SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

THE PREMIER CONFERENCE
& EXHIBITION ON
COMPUTER GRAPHICS &
INTERACTIVE TECHNIQUES

MOVING MOBILE GRAPHIC

SMARTGI: GLOBAL ILLUMINATION WITH SPACE VOXELIZATION ON MOBILE

SHUN CAO (TENCENT GAMES)

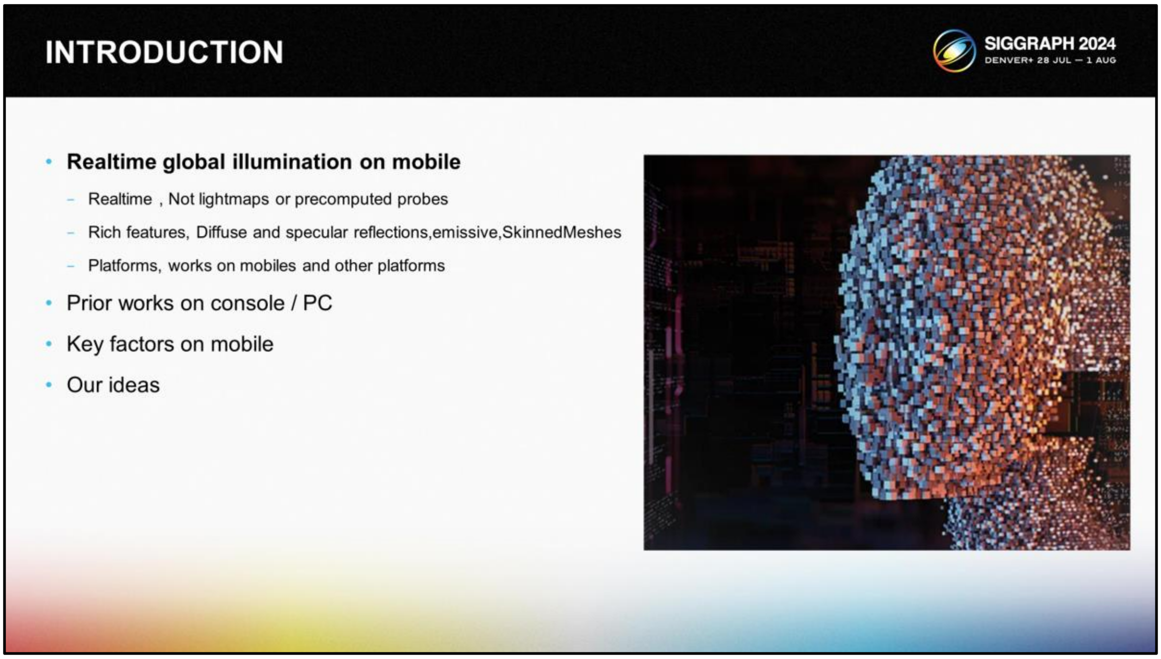
Good afternoon, everyone. I'm Shun from Tencent Games. Today, I'd like to talk about a mobile friendly global illumination solution standing out from the traditional voxelizations.

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

INTRODUCTION

WHY WE DO THIS

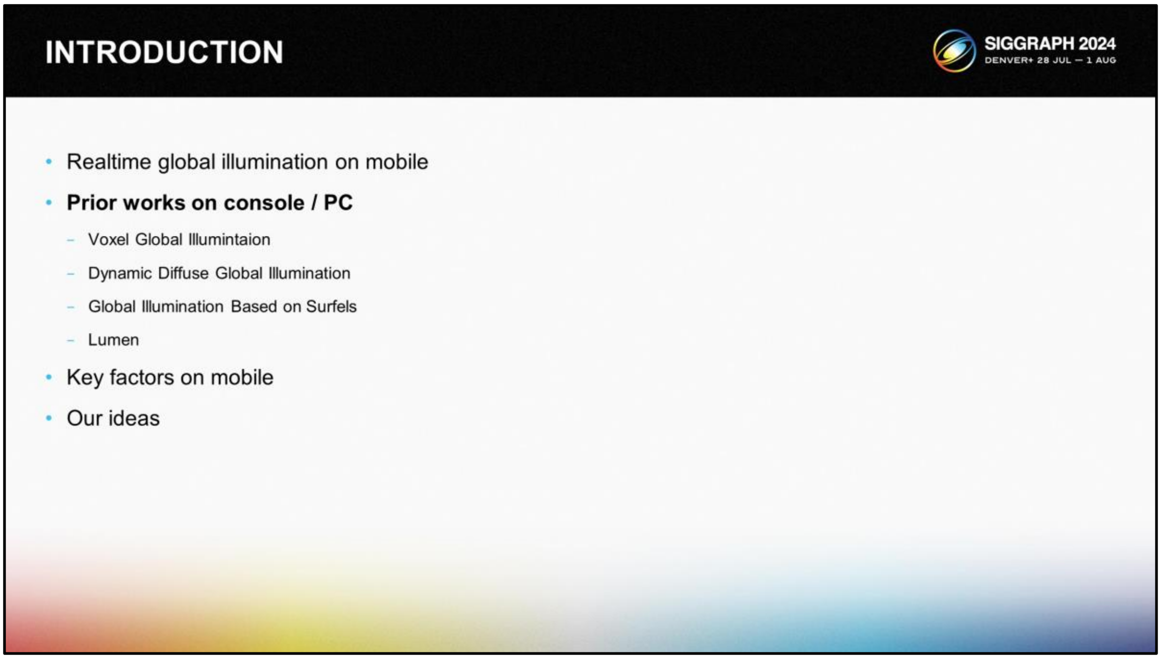
Before diving into any technical details, it might be helpful to look back at why we need to do this and what happened behind the stage.



INTRODUCTION

- **Realtime global illumination on mobile**
  - Realtime , Not lightmaps or precomputed probes
  - Rich features, Diffuse and specular reflections,emissive,SkinnedMeshes
  - Platforms, works on mobiles and other platforms
- Prior works on console / PC
- Key factors on mobile
- Our ideas

As we all know, offline baking solutions are very popular in mobile game engines. Our motivation at the very beginning was to deliver a real-time GI solution, which does not need baking and can achieve various advanced lighting effects, enabling us to create fully dynamic game scenes on today's mobile devices.

