High Performance Texture Streaming and Rendering of Large Textured 3D Cities

Alex Zhang
alex.zhang@fraunhofer.sg
Fraunhofer Singapore

Chen Kan
chen.kan@fraunhofer.sg
Fraunhofer Singapore

Henry Johan
henryjohan@ntu.edu.sg
Nanyang Technological University
Fraunhofer IDM@NTU

Marius Erdt
marius.erdt@fraunhofer.sg
Nanyang Technological University
Fraunhofer Singapore

Figure 1: A bird's eye rendering of the Berlin city dataset (total 5.7M textures) in 5.7ms (4K resolution) with 13.7K texture mipmaps currently streaming at 56.6MB/s. *Top inset:* Gray levels indicate the loaded texture mipmap levels, with lighter and darker levels denoting higher and lower resolution mipmaps. *Bottom inset:* Mesh textures of mipmap level 4 being streamed and cached in the sparse partially-resident image.

ABSTRACT

We introduce a novel, high performing, bandwidth-aware texture streaming system for progressive texturing of buildings in large 3D cities, with optional texture pre-processing. We seek to maintain high and consistent texture streaming performance across different city datasets, and to address the high memory binding latency in hardware virtual textures. We adopt the sparse partially-resident image to cache mesh textures at runtime and propose to allocate memory persistently, based on mesh visibility weightings and estimated GPU bandwidth. We also retain high quality rendering by minimizing texture pop-ins when transitioning between texture mipmaps. We evaluate our texture streaming system on large city datasets, including a tile-based dataset with 56K large atlases and a dataset containing 5.7M individual textures. Results indicate fast and robust streaming and rendering performance with minimal pop-in artifacts suitable for real-time rendering of large 3D cities.

1 INTRODUCTION

Although commercial desktop systems can render large 3D cities in real-time, they have limitations when rendering millions of textures hundreds of gigabytes in size, due to slow data transfers and a restrictive GPU memory capacity. Current systems can automatically page data in and out of system memory (RAM) when GPU memory is oversubscribed but rendering performance will be heavily impacted. A simple method to improve performance is to globally reduce texture resolutions, with textures permanently residing on GPU memory. Another more intelligent method is to adapt GPU-resident texture resolutions to a level-of-detail metric. This method is facilitated by a texture streaming system, where selected texture mipmaps are cached in a virtual texture for rendering. However, such texture streaming systems have difficulties maintaining a consistent streaming performance and a high quality rendering with minimal texturing artifacts.

In this paper, we present a texture streaming system that is fast, consistent, and provides high quality rendering for 3D cities with large quantities of textures. We adopt hardware virtual texturing to cache mesh textures in runtime using a fast atomic counter scheme, and focus on amortizing its high memory binding latency. It is done by adapting texture streaming rate to the estimated GPU load and the visual perceptibility of meshes. Our contributions are:

- A texture streaming system for rendering 3D cities with over five million textures, with or without the need for preliminary processing to generate texture atlases. We propose a single GPU-pass, multithreaded system for texture streaming, and a texture

caching scheme for hardware virtual texturing with persistent memory allocation.
- A GPU bandwidth estimation function derived from previous virtual texture memory binding performance. We also propose functions (robust across datasets) that limit texture streaming rate and texture resolutions using the GPU bandwidth estimation to ensure a stutter free rendering.
- A color blending method to minimize texture pop-ins when texture mipmaps change. We maintain original, per-mesh colors in a City Colormap, which is used to restore correct mesh colors when sampling from low resolution mipmaps.
- An optional mesh discretization scheme to generate texture atlases by grouping nearby and similarly oriented meshes together. These texture atlases improve texture streaming performance and reduce texture caching memory.

We evaluate our texture streaming system performance and rendering quality using large city datasets, including the Berlin dataset [1] with 40GB of compressed texture data and the Helsinki dataset [15] with 56K large texture atlases. Our evaluation reveals a significant improvement in rendering performance and image quality compared to a baseline without adaptive texture streaming and mipmap color blending. We also compare the results of our method to the Unity game engine's using the Helsinki city dataset.

2 RELATED WORK

2.1 City rendering in GIS

Virtual globe visualization systems are capable of rendering large city models and terrain on the web for Geographic Information System (GIS) purposes. These systems stream large amounts of geometry and texture contents to desktop and mobile clients [4, 8], adapting a set of textured geometry tiles to a level-of-detail (LOD) suitable for real-time rendering, based on camera views [7] and rendering time measurements [6]. These methods do not implement virtual texturing using per fragment texture address translation, but rather use a set of pre-generated texture coordinates per surface, resulting in high memory requirements and numerous texture state changes that impact rendering performance.

2.2 Virtual texturing systems

Virtual texturing systems are designed to cache a small subset of the scene textures in fast memory for rendering. They provide a system to move textures in and out of graphics memory depending on LOD factors and resource availability. Tanner et al. [17] proposed the clipmap to keep clipped portions of the terrain mipmap in memory for rendering, while Cline et al. [3] proposed a texture tile caching system for streaming textures in a low bandwidth system. Lefebvre et al. [9] introduced a texture caching system for arbitrary meshes with partial GPU support for visibility detection. Taibo et al. [16] proposed dynamic texturing of terrains using a clipmapping procedure driven by programmable GPU.

