计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
2025,34(5):52−63 [doi: 10.15888/j.cnki.csa.009890] [CSTR: 32024.14.csa.009890]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

面向移动 VR 头显的超高分辨率国画纹理显示方法①

于丰源, 姜忠鼎

(复旦大学 计算机科学技术学院, 上海 200438)
通信作者: 姜忠鼎, E-mail: zdjiang@fudan.edu.cn

摘　要: 高清晰、低延时显示中国画纹理是中国画 VR 展示应用的重要需求, 移动 VR 头显有限的运行时内存、显存难以实现大量高分辨率中国画纹理同时加载及实时显示. 此外, 受限于移动 VR 设备的低显示分辨率和纹理 mimap 管理机制, 用户通过头显直接观察到最清晰纹理细节比较困难. 本文给出一种改进的虚拟纹理方法, 主要对已有虚拟纹理方法的分块请求计算和分块加载两个阶段进行优化. 在分块请求计算阶段, 加入放大辅助视角的分块请求计算, 利用 Compute Shader 并行加速处理分块请求参数纹理, 通过哈希减少 Compute Shader 构建结果缓存的计算量. 在分块加载阶段, 采用无锁队列异步加载提高纹理分块加载效率, 使用数量阈值限定的请求分块直接加载策略减少高清晰纹理分块显示延迟. 本文构建包含单幅和多幅中国画的虚拟观赏场景, 通过模拟用户观赏行为来测试本文方法的运行性能及显示效果. 实验结果表明, 本文方法结合放大辅助视角在移动 VR 设备上可高清晰、低延时呈现多幅超高分辨率中国画纹理. 与 Unreal SVT 等已有虚拟纹理方法相比, 本文方法可在大量纹理分块条件下保持高帧率运行, 实现更低的高清晰纹理分块显示延迟.

关键词: 虚拟现实; 超高分辨率纹理; 虚拟纹理; 中国画展示; 显示优化

引用格式: 于丰源,姜忠鼎.面向移动 VR 头显的超高分辨率国画纹理显示方法.计算机系统应用,2025,34(5):52–63. http://www.c-s-a.org.cn/1003-3254/9890.html

Method for Displaying Ultra-high Resolution Textures of Chinese Paintings on Mobile VR Headsets

YU Feng-Yuan, JIANG Zhong-Ding

(School of Computer Science, Fudan University, Shanghai 200438, China)

Abstract: High-definition, low-latency display of Chinese paintings is essential for Chinese painting VR exhibition applications. Due to the limited memory and GPU resources on mobile VR headsets, it is challenging to display a large number of high-resolution Chinese painting textures simultaneously. Moreover, direct viewing of fine details is hindered by mipmap management and the low resolution of mobile VR devices. This study proposes an improved virtual texture method, optimizing both tile request calculation and tile loading stages based on existing virtual texture methods. In the tile request calculation phase, the method incorporates tile request computations for magnified perspectives. Compute Shader is utilized to parallelize the processing of tile request parameters, and hashing is applied to minimize overhead when constructing result caches. In the tile loading phase, lock-free queues are implemented to enhance loading efficiency. A direct loading strategy for request tiles, constrained by a quantity threshold, reduces display latency. The performance and texture display effects are evaluated in scenarios with single or multiple Chinese paintings, simulating user behavior. Results show that the proposed method supports high-definition, low-latency display of high-resolution

① 基金项目: 教育部第二批“新工科”研究与实践项目 (E-XTYR20200621)
收稿时间: 2024-09-24; 修改时间: 2024-11-07, 2025-01-15; 采用时间: 2025-01-24; csa 在线出版时间: 2025-04-01
CNKI 网络首发时间: 2025-04-02

Chinese painting textures on mobile VR devices. Magnification-assisted perspectives allow for clear viewing of the finest texture details. Compared to existing virtual texture methods, such as Unreal SVT, the proposed method achieves higher frame rates and reduces display latency for high-resolution texture tiles of multiple Chinese paintings.

Key words: virtual reality (VR); ultra-high resolution texture; virtual texture; Chinese painting exhibition; display optimization

具有更成熟显示与交互技术的新一代移动 VR 头显正快速向普通消费者普及, 研究中国画在移动 VR 头显上的呈现方法对向大众宣传我国优秀历史文化具有重要意义[1,2]. 古代画作数字版本借助大幅面扫描仪扫描生成[3].《千里江山图》等大幅面画卷的数字扫描位图通常具有百兆级像素数量超高分辨率, 移动 VR 头显有限的运行时内存、显存无法满足大量高分辨率中国画纹理同时加载并显示的需求.

由 Tanner 等人[4]提出的 Clipmap 是最早解决大量纹理显示问题的方法之一, 其通过仅缓存基于相机距离被裁剪的 mipmap 序列减少存储压力. 虚拟纹理[5]是目前游戏行业或地理信息系统中解决广阔地形纹理加载与显示问题的常用方法[6–8]. 地形显示对纹理细节呈现没有严格要求, 通常以降低纹理画面质量和增加高清晰纹理分块显示延迟为代价提高纹理分块加载与显示性能[9], 而中国画纹理展示需要让用户能够高清晰、低延时观赏到笔触等画卷细节. 此外, 受限于移动 VR 设备低显示分辨率, 在头显中直接观察到最清晰的纹理细节是困难的, 比如在 Quest Pro 头显中观察尺寸为 12 m×0.5 m、分辨率约为 150k×6k 的画卷纹理时, 观察相机需贴近画卷约 7.2 cm 才能观察到 mipmap 层级为 0 的最清晰画卷内容.

针对上述问题, 本文结合用户观赏中国画行为特点和移动 VR 设备自身计算能力, 给出一种改进的虚拟纹理方法, 主要对已有虚拟纹理方法的分块请求计算阶段和分块加载阶段进行优化. 本文主要贡献如下.

(1) 在分块请求计算阶段加入放大辅助视角的分块请求计算, 支持放大辅助观察工具, 并且采用 GPU Driven 的方式加快分块请求获取与处理速度, 利用哈希方式降低 Compute Shader 构建结果缓存计算量.

(2) 在分块加载阶段借助无锁队列和数量阈值限定的未命中分块直接加载策略提高纹理分块加载效率, 实现更低的高清晰纹理分块显示延迟.

(3) 设计模拟真实中国画观赏场景及用户观赏行为的单幅图和多幅图实验场景, 测试方法运行性能及显示效果. 结果表明, 本文方法可在移动 VR 平台上结合放大辅助视角高清晰、低延时、高帧率地呈现大量、超高分辨率中国画纹理. 与 Unreal SVT 等已有方法相比有更低的高清晰纹理分块显示延迟.

