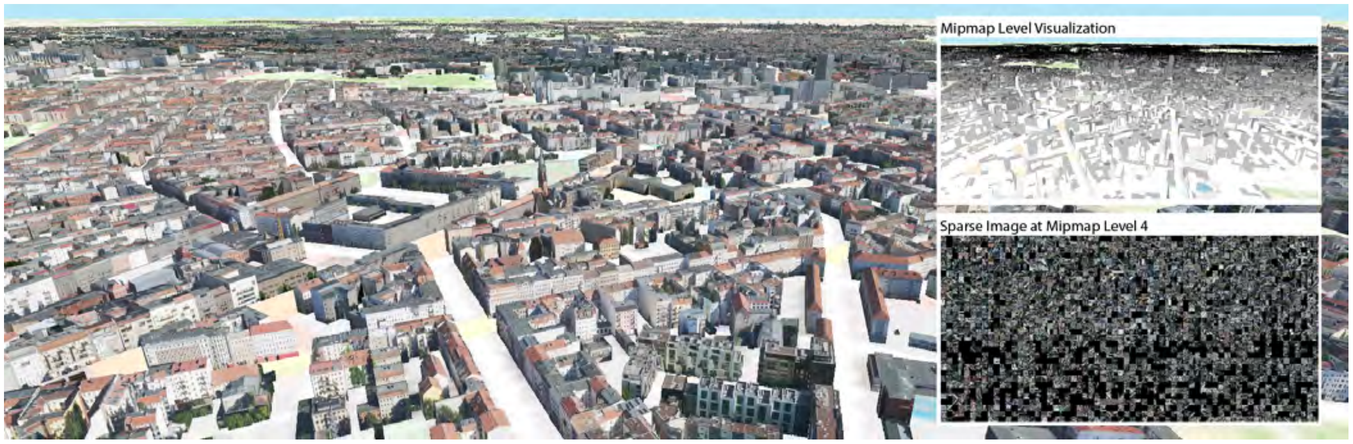

Figure 1: Rendering of the Berlin city dataset. 13.7K texture mipmaps are loaded and cached in the sparse image at a rate of 56.6MB/s. *Top inset:* The resolutions of loaded texture mipmaps are visualized as grayscale (i.e. lighter gray indicating higher resolution). *Bottom inset:* The sparse partially-resident image showing the cached level 4 texture mipmaps.

based on screen-space and camera-space metrics [4, 8], and based on rendering time measurements [7]. These methods use pre-generated texture coordinates for each tile, swapping them in and out of GPU memory on demand, which results in numerous texture state changes that impact rendering performance. Here are some other works that can enhance our proposed texture streaming system. Image epitomes [18] can be used to reduce repeated areas of scene textures, minimizing texture memory space and transmission bandwidth. To further reduce texture pop-ins when switching to higher resolution mipmaps, structure transfer [5] can be used to introduce finer details from high resolution mipmaps while maintaining color consistency. Generating abstractions of building textures [10] to reduce texture size may also be of interest.

2.2 Virtual texturing systems

Virtual texturing systems are designed to cache a relevant subset of the scene texture in GPU memory for rendering, by moving textures in and out of the cache based on geometric factors and resource availability. Tanner et al. [16] proposed the geometry clipmap to cache clipped portions of the terrain based on camera distance. Cline et al. [3] proposed a texture streaming system based on available system bandwidth and used a quadtree mipmap to cache streamed textures. Lefebvre et al. [9] introduced a texture streaming system for arbitrary meshes with GPU support and GPU visibility determination.

Recent virtual texturing methods emulate virtualization of texture memory, by sparsely allocating physical memory for virtual texture blocks (called pages) required for rendering. Page address is translated per-fragment using an indirection texture that maps a page in virtual address to a region in the physical texture cache [11]. Van Waveren [17] proposed software virtual texturing as a mega-texture, where the scene's entire texture space is mapped in its virtual image space, and provided solutions for texture filtering and page caching. Widmark [19] proposed the procedural virtual texture to cache terrain textures and reduce texture blending costs. Chen [2] improved on the prior with adaptive virtual textures to support a very large terrain. Mueller et al. [12] designed a system to compress and stream object-space shading data for remote rendering, using a virtual texture with atomic GPU memory allocation. However, software virtual texturing methods forsake texture filtering in hardware as physical page boundaries and texture coordinates are not contiguous, hence, robustly implementing anisotropic filtering will be difficult [13]. Additionally, page translations invoke multiple in-shader memory accesses that can adversely affect performance.

2.3 Hardware virtual texturing

Hardware virtual texturing [13] is designed to mitigate some disadvantages of software virtual texturing through hardware support. Page address translation is performed in hardware, reducing translation overheads and lowering the memory footprint. Implementation is also simplified as page mappings are no longer handled by the application. Schmitz et al. [14] apply hardware virtual texturing to cache textures for point cloud rendering with predictive page management based on camera movements. Overall, hardware virtual texturing is not widely researched in literature.

The sparse partially-resident image is an interface for hardware virtual texturing on the Vulkan graphics API. In OpenGL, it is named sparse textures, and in DirectX, it is named tiled resources, with equivalent purposes and functionalities. The sparse image is an image object with mipmaps and layers subresources, with the distinction that memory can be mapped to subresource regions at a predefined granularity. A simple example to use the sparse image is to first allocate memory blocks for the destination subresource regions, then bind (i.e. map) allocated memory blocks to those destinations before copying in textures.

On current Windows systems, binding sparse image memory blocks has a high GPU latency cost that causes severe frame rate stuttering. In our tests, we observed a linearly increasing memory block binding cost with every memory block bound, even more so when memory blocks are bound sparsely and irregularly. We suspect this memory binding cost is limited by the underlying system's display drivers, and we amortize this cost using our proposed adaptive texture streaming method. The sparse image is designed for mega-texturing purposes and it assumes that scene textures are pre-generated into an atlas which maps to virtual texture coordinates. To avoid this preprocessing step, we propose a multithreaded sparse image caching scheme that semi-compactly packs textures at runtime.

3 OUR TEXTURE STREAMING PIPELINE

Our texture streaming pipeline (Figure 2) manages texture loading and rendering in real-time. It consists of several CPU and GPU processes that run asynchronously with each other. The heart of the pipeline is the rendering loop. It estimates the amount of texture streaming work for this cycle, retrieves mesh visibility metadata, streams mesh textures to the GPU, and renders the current frame. Within the loop, a non-blocking texture streaming task group is spawned to load designated mesh textures from secondary storage and caches them in the sparse image on the GPU, all in parallel. Note

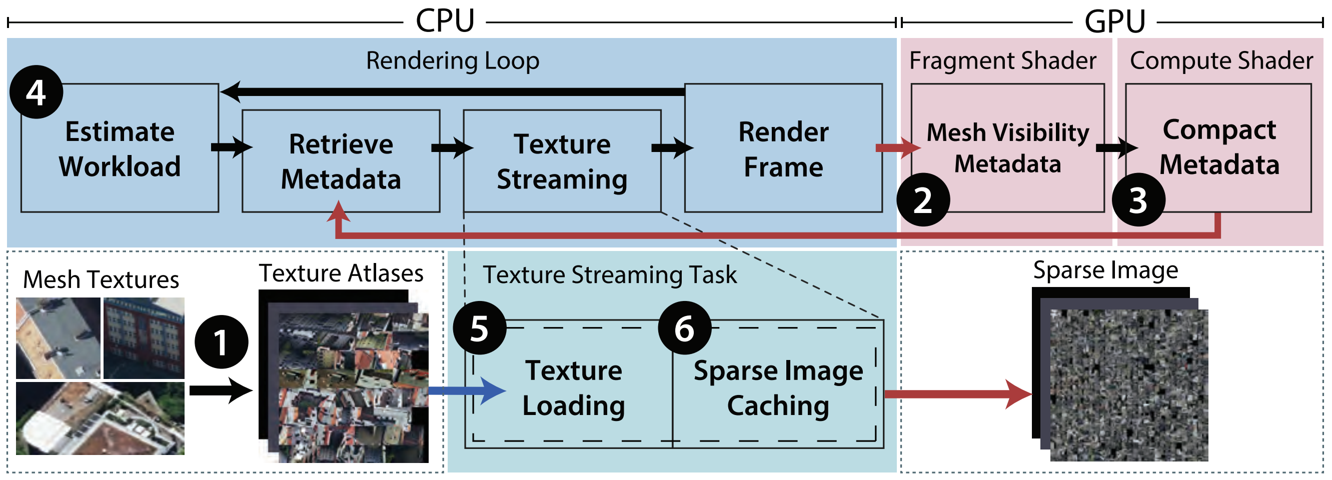

Figure 2: Our single-pass texture streaming pipeline is made up of co-operating CPU and GPU processes. The dotted line boxes define data storage, dashed lines (i.e. subprocesses 5 and 6) define a parallel process, and solid-colored boxes define an asynchronous task group. All arrows indicate the flow of data or process. Red arrows indicate time-expensive data transfers or process calls between the CPU and GPU, and the blue arrow indicates time-expensive data transfer between secondary storage and RAM.

that only one instance of the task group is active at one time. On the GPU end, the fragment shader generates mesh visibility metadata that identifies optimal texture LODs for loading, and a compute shader compacts this metadata before it is retrieved. To meet our requirement for high-performance texture streaming and rendering, we choose to implement our system on the Vulkan graphics API for explicit multithreading and synchronization. We also provide a high-level algorithm (1) of our pipeline that may aid developers in their implementation.