Recent virtual texturing methods emulate virtualization of texture memory, by sparsely allocating physical memory for virtual texture regions (called pages) required for rendering. They use an indirection texture [10] as a page address translation table that maps a page in virtual address to a region in the physical texture cache. Van Waveren proposed software virtual texturing [18] with solutions for texture filtering and feedback rendering (i.e. deciding which pages are made resident in the physical cache). Games engines have also adopted virtual texturing for terrain rendering. Widmark [19] proposed procedural virtual texturing to mitigate texture blending (splatting) costs, by caching splatter results in the virtual texture. Chen [2] improved on the prior with adaptive virtual textures to support a large game world terrain. In the virtual reality field, Mueller et al. [11] designed a parallel framework to compress and stream object-space shading data of game-like scenes 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 trilinear or anisotropic filtering will be difficult [13]. Additionally, page translations invoke multiple in-shader memory accesses that adversely affect performance.

2.3 Hardware virtual texturing

In hardware virtual texturing [13], page address translation and texture filtering are directly supported in hardware. It reduces the page translation overheads and lowers the page table memory footprint, while simplifying implementation without the need to update page mappings in the application end. Schmitz et al. [14] apply hardware virtual texturing to cache textures for point cloud rendering with a predictive page management based on camera movements.

The sparse (partially-resident) image is an interface for hardware virtual texturing on the Vulkan graphics API to cache texture mipmaps for rendering. Like an image object, it has several layers with a full mipmap chain per layer. It is logically divided into blocks where each can be sparsely allocated with physical memory on demand. The sparse image is designed for mega-texturing [18], where the scene's entire texture space is mapped in its virtual image space.

However, using hardware virtual texturing (including the sparse image) is not without drawbacks. On current systems, progressively binding sparse image memory blocks has a high GPU latency cost that causes severe frame rate stuttering. Under testing, we discover that the time taken to bind a memory block increases quadratically with every memory block bound, especially when memory blocks are sparsely bound across more than thirty sparse image layers. This binding cost is limited by the operating system and driver implementations [13], which we seek to amortize using our adaptive streaming method.

The sparse image is also inflexible to program outside of its intended purpose (i.e. mega-texturing) and provides only one set of texture coordinates to address all texture mipmaps. To address the requirement of not having to pre-process city textures and to reduce memory binding latency, we propose a runtime texture caching scheme that semi-compactly packs textures in the sparse image cache.

3 OUR TEXTURE STREAMING PIPELINE

Our texture streaming pipeline is composed of a texture streaming and a rendering path. The texture streaming path works in isolation from the rendering path, loading batches of textures that are pertinent to the current camera view and caching them in the sparse

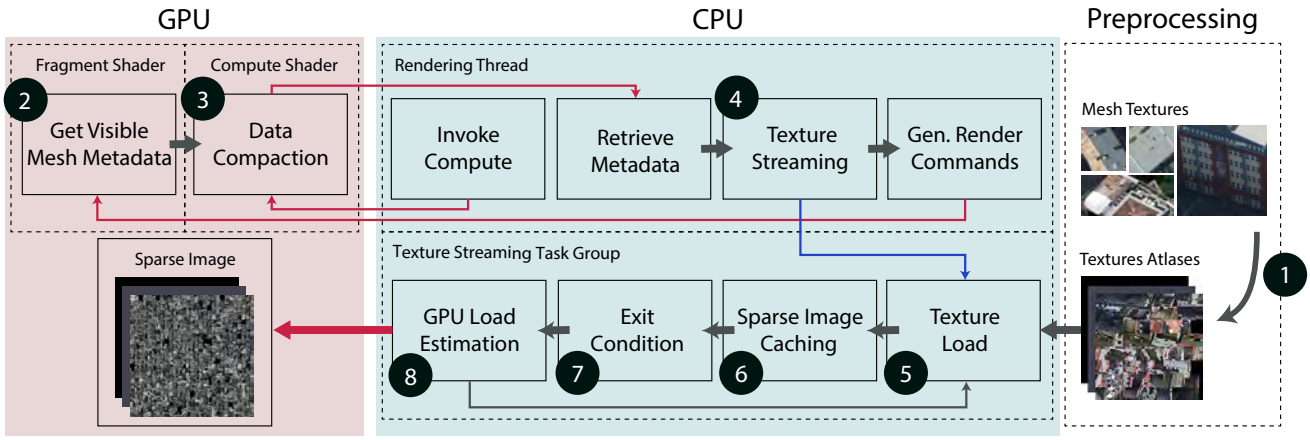

Figure 2: An overview of our texture streaming pipeline. CPU and GPU processes work together to load visible mesh textures from secondary storage and transfer it to the sparse image for rendering. Dotted lined boxes illustrate concurrency of a process or a group of processes. Red arrows denote time-expensive CPU-GPU inter-operations and gray arrows illustrate the flow of data between two processes. The blue arrow invokes the texture streaming task group.

image at the end of the pipeline. We exploit task-based parallel processing in most of the texture streaming subsystems to accelerate texture loading and sparse image caching. We summarize texture preparation, streaming and display as eight distinct subsystems (Figure 2).