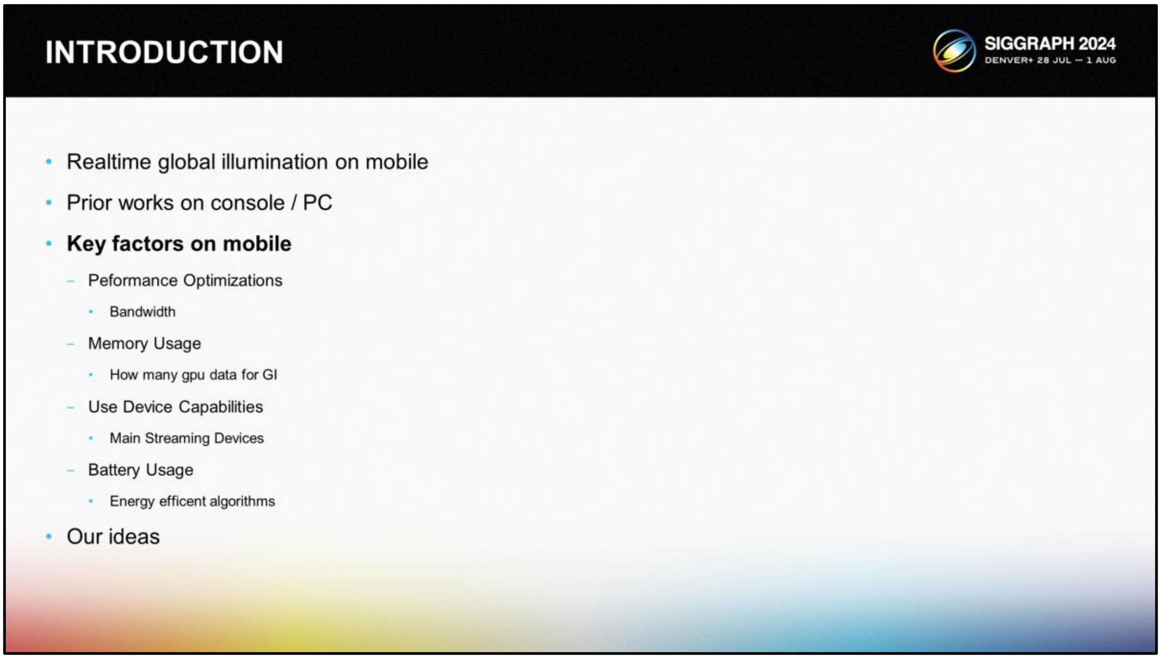


INTRODUCTION

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Realtime global illumination on mobile
- **Prior works on console / PC**
  - Voxel Global Illumintaion
  - Dynamic Diffuse Global Illumination
  - Global Illumination Based on Surfels
  - Lumen
- Key factors on mobile
- Our ideas

We have seen many real-time global illumination solutions shining on Console and PC. We have carefully studied their PROs and CONs and learned from them while developing our own solution.

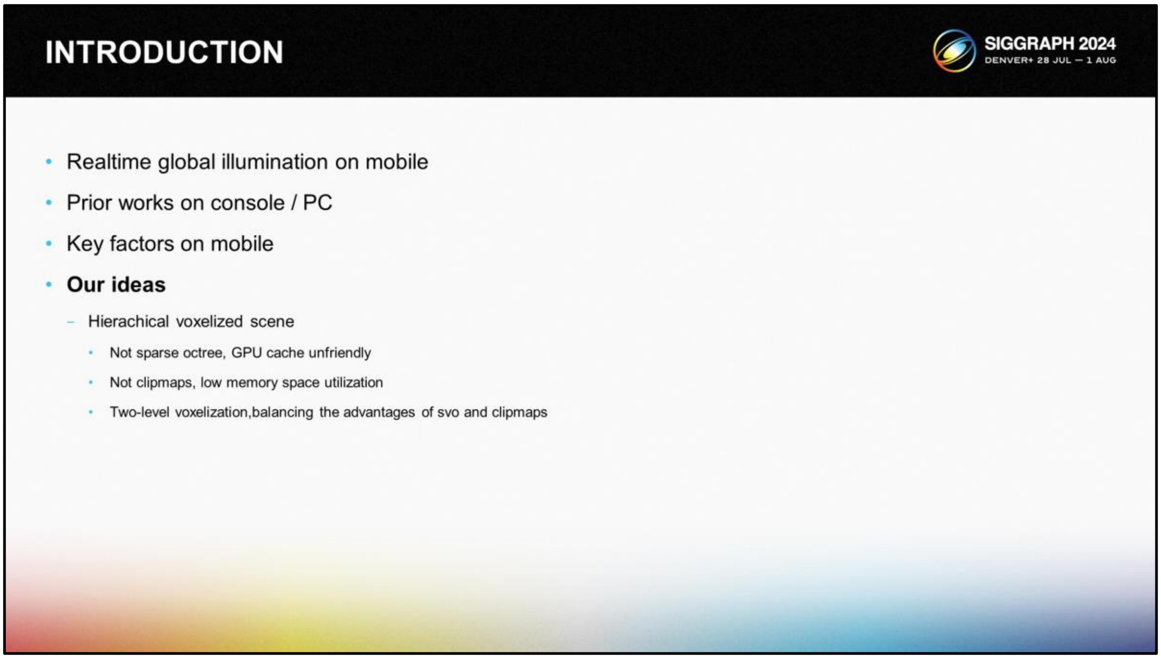


INTRODUCTION

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Realtime global illumination on mobile
- Prior works on console / PC
- **Key factors on mobile**
  - Peformance Optimizations
    - Bandwidth
  - Memory Usage
    - How many gpu data for GI
  - Use Device Capabilities
    - Main Streaming Devices
  - Battery Usage
    - Energy efficent algorithms
- Our ideas

Thanks to all the analyses, we believe that the following factors are important when implementing any mobile GI, such as GPU bandwidth, memory usage, mobile hardware capabilities, and power consumption. We all know that there is still a significant gap between mobile and desktop GPU bandwidth capacities, which matters to performance and power consumption. Memory usage is also crucial; excessive memory footprint can easily lead to a crash. Although high-end mobile GPUs now support raytracing, many devices in the market are left behind. Higher power consumption means battery drainage and overheating coming much earlier, which downclocks the GPU with worse performance.