We design the texture streaming system to balance streaming and rendering performance automatically. We minimize streaming cost (i.e. the frame latency) by regulating the workload of every texture streaming cycle. If the workload is too small, texture streaming rate will be inhibited. If the workload is too large, rendering latency will be impacted, causing frame stuttering. A large workload may also increase texture load-to-display latency, causing displayed textures to be no longer relevant to the camera view. Since GPU performance can fluctuate throughout the rendering, due to viewing city areas of different densities and due to general system performances, fixing the workload limit is not ideal. Therefore, we propose to adaptively adjust the workload based on past rendering performance. When the workload limit is reached, the texture streaming task will end and textures that are already streamed are displayed. In the following sections, we will describe our texture streaming pipeline in detail.

3.1 Generating texture atlases

In this preprocessing step, individual mesh textures are packed into uniform-sized atlases to improve sparse image cache utilization and reduce texture file loading time. We propose a mesh clustering scheme to group proximate meshes with similar normal orientations together for texture packing. First, meshes are grouped by their centers within uniform rectangular grid regions, then within each region, they are further separated into six orientation groups according to the closest global axis and mesh normal alignment (Figure 3). We define a mesh as a set of primitives with one texture map and assume they are mostly planar. For non-planar meshes, we take the mean of the mesh normals as its orientation. This procedure resembles backface culling in geometries, as textures of back-facing meshes are not streamed if they are not rasterized. Finally, meshes in each orientation group have textures of the same height packed together into a square atlas to improve space utilization and to maintain mipmap UV coordinate alignment. Empty spaces in an existing atlas are filled using textures (of the same height) from the next-in-order mesh orientation group. This reduces the number of atlas files and allows additional mesh textures to be cached at no cost. Overall, our mesh clustering scheme minimizes cache utilization and improves texture streaming performance as fewer and smaller texture files are accessed and streamed.

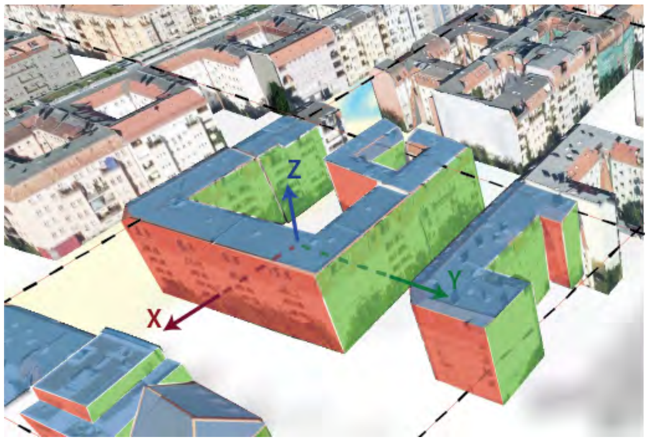

Figure 3: An illustration of our mesh clustering scheme to generate texture atlases for a city dataset. The black dotted lines indicate uniform rectangular grid regions. The red, green, and blue arrows are globally aligned axis. Within each grid region, building meshes are separated into six orientation groups based on their closest normals alignment to the global axis. Back faces are grouped but not illustrated.

3.2 Computing mesh visibility metadata

This subprocess in the fragment shader identifies visible meshes and their texture LOD for streaming. We take advantage of the raster pipeline to perform narrow phase visibility culling automatically, including occlusion culling before the fragment shader stage (via early z-test). In the fragment shader, a **mipmap level** and a **weight value** are computed for every rasterized mesh and stored in a MeshMetaData array buffer. Later in the rendering loop, this array is compacted and copied to system memory for use in texture streaming.

The **mipmap level** metadata defines the mesh texture LOD for streaming. For each fragment that a mesh spans, the mipmap level under anisotropic filtering is computed, as described in Hollemeersch et al. [6], and the lowest mipmap level is selected as the mipmap level metadata. In our system, we define two maximum anisotropy levels A_m and A_s. A_m is the maximum anisotropy used to compute the mipmap level metadata and A_s is the maximum anisotropy used to sample the texture for rendering. In terms of texture streaming, A_m specifies the resolution range of loaded mipmaps in the scene, with a higher value resulting in higher resolution textures being streamed overall. When $A_m > A_s$, texture mipmaps may be preloaded at a higher resolution than what is required for rendering to reduce texture pop-in effects. In our system, we define $A_m = 16$ and $A_s = \frac{A_m}{2}$.

The **weight value** metadata determines mesh texture inclusion in the texture streaming process and the texture's streaming priority, based on the perceptual importance of the mesh. Under perspective camera projection, we can associate the screen space size (i.e. fragment count) of meshes with perceptual importance, since meshes in the foreground with a larger fragment count are more perceivable than faraway or partially occluded meshes. However, this design does not anticipate displayed texture resolutions and human focus, resulting in incongruent distributions of texture resolutions during streaming. We propose a new streaming prioritization based on fragment count, fragment distance from the camera, and loaded mipmap levels to better account for human focus in the camera foreground and to maintain an even distribution of texture resolutions in the foreground to the background. We define the following per-fragment weight W:

$$W = 1 + 8w_d + 8w_m \tag{1}$$

where $W = 1$ is the basic fragment count contribution, w_d is the fragment distance from camera contribution and w_m is the displayed mipmap level contribution. Per-fragment weights W of each rastered mesh are atomically summed (i.e. using the *atomicAdd* function in the fragment shader) to produce the mesh weight value metadata. The ratio of contributions 1:8:8 places significant importance on the mesh distance from the camera and its already loaded texture mipmaps. We choose a value of 8 as it provides a reasonable number of discrete levels to rank textures for streaming (e.g. 2^8 median texture size in city datasets).

The fragment distance from camera contribution w_d prioritizes streaming of mesh textures in the foreground:

$$\begin{aligned} w_d &= (1 - c_w)^4 \\ c_w &= \min\left(\frac{1}{gl\_fragcoord.w}, 1\right) \end{aligned} \tag{2}$$

where c_w is the homogeneous clip space w-coordinate accessed in the fragment shader. It increases weighting for small or occluded meshes near the camera that are quite perceptible to the viewer. This assumes that the viewer's focus is always in the foreground within a view depicting both closeup and faraway meshes. This contribution is scene-independent, and one may adjust the power value 4 to better define the foreground.

The mipmap level contribution w_m prioritizes streaming of smaller mipmaps:

$$w_m = \min\left(\frac{M}{M_{max}}, 1\right) \tag{3}$$

where M is the displayed mipmap level and M_{max} is the highest mipmap level. It regulates texture resolutions within the camera view to achieve a smooth distribution of mipmap resolutions from the foreground to the background during texture streaming (Figure 4). It also prioritizes streaming of lower resolution mipmaps for rapid display when the GPU workload is high.

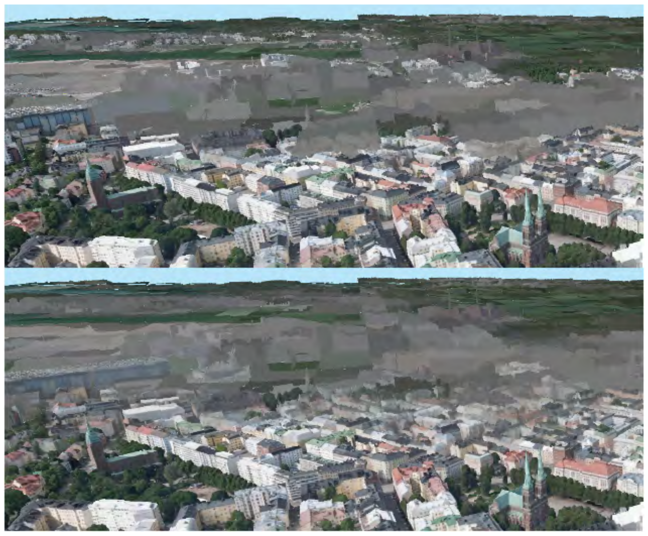
Figure 4: Rendering of the Helsinki city dataset comparing two configurations of the per-fragment weight W. *Top*: Considering only the fragment count contribution ($W = 1$). *Bottom*: Considering fragment count, fragment distance from camera, and mipmap level contributions. Our method indicates a smoother transition of texture resolutions from the foreground to the background during texture streaming, while the other shows incongruent texture resolutions between adjacent meshes in the middle ground.

In addition to our streaming prioritization, we also reduce texture pop-ins at the edges of the viewport by streaming textures outside of the viewport at a low priority. Specifically, we render offscreen at a larger FOV than the intended viewport FOV and assign a per-fragment weight $W = 1$ for meshes that lie outside of the viewport. This design anticipates arbitrary camera movements and allows the system to stream additional textures when the GPU workload is manageable.

3.3 Mesh metadata compaction

The MeshMetaData array buffer contains visibility information for all meshes in the scene. It is a sparsely populated array as the fragment shader only writes rastered (i.e. visible) mesh metadata at the respective array indices. At the end of the fragment shader stage, this array can be retrieved for texture streaming. However, copying the entire array, which can be more than five million elements long, into system memory is slow. Hence, we compact the sparse array using a compute shader prior to transfer. We use a simple atomic counter method [6], where each compute invocation, respective to an index in the sparse MeshMetaData array, copies an element contiguously into the output MeshMetaData array. Afterward, all elements in the sparse MeshMetaData array are reset to prepare for new mesh visibility metadata in the next frame. Note that the compute shader is always run after completion of the fragment shader stage.

Additionally, we propose a weight cutoff threshold τ to further reduce the MeshMetaData array and consequently the number of textures to stream. Small meshes with weights less than the defined threshold τ will not be copied to the output MeshMetaData. For the Berlin (atlas) dataset, we use $\tau = 150$ which balances texture streaming performance and streaming overheads. This threshold value is directly proportional to the per-fragment weight contributions defined previously.