3.1 Generating atlases and the City Colormap

For city datasets with many textures, we recommend packing textures into atlases for faster texture streaming performance. We propose a mesh discretization scheme to pack textures of meshes with similar orientations together in an atlas. Meshes are first discretized into uniform rectangular regions, and then further discretized into six subregions according to the closest axis-aligned normal. This procedure resembles backface culling in geometries, as textures of back facing meshes are not loaded if they are not invoked by the fragment shader. Meshes in each discretized subregion have textures of the same height packed together in an atlas. We pack mesh textures into an atlas in a left-to-right, top-to-bottom order. We also pack textures in the next discrete region into existing atlases with empty spaces to reduce the number of atlases generated. This has the advantage of caching more mesh textures at no additional streaming cost.

We introduce a City Colormap texture that stores a color per mesh in the city dataset. It acts as a color placeholder for mesh textures not yet cached in the sparse image. It is also used to mitigate texture pop-in artifacts when transitioning from a low to high resolution mipmap. This distinct color change is due to sampling errors from smaller mipmaps that are increasingly erroneous. We solve this problem by linear interpolation of the original mesh color in the City Colormap and the color sampled in the sparse image, based on the mipmap level:

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

where c_s is the sampled color, c_o is the original mesh color, M is the sampled 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.

3.2 Generating visible mesh metadata

This subsystem generates metadata that describes visible mesh textures for streaming. We take advantage of the GPU pipeline to perform backface, frustum and occlusion culling (early z-test) automatically before the fragment shader stage. In the fragment shader, we compute a **mipmap level** and a **weight value** (both integers) for all visible meshes and store them in a fixed size array buffer (called the Sparse MeshMetaData array) at the respective mesh index. Later, this array will be compacted and transferred to CPU local memory for use in texture streaming. We also render to an offscreen buffer at a larger camera field of view (FOV) than the viewport FOV to anticipate camera movements during texture streaming. In this way, there will be fewer texture pop-ins at the edges of the viewport during rapid camera movements.

The **mipmap level** metadata specifies the optimal mipmap level to load and stream a mesh texture. We compute the precise mipmap level M_s that mesh textures will be sampled at under a specific anisotropic filtering level F_s, using the implementation by Hollemeersch et al. [5]. Since a mesh spanning multiple fragments can produce different mipmap levels, we select the lowest mipmap level as the mipmap level metadata M, derived as $M = \lfloor M_s \rfloor$ if $\lfloor M_s \rfloor < M$, where the brackets $\lfloor\ \rfloor$ denote the round-down operation. We can control the resolution range of texture mipmaps for texture streaming by adjusting the anisotropic filtering parameter F_s. Note that during rendering, we sample mesh textures at an anisotropic filtering level F_r that is lower or equal to the anisotropic filtering level F_s used to determine M, in order to preload mipmaps and minimize

texture pop-ins. In our implementation, we define $F_r = \frac{F_s}{2}$ where $1 \leq F_r \leq F_s$.

The **weight value** metadata represents the perceptual importance of a mesh. It is used to determine mesh texture inclusion in the texture streaming process and its streaming priority order. In the fragment shader, each fragment invocation contributes some amount of weight, and these per-fragment weights are atomically summed to give the mesh weight value. By default, a +1 per-fragment weight contribution is added for each visible fragment. This contribution, by itself, gives the screen-space size (i.e. fragment count) of a mesh and thereby forms the basis of the streaming priority order. Under perspective projection, nearby meshes will have a larger screen-space size and will have textures loaded prior to faraway or partially occluded meshes. We propose the following weight contributions to meshes within the viewport:

- +0 to σ depending on fragment distance from the camera. This is derived from $\sigma \cdot \text{clamp}(\frac{1}{\max(c_z,\ 1)}, 0, 1)$, where $\sigma = 10$ is the maximum weight contribution, and c_z is the fragment's camera-space depth value, assuming c_z is generally between 0 and 100. The function increases weighting for small meshes near the camera that are quite perceptible despite having a low weight value from the screen-space size contribution. This weight is scene dependent as it is derived from the camera-space distance.
- +0 to ρ depending on fragment's loaded mipmap level. This is derived from $\rho \cdot \min(\frac{M}{\max(M_{max},\ 0.000001)}, 1)$, where $\rho = 8$ is the maximum weight contribution integer, M is the fragment's loaded mipmap level, and M_{max} is the fragment's largest mipmap level. The function increases weighting for meshes with lower resolution mipmaps, in order to maintain similar mipmap levels for nearby meshes, smoothing out resolution disparities during texture streaming.