INTRODUCTION

- Realtime global illumination on mobile
- Prior works on console / PC
- Key factors on mobile
- **Our ideas**
  - Hierachical voxelized scene
    - Not sparse octree, GPU cache unfriendly
    - Not clipmaps, low memory space utilization
    - Two-level voxelization,balancing the advantages of svo and clipmaps

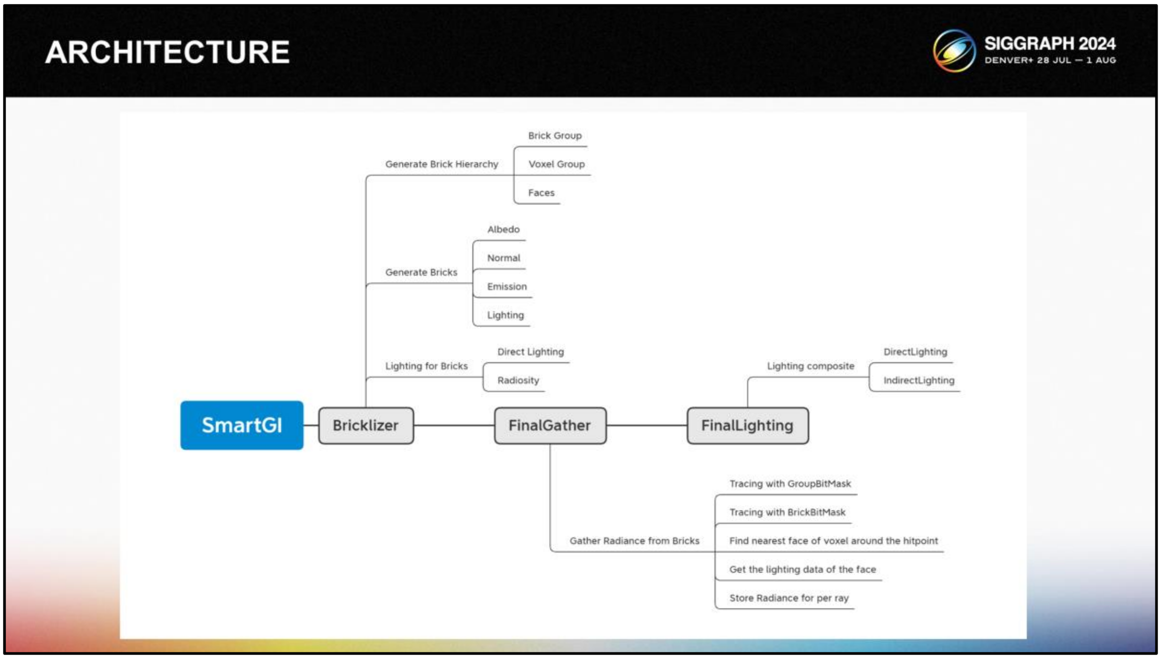
With a considerable amount of learning and struggling, we finally reached the point that a brand new voxel-based algorithm is coming up. It was built on the efficient sparse octrees and cache-friendly clipmaps to achieving a multi-level voxel-based scene representation.

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

METHODOLOGY

HOW WE IMPLEMENT

Now, let's dive into some of the most interesting implementation details.

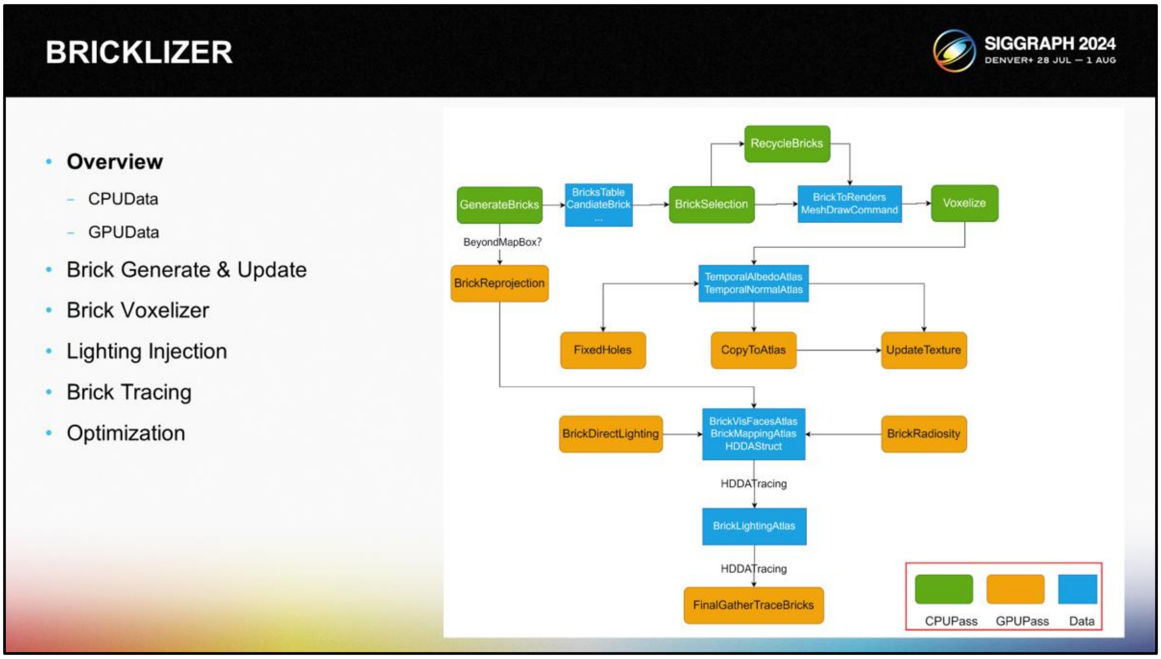


ARCHITECTURE

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- SmartGI
  - Bricklizer
    - Generate Brick Hierarchy
      - Brick Group
      - Voxel Group
      - Faces
    - Generate Bricks
      - Albedo
      - Normal
      - Emission
      - Lighting
    - Lighting for Bricks
      - Direct Lighting
      - Radiosity
  - FinalGather
    - Gather Radiance from Bricks
      - Tracing with GroupBitMask
      - Tracing with BrickBitMask
      - Find nearest face of voxel around the hitpoint
      - Get the lighting data of the face
      - Store Radiance for per ray
  - FinalLighting
    - Lighting composite
      - DirectLighting
      - IndirectLighting

Here is the overall architecture diagram of the solution, which consists of three main parts: Bricklizer, FinalGather, and Lighting Composition. Bricklizer is our approach to achieving hierarchical voxelization. FinalGather is the process of collecting indirect lighting. The final lighting composition process is relatively simple, involving the overlaying of direct and indirect lighting.