Upon completion of the compute work, the compacted MeshMetaData array is copied to system memory for texture streaming. Note that we only retrieve the MeshMetaData array when the texture streaming task group is ready to be invoked, instead of retrieving the array every frame. Afterward, we sort it in descending weight value order for texture streaming. In our implementation, we perform a parallel CPU sort but one may prefer GPU sort if its overhead is justified.

3.4 Estimating texture streaming workload

The first subprocess in a texture streaming cycle is to estimate the texture streaming workload. Having a workload limit prevents overloading the GPU with excess memory binding commands, which causes severe frame rate stuttering. It also limits the time spent processing stale work that is no longer relevant to the current frame. When the workload limit is reached, the texture streaming task group will terminate. We propose two workload limits for early texture loading termination: **load time** (fixed limit) and **adaptive load size** (adaptive limit).

The **load time** limit defines a fixed amount of texture loading time. In the context of operations, it is the minimum interval before new texture streaming commands are submitted to the GPU. A shorter load time results in a lower texture load-to-display latency, ensuring that streamed textures are still relevant to the current camera view. A longer load time allows the GPU to consume a larger batch of streaming commands, improving overall streaming performance. However, this may overload the GPU and increase texture load-to-display latency. In our implementation, we set a maximum load time of 16 milliseconds.

The **adaptive load size** limit defines the maximum size of the loaded texture batch. It restricts the rate of textures transferred to the GPU and their associated sparse image memory bindings, which is the root cause of frame rate stuttering. The adaptive load size is not fixed but rather adaptively adjusted every frame, based on recent measurements of the rendering latency. This design anticipates fluctuations of rendering performance, for example when viewing dense and sparse city areas and when the cost of sparse image memory bindings increases, and adjusts the texture streaming rate accordingly so that frame rate stuttering is minimized.

To further manage the workload, we introduce an **adaptive mipmap bias** to restrict loading higher resolution mipmaps when the streaming bandwidth is low, as higher resolution mipmaps are exponentially more expensive to stream. It also prioritizes streaming smaller mipmaps and may defer streaming larger mipmaps depending on the streaming bandwidth. This design produces an even distribution of mipmap resolutions for any camera view and allows low-resolution mipmaps to be rapidly displayed especially under fast-moving bird's eye views. This mipmap bias value modifies the sampled mipmap value metadata for texture loading.

The adaptive load size s directly controls texture streaming rate:

$$\begin{aligned} s &= \max(l_i \cdot s_{max}, s_{min}) \\ l_i &= \text{clamp}\left(l_{i-1} + \text{sign}(-x)P\Delta t, 0, 1\right) \\ P &= |x| + (10|x|)^2 + (30|x|)^3 + (50|x|)^4 \\ x &= t_{w_i} - t_m \end{aligned} \tag{4}$$

where s_{min} and s_{max} is the minimum and maximum load size in MB respectively, l is the load factor clamped between 0 and 1, sign is a function returning the sign of a real number, Δt is the frame latency, and x is the difference between the weighted maximum frame latency t_w and the maximum frame latency threshold t_m. The adaptive load size s controls texture streaming rate and is computed every rendering cycle based on the relative change of frame latency. It has a fixed limit between s_{min} and s_{max} and is derived from a normalized load factor l clamped within the range of 0 and 1. During rendering, the load factor l is incremented or decremented until x is minimized, in other words, s will produce the optimal texture streaming rate as the weighted maximum frame latency t_w approaches the maximum frame latency threshold t_m. To put things simply, one can think of t_w as the average frame latency and t_m as a user-defined threshold constant. The polynomial P defines the exponential change in l based on the magnitude of x. Through it, texture streaming rate will be exponentially reduced after a spike in frame latency, and likewise, steadily (or exponentially) increased when frame latency remains below the threshold (left of Figure 5). The constants in P are empirically found to work well for s and with other parts of the adaptive streaming algorithm. At the end of the equation for l, we introduce the frame latency factor Δt to ensure a consistent rate of change across computer systems. In our implementation, we set the minimum load size s_{min} to the size of a sparse image memory block (i.e. the granularity of the sparse image selected by the graphics API). We also set a large maximum load size $s_{max} = 64MB$ that will not be reached by most systems. One may also use this variable to hard limit the texture streaming rate to free up computing resources.

The weighted maximum frame latency t_{w_i} reflects the maximum frame latency with an exponential decay:

$$\begin{aligned} t_{w_i} &= \max\left(t_{w_{i-1}} - (30d)^2\Delta t, \Delta t\right) \\ d &= \max\left(t_{w_{i-1}} - \Delta t, 0\right) \end{aligned} \tag{5}$$

where d is the difference between the previous weighted maximum frame latency $t_{w_{i-1}}$ and the current frame latency Δt, clamped to a lower limit of 0. t_w is designed to emphasize the spikes in frame latency for the load size function s. When $\Delta t > t_{w_{i-1}}$, t_{w_i} will immediately assume the high Δt value, and when $\Delta t < t_{w_{i-1}}$, t_{w_i} will (exponentially) decay towards the low Δt value depending on the magnitude of d. This widens the narrow spikes in frame latency which last only a few frames so that the load factor l can be integrated at smaller and more precise steps to meet the maximum frame latency threshold t_m. If t_w directly assumes the spiky Δt, texture streaming rate will be overall reduced as l is frequently under or overcompensating for the difference in frame latencies. We empirically derive the decay rate $(30d)^2$ to produce more reliable l values with fewer integration steps.

The maximum frame latency threshold t_m defines the point when the load factor l is reduced:

$$\begin{aligned} t_m &= \begin{cases} t_{wa}(1-0.85) + \alpha(0.85), & \text{if } t_{wa} > \alpha \\ t_{wa}(1-0.5) + \alpha(0.5), & \text{otherwise} \end{cases} \\ t_{wa} &= \frac{\sum_{j=1}^{200} t_{w_{i-j}}}{200} \end{aligned} \tag{6}$$

where t_{wa} is the average of 200 previous samples of t_w, and α is a user-defined target maximum frame latency. The threshold t_m is derived from the linear interpolation of t_{wa} and α depending on two conditions. When $t_{wa} > \alpha$, t_m will adhere closer to the target maximum frame latency α, and when $t_{wa} < \alpha$, t_m will vary towards the average of the weighted maximum frame latency t_{wa}. Since the load size s is designed to change based on relative frame latencies, we have to pin the threshold t_m to a maximum frame latency desirable to the user and also relative to t_w. We introduce some variation to t_m as it produces a smoother change in the load factor l compared to having $t_m = \alpha$, and this further reduces frame stuttering caused by a sharp increase in texture streaming rate when the prior average frame latency is low. We use 200 samples of t_w to compute t_{wa} as it best defines the average range of t_w under two seconds.

The adaptive mipmap bias b limits the maximum resolution of textures for streaming:

$$\begin{aligned}
b &= \beta \cdot \text{smoothstep}(0.96 t_{bl}, 1.2 t_{bu}, t_b) \\
t_{bl} &= \frac{\sum_{j=1}^{40} \left(t_{w_i}(1-0.8) + \min(t_{bl_{i-j}}, t_{w_i})(0.8) \right)}{40} \\
t_{bu} &= \frac{\sum_{j=1}^{40} \left(t_{w_i}(1-0.8) + \max(t_{bu_{i-j}}, t_{w_i})(0.8) \right)}{40} \\
t_b &= \frac{\sum_{j=1}^{5} t_{w_{i-j}}}{5}
\end{aligned} \tag{7}$$

where β is a user-defined maximum mipmap bias, smoothstep is the Hermite interpolation function defined in the fragment shader, t_{bl} and t_{bu} are respectively the lower and upper edges of the Hermite function, and t_b is the average of 5 previous weighted maximum frame latency t_w. The mipmap bias b is smoothly distributed (from 0 to β) using the normalizing smoothstep function that captures the fluctuations of t_b between the t_{bl} and t_{bu} edges. These two edges define the limits of the mipmap bias b and are computed by weighting the mean of t_w by a factor of 0.8 towards the minimum or maximum t_w over 40 samples, which is about one third of the frames rendered in a second. We use 5 samples of t_w for a smoother variation of b compared to using only one sample.

The adaptive mipmap bias b is designed to increase when the average frame latency t_b is increasing, and decrease when t_b is decreasing (right of Figure 5). We designate a user parameter β that scales the distribution of mipmap bias to better fit city datasets with different mipmap size distributions. It provides another level of control to optimize texture streaming, delaying the display of higher resolution mipmaps to minimize frame rate stuttering. In computing the edge values, we use a small number of 40 t_w samples to define the period under one second before b changes. This creates a frequent but stable variation of the mipmap bias, allowing more higher resolution mipmaps to be streamed while minimizing frame stuttering. To further reduce frame stuttering, we lower t_{bl} by a factor of 0.96 which increases the average mipmap bias and maintains a mipmap bias close to 0 when the streaming load is low. t_{bu} is also shifted by a factor of 1.2 to reduce the occurrence of high mipmap bias unless necessary, relaxing the importance of selecting β values.