1 相关工作

1.1 基于软件的虚拟纹理方法

Mittring 等人[5]对早期虚拟纹理技术进行了讨论. Barrett 等人[10]介绍了基本虚拟纹理方法. Hollemeersch 等人[11]提出利用 GPU 并行计算加速虚拟纹理执行性能的优化方法. van Waveren[9]和 Chajdas 等人[12]对基于软件的虚拟纹理方法的良好实践进行总结. Cheng[13]提出基于相机到地形表面距离来自适应调整页表纹理到物理纹理映射关系的 AVT (adaptive virtual textures). Revanth 等人[14]给出利用分布式渲染系统提高虚拟纹理执行性能的方法. Magro 等人[15]结合虚拟纹理技术给出一种可部署在云上的基于分布式渲染管线的高质量纹理显示方法. 主流游戏引擎 Unity 和 Unreal 近年来也给出各自的虚拟纹理方案: Unity SVT (streaming virtual texture)[16]和 Unreal SVT [17], Unity SVT 不支持移动平台.

上述方法主要适用于 PC 等计算性能较好的设备, 对于移动 VR 头显等低计算性能设备, Lukas[18]给出早期移动设备上基于 C/S 架构结合虚拟纹理技术渲染可交互地图的方法. Ma 等人[19]给出一种能在低性能设备上高效渲染有限尺寸纹理的虚拟纹理方法. 主流游戏引擎 Unreal 的 SVT[17]方案也支持移动端设备. Nourai 等人[20]和 Funt 等人[21]给出在云上结合虚拟纹理技术提高低性能 VR 设备纹理渲染能力的方法. 相较于 Ma 和 Unreal SVT 等仅利用设备自身算力的方法, 文献[20]和文献[21]的方法具有更高设备成本及系统复杂度.

1.2 基于硬件的虚拟纹理方法

相较于基于软件的虚拟纹理方法, 借助虚拟纹理硬件辅助图形接口可减少页表等模块带来的复杂性并提高虚拟纹理执行效率[22]. Schmitz 等人[23]采用基于硬

件的虚拟纹理渲染点云, Zhang 等人[24,25]提出一种基于硬件虚拟纹理技术的高性能超大规模 3D 城市纹理显示方法. Paar 等人[26]利用基于硬件的虚拟纹理渲染高精度隧道模型表面纹理. 然而, 基于硬件的虚拟纹理图形接口扩展在移动端设备上的普及度不高, 比如 Quest Pro、Pico4 Ultra、Quest3 等主流移动 VR 设备不支持 OpenGL ES 的 GL_EXT_sparse_texture 扩展和 Vulkan 的 Sparse Residency Image 特性. 本文方法采用基于软件的方式以适配主流移动 VR 设备.

1.3 纹理分块请求计算方法

虚拟纹理通常使用独立渲染通道逐片元计算分块请求参数, 请求参数存储在渲染结果纹理各像素中. 文献[10]给出请求参数结果纹理的基本处理方法: 同步等待 GPU 回读请求参数纹理至 CPU, 然后逐像素读取请求参数纹理构建不包含重复元素的分块请求集合. 分块请求集合与页表信息结合, 构建分块请求响应集合和分块加载指令集合. 同步等待 GPU 回读和逐像素读取请求参数的操作使得该方法执行效率较低.

文献[11]给出在 GPU 上并行分析请求参数结果纹理加速构建分块请求集合的方法, 但其为保证 GPU 核心计算同步性, 使用了一个分块总数量长度的标记数组, 当分块较多时, 遍历该标记数组构建请求集合的计算负担会增加. Unreal SVT[17]异步回读请求参数纹理至 CPU, 多线程异步分析请求参数纹理构建请求集合. 其利用多线程异步策略避免了逐像素遍历请求参数纹理的时间消耗, 但增加了 CPU 计算量及分块请求集合获取延迟. 文献[19]考虑到低性能设备上处理分块请求参数纹理效率较低, 改用在 CPU 上基于空间距离直接逐分块计算分块请求, 该方法在分块数量较少时能有效提高分块请求集合构建效率, 但随分块数量增多, 逐分块计算分块请求的计算量增大, 程序运行性能下降. 本文受到文献[11]启发, 利用 Compute Shader 优化纹理分块请求获取与处理过程, 并使用哈希方法解决使用标记数组构建结果缓存时, 随分块数量增多程序性能下降的问题.

2 方法概述

虚拟纹理技术将原纹理的各 mipmap 划分为大小相同的纹理分块, 使用类似虚拟内存的方式管理纹理分块. 虚拟内存技术的虚拟内存、物理内存、页表概念分别有虚拟纹理、物理纹理、页表与之对应.

虚拟纹理包含数据准备和运行时两个阶段, 数据准备阶段负责生成纹理分块打包文件和相关配置文件, 生成的文件存储在大容量存储设备上等待运行时被加载. 运行时包含分块请求计算、分块加载、物理纹理更新、页表纹理更新、纹理分块显示 5 个子阶段. 首先, 运行时借助独立渲染通道完成分块请求计算, 分块请求与页表信息结合发布分块加载指令, 用于调度分块加载, 新加载分块数据到达后执行页表和物理纹理更新, 最后根据页表将物体表面纹理坐标重映射至物理纹理空间, 利用重映射后的纹理坐标值采样物理纹理渲染物体表面.

本文结合中国画纹理展示特点和移动 VR 设备特性, 在文献[9]方法基础上, 主要对运行时的分块请求计算、分块加载两个子阶段进行优化, 本文改进的虚拟纹理方法如图 1 所示.

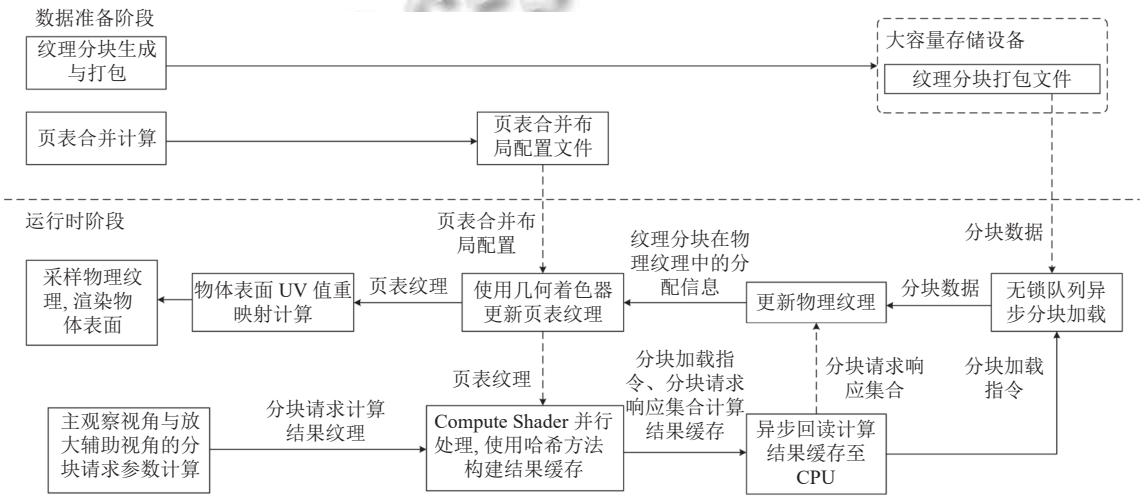

图 1 本文虚拟纹理方法

在分块请求计算子阶段, 传统虚拟纹理方法仅支持单一观察视角的分块请求计算[9], 本文使用额外渲染通道加入放大辅助视角的分块请求计算, 并利用 Compute Shader 加速处理分块请求参数纹理, 使用哈希映射构建计算结果集合, 避免文献[11]方法中标记数组带来的分块数量较多时程序运行性能下降的问题.