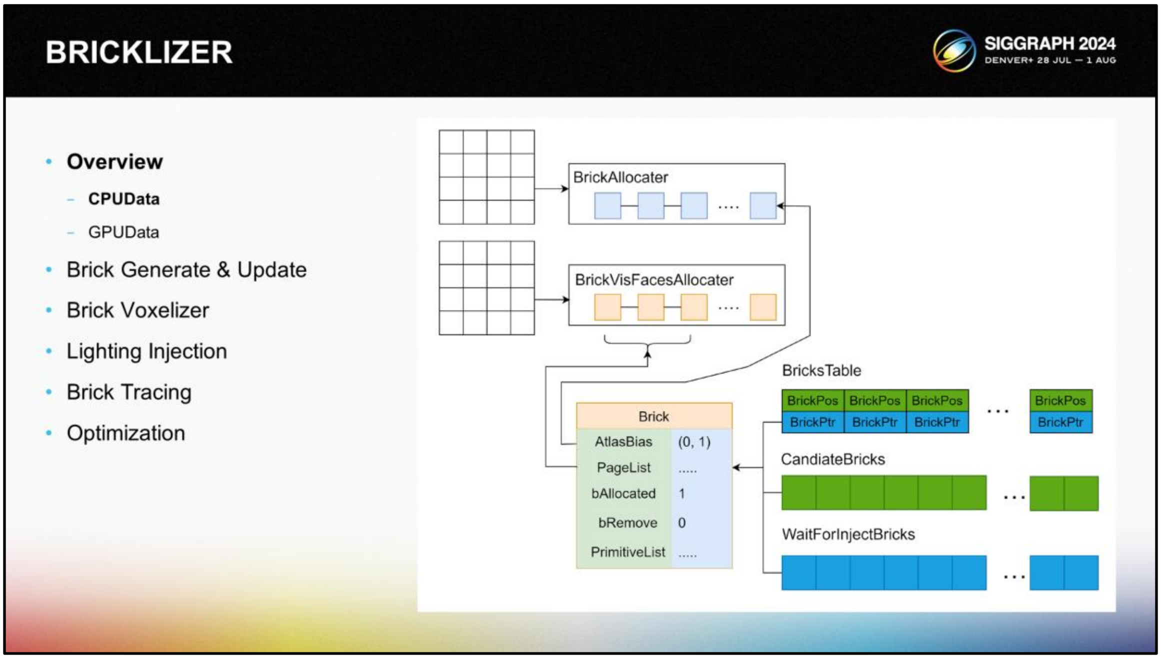


BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- **Overview**
  - CPUData
  - GPUData
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- Brick Tracing
- Optimization

First, let's take a look at the overall process of Bricklizer. On the CPU side, we generate candidate blocks, referred to as bricks, based on the visible range. These candidate bricks are placed in a candidate queue, and a fixed number of bricks are voxelized each frame based on a certain priority. After voxelization, we have a process to repair and optimize the voxelization, such as filling holes in specific directions of the voxels and eliminating invalid faces between two voxels. Once the brick voxelization is complete, we perform lighting injection, including sampling direct light sources and reflecting light from other voxels onto the current voxel. Finally, we can use the HDDA algorithm to calculate the radiance of the screen probes.

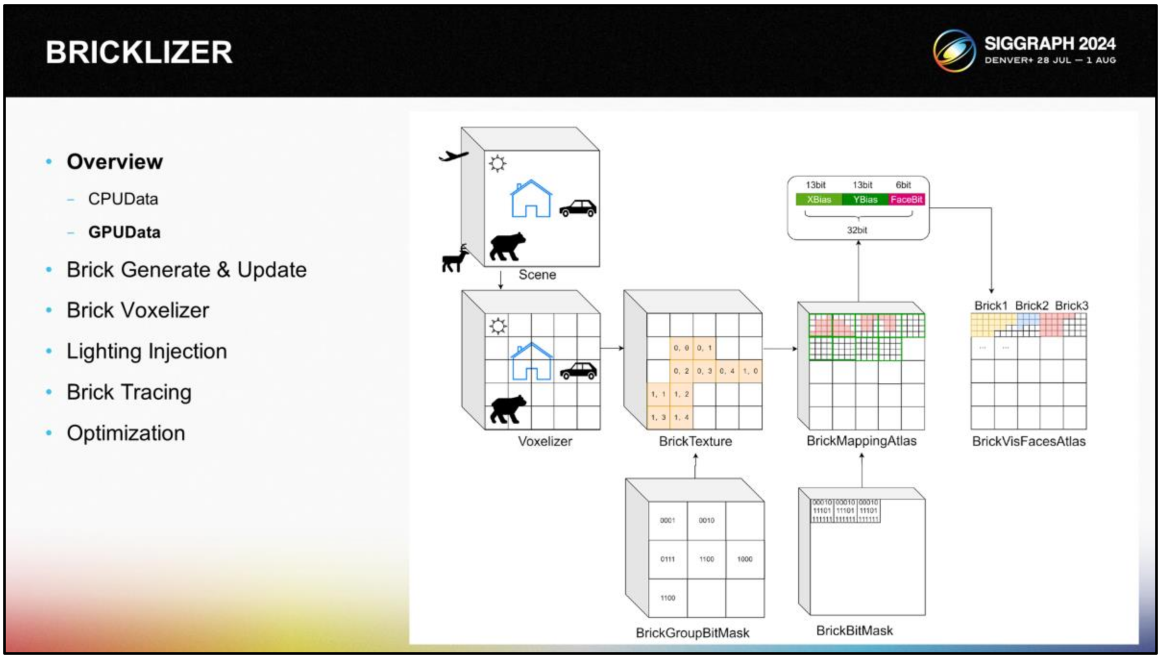


BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- **Overview**
  - **CPUData**
  - GPUData
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- Brick Tracing
- Optimization

The core data structure in the CPU is illustrated as follows. During initialization, two allocators are initialized based on the given texture sizes of BrickMappingAtlas and BrickVisFacesAtlas, and an array is created by stringing together the Bias as the base elements. Some important parameters in Brick are:
AtlasBias: The Bias obtained from BrickAllocator, used for mapping to BrickMappingAtlas.
PageList: Maintains the PageBias of several Bricks in VisFacesAtlas in the form of an array.
bAllocated: Indicates whether space has been allocated and is also used to distinguish whether capture is complete.
bRemove: Bricks exist in three lists in the form of smart pointers. If a brick needs to be deleted, set this flag to 1, and then delete it from the BricksTable. The other two lists will check this flag before processing. PrimitiveList contains all the primitives intersect with the brick.