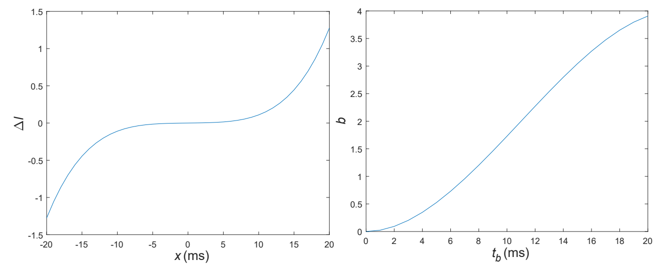

Figure 5: *Left*: The change of load factor l based on x, representing how much t_w deviates from t_m (Equation 4), assuming $\Delta t = 1ms$. *Right*: The mipmap bias b when t_b increases, assuming $\beta = 4$, $t_{bl} = 0ms$, and $t_{bu} = 16.67ms$ (Equation 7).

3.5 Texture image loading and transfer

The conjoined texture loading and sparse image caching subprocess loads texture images in secondary storage and transfers them to the sparse image in GPU memory. We begin this subprocess by parallelly iterating the MeshMetaData array in descending weight order, then load and stage texture mipmaps in the system memory, up to the defined workload limit. Afterward, we record texture copy commands to transfer staged textures into specific sparse image locations defined by our sparse image caching algorithm. We also allocate GPU memory and record memory binding commands for the appropriate regions in the sparse image. These commands will be submitted to the GPU at the end of the subprocess. Note that textures are transferred to the GPU asynchronously thru direct memory access (DMA) without further CPU intervention.

We design a texture state system to load texture mipmaps incrementally. Each texture streaming task cycle loads only a mipmap per mesh texture, starting from the initial mipmap level (i.e. the defined lowest resolution) down to the optimal mipmap level (defined in the MeshMetaData) modified by the adaptive mipmap bias. Specifically, if the modified mipmap level is higher or equal to the currently displayed mipmap level, then the next lower mipmap will not load. This design allows gradual refinement of textures, rather than immediately displaying the intended mipmap level, causing texture pop-ins. For each texture, we define three atomic object states in order: **Initial**, **Active**, and **Final**. Each object state will progress to the next if conditions are met at the end of an iteration.

The **Initial** state indicates that a texture is loaded for the first time. A unique sparse image location is assigned to the texture object designating the transfer destination for all its mipmaps. Then, the initial mipmap transfer command is recorded and memory is allocated and bound to the underlying sparse image location. Finally, the sparse image texture coordinate pointing to this mipmap is updated in the texture data buffer for fragment shader access. The **Active** state indicates that the texture has a mipmap to be transferred. The next higher resolution mipmap is transferred to the designated sparse image location. Just like the Initial state, the transfer command is recorded and appropriate sparse image memory is allocated and bound. The texture data buffer is also updated. The **Final** state indicates that the texture has transferred its base mipmap or is invalidated in the sparse image cache.

As an optimization, we copy mipmaps into system memory at the exact memory location in the image file to exponentially reduce file transfer time during texture streaming, as opposed to copying the entire image file into memory which can be megabytes in size. This greatly improves texture streaming performance, especially when requesting many smaller mipmaps for new camera views. To further reduce file access time, one may also pack the scene textures (or texture atlases) into a single file and directly access required memory locations.

3.6 Caching textures in the sparse image

We propose a parallel sparse image caching algorithm to cache loaded textures in GPU memory for rendering. It is intended for city datasets with uniform and non-uniform texture dimensions that are of powers of two. Our algorithm essentially packs textures with the same maximum extent, $\max(width, height)$, together in the same sparse image layer. For example, a layer designated with a maximum extent of 32 can only store textures with extent combinations of 4x32, 8x32, 16x32, and 32x32. In each sparse image layer, textures are uniformly placed in square regions where each region stores only one texture. While some rectangular textures do not fit the square regions optimally, this design is still necessary to align mipmap UV coordinates for all textures in the layer.

During runtime, our sparse image caching algorithm requests an available layer region to store a texture. If no layers are designated with that texture's maximum extent, or if previously designated layers are filled up, the next available layer in index order will be assigned. If all sparse image layers are assigned, the oldest layer in a ring index order is assigned instead and its existing textures are invalidated. Invalidated textures are no longer displayed and will have to be reloaded in the future. We use an unordered map (i.e. the *stl::unordered_map* data structure) to keep track of available sparse image regions to insert new textures. The map defines a key-value pair $\{extent, (layer, index)\}$ that maps a unique maximum texture *extent* to an available square region *index* at the sparse image *layer*. To support parallel sparse image caching, we use a parallel unordered map (i.e. the stl::unordered_map data structure extended with parallel search and insert methods), and define *layer* and *index* as atomic counters pointing to a sparse image *layer* (in a ring order) and a square region *index* within that layer (in left to right, top to bottom order). We also use a third atomic counter *nextLayer* to track the next available sparse image layer for assignment.

Overall, our sparse image caching algorithm enables semi-compact packing of textures and the use of atomic counters for fast caching performance. The number of sparse image layers is predefined and guarantees a bounded memory limit. One may better utilize texture caching space by dedicating unique layers for rectangular texture extents. However, this will further inflate the number of sparse image layers and memory binding costs and thus impact texture streaming performance.

3.7 City Colormap and texturing artifacts

The City Colormap buffer stores a color per mesh texture for the entire city dataset downsampled from the original textures. It remains permanent in GPU memory, acting as a color placeholder when sampling textures not yet cached in the sparse image. However, transitioning from a single mesh color to low-resolution mipmaps produces significant texture pop-in artifacts. This distinct color change is due to sampling errors in smaller mipmaps that are increasingly erroneous, and also due to preexisting mipmapping artifacts in city datasets such as erroneous sampling of black borders and gamma adjustments. We propose to minimize texture pop-in artifacts by linearly interpolating the sampled sparse image colors with the original colors stored in the City Colormap, based on the displayed mipmap level:

$$c = \left(1 - \frac{M}{M_{max}}\right) c_s + \left(\frac{M}{M_{max}}\right) c_o \tag{8}$$

where c_s is the sampled color, c_o is the original color, M is the displayed mipmap level, and M_{max} is the maximum mipmap level of the mesh texture. This method retains correct low-frequency colors for smaller mipmaps while smoothly blending higher frequency details from higher-resolution mipmaps, significantly reducing texture pop-in artifacts.

4 Evaluation

Our texture streaming system performance and rendering quality are evaluated using three large city datasets (Table 1). We assess our adaptive streaming algorithms using a designed benchmark and show how the various system parameters (Table 2) affect texture streaming and rendering performance. Rendering quality in terms of texture pop-in errors is evaluated. We also compare our new texture streaming system to our previous method [20], as well as the Unity game engine. Our test system is equipped with an Intel i7-8700K CPU, a Nvidia RTX 2080Ti GPU, and a Samsung 970 EVO 1TB SSD, running on Windows 10 and renders into a 3840×2160 resolution framebuffer. We provide a video capturing real-time rendering results as part of the supplementary material. You may also download a higher resolution results video here: https://drive.google.com/file/d/1UwAW_2jWgVZf-ErBzpeKLc3GPF7Ni3cK/view?usp=sharing.

4.1 Benchmarking methodology

All of our texture streaming performance measurements are recorded using a benchmark, where a virtual camera moves in a fixed path through the city, covering sparse and dense building areas. The benchmark is designed to thoroughly evaluate our texture streaming system and imitate some aspects of real-world usage. We fix the camera at a bird's eye view (Figure 1), capturing buildings with the highest mipmap resolutions in the foreground and the full range of resolutions in the background. This allows us to fully evaluate streaming performance and the evenness of mipmap resolution distribution under our mipmap prioritization system. At four determined points of the flythrough, the camera will pause for two seconds before continuing, causing more high-resolution mipmaps to be streamed. This tests our adaptive streaming algorithm's performance when transitioning between streaming lower and higher resolution mipmaps. At another four determined points of the flythrough, the camera will immediately switch to a new view, forcing the adaptive streaming algorithm to catch up on the streaming work. This tests the algorithm's ability to minimize frame rate stuttering without over-throttling texture streaming rate. Each benchmark is standardized across different computer systems and for each city dataset. For the Berlin and the Berlin (atlas) dataset, the benchmark duration is 68.6 seconds, and for the Helsinki dataset, the duration is 43.7 seconds.

We measure the median, average, and maximum frame latency and mipmap transfer rate to judge the overall streaming performance. The average and maximum frame latency is used as an indicator of frame rate stuttering. We can estimate the amount of frame stuttering by taking the difference between the average frame latency without texture streaming and the measured average frame latency. We can also estimate the amount of frame stuttering from the maximum frame latency value, where a large value will result in an observable frame rate stutter. Here we take the maximum t_w latency instead for a more accurate result. The transfer rate measurement is a performance indicator that shows the count and size of mipmaps transferred per second. The relation of mipmap count and size can indicate lower or higher resolution mipmaps being streamed when different system parameters are used.

4.2 Texture streaming performance

The following charts (Figure 6) show the measurements of a benchmark for the Berlin (atlas) dataset, recording changes in the adaptive streaming algorithm's parameters and the resulting texture streaming performance. Throughout the benchmark, we capture a measurement at every 0.10423 seconds interval. The top-left chart shows the weighted maximum frame latency t_w and how it varies with the maximum frame latency threshold t_m to produce the load size. We can see the changes in the load size and the proportional transfer rates in MB/s based on how much t_w deviates from t_m. The top-right chart shows the smoothstep function parameters t_b, t_{bu} (upper edge), and t_{bl} (lower edge) that control the mipmap bias. t_b tends to fluctuate near the lower edge when the frame latency is decreasing or stabilizing, and near the upper edge when frame latency is increasing, producing a relatively turbulent mipmap bias compared to the load size. The rapid fluctuation in mipmap bias is designed to regulate the number of high-resolution mipmaps streamed without completely stalling mipmap refinement when the frame latency is high, minimizing the perceivable texture pop-ins.