在分块加载子阶段, Unreal SVT[17]等方法采用基于锁机制的多线程异步加载方式加载分块, 本文采用无锁队列方式避免锁竞争在分块加载短时间作业任务中的性能损耗. 当分块请求未命中时, 传统逐 mipmap 层级分块加载策略优先加载包含目标分块所在区域仍未加载的 mipmap 层级最大的分块[9], 这会增加高清晰度纹理分块显示延迟. 考虑到中国画观赏场景中用户通常以连续低速运动方式观赏画卷, 帧间分块请求差异较少, 本文方法使用直接加载未命中请求分块策略以获得更低的高清晰分块显示延迟, 并通过限定单帧最大分块加载数量避免大量突发分块加载请求降低程序执行稳定性.

3 数据准备阶段

3.1 纹理分块生成与打包

中国画扫描纹理的分辨率通常非 2 的幂, 而虚拟纹理技术需要 2 的幂尺寸纹理才能构建 mipmap 并生成纹理分块. 本文方法通过空数据填充和物体表面纹理坐标缩放的方式避免画卷纹理向 2 的幂尺寸缩放带来的画面失真.

纹理分块存储有 RGB24、ASTC、DCT 等多种压缩或无压缩格式. 为满足中国画高保真显示需求, 纹理分块应采用 RGB24 等无压缩纹理格式存储. 为避免纹理分块边界的走样问题, 纹理分块需增加边界扩展像素[9]. 生成的纹理分块打包到一个或多个二进制纹理分块打包文件中, 等待运行时被动态加载.

3.2 页表合并

本文页表采用 mip-mapped 纹理形式, 相较于四叉树、哈希表等形式, mip-mapped 纹理形式页表可在着色器中并行访问, 有更高的访问效率[9]. 每张画卷若使用独自的页表纹理会增加页表纹理管理的复杂性, 且受图形 API 纹理采样器数量限制.

通过页表合并可解决上述问题. Unreal SVT[17]基于树结构在运行时对新增纹理动态分配页表空间进行页表合并. 考虑到国画展示场景一般不会有运行时新增画卷的情况, 本文使用 BRKGA (biased random-key genetic algorithm)[27]矩形装箱算法在数据准备阶段完成页表合并布局构建, 将页表合并结果输出到配置文件中, 等待运行时被访问.

4 运行时阶段

4.1 纹理分块请求获取与处理

4.1.1 放大辅助视角分块请求计算

在中国画展示场景中构建放大镜工具需要在场景中添加额外观察相机, 即放大辅助视角, 调整观察相机 FOV (field of view) 可控制放大倍数. 但传统虚拟纹理方法仅支持单一观察视角的分块请求计算[9].

为支持放大辅助视角的分块请求计算, 本文方法在主观察视角的分块请求计算渲染通道外, 增加放大辅助视角的逐片元分块请求参数计算渲染通道, 放大辅助视角的分块请求参数存储在其渲染通道的渲染结果纹理中, 随后与主观察视角的请求参数渲染结果纹理一同被 Compute Shader 处理.

4.1.2 基于 Compute Shader 处理请求参数纹理

本文使用 Compute Shader 并行加速处理请求参数纹理. Compute Shader 计算并输出分块请求响应集合和分块加载指令集合两个结果缓存, 计算结果存储在 Compute Buffer 中, 异步回读至 CPU 后用于调度分块加载和更新物理纹理. 由于页表为纹理格式, Compute Shader 可直接访问.

图 2 给出 Compute Shader 处理主观察视角和放大辅助视角分块请求参数纹理的流程. 首先读取像素值获取分块请求参数, 然后根据分块参数访问页表获取响应分块参数, 响应分块指当前请求分块位置处已加载的 mipmap 层级更大或相等的分块; 若响应分块不为当前请求分块, 则需构建分块加载指令; 最后将分块加载指令添加到分块加载指令集合结果缓存, 将响应分块参数添加到分块请求响应集合结果缓存.

4.1.3 哈希函数

本文方法采用哈希的方式省去文献[11]方法先写标记数组, 再遍历标记数组的过程.

$$h = a \cdot (B \cdot F) \bmod m \tag{1}$$

$$B = p \cdot 2^{28} + l \cdot 2^{24} + morton(x, y) \tag{2}$$

本文使用线性同余法构建从分块参数到结果缓存索引之间的哈希映射函数, 如式 (1). 其中, h 表示结果缓存索引值, B 表示分块编码, F 表示当前帧数, a 和

m 为线性同余计算系数, 文献[28]给出具有良好分布均匀性的线性同余系数参考值. B 由式 (2) 计算得到, 式中 p 表示画卷编号, l 表示 mipmap 层级, x 和 y 表示画卷分块坐标, 分块 x 和 y 坐标值以莫顿码 (Morton code) 形式排列. *morton* 函数返回二维莫顿编码值. 出现哈希冲突时, 在结果缓存中保留 mipmap 层级较小的分块计算数据. 在哈希计算输入中加入帧值参数, 使同一分块参数在不同帧的哈希计算结果不同, 保证连续帧内分块请求不会因哈希冲突丢失.

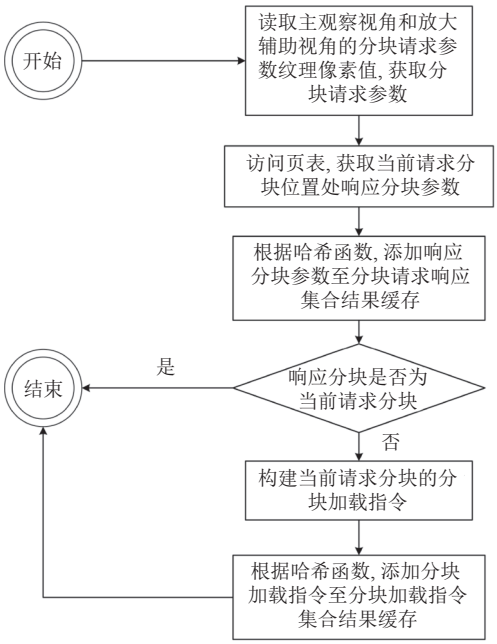

图 2 Compute Shader 处理分块请求参数纹理流程

4.2 纹理分块加载

4.2.1 纹理分块加载方法

Unreal SVT 利用其高效多线程框架异步加载纹理分块[17], 但分块加载任务是短时间任务, 线程间同步引入的锁竞争会带来较大性能损耗. 本文使用无锁队列方法[29]进行纹理分块异步加载. 无锁队列是减少锁竞争的有效方法, 且可避免对移动 VR 平台上有限线程资源的过度占用.