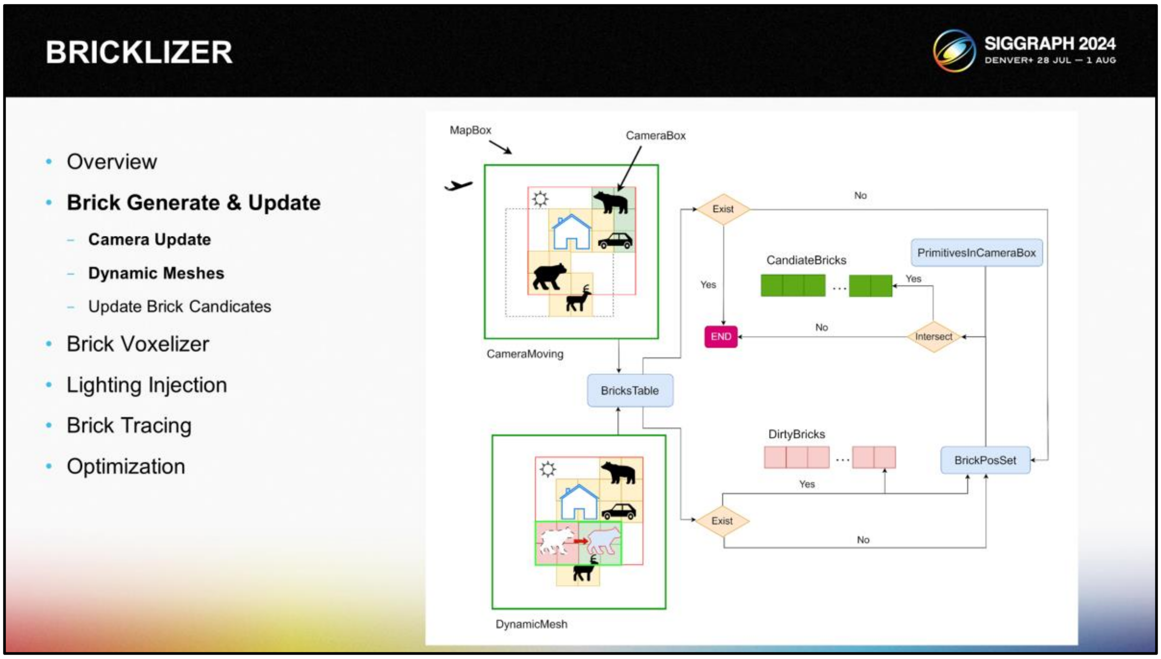


BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- **Overview**
  - CPUData
  - **GPUData**
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- Brick Tracing
- Optimization

The resource organization structure in the GPU is illustrated as follows. For the 3D scene shown, BrickTexture stores the actual storage locations of each Brick with a value in BrickMappingAtlas. The default configuration is that each Brick covers a range of 4x4x4 m, with a total coverage of 512x512x128 m. BrickMappingAtlas stores data in Voxel units, where each Voxel stores a mapping pointer to the next level, composed of 32 bits. 26 bits are used to represent x and y-axis offsets, and 6 bits indicate the presence or absence of each face. The actual storage carrier for each voxel face is a 2D Atlas called BrickVisFacesAtlas, which tightly stores every valid face (where the adjacent voxel is empty or translucent). Only the last page of each Brick may contain intra-page fragments, so the space utilization is extremely high. The following two 3D Textures are auxiliary structures for HDDA Tracing. Among them, BrickGroupTexture uses 4x4x4 Bricks as a composition unit, with each grid storing a 64-bit BitMask used to indicate the presence or absence of Bricks. Similarly, BrickBitMask uses BitMask to represent the presence or absence of corresponding voxels.



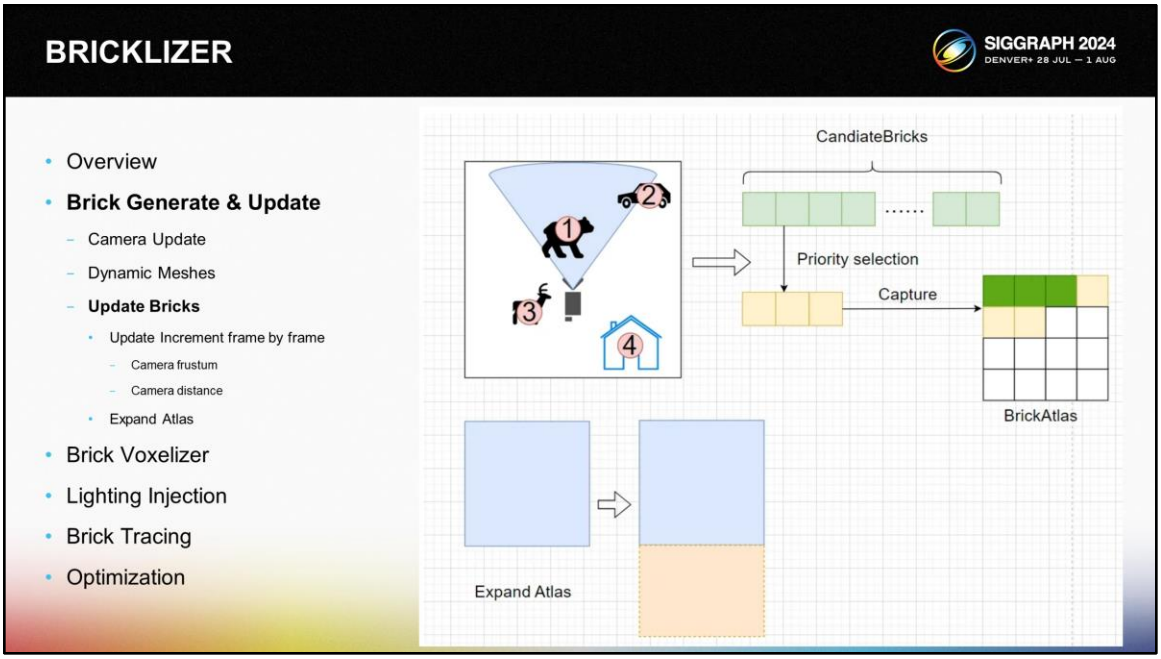
BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Overview
- **Brick Generate & Update**
  - **Camera Update**
  - **Dynamic Meshes**
  - Update Brick Candicates
- Brick Voxelizer
- Lighting Injection
- Brick Tracing
- Optimization

Besides full updates caused by entering the scene or BrickReset, only two changes can lead to Brick updates: one is the addition of Bricks caused by camera movement, and the other is updates caused by Mesh changes, including Mesh deletion, addition, and movement. Full updates and camera movement updates can be attributed to updates based on bounding boxes, requiring normalization of the bounding boxes to ensure that their boundaries align with Brick boundaries. Then, for each BrickPos inside, use BricksTable to determine whether the corresponding Brick exists. If not, add it to the BrickPosSet.