Frame latency is related to the load size to a certain degree. When the average load size increases, the average frame latency also increases, and well as the magnitude of frame latency spikes. We notice a tendency of frame stuttering when switching to a new camera view, caused by an abrupt increase in streaming workload, and also randomly occurring stutters at different times of the same benchmark. Unfortunately, these frame latency spikes cannot be fully eliminated even when the load size is set to a constant low value of 0.05MB. Hence, we conclude that binding sparse image memory blocks has an unknowable system-level performance cost. Nevertheless, our adaptive streaming algorithms can quickly react and minimize any frame latency spikes without over-throttling texture streaming rate, by rapidly adjusting the load size where necessary.

In the series of benchmarks (Table 3, Table 4, Table 5), we analyze our adaptive texture streaming performance for the Berlin

Table 1: Attributes of the three city datasets that are used to evaluate our adaptive streaming system.

Dataset	Triangles	Geometry size (MB)	Textures	Texture size (GB)	Format	Texture extents (2^x)
Berlin [1]	15,363,486	935	5,712,309	41.6	BC1	2 to 9
Berlin (atlas)	15,363,486	1092	233,110	40.8	BC1	9
Helsinki [15]	115,112,748	6096	25,354	3.5	BC1	8, 9

Table 2: Optimized system parameters used in all our texture streaming performance benchmarks unless specified. Given any city dataset and computer system, users will only need to specify the α parameter and optionally fine-tune the streaming performance using the β parameter.

Parameters	Value	Description
Load time	16ms	Maximum load time
s_{min}	0.065MB	Minimum load size
s_{max}	64MB	Maximum load size
α	16ms	Target maximum frame latency
β	8	Maximum mipmap bias
F_s	16	Anisotropic filtering level
τ	150	Mesh weight cutoff threshold

(atlas), Berlin, and Helsinki datasets, and compare them to a baseline without adaptive texture streaming. Without adaptive streaming, textures are streamed as fast as possible, producing the highest texture transfer rate measurement. We also observe a significant amount of frame stuttering after about 5 seconds into the benchmark, with maximum frame latencies surpassing 100ms, resulting in a non-interactive rendering. With adaptive streaming ($\alpha = 12ms$ in Table 3), the average texture streaming rate in MB/s is reduced by 47% to support a low, stutter-free average rendering latency of 5.3ms, which is an 88% reduction from the baseline. The number of mipmaps streamed per second remains similar to the baseline, and even increases as α approaches 16ms, indicating the deferment of streaming higher resolution mipmaps by our adaptive mipmap bias algorithm. Adaptive texture streaming performance is also similarly reflected for the Berlin and Helsinki datasets ($\alpha = 14ms$ in Table 4, $\alpha = 25ms$ in Table 5). In the texture-heavy Berlin dataset, we even see a 9% increase in mipmaps streamed per second which results in a quicker display of lower resolution textures for new camera views with fewer texture pop-ins. Additionally, frame stuttering is significantly reduced in all three datasets, both from measured maximum frame latency and from detailed observations of the rendering. We measure a 97%, 51%, and 80% reduction in maximum frame latency using the α parameters of $12ms$, $14ms$, and $25ms$ in the Berlin (atlas), Berlin, and Helsinki datasets respectively. Also from the benchmarks, we can see that the texture transfer rate and frame latency increase as the target maximum frame latency α increases, supporting the design of our adaptive streaming algorithms.

4.3 Texture streaming parameters

To determine the performance, robustness, and ease of use of our adaptive streaming algorithms, we evaluate the target maximum frame latency parameter α in the three structurally different city datasets. For each dataset, we tested a reasonable range of α values that is about two times the average rendering latency without texture streaming and fixed the β parameter to 8, which performs well for the Berlin (atlas) dataset. Results for the Berlin (atlas) dataset (Table 3) distinctly show a steady increase in frame latency and transfer rate as α increases from 12 to 18. At $\alpha = 12$, there is few observable frame rate stuttering even under extreme camera movements, and at $\alpha = 18$, frame rate stuttering is noticeable during the benchmark. We choose $\alpha = 16$ for a better texture streaming performance to suit our geospatial visualization needs. While small frame stutters can occasionally occur when switching camera views, it is a worthy tradeoff for increased texture streaming rate and reduced texture pop-in artifacts. For other purposes that require smoother frame rates, a lower α can be used.

Results for the Berlin dataset (Table 4) show a similar increase in frame latency and transfer rate as α increases, albeit at a smaller magnitude with substantially lower texture transfer rates compared to the Berlin (atlas) results. This is due to the high texture file access overhead since the Berlin dataset contains millions of individual textures. For the Helsinki dataset (Table 5), texture transfer rate in MB/s increases with α, but frame latency remains fairly constant. Here, we use a higher α range due to the high average rendering latency, as the Helsinki dataset has a much larger geometry count. We noticed that higher resolution mipmaps tend to be delayed more than those in the Berlin datasets, due to the presence of high (median and average) mipmap bias during streaming. Since the Helsinki dataset contains a larger texture extent on average compared to the Berlin datasets, introducing a high mipmap bias will delay streaming of higher resolution mipmaps. Therefore, we define the maximum mipmap bias β parameter to accommodate datasets with different mipmap extent distributions.

We evaluate the influence of the β parameter using the Helsinki dataset with $\alpha = 30$ (Table 6). As β decreases from 8 to 0, texture transfer rate increases while frame latency remains relatively steady. However, the maximum frame latency, as well as observed frame rate stuttering, has substantially reduced when $2 \geq \beta \leq 6$. This is because the fluctuation of mipmap bias is within a smaller range suitable for the dataset, where the adaptive load size algorithm will not overcompensate by streaming many large mipmaps in the subsequent time instances. To determine the β parameter, users can take the default value of 8 and lower it until the mipmap presentation delay is tolerable, or increase the value until the desired transfer rate is met. Overall, our adaptive texture streaming system can adapt to different datasets, greatly reducing frame rate stuttering, even when using non-optimal parameters. System parameters can be easily and intuitively adjusted by setting the maximum target latency parameter α and the maximum mipmap bias parameter β.

Additional parameters (Table 2) can be tuned to further improve streaming or rendering performance to better match the local system's computational bandwidth. The fixed load time of 16ms and the maximum load size s_{max} can be used to hard limit texture streaming rate which reduces frame rate stuttering, and also can be used to free up system resources. Likewise, adjusting the τ value modifies the camera range of mesh textures being streamed, affecting the maximum streaming cycle workload.

4.4 Rendering quality

Our texture streaming system minimizes texture pop-in artifacts and maintains visual coherency of proximate meshes using our adaptive streaming algorithms, our mipmap color blending method, and our streaming prioritization method. In a fly-through of the Berlin (atlas) dataset that covers all areas of the city, we consistently observed minimal frame rate stuttering and very few texture pop-in artifacts. Texture mipmaps are rapidly streamed and displayed up to the optimal mipmap level (at 16x anisotropic filtering) with minimal perceivable texture pop-ins, except during some instances when the camera is very quickly strafed, rotated, or snapped into a new view.

Table 3: Adaptive streaming performance for the Berlin (atlas) dataset using various α parameter with $\beta = 8$.

	Median	Average	Max
No adaptive algorithms			
Frame latency (ms)	6.8	42.5	471.16
t_w latency (ms)	27.03	56.22	471.16
Transfer rate (Mip/s)	7997	9628	32807
Transfer rate (MB/s)	74.96	79.95	207.55
Load size s (MB)	64	64	64
Mipmap bias b	0	0	0
$\alpha = 12ms$			
Frame latency (ms)	4.55	5.3	15.98
t_w latency (ms)	9.53	9.73	17.36
Transfer rate (Mip/s)	8621.5	9297.1	22310
Transfer rate (MB/s)	41.82	42	93.67
Load size s (MB)	0.75	0.74	1.75
Mipmap bias b	0.49	1.31	8
$\alpha = 14ms$			
Frame latency (ms)	4.58	6.1	24.59
t_w latency (ms)	10.91	11.01	24.59
Transfer rate (Mip/s)	7818	9574.7	21914
Transfer rate (MB/s)	41.96	47.38	136.87
Load size s (MB)	0.81	0.96	2.64
Mipmap bias b	0.4	1.21	8
$\alpha = 16ms$			
Frame latency (ms)	4.52	6.47	26.16
t_w latency (ms)	12.21	12.26	26.16
Transfer rate (Mip/s)	8936.5	9788.26	21075
Transfer rate (MB/s)	51.15	51.94	146.8
Load size s (MB)	0.96	1.17	3.32
Mipmap bias b	0.38	1.2	8
$\alpha = 18ms$			
Frame latency (ms)	4.61	7.18	29.51
t_w latency (ms)	13.29	13.37	29.51
Transfer rate (Mip/s)	8455	9815.76	25873
Transfer rate (MB/s)	46.3	53.02	154.32
Load size s (MB)	1.19	1.32	3.79
Mipmap bias b	0.4	1.16	8

Table 4: Adaptive streaming performance for the Berlin dataset using various α parameter with $\beta = 8$.

