VRSNS向けメモリ拡張型マルチGPU描画フレームワーク VRAM-Xの提案

橋本 空輝　湯村 翼
(北海道情報大学)

1　はじめに

VRChat や Cluster といった VRSNS の普及に伴い，利用者は自作 3D アバターを用いた仮想空間内での自己表現や他者との交流を日常的に行えるようになった．しかし商用ゲームと異なり，利用者が仮想空間に持ち込むアバターモデルは最適化が施されていない場合が多い．利用者は外見の精細さを追求するあまり，微細な装飾品に対しても非圧縮の高解像度テクスチャを割り当て，過剰なポリゴン数を設定する傾向にある．数十名規模の利用者が同一の空間（インスタンス）に密集すると，VRAM の消費量が単一 GPU の搭載容量（一般的な家庭用 PC では 8 GB から 12 GB 程度）を即座に超過する．VRAM が枯渇すると，グラフィックスドライバが描画アセットをメインメモリへ退避させるため，PCIe バスを介したデータ転送がボトルネックとなり，激しいスタッターやフレーム落ちを引き起こす．HMD の装着環境におけるフレームレートの低下は，視覚情報と前庭感覚の不整合を引き起こし，深刻な VR 酔いを誘発する要因となる．

従来，この VRAM 枯渇問題に対しては主に二つの対症療法が用いられてきた．第一の手法は，描画設定によるアバターの非表示化や簡易モデル（灰色の共通モデル等）への置き換えといった受信側のセーフティ機能である．しかしこの手法は，閲覧側の利用者に妥協を強いるだけでなく，制作者がアバターに込めた自己表現やディテールを不可視化し，VRSNS の根幹である臨場感やコミュニケーション価値を著しく損なう．第二の手法は，空間の管理者が負荷指標（アバターパフォーマンスランク等）の上限を設定し，基準を満たさないアバターの表示を一律に制限する運用である [1]．だが過度な規制は利用者の表現活動を萎縮させ，コミュニティからの離脱を招く懸念がある．このように，閲覧側と制作側の双方が体験の質を犠牲にせざるを得ない状況が続いている．

他方，ハードウェア面での負荷分散としてマルチ GPU 技術が存在するが，従来の描画並列化手法は VRAM 容量の拡張を目的として設計されていない．後述するように，既存の並行描画技術は全 GPU 間で同一の描画リソースを複製するメモリミラーリングを前提としており，有効なメモリ空間を倍増させることはできない．

そこで我々は，一般利用者の PC 環境に眠る旧世代のグラフィックスカードや統合 GPU などの異種 GPU を有効活用し，アバター単位で描画処理とメモリ消費を分散させることで VRAM 容量を実質的に拡張するマルチ GPU フレームワーク VRAM-X を提案する．本フレームワークでは，メイン GPU がワールド背景や注視オブジェクトを担当し，サブ GPU が注視外や遠方のアバターの描画とメモリ保持を担当する．オブジェクト単位での描画分担には，パイプラインの最終段で画像を結合する Sort-Last 方式を採用する．しかし異種 GPU を独立プロセスとして非同期に並行動作させると，両者の描画完了タイミングの違いにより，視点移動時にサブ側のアバター表示が遅れて追従するという知覚的遅延が生じる．

本稿では，VRAM-X のアーキテクチャ設計，Win32 共有メモリを用いたプロセス間通信プロトコル，および初期プロトタイプによる深度合成の実証について述べる．さらに，非同期分散描画で生じる追従遅延を隠蔽するため，サブ側から受信したアバター画像層のみを最新のカメラ視点へ幾何学的に写し直す局所深度再投影手法を提案し，そのアルゴリズムと検証結果を報告する．

2　関連研究

2.1　並列レンダリングの分類

並列レンダリング手法は，描画パイプラインのどの段階でデータを分配し結合するかによって Sort-First，Sort-Middle，Sort-Last 方式に分類される [2]．Sort-Last 方式は，各 GPU が割り当てられたオブジェクトを独立して描画し，パイプラインの最終段でピクセルごとの深度値（Z 値）を比較して画像を合成する手法である．先行研究である PixelFlow[3] は，Z バッファを用いた画像合成ネットワークによりオブジェクト単位の負荷分散を実現した．しかし PixelFlow は隣接する基板上のカスタムチップ間を専用の超高帯域ネットワークで接続する専用ハードウェアを前提としており，汎用 PC には普及しなかった．VRAM-X は，この Sort-Last 方式の概念を一般的な民生用 PC 上の独立プロセスと共有メモリ通信を用いて実現する点に特徴がある．