Next, obtain the updated bounding box of the Mesh, expand it to align its boundaries with Brick boundaries, and then iterate through BrickPos. If it already exists, it needs to be cleared and regenerated. If it does not exist, add it to the BrickPosSet for subsequent processing. All Bricks waiting for updates are added to the BrickPosSet, and then multithreading is used to initially cull Meshes, retaining only those intersecting with the updated bounding box. Then, multithreading is used to traverse all Bricks and Primitives. After traversal, each Brick will have its own Mesh list, and Bricks that do not intersect with Meshes can be deleted. The rest are added to CandiateBricks. In the above steps, Bricks that need to be deleted in this frame will also be added to DirtyBricks for processing.

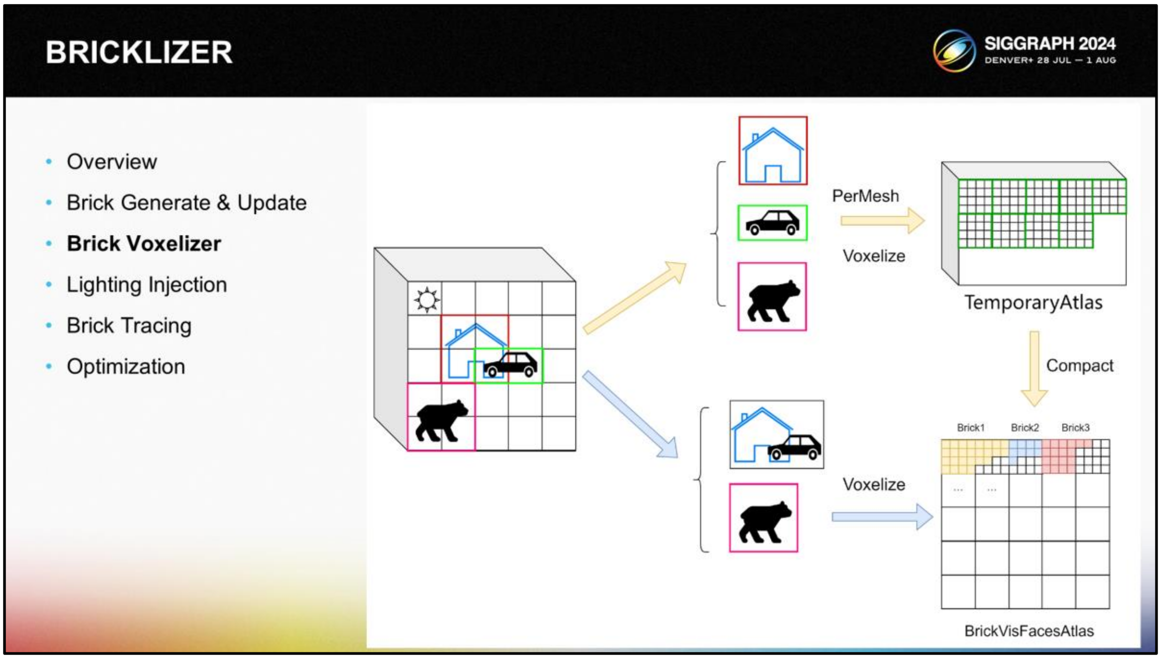


BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Overview
- **Brick Generate & Update**
  - Camera Update
  - Dynamic Meshes
  - **Update Bricks**
    - Update Increment frame by frame
      - Camera frustum
      - Camera distance
    - Expand Atlas
- Brick Voxelizer
- Lighting Injection
- Brick Tracing
- Optimization

After updating the Bricks, it is necessary to select k Bricks from the CandiateBricks for updating in the current frame. The selection strategy can rely on factors such as the distance from the camera or the distance within the view frustum. As shown in the figure below, priority should be given to updating the bricks within the view frustum, followed by bricks closer to the camera. After the selection, it is essential to ensure that there are remaining Bricks and Pages in both Allocators. If there are not enough, it will trigger the recycling process or memory expansion logic. The recycling process will reclaim the space of all Bricks outside the update range. Memory expansion will double the size of BrickMappingAtlas or BrickVisFacesAtlas, and an additional pass will be required to move the original Atlas data to the corresponding locations.

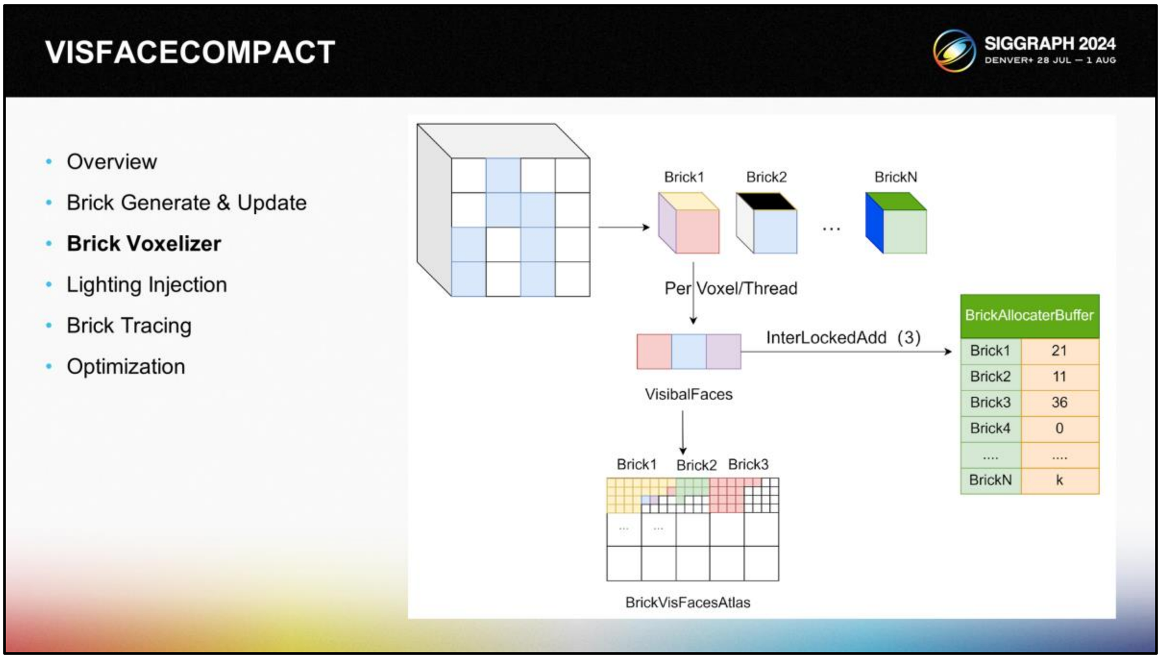


BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Overview
- Brick Generate & Update
- **Brick Voxelizer**
- Lighting Injection
- Brick Tracing
- Optimization

The voxelization process currently has significant optimization potential and is also the most time-consuming part. In the current strategy, each brick generates a corresponding MeshDrawCommand based on its Mesh list. Each Brick then undergoes voxelization in three directions, and the results are temporarily written into a temporary 3D texture generated in the current frame for subsequent processing. When voxelizing, it is advisable to combine MultiView and voxelize in three directions simultaneously to reduce DrawCalls and other operations.