3.3 Data compaction in the compute shader

The Sparse MeshMetaData array contains mesh descriptors for all meshes in the scene. It is a sparsely populated array since the fragment shader only writes mesh descriptors for visible meshes. This sparse array is entirely transferred from GPU to CPU memory every texture streaming cycle, which is slow as the array can be over five million long. Therefore, we propose to compact the sparse array on the GPU using a compute shader before transferring it to CPU memory. We use a simple atomic counter method [5], where each compute invocation, respective to an index in the Sparse MeshMetaData array, copies a mesh descriptor to the output MeshMetaData array at an atomically incrementing index.

To further reduce the size of the MeshMetaData array and consequently the number of textures to stream, we introduce a threshold value τ to cutoff meshes (that are faraway or span a few pixels) from the texture streaming process. We use a threshold value $\tau = 300$ which gives a good balance between the number of textures displayed and the texture streaming performance. This threshold value is proportional to the per-fragment weight contributions listed in the previous section.

3.4 Texture streaming strategy

Our texture streaming system is designed to maximize streaming and rendering performance and minimize runtime memory consumption. We propose to stream a regulated number of textures per batch to keep both the CPU and GPU reasonably busy. Streaming too small a batch inhibits texture transfer rate and streaming too large a batch increases texture load-to-display latency, meaning that newly loaded textures may not be relevant to the current camera view anymore. Another reason to regulate texture streaming is frame rate stuttering. Although the texture streaming tasks operate independently from the rendering path, issuing too many sparse image memory block binding commands will cause long internal GPU synchronizations that lead to stuttering frame rates. Therefore, we propose to stream textures in manageable batches controlled by the batch size exit conditions. These exit conditions estimate the amount of work the GPU can consume in the current cycle and if conditions are met, stops texture streaming for the current batch. GPU commands for this batch are submitted, to allocate and bind sparse image memory blocks and to transfer mesh textures to the sparse image. Modern graphics APIs expose transfer queues that take advantage of direct memory access (DMA) for asynchronous data transfers without further CPU intervention.

3.5 Loading texture images

The first subsystem in the texture streaming task group is texture loading, which deals with loading a batch of visible texture mipmaps into system memory (RAM) for transfer to the sparse image (VRAM). We propose to load textures in parallel. Mipmaps are loaded in order of descending mesh weights, incrementally, one mipmap level per batch processing cycle, beginning from the initial mipmap level to the base mipmap level. We load mipmaps into system memory at the exact memory offset in the image file to support exponentially faster texture loading times for smaller mipmaps.

Three texture objects states are defined: **Initial**, **Mipmap**, and the **Final** state. All texture object state begins at the Initial state and progresses to the next state at the end of a batch processing cycle if conditions are met.

- **Initial**: The Initial state indicates that a texture will be loaded for the first time. A unique location on the sparse image is assigned to the texture including all its mipmaps. Then, the initial texture mipmap (i.e. the 4×4 mipmap level) is transferred to the sparse image with the underlying memory blocks allocated and bound. Finally, the mesh metadata used to identify and sample this texture is generated and stored in a shader storage buffer for fragment shader access.
- **Mipmap**: The Mipmap state indicates that the next higher resolution mipmap is to be transferred. Sparse image memory blocks are allocated and bound for the newly transferred mipmap and the mesh metadata buffer is updated.
- **Final**: This state indicates that the base mipmap is loaded or that the texture is invalidated in the sparse image cache.

3.6 Texture caching in the sparse image

We design an algorithm to cache textures in the sparse image for scenes with texture atlases of a uniform extent, as well as for scenes with textures of varying extents. It is a parallel algorithm that uses

atomic counters for fast caching performance, and it packs texture mipmaps semi-compactly to optimize caching space. The main idea of the algorithm is to pack textures with the same maximum extent in the same sparse image layer. Each sparse image layer defines a grid of non-overlapping uniform square regions where each region stores only one texture. A sparse image layer can only store textures with the same maximum extent value $v = \max(width, height)$. For example, a layer designated with $v = 32$ can only store textures with extent combinations of 32x32, 32x16, 32x8 and 32x4. We require all scene textures to be in powers of two to limit the number of possible combinations. Although packing rectangular textures in square regions wastes caching space, it simplifies implementation and ensures that mipmaps are all aligned to the same texture coordinate. One may designate more sparse image layers to cache only rectangular textures, but this will inflate the pool of layers resulting in overall caching performance loss.

We use a parallel unordered map to keep track of available sparse image regions for caching new textures. This map defines a key-value pair $\{v, (layer, index)\}$ for every unique v value. The mapped value $(layer, index)$ contains atomic counters describing a sparse image $layer$ and a square region $index$ to cache a new texture with the associated v value key. After caching the new texture, the $index$ counter is incremented so that it points to a new region in a left to right, top to bottom order. A third atomic counter, $nextLayer$, tracks the next available sparse image layer to store textures. When a sparse image layer is filled up, the $layer$ will point to a new sparse image layer fetched from the $nextLayer$ counter, and then the $nextLayer$ counter is incremented.