	Median	Average	Max
No adaptive algorithms			
Frame latency (ms)	3.58	6.07	51.72
t_w latency (ms)	16.37	16.86	51.72
Transfer rate (Mip/s)	14316	14943.75	29458
Transfer rate (MB/s)	15.98	16.69	43.45
Load size s (MB)	64	64	64
Mipmap bias b	0	0	0
$\alpha = 14ms$			
Frame Latency (ms)	3.71	4.99	25.13
t_w latency (ms)	11.2	11.39	25.52
Transfer rate (Mip/s)	15680	16384.59	30354
Transfer rate (MB/s)	11.35	11.67	31.27
Load size s (MB)	0.37	0.79	3.74
Mipmap bias b	0.37	1.61	8
$\alpha = 16ms$			
Frame Latency (ms)	3.71	5.38	22.33
t_w latency (ms)	12.26	12.39	35.11
Transfer rate (Mip/s)	17192	17486.05	31961
Transfer rate (MB/s)	12.24	12.78	29.18
Load size s (MB)	0.43	0.93	5.49
Mipmap bias b	0.34	1.46	8
$\alpha = 18ms$			
Frame Latency (ms)	3.67	5.21	26.59
t_w latency (ms)	13.02	13.07	26.59
Transfer rate (Mip/s)	16610	17110.94	30636
Transfer rate (MB/s)	12.19	12.59	30.07
Load size s (MB)	0.55	2.01	10.78
Mipmap bias b	0.34	1.44	8

Also when viewing city areas with high building densities or when the streaming workload is too high, the adaptive streaming algorithms may delay refinement of higher resolution mipmaps, or omit streaming of low priority meshes completely. This leaves prominent areas such as the foreground of the camera view with fuzzy, low-resolution textures, which causes texture pop-in artifacts when the streaming system eventually catches up. Generally, the delay in texture refinement is still tolerable for already loaded textures of a high resolution, especially under our mipmap color blending method. However, fully delayed streaming of textures may cause the texture resolution of proximate meshes to differ greatly, disrupting the visual coherency of meshes. Nevertheless, we can mitigate this effect for prominent, foreground textures using streaming prioritization, but ultimately, it is constrained by the system's capabilities. We conclude that a high texture streaming performance is most important to minimize texture pop-in artifacts, since any visible mipmap refinement will be resolved quickly, but not at the expense of increased frame stuttering, as it greatly disrupts visual coherency.

In the following figure (Figure 7), we show the effectiveness of our mipmap color blending method compared to direct mipmap sampling to reduce texture pop-in artifacts. We capture a static view at increasing mesh texture resolution per frame and measure the differences of pixel color intensities between each frame under both methods. From frames 0 to 7, we can see that direct mipmap sampling produces a significantly greater change of color intensities with a maximum mean squared error (MSE) of 438 compared to our color blending method which produces a maximum MSE of 62. Our method reduces maximum MSE by 86%, indicating a significant reduction of texture pop-in artifacts, especially at lower mipmap resolutions (shown in frames 1 and 2). Overall, our method manages to retain low frequency, original mesh colors while gradually introducing higher frequency texture details, greatly suppressing any mipmapping artifacts.

4.5 Comparison with our previous method

Similar to our new texture streaming method, our previous method [20] adjusts the workload per streaming cycle by computing the load size s and mipmap bias b based on previously measured streaming latencies. However, s and b are derived from streaming latency measurements which are data and system dependent, using fixed functions with parameters determined by the user to work optimally. In our new method, we derive s and b based on the relative changes in rendering latency, using adaptive functions that maintain these changes in the load factor l. We also condense the number of system parameters the user has to consider to two (the target maximum frame latency α and the maximum mipmap bias β) and adopt a standardized weighted maximum frame latency t_w as input to all adaptive streaming functions. The parameters α and β corresponds

Table 5: Adaptive streaming performance for the Helsinki dataset using various α parameter with $\beta = 8$.

	Median	Average	Max
No adaptive algorithms			
Frame latency (ms)	9.94	10.41	146.76
t_w latency (ms)	18.7	18.25	146.76
Transfer rate (Mip/s)	1591	2366.22	16149
Transfer rate (MB/s)	33.63	34.32	92.02
Load size s (MB)	64	64	64
Mipmap bias b	0	0	0
$\alpha = 25ms$			
Frame latency (ms)	10.25	10.99	28.94
t_w latency (ms)	18.74	18.46	38.64
Transfer rate (Mip/s)	1367	2288.61	14592
Transfer rate (MB/s)	16.52	24.71	99
Load size s (MB)	7.49	16.03	63.82
Mipmap bias b	0.75	1.56	8
$\alpha = 30ms$			
Frame latency (ms)	10.28	11.22	74.41
t_w latency (ms)	18.49	18.38	74.41
Transfer rate (Mip/s)	1318	2260.06	12945
Transfer rate (MB/s)	21.09	25.32	103.91
Load size s (MB)	16.58	21.31	59.78
Mipmap bias b	1.09	1.81	8
$\alpha = 35ms$			
Frame latency (ms)	10.17	10.65	35.57
t_w latency (ms)	18.44	17.81	35.57
Transfer rate (Mip/s)	1340	2277.58	12440
Transfer rate (MB/s)	26.57	27.5	97.15
Load size s (MB)	44.4	40.73	64
Mipmap bias b	0.94	1.78	8

Table 6: Adaptive streaming performance for the Helsinki dataset under various β parameter with $\alpha = 30ms$.

	Median	Average	Max
$\beta = 0$			
Frame latency (ms)	10.14	10.92	147.09
t_w latency (ms)	18.75	18.27	147.1
Transfer rate (Mip/s)	1606	2390.74	17147
Transfer rate (MB/s)	32.94	34.24	97.79
Load size s (MB)	30.25	28.56	64
Mipmap bias b	0	0	0
$\beta = 2$			
Frame latency (ms)	10.17	11.41	26.1
t_w latency (ms)	18.84	18.52	38.36
Transfer rate (Mip/s)	1316	2294.75	14589
Transfer rate (MB/s)	28.78	30.16	114.76
Load size s (MB)	11	14.29	35.39
Mipmap bias b	0.22	0.45	2
$\beta = 4$			
Frame latency (ms)	9.94	9.99	34.02
t_w latency (ms)	18.45	18.06	35.09
Transfer rate (Mip/s)	1314	2288.19	14320
Transfer rate (MB/s)	25.93	27.22	87.2
Load size s (MB)	11.76	15.72	37.14
Mipmap bias b	0.57	1	4
$\beta = 6$			
Frame latency (ms)	10.19	10.84	37.5
t_w latency (ms)	18.56	18.43	37.5
Transfer rate (Mip/s)	1364	2279.1	11733
Transfer rate (MB/s)	23.59	26.01	79.33
Load size s (MB)	10.72	13.94	31.01
Mipmap bias b	0.85	1.38	6

to the ideal binding time and the maximum mipmap bias defined in our previous method. The difference is that we more precisely consider the frame latency instead of the streaming latency, as the frame latency accounts for all GPU operations including the sparse image binding time. Furthermore, adjusting α based on frame latency is more intuitive for users as it is a common measurement.

We measure the rendering quality between our new and old methods using the same benchmarking setup (Table 7). Our new method is visibly faster (shown in video) to refine textures to the highest resolution, especially for textures prominently located in the foreground, resulting in less visible texture pop-ins. Generally, texture streaming performances of our new and old methods are quite comparable in all three datasets. For example in the Berlin (atlas) dataset, the old method's transfer rate of 46.94MB/s is similar to our new method's transfer rate of 47.38MB/s under $\alpha = 14ms$. While our average frame latency is 13% higher, our maximum frame latency is 17% lower which results in less frame stuttering. Also, the transfer rate in Mip/s is lower in our method, indicating higher resolution mipmaps being streamed. For the Helsinki dataset, we observed a 20% reduction of transfer rate in MB/s at similar average and maximum frame latencies when using our new method ($\alpha = 30ms$, $\beta = 4$). This is because the fixed functions of the old method can be more efficient when the system and dataset-specific streaming workload is known but requires users to determine it and tune the system parameters accordingly. In our new method, we can automatically determine and adjust the streaming workload at runtime, making our method more robust. Additionally, our new method is faster to determine and refine texture resolutions which are more important than displaying more higher resolutions textures.

4.6 Limitation

Since our adaptive texture streaming method automatically adjusts the streaming rate based on rendering latency, it depends on having constantly fluctuating frame latencies. It does not work well with applications that require limiting the maximum rendering rate, most commonly through vertical synchronization. In our testing of the Berlin (atlas) dataset, using $\alpha = 18$ with a 16.68ms rendering latency limit, we measured a 43% reduction in texture transfer rate at 30MB/s compared to our benchmark's rate at 53.02MB/s. Nevertheless, our adaptive streaming algorithm remains performant in suppressing frame rate stuttering, rapidly reducing the workload during periods of high rendering latency, while slowly increasing transfer rate when frame latency is at the upper limit of 16.68ms. Texture streaming rate increases slowly because rendering latency is mostly invariant at the upper limit. We like to resolve this in future work by automatically adapting our texture streaming rate to the memory binding latency as well as the rendering latency.

Another common limitation is cache trashing, which occurs in the sparse image cache when camera visible textures are being constantly replaced by newer textures streaming in, due to insufficient cache size to hold all visible textures. As textures are cached in a ring buffer order, textures (that are still camera visible) in the oldest sparse image layer are invalidated by the new textures streaming in, causing this cycle to repeat. We can simply resolve this by increasing the cache, specifically by increasing the number of sparse image layers at the expense of increased sparse image memory binding

Table 7: Texture streaming performance of our previous method [20], tested using the camera fly-through benchmark in this paper. The system parameters specified in our previous paper are used.