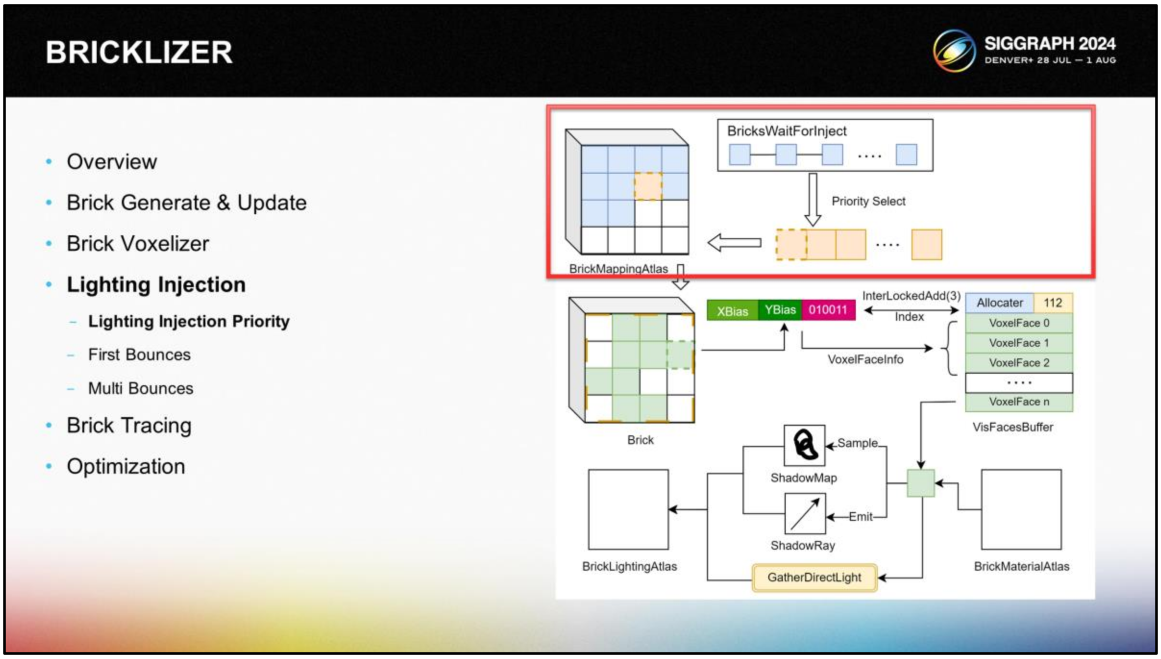
VISFACECOMPACT

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Overview
- Brick Generate & Update
- **Brick Voxelizer**
- Lighting Injection
- Brick Tracing
- Optimization

BrickAllocaterBuffer	
Brick1	21
Brick2	11
Brick3	36
Brick4	0
....	....
BrickN	k

Each brick is pre-allocated with 6 to 8 pages, and each page occupies 8x8 pixels. The compression and allocation of all valid faces for voxels within a brick are handled here. The specific bias for each voxel's visface is tightly arranged within these pages, ensuring that only the last page may have unused space. If the 8 pages are insufficient, we will feedback to the CPU to request additional pages. The purpose of using large-grained pages for allocation is to facilitate efficient page recycling.

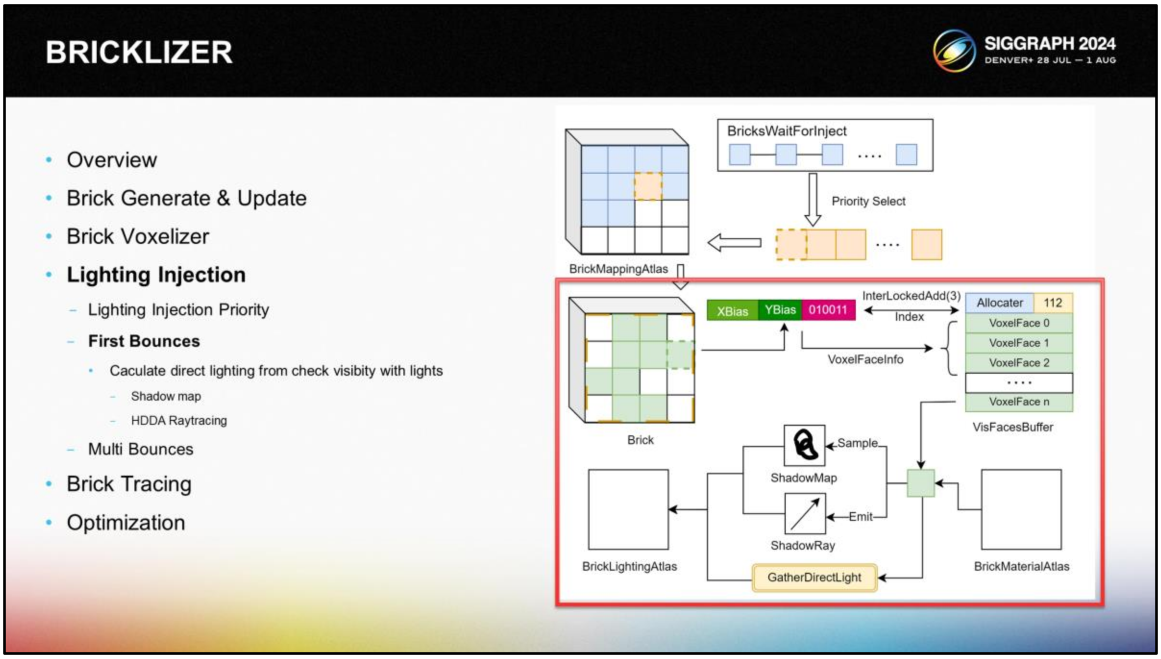


BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Overview
- Brick Generate & Update
- Brick Voxelizer
- **Lighting Injection**
  - **Lighting Injection Priority**
  - First Bounces
  - Multi Bounces
- Brick Tracing
- Optimization

Lighting calculations can be divided into two parts for processing: direct lighting and indirect lighting.
Before calculating direct lighting for each frame, k Bricks are selected from a candidate list to undergo direct lighting computations. The selection can be based on sorting factors such as their position within the view frustum or their distance from the camera.