每帧主线程将分块加载指令加入分块加载指令无锁队列中, 加载线程从分块加载指令无锁队列中获取加载指令进行分块加载任务. 其次, 加载线程完成分块加载后, 将分块数据加入分块数据无锁队列中等待主线程获取分块数据.

4.2.2 数量阈值限定的请求分块直接加载策略

当分块请求未命中时, 传统逐 mipmap 层级分块加载策略会优先加载包含目标分块所在区域仍未加载的 mipmap 层级最大的分块[9], 这会增加高清晰度纹理分块显示延迟, 影响中国画观赏体验.

本文方法使用直接加载未命中请求分块的策略, 并通过限定单帧最大分块加载数量, 避免大量突发分块加载请求降低程序执行稳定性. 当请求分块超过单帧最大加载数量阈值时, 当前帧停止向加载指令队列填充分块加载指令, 剩余未发布指令在下一帧继续发布, 直到获取到新一帧分块加载指令集合.

4.3 物理纹理与页表纹理的更新

本文物理纹理采用 LRU (least recently used) 策略来更新纹理分块[9]. 物理纹理空间中还需常驻各画卷 mipmap 层级最大的纹理分块, 确保任意分块请求到达时总有已加载分块进行响应.

页表纹理需每帧根据物理纹理中已加载分块信息执行更新. 逐像素更新页表纹理会给 CPU 带来极大计算负担. 本文采用文献[11]的方法, 利用几何着色器加速页表纹理更新.

4.4 纹理分块显示

渲染物体表面时根据页表纹理查找请求分块的分配信息, 若请求分块已加载至物理纹理, 采样并显示对应分块数据; 若请求分块未加载至物理纹理, 虚拟纹理会选择请求分块所在区域 mipmap 层级更高的已加载分块作为响应分块进行显示.

渲染物体表面显示画卷纹理的计算过程如下.

步骤 1. 利用式 (3) 和式 (4) 来将物体表面 UV 值 (u_m, v_m) 缩放至画卷纹理尺寸内的纹理坐标 (u_p, v_p), 剔除空填充数据. 其中, u_m 和 v_m 表示原纹理坐标值, u_p 和 v_p 表示缩小后画卷纹理尺寸内的纹理坐标值, W_o 和 H_o 表示画卷纹理原尺寸宽高, W_{fill} 和 H_{fill} 表示填充后 2 的幂纹理尺寸宽高.

$$u_p = \frac{W_o}{W_{\text{fill}}} \cdot u_m \tag{3}$$

$$v_p = \frac{H_o}{H_{\text{fill}}} \cdot v_m \tag{4}$$

步骤 2. 根据 (u_p, v_p) 及页表合并布局计算页表纹理采样坐标 (u_t, v_t), 如式 (5) 和式 (6).

$$u_t = \frac{floor\left(\frac{M_p}{2^l}\right) + floor\left(\frac{M_{\text{offset}}}{2^l}\right)}{floor\left(\frac{M_c}{2^l}\right)} \tag{5}$$

$$v_t = \frac{floor\left(\frac{N_p}{2^l}\right) + floor\left(\frac{N_{\text{offset}}}{2^l}\right)}{floor\left(\frac{N_c}{2^l}\right)} \tag{6}$$

其中, 画卷页表尺寸为 $M_p \times N_p$, 合并页表尺寸为 $M_c \times N_c$, 画卷页表在合并页表中位置偏移值为 $(M_{\text{offset}}, N_{\text{offset}})$, 分块的 mipmap 层级为 l. *floor* 函数返回输入浮点数向下取整结果.

步骤 3. 根据 (u_t, v_t) 和 l 采样页表纹理获取响应分块在物理纹理中的坐标 $(x_{\text{phy}}, y_{\text{phy}})$.

步骤 4. 根据 (u_p, v_p) 计算分块内偏移值 $(u_{\text{tile}}, v_{\text{tile}})$, 如式 (7) 和式 (8) 所示, 其中, 物体表面分块划分尺寸为 $X_p \times Y_p$, 纹理分块尺寸为 $X_{\text{tile}} \times Y_{\text{tile}}$, 纹理分块边界扩展像素数为 E.

$$u_{\text{tile}} = \left(u_p \cdot X_m - floor\left(u_p \cdot X_m\right)\right) \cdot \frac{X_{\text{tile}} - 2 \cdot E}{X_{\text{tile}}} + \frac{E}{X_{\text{tile}}} \tag{7}$$

$$v_{\text{tile}} = \left(v_p \cdot Y_m - floor\left(v_p \cdot Y_m\right)\right) \cdot \frac{Y_{\text{tile}} - 2 \cdot E}{Y_{\text{tile}}} + \frac{E}{Y_{\text{tile}}} \tag{8}$$

步骤 5. 根据 $(x_{\text{phy}}, y_{\text{phy}})$ 和 $(u_{\text{tile}}, v_{\text{tile}})$, 计算物理纹理采样坐标 $(u_{\text{phy}}, v_{\text{phy}})$, 如式 (9) 和式 (10) 所示, 其中物理纹理分块划分尺寸为 $X_{\text{phy}} \times Y_{\text{phy}}$.

$$u_{\text{phy}} = \left(\frac{x_{\text{phy}} + u_{\text{tile}}}{X_{\text{phy}}}\right) \tag{9}$$

$$v_{\text{phy}} = \left(\frac{y_{\text{phy}} + v_{\text{tile}}}{Y_{\text{phy}}}\right) \tag{10}$$

步骤 6. 根据 $(u_{\text{phy}}, v_{\text{phy}})$ 采样物理纹理渲染物体表面.

5 实验结果与分析

5.1 实验环境

本节实验将本文改进的虚拟纹理方法在 Unity 内置渲染管线上进行实现, Unity 版本为 2021.3.33f1c1, 使用 C#语言编写.

实验用到的移动 VR 头显设备为 Quest Pro, Quest Pro 设备的计算芯片是 Snapdragon(TM) XR2+, 12 GB 运行时内存, 操作系统版本是基于 Android 的 Quest OS v68, 单眼显示分辨率为 1800×1920. 本节部分实验使用头显自带录屏功能录制程序运行画面, 录制视频分辨率为 1024×1024, 帧率为 60 f/s, 码率为 15 Mb/s.

实验均采用 128×128 纹理分块尺寸及 4 像素边界扩展, 纹理格式为 RGB24. 在单幅图实验场景中, 实验设置物理纹理大小为 30×30 个纹理分块, 约占 47 MB, 在多幅图实验场景中设置物理纹理大小为 45×45 个纹理分块, 约占 95 MB.

5.2 不同分块加载方法加载效率对比

本节实验对多线程异步加载、无锁队列异步加载两种分块加载方法的分块加载效率进行对比. 实验在空场景中放置尺寸为 0.5 m×12 m 的《千里江山图》画卷, 纹理分辨率为 152089×6083, 控制观察相机距离 10 cm 正对画卷头部, 以 1 m/s 的速度匀速从画卷头部运动到画卷尾部然后跳转回头部循环运动. 此布局下每帧分块加载请求数量约 220 个, 将分块加载指令拷贝 10 份后添加到分块加载指令队列中等待加载线程加载, 确保有足够的待加载分块请求供加载线程访问. 主线程获取加载完成的纹理分块数据后直接丢弃, 而不是渲染至物理纹理. 统计该过程中多线程异步加载、无锁队列异步加载两种分块加载方法加载完成 5 万个纹理分块的耗时, 统计结果如表 1 所示.