Since the number of sparse image layers for caching textures is limited, we reuse the 0^{th} index layer when the last layer is filled up, just like a ring buffer. Textures in an old index layer are invalidated and no longer displayed and will need to be reloaded in the future when required. This ring buffer design guarantees a bounded GPU memory limit for caching textures which also improves texture streaming performance.

3.7 Texture streaming exit conditions

The last subsystem in the texture streaming task group controls the early termination of the streaming process.

- Batch time (fixed) – The task group is terminated when a fixed amount of batch processing time has elapsed. It defines the interval at which texture streaming commands are generated and consumed by the GPU. A short time interval gives perceivable responsiveness of the texture streaming process, ensuring a consistent display of camera relevant textures. A longer time interval reduces texture streaming overheads, allowing a larger batch of textures to be streamed, but this increases sparse image memory binding operations leading to stuttering frame rates. In our implementation, we use a batch processing time of 0.016s.
- Batch size (adaptive) – The task group is terminated when the size of loaded texture mipmaps exceeds a batch size threshold. It effectively limits the number of sparse image memory binding operations for a batch of work. A small batch size guarantees low latency and stutter-free rendering but restricts the rate of texture streaming, while a large batch size increases the rate of texture streaming at the expense of stuttering frame rates. In our implementation, we adaptively adjust the batch size based on estimated GPU load.

To further reduce frame rate stuttering, we introduce an adaptive mipmap bias parameter to regulate the number of large texture mipmaps requested for loading. It is designed to maintain visual coherency of neighboring mesh textures under high GPU load by giving smaller mipmap texture loading priority. This is managed in the texture loading subsystem.

3.8 Adaptive batch size and mipmap bias

Our proposed adaptive batch size and adaptive mipmap bias components are essential to minimize frame rate stuttering during texture streaming. The adaptive batch size S is derived:

$$S = \text{clamp}(s \cdot f, s_{min}, s_{max}) \tag{2}$$

where $s_{min} \leq S \leq s_{max}$, s is the batch size scale in MB that will be modulated by the exponential batch size factor f, and s_{min} and s_{max} is the minimum and maximum batch size in MB respectively. Since we derive GPU load measurements from texture streaming operations, we have to define a minimum boundary $s_{min} = 0.01$ to ensure constant texture streaming even under high GPU load. The maximum boundary $s_{max} = 64$ is the maximum texture streaming performance that we have defined for our system. The batch size modulation factor f is derived:

$$f = \frac{1}{\max((1 + L_f - t_b)^{\alpha}, 1)} \tag{3}$$

where $0 \leq f \leq 1$, L_f is the GPU load measurement derived from the sparse image binding time t, t_b is the ideal sparse image binding time, and α is the exponent. The batch size factor ensures that texture streaming rate is highest when $L_f \leq t_b$, and exponentially reduces when $L_f > t_b$. In our implementation, we use a large batch size scale $s = 5000$ and a maximum batch size $s_{max} = 64$MB for the highest possible texture streaming rate under low GPU load, and set $\alpha = 5$ to exponentially reduce the texture streaming rate under heavy GPU load. We set $t_b = 5$ms to begin reducing texture streaming rate before stuttering is noticeable.

The optimal mipmap level is modified by a mipmap bias:

$$b = smoothstep(t_b, t_{max}, L_b) \cdot b_{max} \tag{4}$$

where $0 \leq b \leq b_{max}$, t_b is the ideal sparse image binding time, t_{max} is the maximum sparse image binding time, L_b is the GPU load measurement, and b_{max} is the maximum mipmap bias. This function introduces no bias when $L_b \leq t_b$, and introduces an increasing amount of bias (along a Hermite curve) as L_b approaches t_{max}. We apply the mipmap bias to only allow the next lower mipmap level to load if $round(m_o + b) < m$, where m_o is the optimal mipmap level (from the MeshMetaData array), and m is the currently displayed mipmap level. In our implementation, we set $t_{max} = 16$ms and $b_{max} = 10$ to load mostly smaller mipmaps under high GPU load.

Both f and b have a GPU load estimation measurement, L_f and L_b respectively, derived from the sparse image binding time sampling function:

$$L = \max\left(\frac{\sum_{i=1}^{n} L_i \cdot r}{n}, t\right) \tag{5}$$

The function samples n previous sparse image binding time, $L_i \cdot r$, to estimate the GPU load cost L when executing future texture streaming operations. A high L value indicates high GPU load, and a low L value indicates low GPU load. When the sparse image binding time t is high, L will immediately assume the high t value through the max function, and when t is low, L will steadily falloff towards the low t value based on the falloff rate r. This design minimizes frame rate stuttering by immediately reducing texture streaming rate when GPU load is high, and steadily increasing texture streaming rate when GPU load is consistently low. For L_f, we use $n = 10$ and $r = 0.87$. For L_b, we use $n = 30$ and $r = 0.99$. Note that for L_f, we only integrate those binding time L_i when texture streaming has been performed, to reflect a more accurate GPU load measurement for future streaming operations. For L_b, we have to constantly refine mipmaps by integrating all binding time L_i including times when texture streaming is idle.