2.2　既存のマルチ GPU 技術の限界

一般の PC ゲーム向けに普及したマルチ GPU 技術には，AFR（Alternate Frame Rendering）や SFR（Split Frame Rendering）が存在する．しかしこれらの手法は，いずれも VRAM の容量拡張には寄与しない．

AFR は，偶数フレームと奇数フレームを複数の GPU に交互に描画させる時間的並列化手法である．各 GPU がシーン全体の描画を担当するため，全アセットをすべての GPU の VRAM 上に保持するメモリミラーリングを余儀なくされ，GPU を追加しても利用可能なメモリ容量は増えない．さらにフレーム生成がパイプライン化

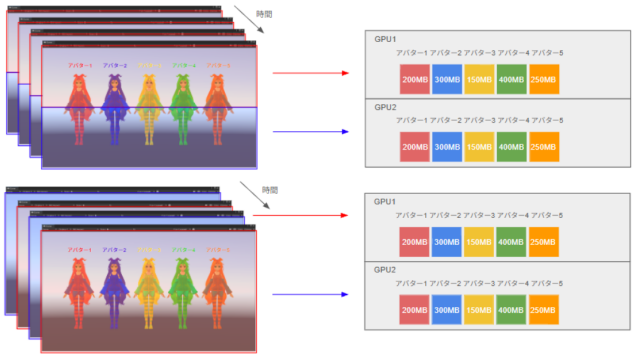


Fig. 1 VRAM-X のシステム構成概要

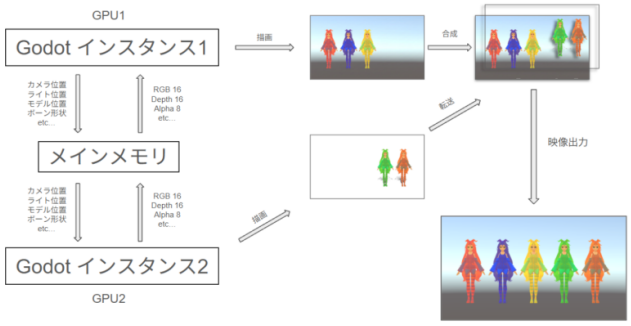


Fig. 2 VRAM-X のシステム構成概要

されるため遅延が増大し，VR 酔いを悪化させる欠点を持つ．

一方，SFR は画面領域を分割して各 GPU に割り当てる空間的並列化手法である．理論上は画面内のオブジェクトだけを該当 GPU に読み込めばよいように見えるが，VRSNS では首振りや移動によってアバターが領域境界を頻繁に横断する．事前に全アセットを双方の GPU に読み込んでおかなければ，境界を横切る瞬間に PCIe 経由の読み込みが発生して激しい画面停止を引き起こすため，実用上はメモリミラーリングが避けられない．

したがって，時間分割（AFR）や画面分割（SFR）では VRAM 不足を解決できない．有効メモリを増やすには，アバター単位で描画とメモリ保持を完全に分担する Sort-Last 方式が有効となる．

2.3 視点再投影技術と本手法の相違点

先行描画された画像を最新の視点姿勢に合わせて写し直す視点再投影は，描画遅延を幾何学的に補正する有力な手段である．VR 向け技術として実用化されている ASW（Asynchronous Spacewarp）や PTW（Positional Timewarp）は，描画が間に合わない場合に前フレームの画像や深度，動きベクトルを用いて画面全体の中間フレームを生成する．これに対し本研究の局所深度再投影は，画面全体のフレーム生成や動きベクトル推定を伴わない．メイン側の描画は常に最新姿勢で遅延なく実行し，独立して非同期に届くサブ側のアバター画像層のみを対象として，カメラ姿勢差と深度値に基づき局所的に再配置する．

3 VRAM-X

3.1 独立 2 プロセスによる異種 GPU 制御

VRAM-X は，オープンソースのゲームエンジンである Godot Engine 4[4] を基盤として稼働する．エンジン本体の描画パイプラインは単一 GPU による描画ループを前提として設計されているため，単一プロセス内で複数 GPU を並列制御することは困難である．そこで本フレームワークでは，OS レベルで物理 GPU ごとに独立した 2 つのプロセス（メインプロセスとサブプロセス）を起動する設計を採用した．起動時引数により Vulkan レンダリングバックエンドの物理デバイス番号（‘VkPhysicalDevice‘）を個別に指定することで，異なる GPU アダプタを各プロセスへ排他的に割り当てる．本フレームワークの全体構成を Fig.2 に示す．