表 1 不同分块加载方法的加载效率比较

分块加载方式	加载线程数量 (个)	加载完成5万分块耗时 (s)
多线程异步加载	1	11.45
	2	10.5
	3	10.51
	4	10.87
无锁队列异步加载	1	4.81

从数据看出, 多线程异步加载在加载线程数量为 2 时耗时最少, 耗时为 10.5 s, 明显高于无锁队列异步加载的 4.81 s 耗时, 说明无锁队列异步加载相较于基于锁同步机制的多线程异步加载策略, 避免了锁竞争带来的性能损耗, 在处理分块加载短时间作业任务时分块加载方法效率更高.

5.3 单幅画卷场景实验

5.3.1 实验场景设计

本文使用简单长方体和平面来构建如图 3 所示的《千里江山图》单张中国画观赏模拟场景, 画卷尺寸为 0.5 m×12 m, 纹理分辨率为 152089×6083. 主观察相机距离 40 cm 正对画卷, 放大镜放置在主观察相机与画卷之间, 放大辅助视角分块请求渲染结果纹理分辨率为 512×512. 放大镜可将画卷约 2.5 cm×2.5 cm 的区域放大为 10 cm×10 cm 显示. 此场景布局下, 主视角请求 mipmap 层级为 2 的纹理分块, 放大镜请求 mipmap 层级为 0 的纹理分块.

图 3 单幅画卷实验场景

5.3.2 放大辅助视角的观察效果测试

图 4 和图 5 给出结合放大辅助视角和未结合放大辅助视角的虚拟纹理方法画卷纹理细节观察效果对比. 图 4 中结合放大辅助视角的虚拟纹理方法对主观察视角和放大辅助视角均进行了纹理分块请求计算, 明显提升了画卷纹理高清晰细节呈现能力.

图 4 结合放大辅助视角的虚拟纹理方法的观察效果

图 5 未结合放大辅助视角的虚拟纹理方法的观察效果

5.3.3 不同请求参数纹理处理方法对比

本节实验对 CPU 多线程异步、使用标记数组的 Compute Shader 及使用哈希方法的 Compute Shader 这 3 种不同分块请求参数纹理处理方法的执行性能和显示延迟进行对比. 在单幅图模拟场景下, 控制主观察相机和放大镜一同从左向右以 1 m/s 的速度匀速运动 10 s, 统计该过程中 3 种不同方法的平均程序运行帧率、平均分块请求参数纹理处理延迟及平均分块请求命中率, 并使用头显自带录屏功能录制程序运行画面, 录制帧率为 60 f/s. 直接加载数量阈值设置为 16.

表 2 给出不同请求参数纹理处理方法下程序运行性能参数对比, 从数据看出, 两种使用 Compute Shader 的方法相较采用 CPU 多线程异步的方法有更低的请求参数纹理处理延迟及更高的分块请求命中率.

表 2 单幅画卷实验场景下, 不同请求参数纹理处理方法的运行性能对比

请求参数纹理处理方法	平均运行帧率 (f/s)	平均请求参数纹理处理延迟 (帧)	平均分块请求命中率 (%)
CPU多线程异步	89.97	13.24	76.96
使用标记数组的 Compute Shader	89.73	2.00	86.65
使用哈希方法的 Compute Shader	89.34	2.00	86.40

某一观察视角下所有纹理分块请求的显示延迟由请求集合获取延迟和分块加载延迟两部分组成, 其中, 请求集合获取延迟又包含等待请求参数纹理处理指令延迟和请求参数纹理处理延迟.

在本实验的观察视角位置及运动速度条件下, 平均每帧向右运动新增约 3 个主观察视角分块请求和 16 个放大辅助视角分块请求. 对于两种使用 Compute Shader 的虚拟纹理方法, 平均分块请求参数纹理处理延迟帧数为 2, 使得等待请求参数纹理处理指令延迟帧数为 0 或 1. 每次分块请求集合计算结果包含当前帧放大辅助观察视角的分块请求和平均 2 个运行帧内因等待延迟未被请求的新增主观察视角分块请求, 共计 22 个分块请求, 分块加载延迟为 2 个运行帧. 综上, 使用 Compute Shader 的方法某一观察视角下所有纹理分块请求显示延迟为 4–5 个运行帧, 约 44–56 ms. 图 6 和图 7 给出两种使用 Compute Shader 的方法在相机运动 10 s 后, 从静止第 1 帧开始到视角内纹理完全清晰过程的部分录制视频帧帧号及帧画面. 分块请求集合获取延迟期间视频帧画面没有变化, 分块加载延迟期间画面逐渐变得清晰. 假设静止在当前视角的第 1 帧帧号为 1, 红框标记了录制视频帧画面中的模糊区域. 两种使用 Compute Shader 的方法均在 3 个视频帧 (约 50 ms) 内完成所有高清晰纹理分块的加载与显示, 在 44–56 ms 之间, 符合预期计算结果.

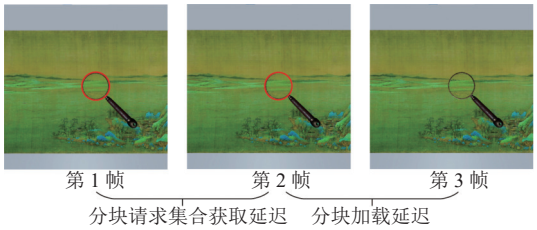

图 6 使用标记数组的 Compute Shader 处理分块请求参数纹理的虚拟纹理方法的静止画面变化

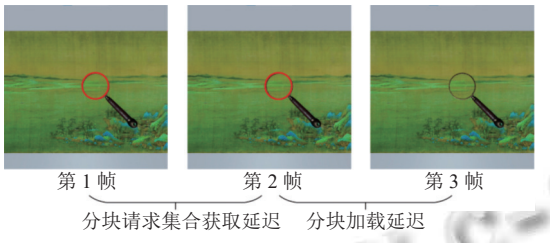

图 7 使用哈希方法的 Compute Shader 处理分块请求参数纹理的虚拟纹理方法的静止画面变化

对于 CPU 多线程异步的请求参数纹理处理方法, 平均请求参数纹理处理延迟帧数为 13.24, 以同样方法计算, 某一观察视角下所有纹理分块请求显示延迟平均为 17–29 个运行帧, 约 189–322 ms. 图 8 为 CPU 多线程异步分块请求参数纹理处理方法的静止画面变化过程. 使用 CPU 多线程异步的方法需 14 个视频帧才能完成视角内所有高清晰纹理分块的加载与显示, 约 233 ms, 符合预期计算结果, 但高于两种使用 Compute Shader 方法的延迟. 从实际视觉效果上, 也说明使用 Compute Shader 处理请求参数纹理相较于使用 CPU 多线程异步处理请求参数纹理有更低的高清晰纹理分块显示延迟.