4 EVALUATION

We evaluate our texture streaming system in terms of streaming and rendering performances using three large city datasets (Table 1). We assess the performance of our system and show how the various system parameters affect streaming and rendering performance. We also review the image quality improvements when sampling small mipmaps using our color blending method and compare our texture streaming system to the Unity game engine using the Helsinki city dataset. Our test system is an Intel i7-8700K CPU with a Nvidia RTX2080 Ti GPU and a Samsung 970 EVO 1TB SSD, on Windows 10, rendering into a 3840×2160 resolution framebuffer.

4.1 Streaming and rendering performance

We evaluate our texture streaming performance with a rapid fly-through of the Berlin city scene with texture atlases (Figure 1), using the system parameters shown in Table 2. We record an average rendering latency of 5.66ms and a peak latency of 15.37ms, with 13708 mipmap transfers per second (56.58MB/s) on average, and 28896 mipmap transfers per second (166.19MB/s) at peak (Figure 3). We notice that rendering latency tends to spike sharply during texture streaming, even when we are streaming a small number of textures (0.05MB) per batch. Therefore, we conclude that sparse image memory block binding has a system level performance cost.

Our adaptive texture streaming functions are effective in reducing frame rate stuttering while maintaining a constant average rendering latency. In the series of benchmarks (Table 3), we analyze the reductions of frame rate stuttering by adjusting the α exponent value of the adaptive batch size function (Equation 3). The amount of frame rate stuttering is measured as the maximum of the first-order differences of rendering latency. We notice the decrease in average mipmap bias when α increases, since GPU load is reduced overall. This may cause slightly uneven mipmap resolutions in neighboring meshes which can be remedied by increasing the average mipmap bias to 1.5. The α exponent parameter is system performance dependent and we recommend setting it to a lower value to suppress most frame rate stuttering.

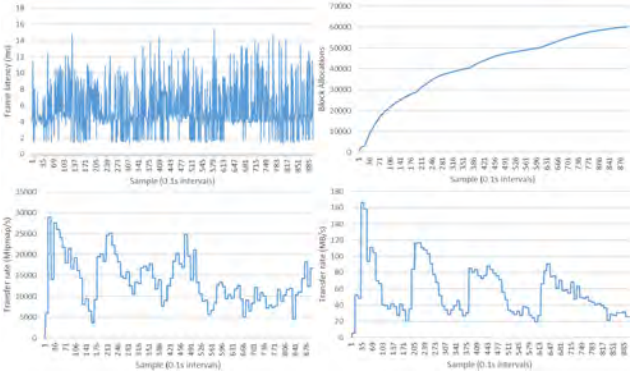

Figure 3: Texture streaming benchmark for the Berlin city with texture atlases. Left to right, top to bottom: Rendering latency in milliseconds, number of allocated sparse image memory blocks, mipmaps transferred per second, size of mipmaps transferred per second. We observe fluctuating rendering latency when binding sparse image memory, and fluctuating texture streaming rates when viewing areas of different building densities. Generally, peak texture transfer rate decreases as more sparse image memory blocks are allocated.

4.2 Rendering quality

Our texture streaming system is designed to maintain the visual coherency of mesh textures during their transition from low to high resolutions. We use the adaptive mipmap bias parameter to regulate mipmap streaming and maintain evenly distributed texture resolutions in the current view, at the cost of slightly delaying the streaming of higher resolution mipmaps. In a rendering view, our method results in a mipmap resolution spread of only 128 extents, while the native method results in a mipmap resolution spread of 64 to 512 extents. Without our adaptive mipmap bias feature, adjacent meshes will have visually incongruent mipmap levels during texture streaming.

From the rendering captures (Figure 4, right column), we observe a significant change of color intensity between frame 0 and 1 when switching from the original mesh colors to the highest mipmap level of the texture atlases. Using our color blending method, we managed to retain low frequency, original mesh colors (provided by our City Colormap) while gradually introducing high frequency details from the higher resolution mipmaps. It reduces the mean squared color differences (MSE) between frames, measuring at a maximum MSE of 26 using our method and a maximum MSE of 124 without, hence, significantly reducing texture pop-in artifacts.

4.3 Robustness: evaluating other datasets

To determine the robustness of our texture streaming system, we tested two structurally different datasets: the Berlin dataset with fragmented textures and the Helsinki dataset. We apply the system parameters from the Berlin (atlas) benchmark to investigate the sensitivity of our adaptive streaming methods. Texture streaming appears sluggish as it takes about two seconds to display mipmaps

Table 1: Attributes of the three city datasets tested in our system.

	Triangles	Geometry size (MB)	Textures	Format	Texture size (GB)	Texture extent per-axis
Berlin	15,363,486	935	5,712,309	BC1	41.6	4, 8, 16, 32, 64, 128, 256, 512
Berlin (atlases)	15,363,486	1092	233,110	BC1	40.8	512
Helsinki	115,112,748	6096	25,354	BC1	3.5	256, 512

Table 2: Texture streaming system parameters used in all our performance benchmarks unless stated so.