メインプロセスは高性能なメイン GPU（RTX 4060 Laptop GPU や RTX 2080 Ti 等）を制御し，ワールド背景や UI，利用者が注視するオブジェクトの描画を担当する．サブプロセスは手元にある余剰のサブ GPU（Intel UHD Graphics や GTX 1080 等）を制御し，注視外や遠方に位置するアバターモデルのオフロード描画を担当する．未最適化アバターの巨大なテクスチャやメッシュリソースをサブ GPU 側のメモリ空間へ逃すことで，メイン側の VRAM 枯渇を根本から回避する．

3.2 共有メモリ通信パイプライン

プロセス間通信に伴うレイテンシを最小化するため，Win32 API の共有メモリ（名前付きファイルマッピング）を採用した．同期機構には Win32 イベントオブジェクトを用い，ダブルバッファリング構成とすることで，互いの描画ループをブロックしないノンブロッキングなバッファ転送を実現した．

メインプロセスは，現フレームにおけるカメラの位置，姿勢クォータニオン，画角，クリップ面，およびオフロード対象アバターのワールド変換行列を書き込み，サブプロセスへ通知する．サブプロセスは受信したパラメータに基づいてアバターを描画し，出力結果として RGBA16F 形式のカラー画像バッファと，FP16 形式の 逆 Z 深度画像バッファを生成する．これらの描画バッファを描画時点のカメラパラメータとともに共有メモリへ書き戻し，メインプロセスへ返送する．メインプロセスは受信した生データを GPU テクスチャへ展開し，合成シェーダーへ引き渡す．

3.3 初期プロトタイプによる深度合成検証

本フレームワークの根幹である Sort-Last 方式の深度合成を確認するため，RTX 2080 Ti（メイン GPU）と GTX 1080（サブ GPU）を用いた初期プロトタイプ環境

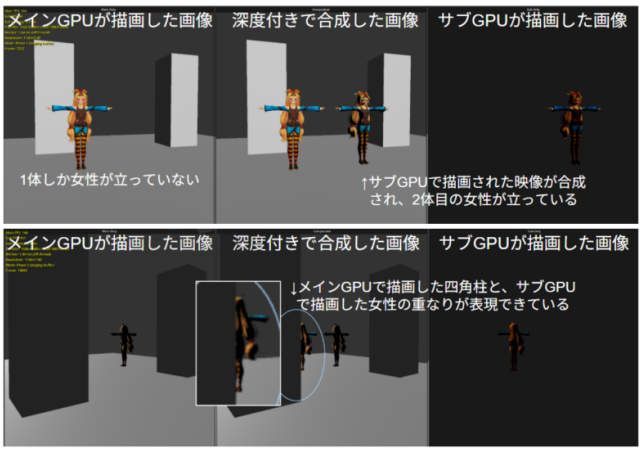


Fig. 3 深度テスト付き合成のテスト結果

を構築した．テストシーンとして，メインプロセスで障害物の柱を描画し，サブプロセスでアバターモデルを描画した．メイン側で動作する合成シェーダーがピクセル単位で深度比較を行った結果，柱の背後にアバターが正しく隠れる遮蔽合成を確認した（Fig.3）．この検証により，共有メモリ通信を用いたプロセス間の深度付き映像合成が実現可能であることを実証した．

4 非同期遅延の課題と局所深度再投影

4.1 非同期描画における視点追従遅延

初期プロトタイプの実証により深度合成の成立を確認した一方，性能の異なる異種 GPU を非同期に並行動作させると，新たな課題が判明した．両プロセスが独立した描画周期で動作するため，視点移動時にサブプロセスから届くアバター画像が過去のカメラ姿勢のまま合成される．その結果，メイン側の背景に対してアバターの追従が遅れ，視覚的な違和感を引き起こす．実際に，幅 2048 px，視野角 60 度，カメラ旋回速度 0.36 rad/s の条件で測定したところ，アバター輪郭の水平中心位置に中央値 32.5 px（p95 で 40.7 px）の追従遅延が発生した．