5.3.4 不同分块加载策略对比

本节实验对逐 mipmap 层级加载、数量阈值限定的直接加载两种不同分块加载策略的程序执行稳定性及画面显示延迟进行对比. 观察相机位置及运动方式与第 5.3.3 节实验相同, 直接加载策略的数量阈值设置为 16.

表 3 给出两种使用不同分块加载策略的虚拟纹理方法的运行性能对比, 从数据看出, 单帧加载数量阈值保证了直接加载策略的程序运行稳定性, 两种加载策略下均有较小的帧率统计标准差值.

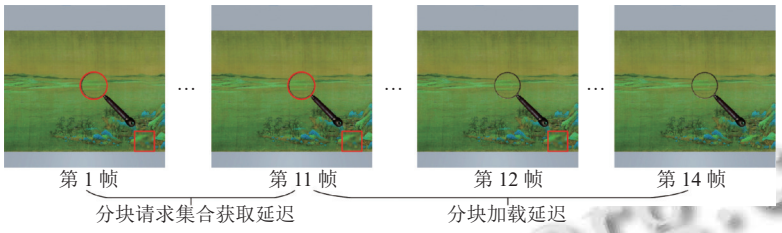

图 8 使用 CPU 多线程异步处理分块请求参数纹理的虚拟纹理方法的静止画面变化

表 3 不同分块加载策略的程序运行性能比较

分块加载策略	平均运行帧率 (f/s)	帧率统计标准差
逐mipmap层级加载	89.76	0.65
数量阈值限定的直接加载	89.81	0.72

图 9 给出使用逐 mipmap 层级分块加载策略的虚拟纹理方法在相机运动 10 s 后从静止的第 1 帧开始到视角内纹理完全清晰过程的部分视频帧帧号及帧画面, 假设静止在当前视角时的第 1 帧帧号为 1. 使用逐 mipmap 层级分块加载策略的虚拟纹理方法需要 6 个视频帧 (约 100 ms), 高于使用数量阈值限定的直接加载策略 3 个视频帧 (约 50 ms) 延迟, 见图 7, 说明数量阈值限定的未命中分块直接加载策略相较于逐 mipmap 层级分块加载策略降低了高清晰纹理分块显示延迟.

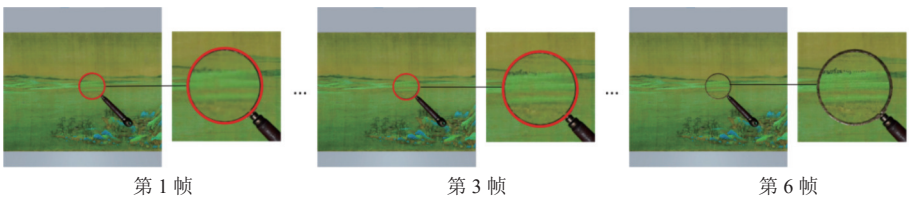

图 9 使用逐 mipmap 层级分块加载策略的虚拟纹理方法的静止画面变化

5.4 多幅画卷场景实验

5.4.1 实验场景设计

本文使用简单长方体和平面构建图 10 所示的多幅中国画观赏场景, 场景中包含 8 张超高分辨率画卷纹理的 5 份数据拷贝, 共 40 份超高分辨率纹理, 各画卷纹理分辨率参数如表 4 所示, 40 幅画卷总存储大小占用超 150 GB. 场景将 40 幅画卷分为 5 组, 每组 8 幅画卷平铺在高 70 cm 的桌子上. 场景设计 8、16、24、32、40 这 5 组不同画卷数量下的观察相机运动路线, 如图所示, 观察相机距离地面 170 cm, 并倾斜一定视角可观察到相机前方摆放的所有画卷.

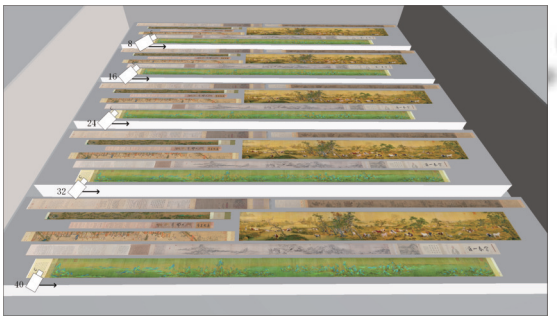

图 10 多幅画卷实验场景

表 4 超高分辨率中国画纹理实验测试用图参数

序号	分辨率	画卷名称
1	152089×6083	《千里江山图》
2	196301×5197	《富春山居图》
3	159010×4339	《清明上河图》
4	102680×5229	《汉宫春晓图》
5	109321×7183	《步辇图》
6	117968×4722	《五牛图》
7	91403×11161	《百骏图》
8	119177×3858	《洛神赋》

5.4.2 哈希方法优化策略的效果测试

本节实验对使用标记数组的 Compute Shader 以及使用哈希方法的 Compute Shader 两种不同分块请求参数纹理处理方法在呈现不同画卷数量时的执行性能进行对比. 在 5 组不同画卷数量实验条件下, 控制观察相机从左向右以 1 m/s 的速度匀速运动 10 s, 统计该过程中两种不同方法的平均运行帧率.

图 11 给出不同画卷数量时两种方法的程序平均运行帧率对比, 使用标记数组的 Compute Shader 处理方法当分块较多时, 遍历标记数组构建当前帧请求集合的计算负担会增加, 程序运行帧率出现下降. 而使用哈希方法优化 Compute Shader 请求参数纹理处理方法的程序运行帧率并没随画卷数量增加而下降, 说明本文给出的哈希优化 Compute Shader 请求参数纹理处理方法在多图场景下运行性能更优.

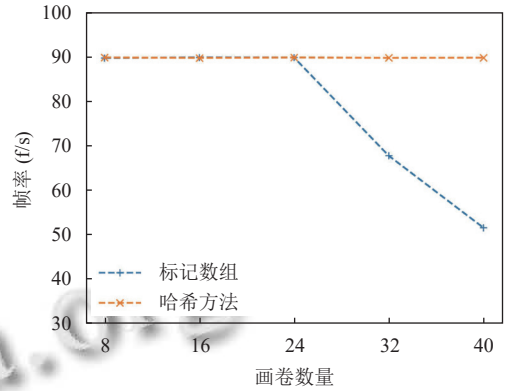

图 11 多幅画卷实验场景下, 不同分块请求参数纹理处理方法的运行性能对比

5.4.3 本文方法各执行阶段耗时

表 5 给出本文方法随画卷数量增多各执行阶段耗时统计. 由于画卷数量增多需要更多最大 mipmap 层级纹理分块常驻在物理内存中, 导致更少的可用物理纹理空间, 使得物理纹理空间分块替换更频繁, 随画卷数量增多, 物理纹理更新及页表纹理更新的耗时会增多. 分块请求计算时需要对场景进行逐物体的分块请求参数计算渲染, 随着画卷数量增多, 场景需要遍历的物体数量变多, 计算耗时增加. 由于本文方法采用数量阈值限定的分块直接加载策略, 每帧可发布分块加载指令数量受阈值限制, 所以随着画卷数量增多, 分块加载指令发布耗时无变化.