Parameters	Value	Description
F_s	2	Anisotropic filtering level
τ	300	Mesh weight cutoff threshold
Batch time	0.016s	Max processing time per batch
Adaptive batch size (Equation 2)		
s	5000MB	Batch size scale
s_{min}	0.01MB	Minimum batch size
s_{max}	64MB	Maximum batch size
Batch size modulation factor (Equation 3)		
t_b	5ms	Ideal binding time
α	5	Exponent
n	30	Previous binding time samples
r	0.99	Falloff rate
Mipmap bias (Equation 4)		
t_b	5ms	Ideal binding time
t_{max}	16ms	Maximum binding time
b_{max}	10	Maximum mipmap bias
n	10	Previous binding time samples
r	0.87	Falloff rate

Table 3: Performance benchmark of the Berlin scene (atlases) under various adaptive batch size exponent value α.

	Median	Average	Max
$\alpha = 1$			
Latency (ms)	4.69	5.94	67.06
Transfer rate (Mip/s)	18065	19743	35598
Transfer rate (MB/s)	54.49	55.08	163.89
Batch size (MB)	64	64	64
Mipmap bias	1.42	2.19	10
$\alpha = 5$			
Latency (ms)	4.88	5.8	21.02
Transfer rate (Mip/s)	18587	19609	35411
Transfer rate (MB/s)	59.27	58.72	163.79
Batch size (MB)	1.36	5.18	64
Mipmap bias	0.78	1.53	8.58
$\alpha = 9$			
Latency (ms)	4.82	5.19	11.88
Transfer rate (Mip/s)	12675	14190	35174
Transfer rate (MB/s)	38.41	45.99	196.14
Batch size (MB)	0.44	3.99	64
Mipmap bias	0.07	0.46	6.9

until the optimal level. Nonetheless, our adaptive streaming methods under non-optimal parameters manage to suppress frame rate stuttering and maintain an even distribution of mipmap levels, while performing close to the maximum texture streaming rate.

4.4 Limitation

One limitation of our texture streaming method is texture trashing, where currently visible textures cached in the sparse image are constantly replaced by newer textures streaming in. This occurs when there are too few sparse image layers available to cache all visible textures in the current view, causing the caching algorithm to invalidate all textures in the previous sparse image layer which are still visible in the current view. We can resolve this issue by having enough sparse image layers at the expense of impacted sparse image memory binding performance and increased memory pool.

4.5 Comparison with the Unity game engine

Professional GIS software tools are increasingly adopting gaming technologies to enhance user experiences and workflows [12]. They leverage game engine's (e.g. Unity and Unreal) high performance and high quality real-time rendering capabilities for analysis and presentation of geospatial data, especially in virtual reality. For these reasons, we choose to compare our system's texture streaming performance and rendering quality to those of the Unity (2020) game engine.

Rendering results of the Helsinki scene from a bird's eye view to a closeup view show an average rendering latency of 28ms using the Unity engine compared to our system at 19ms average rendering latency. In Unity, mesh textures are initially presented at low mipmap resolutions with the presence of erroneous mipmap colors. During texture streaming, these mipmaps are refined directly to the optimal mipmap level for the current camera view which produces distinct texture pop-ins. Nonetheless, we did not notice any further texture pop-ins when zooming in to a closeup camera view unless when moving at very high speeds. In our system, mesh textures are initially presented as single colors via the City Colormap, giving the scene a crude appearance. Mesh textures are subsequently refined with minimum texture pop-in effects due to our color blending method. When transitioning to the closeup view, some texture refinement is still observable especially when moving the camera at high speeds.