4.2 局所深度再投影のアルゴリズム

この追従遅延を解消するため，受信したアバター層のみを最新視点へ幾何学的に写し直す局所深度再投影パイプラインを開発した．

本手法はアバターの骨格運動を予測するものではなく，カメラ姿勢の変化のみを GPU コンピュートシェーダー上で幾何補正する．第 1 段階として，シェーダーがサブ画像画素をワールド座標へ逆算し，現フレームの最新視点へと前方投影する．サブ画像上の各画素位置と逆 Z 深度から描画時のクリップ座標 $\mathbf{x}_{\text{sub}}$ を求め，現フレームのクリップ座標 $\mathbf{x}'$ を次式で算出する．

$$\mathbf{x}' = \mathbf{P}_{\text{main}}\mathbf{V}_{\text{main}}\mathbf{V}_{\text{sub}}^{-1}\mathbf{P}_{\text{sub}}^{-1}\mathbf{x}_{\text{sub}} \tag{1}$$

ここで $\mathbf{V}_{\text{sub}}, \mathbf{P}_{\text{sub}}$ はサブ描画時のビュー行列と射影行列であり，$\mathbf{V}_{\text{main}}, \mathbf{P}_{\text{main}}$ は現フレームの行列である．サブ側の行列の逆行列を掛けてワールド座標を復元し，メイン側の最新行列を掛けることで現在の視点へと座標を写し直す．

シェーダーは得られた座標を透視除算し，投影先の 2×2 画素領域へ atomicMax を用いて深度値を書き込む．単一画素投影では視点移動時に隙間が生じやすいため，2 × 2 画素への展開によって面の連続性を確保する．また逆 Z 深度表現では手前が 1.0，奥が 0.0 となるため，浮動小数点の精度が手前側で極めて高くなり，最大値更新によってカメラに最も近い表面の深度値が一意に定まる．

第 2 段階として現視点の各出力画素から元画像へ逆参照サンプリングを行い，第 3 段階で元深度と再投影深度を比較する．サンプリングした深度が再投影深度と一致しない画素や画面外，視点移動で新たに露出した領域は，アルファ値を 0 に設定して透明化し，遮蔽抜けによる描画破綻を抑える．最終段では，メイン画像とアバターの深度値を画素ごとに比較し，メイン側の物体が手前にある場合はメイン画像を表示し，アバターが手前にある場合はアルファブレンド合成を行う．

4.3 本手法の技術的制約

本手法には二つの技術的制約が存在する．第一に，画素ごとに単一深度しか保持できないため，半透明アバター内部の微細な重なりは最前面の単一深度に縮退する．第二に，視点移動に伴う欠損領域の発生である．サブ描画時に隠れていた部位が視点移動によって露出した場合，サブ画像にはテクスチャ情報が存在しないためアルファ値 0 により背景が透過する．視点差が大きくなるほど，アバター輪郭周辺に透過欠損が生じる．

5 評価

5.1 検証環境と前後遮蔽の合成検証

Windows 11 ノート PC（RTX 4060 Laptop GPU と Intel UHD Graphics）のデスクトップ描画出力で動作を検証した．なお客観的な表示遅延測定や HMD 実機装着実験は含まない．単眼解像度 1140 × 1140 px の仮想空間で，前景の赤色立方体，2 体のアバター，後景の緑色立方体を配置した．合成結果を Fig.4 に示し，各 GPU のプレビューを Fig.5 に示す．Fig.5 の左列はメイン GPU 出力，中央列はサブ GPU によるアバター出力，右列は合成出力を表す．赤色立方体がアバター手前に描かれ，緑色立方体がアバターに遮蔽される正しい前後関係を確認した．

5.2 カメラ旋回時の輪郭一致率と両眼プレビュー

カメラ旋回速度 0.36 rad/s の条件で，同一フレームのメイン基準画像（全物体をメイン GPU で同時描画した画像）と合成画像を比較した．輪郭一致率を表す IoU（Intersection over Union）の中央値は，再投影前の 0.0486 から適用後は 0.8615 に向上した．水平中心位置