表 5 多幅画卷实验场景下, 本文方法各阶段执行时间统计 (ms)

画卷数量	总执行时间	分块请求计算	分块加载指令发布	物理纹理更新	页表纹理更新
8	2.79	0.89	0.12	0.14	1.17
16	2.93	0.92	0.12	0.18	1.21
24	3.18	1.03	0.12	0.24	1.25
32	3.97	1.18	0.12	0.37	1.63
40	4.41	1.21	0.12	0.47	1.89

5.5 与已有方法的对比实验

5.5.1 与已有方法运行性能对比

本节实验比较本文方法与虚拟纹理基本方法[10]、Ma 等人[19]给出的适配低性能设备的虚拟纹理方法以及 Unreal SVT[17]这 3 种已有虚拟纹理方法在不同纹理分块总数量条件下程序运行性能. Ma 等人[19]给出的虚

拟纹理方法在本实验中简称为 Ma VT. 其中, 基本方法和 Ma VT 两个对比项根据文献[10]和文献[19]在 Unity 内置渲染管线上进行实现. Unreal SVT 使用 UE 5.3 版本进行实验, 采用引擎默认配置. 由于已有虚拟纹理方法不支持放大辅助视角, 所以本实验模拟观赏者不使用放大辅助工具时近距离观赏画卷的静止观赏行为. 实验使用与单幅画实验场景相同的场景布局, 控制观察视角距离 25 cm 正对画卷, 保持观察相机静止. 通过对《千里江山图》画卷纹理进行裁剪, 构建 4 096 个分块数量步长下的 5 组不同纹理分块总数量纹理分块数据, 统计随纹理分块总数量增加, 不同虚拟纹理方法的平均程序运行帧率.

实验结果如图 12 所示, 本文方法、基本方法以及 Unreal SVT 的分块请求集合计算采用基于分块请求参数纹理的方法, 请求参数纹理尺寸仅与显示设备分辨率有关, 程序运行帧率随分块数量增加无明显变化. 本文方法和 Unreal SVT 均以 90 f/s 满帧率运行, 但基本方法由于逐像素处理请求参数纹理性能较低, 运行帧率较低. Ma VT 的分块请求计算采用基于空间距离的逐分块计算方法, 在分块数量较少时, 程序可以满帧率运行, 但随分块数量增多, CPU 上逐分块计算分块请求的计算量增加, 程序运行帧率逐渐下降.

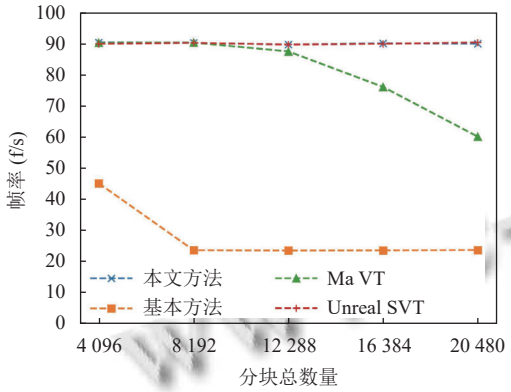

图 12 4 种虚拟纹理方法在不同纹理分块总数量时的平均运行帧率对比

实验表明, 在纹理分块数量较多时, 本文方法相较于虚拟纹理基本方法和 Ma 等人提出的适配低性能设备的虚拟纹理方法具有更好的性能表现, 且与 Unreal SVT 的运行性能是可竞争的.

5.5.2 与 Unreal SVT 高清晰纹理分块显示延迟对比

本节实验比较本文方法与 Unreal SVT 在模拟观赏者近距离观赏画卷行为时的高清晰纹理分块显示延迟. 实验使用与单幅画实验场景类似的场景布局, 不使用放大辅助工具, 控制观察视角距离 25 cm 正对画卷, 以 0.5 m/s 的速度从左向右匀速运动一段时间后停止运动.

由于 Unreal SVT 在移动 VR 平台上受限于打包 Apk 文件 2 GB 大小限制, 所以 Unreal SVT 实验场景中仅展示半张《千里江山图》纹理, 如图 13 所示. 使用头显自带录屏功能录制程序运行画面, 录制视频帧率为 60 f/s.

图 13 Unreal SVT 高清晰纹理分块显示延迟测试实验场景

本节实验的相机观察行为下, 本文方法平均每运行帧观察视角内新增约 7 个分块加载请求, 可在 1 个运行帧内完成所需分块加载. 以第 5.3.3 节同样的显示延迟计算方法计算, 本文方法在某一观察视角下所有纹理分块请求显示延迟为 3–4 个运行帧, 约 33–44 ms.

图 14 给出上述实验过程中, 本文方法从相机静止第 1 帧开始到视角内纹理完全清晰过程的部分录制视频帧帧号及帧画面, 假设静止在当前视角的第 1 帧帧号为 1. 由于录制视频对观察画面有一定裁剪, 导致随相机向右运动观察画面最右侧新出现的纹理区域无法在录制视频画面中及时呈现, 当最右侧画面在录制视频画面中出现时已完成分块加载, 所以本文方法在该实验的整个运行录制视频画面中无模糊分块区域, 比如图 14 中静止前 1 帧、第 1 帧和第 2 帧画面.

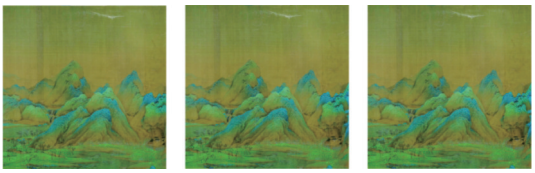

图 14 本文方法在模拟近距离观察行为时的静止画面变化

图 15 给出实验过程中 Unreal SVT 从相机静止第 1 帧开始到视角内纹理完全清晰过程的部分录制视频帧帧号及帧画面, 红框标记了录制视频帧画面中存在的模糊区域. Unreal SVT 需要 30 个视频帧 (约 500 ms), 而本文方法在 44 ms 内即可完成视角内所有高清晰纹理分块加载与显示, 说明本文方法在高清晰纹理显示延迟方面有明显优势.