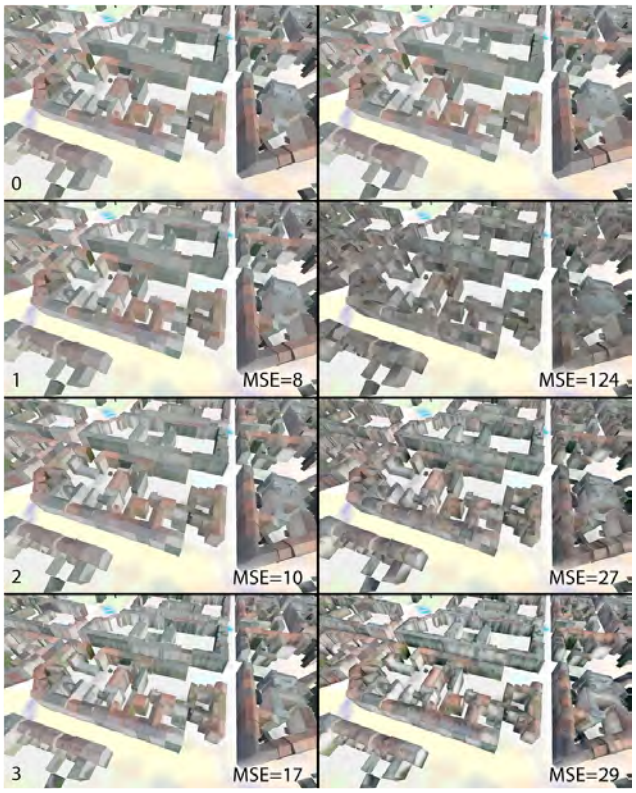

Figure 4: Frame captures of the Berlin atlases scene using our color bending method (left column) vs one without (right column). In the left column, we blend original mesh colors with the sampled texture mipmaps to minimize color errors and preserve visual continuity. In the right column, we directly sample from the texture mipmaps. Frame 0 shows the scene sampled only using the City Colormap. Frames 1, 2, 3 show progressively texturing using mipmap resolutions of 4, 8, 16 extents respectively. We show the mean squared error (MSE) between the current and the previous frame.

5 CONCLUSION AND FUTURE WORK

Our proposed texture streaming pipeline adopts a multithreaded, GPU-driven system for high performance texture streaming. It uses the fragment shader to determine mesh visibility and the optimal mipmap level for texture streaming. Textures are loaded and cached into the hardware supported sparse partially-resident image for sampling, using a batch processing strategy that operates in parallel with rendering. We have proposed two adaptive streaming methods: the adaptive batch size, and the adaptive mipmap bias function, to minimize frame rate stuttering and regulate mipmap display. Furthermore, we have presented a method to significantly reduce texture pop-in artifacts by blending in original mesh colors stored in the City Colormap. We also proposed to generate texture atlases based on similar orienting normals to vastly improve texture streaming performance. Our texture streaming system is fast and produces high quality rendering. It is also robust enough to support different city datasets without having to accurately tune system parameters.

As for future work, we like to apply our texture streaming method to remote rendering, where the server is tasked with texture selection, encoding, and transfer to the clients for fast decode and rendering. As compute support is limited in WebGL, we can explore shader based decoding and software virtual textures to cache mesh textures on client side.

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] Berlin Partner for Business and Technology Berlin Senate Department for Economics, Energy and Public Enterprises. [n.d.]. Berlin 3D. https://www.businesslocationcenter.de/en/economic-atlas/download-portal/. Accessed: 2019-10-01.
[2] K Chen. 2015. Adaptive Virtual Texture Rendering in Far Cry 4. In *Game Developers Conference*. 869.
[3] David Cline and Parris K Egbert. 1998. *Interactive display of very large textures*. IEEE.
[4] Patrick Cozzi and Kevin Ring. 2011. *3D engine design for virtual globes*. Crc Press.
[5] Charles Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle. 2010. Accelerating virtual texturing using cuda. *GPU Pro: advanced rendering techniques* 1 (2010), 623–641.
[6] Wumeng Huang and Jing Chen. 2018. A virtual globe-based time-critical adaptive visualization method for 3D city models. *International journal of digital earth* 11, 9 (2018), 939–955.
[7] Seokchan Kang and Jiyeong Lee. 2017. Developing a tile-based rendering method to improve rendering speed of 3d geospatial data with html5 and webgl. *Journal of Sensors* 2017 (2017).
[8] Michel Krämer and Ralf Gutbell. 2015. 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*. ACM, 189–197.
[9] Sylvain Lefebvre, Jérome Darbon, and Fabrice Neyret. 2004. Unified texture management for arbitrary meshes. (2004).
[10] Martin Mittring and Crytek. 2008. Advanced virtual texture topics. In *ACM SIGGRAPH 2008 Games*. ACM, 23–51.
[11] Joerg H Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading atlas streaming. In *SIGGRAPH Asia 2018 Technical Papers*. ACM, 199.
[12] Pascal Mueller. 2018. High-end 3D Visualization with CityEngine, Unity, and Unreal. *ESRI Developer Summit https://proceedings.esri.com/library/userconf/devsummit18/papers/dev-int-142.pdf (2018-03-07)* (2018).
[13] Juraj Obert, JMP van Waveren, and Graham Sellers. 2012. Virtual texturing in software and hardware. In *ACM SIGGRAPH 2012 Courses*. ACM, 5.
[14] Patric Schmitz, Timothy Blut, Christian Mattes, and Leif Kobbelt. 2020. High-Fidelity Point-Based Rendering of Large-Scale 3D Scan Datasets. *IEEE Computer Graphics and Applications* (2020).
[15] Helsinki Region Infoshare (HRI) service. [n.d.]. Reality mesh of entire Helsinki (2017). https://hri.fi/data/en_GB/dataset/helsingin-3d-kaupunkimalli. Accessed: 2019-10-01.
[16] Javier Taibo, Antonio Seoane, and Luis Hernández. 2009. Dynamic virtual textures. (2009).
[17] Christopher C Tanner, Christopher J Migdal, and Michael T Jones. 1998. The clipmap: a virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM, 151–158.
[18] JMP van Waveren. 2012. Software Virtual Textures. (2012).
[19] Mattias Widmark. 2012. 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).