	Median	Average	Max
Berlin (atlas)			
Frame latency (ms)	4.34	5.33	29.79
t_w latency (ms)	10.43	10.51	29.79
Transfer rate (Mip/s)	8800	10121.74	32413
Transfer rate (MB/s)	42.79	46.94	171.19
Load size s (MB)	0.89	3.1	64
Mipmap bias b	0.83	1.51	10
Berlin			
Frame latency (ms)	3.71	4.84	18.86
t_w latency (ms)	10.32	10.47	28.62
Transfer rate (Mip/s)	17588	18539.91	30573
Transfer rate (MB/s)	10.74	11.85	32.21
Load size s (MB)	0.33	7.84	64
Mipmap bias b	0.7	1.35	10
Helsinki			
Frame latency (ms)	9.96	10.27	33.89
t_w latency (ms)	18.8	17.98	33.89
Transfer rate (Mip/s)	1667	2384.52	16175
Transfer rate (MB/s)	33.1	34.2	92.85
Load size s (MB)	64	55.06	64
Mipmap bias b	0	0.06	10

time and reduced texture streaming performance.

4.7 Comparison with the Unity game engine

Game engines such as Unity and Unreal are widely adopted to produce and visualize geospatial data. They feature texture streaming capabilities to handle scenes with large amounts of texture data. We briefly compare Unity's (v2020.2.0) texture streaming and rendering capabilities to ours, using the Helsinki dataset with a benchmark that moves the camera from a bird's eye view to a closeup view. Unity's texture streaming parameters are also set similarly. From the Unity rendering (provided in video), we can see mesh textures being initially presented at very low resolutions with erroneous mipmap colors. As the camera moves closer to the meshes, textures are quickly refined to the highest resolution which causes texture pop-in artifacts. They are refined in an arbitrary order which causes low-resolution, blotchy areas in the scene (Figure 8). Compared to Unity, our adaptive streaming and color blending method consistently refines mesh textures at all mipmap levels with minimal texture pop-ins. In terms of rendering performance, we are unable to make a fair comparison as Unity appears to allocate GPU memory in advance for texture streaming.

5 Conclusion and Future Work

In this paper, we have proposed an adaptive, multithreaded, GPU-driven texture streaming pipeline for high-performance texture streaming. Our main contribution is an algorithm that automatically and adaptively adjusts texture streaming workload based on measured frame latencies, minimizing sparse image memory binding costs and frame rate stuttering. Our pipeline determines mesh visibility and streaming priority in shader and caches textures in the sparse image semi-compactly in parallel at runtime, without requiring a preprocessing step. Furthermore, we significantly reduce texture pop-in artifacts by blending sampled colors with the original mesh colors stored in a City Colormap based on mipmap levels. We also proposed to generate texture atlases based on similarly orienting normals to improve texture streaming performance. Our new method indicates clear improvements in texture streaming rate and rendering quality compared to the baseline, the Unity game engine, and our prior method. It is also robust to support different city datasets and computer systems with intuitive and easy-to-tune system parameters.

For future work, we like to explore the idea of further reducing frame stuttering by explicitly amortizing the texture streaming workload over time. Currently, we specify a fixed time limit for each cycle of texture streaming, and adaptively adjust the load size and mipmap bias to indirectly delay streaming work over time. However, there may be a case involving many texture streaming cycles with small workloads causing GPU overload, and we like to resolve this issue by adapting workload across time.

We also want to explore how well our texture streaming approach performs in general scenes, specifically, scenes with close-up camera view with higher resolution textures. We believe our proposed approach can be directly applied to general scenes, as mesh occlusion and perceptibility are already considered during texture streaming. To further minimize frame rate stuttering when streaming many high-resolution mipmaps with extents over 1024x1024, we propose to stream them at smaller discrete subregions, taking into account subregion resolution distributions and texture file accesses to improve rendering quality and performance.

In our current texture streaming pipeline, we use a 4090x4090 City Colormap texture for the Berlin (atlas) dataset to sample original mesh colors in shader. As arbitrarily sampling a large texture can impact rendering performance due to memory cache misses, we recommend using a more cache-aware method when sampling large textures frequently. We want to explore permanently caching per-mesh colors in the mip tail regions of the sparse image which should facilitate efficient memory management.

Appendix

This article is an author accepted version. You can find the published version at: https://doi.org/10.1007/s00371-021-02152-z.

Acknowledgments

This research is supported by the National Research Foundation, Singapore under its International Research Centres in Singapore Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

References

[1] B. Berlin Senate Department for Economics, Energy and Public Enterprises. Berlin 3d. https://www.businesslocationcenter.de/en/economic-atlas/download-portal/. Accessed: 2019-10-01.

[2] K. Chen. Adaptive virtual texture rendering in far cry 4. In *Game Developers Conference*, p. 869, 2015.

[3] D. Cline and P. K. Egbert. *Interactive display of very large textures*. IEEE, 1998.

[4] P. Cozzi and K. Ring. *3D engine design for virtual globes*. Crc Press, 2011.

[5] C. Han and H. Hoppe. Optimizing continuity in multiscale imagery. *ACM Transactions on Graphics (TOG)*, 29(6):1–10, 2010.

[6] C. Hollemeersch, B. Pieters, P. Lambert, and R. Van de Walle. Accelerating virtual texturing using cuda. *GPU Pro: advanced rendering techniques*, 1:623–641, 2010.

[7] W. Huang and J. Chen. A virtual globe-based time-critical adaptive visualization method for 3d city models. *International journal of digital earth*, 11(9):939–955, 2018.

[8] M. Krämer and R. Gutbell. A case study on 3d geospatial applications in the web using state-of-the-art webgl frameworks. In *Proceedings of the 20th International Conference on 3D Web Technology*, pp. 189–197. ACM, 2015.

[9] S. Lefebvre, J. Darbon, and F. Neyret. Unified texture management for arbitrary meshes. *INRIA*, 2004.

[10] A. Loya, N. Adabala, A. Das, and P. Mishra. A practical approach to image-guided building facade abstraction. In *Computer Graphics International*. Citeseer, 2008.

[11] M. Mittring and Crytek. Advanced virtual texture topics. In *ACM SIGGRAPH 2008 Games*, pp. 23–51. ACM, 2008.

[12] J. H. Mueller, P. Voglreiter, M. Dokter, T. Neff, M. Makar, M. Steinberger, and D. Schmalstieg. Shading atlas streaming. In *SIGGRAPH Asia 2018 Technical Papers*, p. 199. ACM, 2018.

[13] J. Obert, J. van Waveren, and G. Sellers. Virtual texturing in software and hardware. In *ACM SIGGRAPH 2012 Courses*, p. 5. ACM, 2012.

[14] P. Schmitz, T. Blut, C. Mattes, and L. Kobbelt. High-fidelity point-based rendering of large-scale 3d scan datasets. *IEEE Computer Graphics and Applications*, 2020.

[15] H. R. I. H. service. Reality mesh of entire helsinki (2017). https://hri.fi/data/en_GB/dataset/helsingin-3d-kaupunkimalli. Accessed: 2019-10-01.

[16] C. C. Tanner, C. J. Migdal, and M. T. Jones. The clipmap: a virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 151–158. ACM, 1998.

[17] J. Van Waveren. Software virtual textures. *Id Software LLC, Tech. Rep*, 2012.

[18] H. Wang, Y. Wexler, E. Ofek, and H. Hoppe. Factoring repeated content within and among images. *ACM Transactions on Graphics (TOG)*, 27(3):1–10, 2008.

[19] M. Widmark. Terrain in battlefield 3: A modern, complete and scalable system. *GDC Presentation* http://publications.dice.se/attachments/GDC12_Terrain_in_Battlefield3.pdf *(2012-05-31)*, 2012.

[20] A. Zhang, K. Chen, H. Johan, and M. Erdt. High performance texture streaming and rendering of large textured 3d cities. In *2020 International Conference on Cyberworlds (CW)*, pp. 17–24. IEEE, 2020.