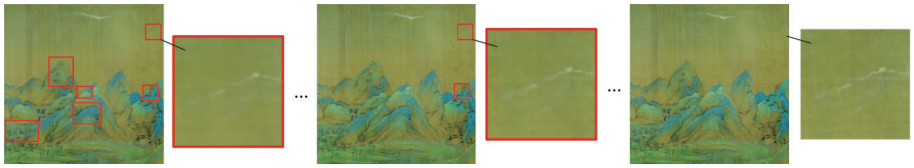

图 15 Unreal SVT 在模拟近距离观察行为时的静止画面变化

6 结束语

本文给出了一种改进的虚拟纹理方法, 在移动 VR 头显上可以高清晰、高帧率、低延时显示大量、超高分辨率国画纹理. 本文方法主要对已有虚拟纹理方法的分块请求计算阶段和分块加载阶段进行优化. 在分块请求计算阶段加入放大辅助视角请求计算, 并基于 Compute Shader 和哈希方法加速处理分块请求参数纹理; 在分块加载阶段, 本文方法利用无锁队列和数量阈值限定的请求分块直接加载策略提高分块加载效率, 降低高清晰纹理分块显示延迟. 与已有方法相比, 本文方法可在画卷纹理分块总数量较多条件下保持高帧率运行, 并具有更低的高清晰纹理分块显示延迟. 未来将研究解决当前方法不支持遮挡或半透明物体的问题, 以在更复杂应用场景中, 实现超高分辨率中国画纹理正确、高清晰、低延时显示.

参考文献

1 Jin S, Fan M, Wang YC, *et al*. Reconstructing traditional Chinese paintings with immersive virtual reality. Proceedings of Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems. Honolulu: ACM, 2020. 1–8. [doi: 10.1145/3334480.3382934]

2 Yuan C, Yun Z. Tunable, a VR reconstruction of “Listening to a Guqin” from emperor Zhao Ji. Proceedings of the 2016 SIGGRAPH ASIA VR Showcase. Macao: ACM, 2016. 1–2. [doi: 10.1145/2996376.2996379]

3 王赫. 故宫古书画复制的数字化生存. 丝网印刷, 2014(5): 28–33.

4 Tanner CC, Migdal CJ, Jones MT. The clipmap: A virtual mipmap. Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques. ACM, 1998. 151–158. [doi: 10.1145/280814.280855]

5 Mittring M, GmbH C. Advanced virtual texture topics. Proceedings of the 2008 ACM SIGGRAPH Games. Los Angeles: ACM, 2008. 23–51. [doi: 10.1145/1404435.1404438]

6 Masood Z, Jiangbin Z, Ahmad I, *et al*. High-performance adaptive texture streaming for planetary-scale high-mobility information visualization. Journal of King Saud University-Computer and Information Sciences, 2022, 34(10): 8336–8349. [doi: 10.1016/j.jksuci.2022.08.014]

7 Masood Z, Jiangbin Z, Irfan M, *et al*. High-performance virtual globe GPU terrain rendering using game engine. Computer Animation and Virtual Worlds, 2023, 34(2): e2108. [doi: 10.1002/cav.2108]

8 Liu T, Wang S, Lei ZL, *et al*. Trajectory risk cognition of ship collision accident based on fusion of multi-model spatial data. The Journal of Navigation, 2022, 75(2): 299–318. [doi: 10.1017/s0373463322000066]

9 van Waveren JMP. Software virtual textures. https://www.mrelusive.com/publications/papers/Software-Virtual-Textures.pdf. (2012-02-25)[2024-09-01].

10 Barrett S. Sparse virtual textures. https://silverspaceship.com/src/svt/. [2024-09-01].

11 Hollemeersch CF, Pieters B, Lambert P, *et al*. Accelerating virtual texturing using CUDA. Proceedings of the 2009 GPU Technology Conference. San Jose, 2009.

12 Chajdas MG, Eisenacher C, Stamminger M, *et al*. Virtual texture mapping 101. In: Engel W, ed. GPU Pro 360 Guide to Rendering. New York: A K Peters/CRC Press, 2018. 69–79. [doi: 10.1201/9781351261524-4]

13 Chen K. Adaptive virtual textures. In: Engel W, ed. GPU Pro 360 Guide to Rendering. New York: A K Peters/CRC Press, 2018. 513–526. [doi: 10.1201/9781351261524-30]

14 Revanth NR, Narayanan PJ. Large-scale virtual texturing on a distributed rendering system. Proceedings of the 5th National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG). Patna: IEEE, 2015. 1–4. [doi: 10.1109/ncvpripg.2015.7490050]

15 Magro M, Bugeja K, Spina S, *et al*. Atlas shrugged: Device-agnostic radiance megatextures. Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications. Valletta, 2020. 255–262. [doi: 10.5220/0008954902550262]

16 Unity. Streaming virtual texturing. https://docs.unity3d.com/Manual/svt-streaming-virtual-texturing.html. [2024-09-01].

17 Unreal Engine. Virtual texturing. https://dev.epicgames.com/

documentation/en-us/unreal-engine/virtual-texturing-in-unreal-engine. [2024-09-01].

18 Lukas R. Rendering interactive maps on mobile devices using graphics hardware [Master's Thesis]. Wien: Technische Universität Wien, 2012.

19 Ma JZ, Fang ZY, Zheng JB, *et al*. A virtual texture technology realized on low computational power platforms. Proceedings of the 11th International Conference on Dependable Systems and Their Applications (DSA). Suzhou: IEEE, 2024. 324–330. [doi: 10.1109/dsa63982.2024.00050]

20 Nourai R, Aksoy V, Htet Z. Cloud rendering of texture map: U.S., 20220319094. 2023-06-13.

21 Funt B, Nourai R, Aksoy V, *et al*. Latency-resilient cloud rendering: U.S., 20220139026A1. 2022-05-05.

22 Obert J, van Waveren JMP, Sellers G. Virtual texturing in software and hardware. Proceedings of the 2012 ACM SIGGRAPH Courses. Los Angeles: ACM, 2012. 5. [doi: 10.1145/2343483.2343488]

23 Schmitz P, Blut T, Mattes C, *et al*. High-fidelity point-based rendering of large-scale 3D scan datasets. IEEE Computer Graphics and Applications, 2020, 40(3): 19–31. [doi: 10.1109/mcg.2020.2974064]

24 Zhang A, Chen K, Johan H, *et al*. High performance texture streaming and rendering of large textured 3D cities. Proceedings of the 2020 International Conference on Cyberworlds (CW). Caen: IEEE, 2020. 17–24. [doi: 10.1109/cw49994.2020.00011]

25 Zhang A, Chen K, Johan H, *et al*. High-performance adaptive texture streaming and rendering of large 3D cities. The Visual Computer, 2022, 38(4): 1245–1262. [doi: 10.1007/s00371-021-02152-z]

26 Paar G, Mett M, Ortner T, *et al*. High-resolution real-time multipurpose tunnel surface 3D rendering. Geomechanics and Tunnelling, 2022, 15(3): 290–297. [doi: 10.1002/geot.202100066]

27 Gonçalves JF, Resende MGC. A biased random key genetic algorithm for 2D and 3D bin packing problems. International Journal of Production Economics, 2013, 145(2): 500–510. [doi: 10.1016/j.ijpe.2013.04.019]

28 L'ecuyer P. Tables of linear congruential generators of different sizes and good lattice structure. Mathematics of Computation, 1999, 68(225): 249–260. [doi: 10.1090/s0025-5718-99-00996-5]

29 Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, 1979, C-28(9): 690–691. [doi: 10.1109/TC.1979.1675439]

(校对责编: 张重毅)