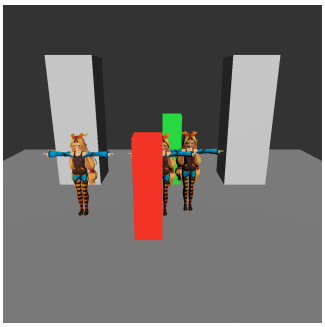


Fig. 4　単眼環境での赤色立方体，2 体のアバター，緑色立方体の合成結果

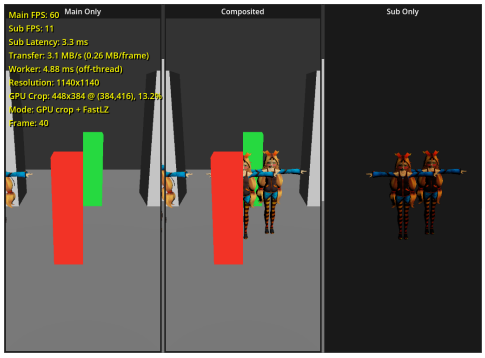


Fig. 5　単眼環境での 3 カラム表示による各 GPU の描画プレビュー

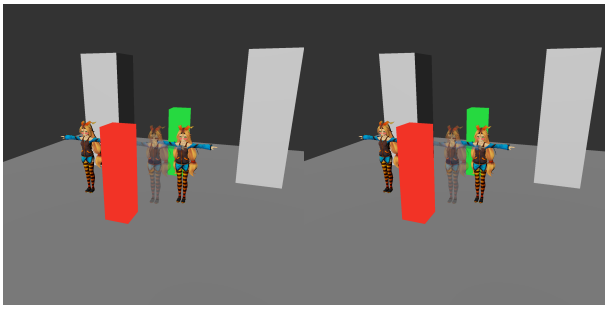


Fig. 6　二眼プレビュー合成結果

のずれ中央値も 32.5 px から 0.5 px へ減少し，局所深度再投影による追従遅延の低減を確認した．

また，立体視を想定した二眼描画の合成を検証した．左右各眼に 1600 × 1600 px（計 3200 × 1600 px），IPD（瞳孔間距離）0.064 m を設定して左右視点から合成処理を実行した．二眼プレビュー結果を Fig.6 に示す．水平視差を伴う立体視環境でも，前後関係を維持した合成を確認した．

5.3　描画性能の計測と観察結果

短時間ログの計測では，メインプロセスの描画速度は 24 から 32 FPS，サブプロセスの描画速度は 15 から 21 FPS で推移した．両 GPU のフレームレート差が大きい場面では同一画像の再投影角度が広がり，輪郭の透過欠損が目立った．

6　考察と今後の課題

局所深度再投影により，メインプロセスはサブプロセスの完了を待たずに最新視点でアバターを表示できる．ただし補正対象はカメラ姿勢変化に限られ，アバター自身の骨格運動は補正できない．また，両 GPU の速度差が開くほど新規露出部の透明化破綻が目立つ．

今後の課題として，次の 4 点に取り組む．第一に，PCIe 帯域の圧迫を防ぐための通信最適化である．現在の手法では非圧縮転送を行うため，透明画素の転送棄却や，サブ側を低解像度で描画し超解像技術（DLSS 等）で補完する手法を導入する．第二に，視線や距離に応じ，注視アバターをメイン GPU へ動的に戻す適応的描画割り当ての導入である．第三に，HMD 実機での遅延計測と，既存 SLI や単一 GPU 構成との主観評価実験である．第四に，Godot Engine 4 での成果を既存 VRSNS へ組み込む統合開発である．

7　おわりに

本稿では，VRSNS での VRAM 不足に対処するため，2 プロセス構成によるメモリ拡張型マルチ GPU フレームワーク VRAM-X を提案し，実装した．VRAM-X では，非同期描画に起因する遅延を補償する局所深度再投影を導入した．異種 GPU 構成により，単眼および二眼での正確な前後遮蔽合成を実現し，旋回時の位置ずれを中央値 32.5 px から 0.5 px へ低減し，IoU を 0.0486 から 0.8615 へ向上させた．今後は通信帯域の削減と HMD 実機評価を進める．

参考文献

[1] Strasz: Developer Update - 9 April 2026, `https://ask.vrchat.com/t/developer-update-9-april-2026/48173` (2026).

[2] Molnar, S., Cox, M., Ellsworth, D. and Fuchs, H.: A Sorting Classification of Parallel Rendering, *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, pp. 23–32 (1994).

[3] Molnar, S., Eyles, J. and Poulton, J.: PixelFlow: High-Speed Rendering Using Image Composition, *ACM SIGGRAPH Computer Graphics*, Vol. 26, No. 2, pp. 231–240 (1992).

[4] Godot Engine Community: Godot Engine 4.x Documentation, `https://docs.godotengine.org/` (2026).