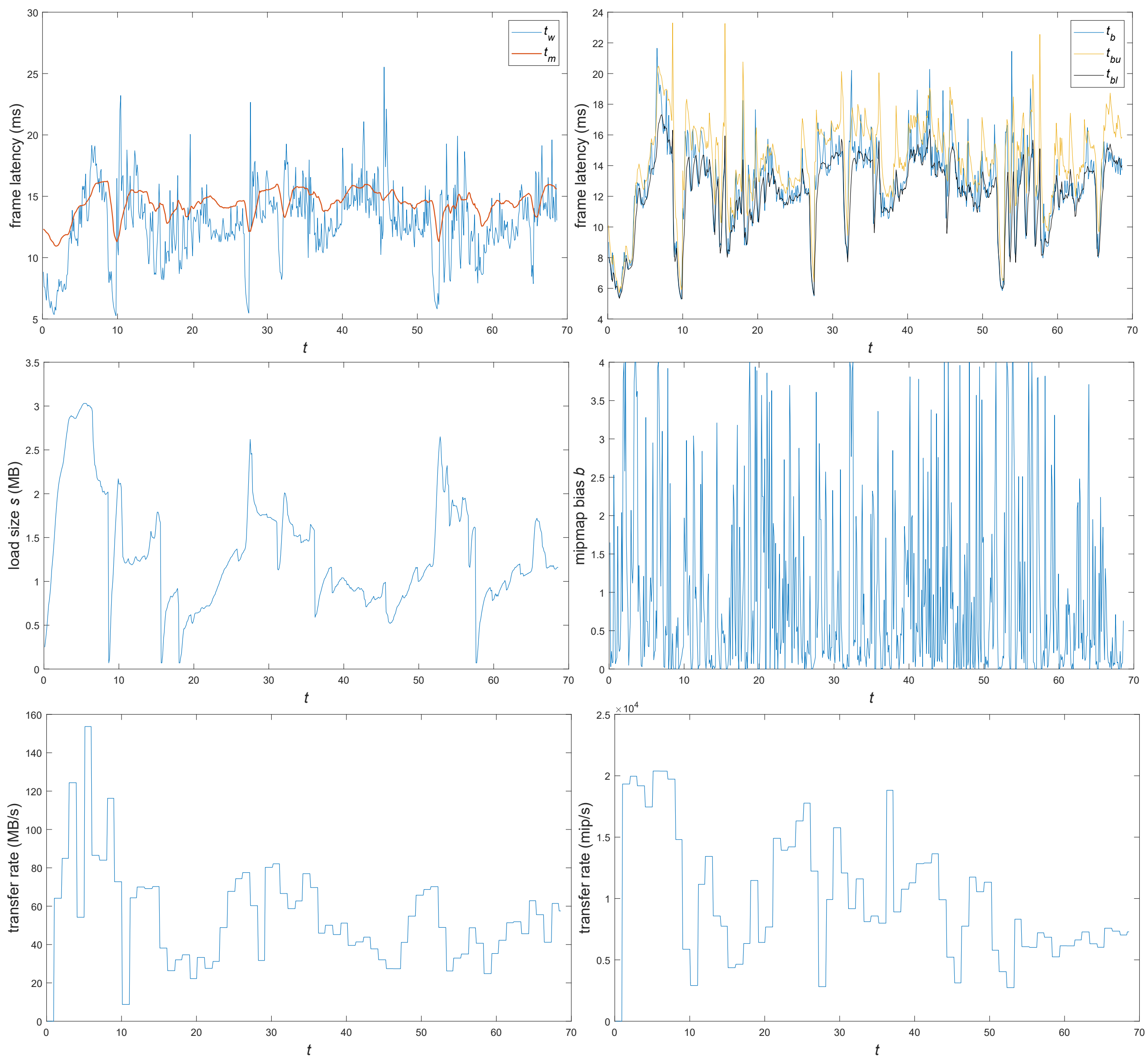

Figure 6: Our texture streaming system performance over 68 seconds for the Berlin city (atlas) dataset. *Top-left*: Measurements of the weighted maximum frame latency t_w and the maximum frame latency threshold t_m, which affects the load size s. *Top-right*: Measurements of the average weighted maximum frame latency t_b, the lower edge t_{bu}, and the upper edge t_{bl}, which affects the mipmap bias b. *Middle*: Measurements of the load size s and the mipmap bias b. *Bottom*: Measurements of texture streaming rate in megabytes transferred per seconds and in mipmaps transferred per second.

Algorithm 1: High-level pseudocode for our texture streaming pipeline.

```
// Texture streaming operations in the rendering loop.
while $isRendering$ do
    // Invoke compute shader to compact mesh metadata (Section 3.3).
    // Synchronization:  Run after fragment shader stage (Section 3.2).
    InvokeCompactMetadata();
    // Compute the adaptive load size and mipmap bias (Section 3.4).
    $(s, b) \longleftarrow$ EstimateWorkload();
    // Check if a texture streaming task is running.
    if $isStreaming$ = false then
        // Retrieve the compacted mesh metadata and store in array (Section 3.3).
        // The array element type is (mipmap level, weight value, meshID).
        $arrayMetadata \longleftarrow$ RetrieveMetadata();
        // Sort the array by descending weight value.
        SortArray($arrayMetadata$);
        // Enqueue a texture streaming task for asynchronous execution.
        // The atomic boolean, isStreaming, allows only one concurrent instance of the task.
        begin asynchronous task
            $isStreaming \longleftarrow$ true;
            // Set invalidated mesh textures to not be sampled in shader.
            UnloadTextures();
            // Load visible mesh textures into the sparse image.
            LoadTextures($arrayMetadata$, $s$, $b$);
            $isStreaming \longleftarrow$ false;
        end
    end
end
// Load and cache mesh textures in the sparse image.
Function LoadTextures($arrayMetadata$, $s$, $b$):
    $loadedBatchSize \longleftarrow 0$;
    parallel for $metadata \in arrayMetadata$ do
        $texture \longleftarrow listTextures[metadata.meshID]$;
        // Managing texture states to load texture mipmaps (Section 3.5).
        if $texture.state \neq$ Final then
            $mipLevel \longleftarrow$ -1;
            if $texture.state =$ Initial then
                // Read image metadata (e.g.  image extent and miplevels).
                LoadImageMetaData($texture$);
                // Set the initial mipmap level to load.  We use the 4x4 mipmap.
                $mipLevel \longleftarrow texture.mipLevelMax$;
                $texture.state$ = Active;
            else if $texture.state =$ Active then
                if $texture.mipLevelSampled + b < texture.mipLevel$ then
                    // Specify the next higher resolution mipmap for streaming.
                    $mipLevel \longleftarrow texture.mipLevel - 1$;
                    // Set to Final state as the highest resolution mipmap will be streamed.
                    if $mipLevel = texture.mipLevelMin$ then
                        $texture.state$ = Final;
                    end
                end
            end
            if $mipLevel \neq$ -1 then
                // Stop streaming textures when the adaptive load size is reached.
                if $loadedBatchSize > s$ then
                    break;
                end
                // Load the mipmap image data.
                LoadImageData($texture$, $mipLevel$);
                $loadedBatchSize \longleftarrow loadedBatchSize + texture.dataSize$
                // Cache the loaded mipmap in the sparse image (Section 3.6).
                CopyToSparseImage($texture$, max($texture.width$, $texture.height$));
            end
        end
    end
return
```

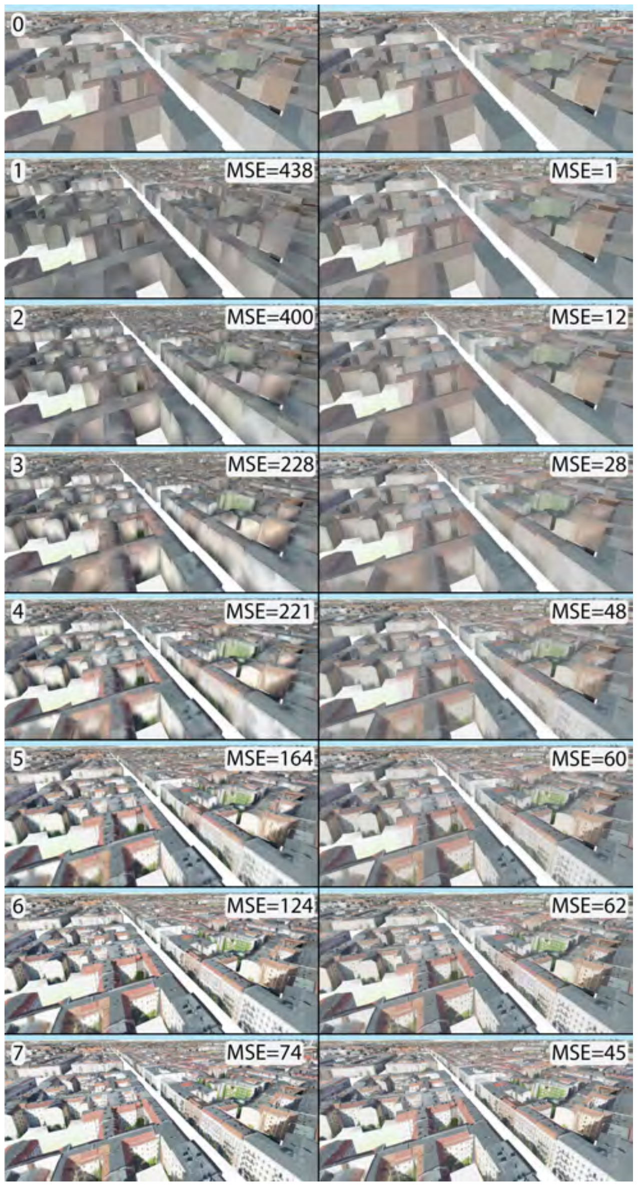

Figure 7: Frame captures of the Berlin (atlas) scene comparing direct mipmap sampling and our color blending method. *Left column*: Sampling the sparse image directly. *Right column*: Blending sampled colors in the sparse image with original mesh colors (shown in frame 0) stored in the City Colormap. Frames 1 to 7 indicate progressive sampling from the base sparse image mipmap level. The mean squared error (MSE) between the current and the previous frame is indicated.

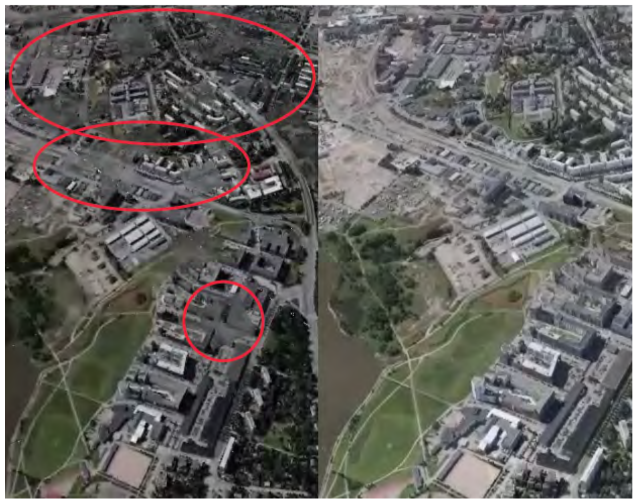

Figure 8: Comparison of rendering quality during texture streaming between Unity (*left*) and our method (*right*). In Unity, texture pop-ins are visible as low resolution mipmaps are immediately refined into high resolutions ones. In our method, we refine mipmaps at all levels using our mipmap blending method and adaptive algorithms, resulting in less texture pop-ins and more uniform texture resolutions.