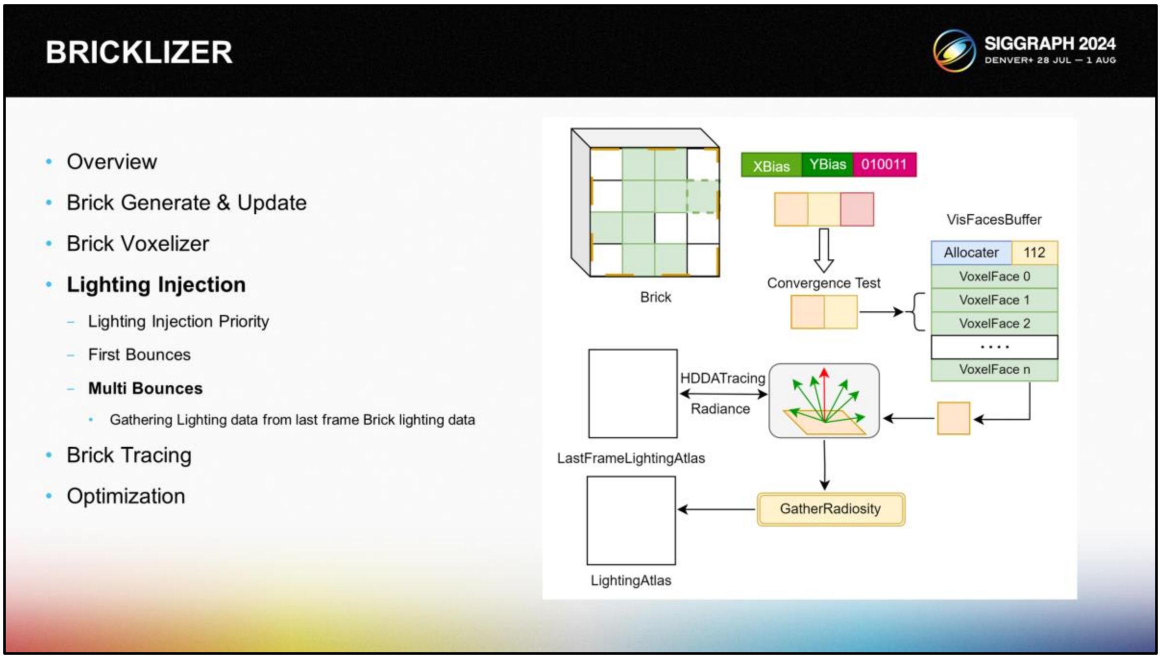
BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Overview
- Brick Generate & Update
- Brick Voxelizer
- **Lighting Injection**
  - Lighting Injection Priority
  - **First Bounces**
    - Caculate direct lighting from check visibity with lights
      - Shadow map
      - HDDA Raytracing
  - Multi Bounces
- Brick Tracing
- Optimization

Prior to the direct lighting calculation, to maximize hardware utilization, all valid faces within a brick are compacted into a VisBuffer. As illustrated in the diagram below, the upper half depicts the compaction logic. Since the BrickMappingAtlas already stores the storage information for each face corresponding to a voxel, this value can be directly retrieved. If a voxel exists and has three valid faces, three consecutive spaces are requested from the Allocator, and the index values of these valid faces in the VisFacesAtlas are written into the buffer.

Based on the Allocator, the number of thread groups can be determined, with each thread handling one valid VoxelFace. The basic material properties are fetched from the BrickAtlas. If the ShadowMap is valid, a direct sample can be taken; otherwise, a separate ShadowRay needs to be cast to determine if the point is in shadow. Subsequently, direct lighting is calculated based on factors such as normal weights, light source type, and material information. Finally, the results are written into the LightingAtlas.

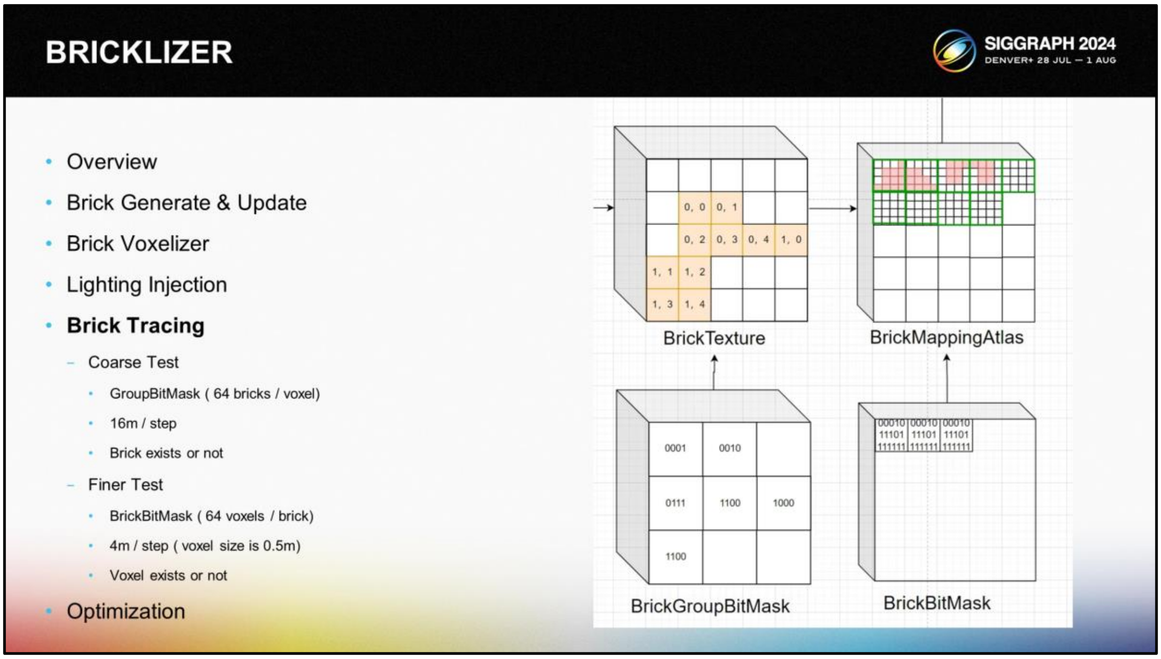


BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Overview
- Brick Generate & Update
- Brick Voxelizer
- **Lighting Injection**
  - Lighting Injection Priority
  - First Bounces
  - **Multi Bounces**
    - Gathering Lighting data from last frame Brick lighting data
- Brick Tracing
- Optimization

Multi Bounces:
The process for indirect lighting calculations is similar to direct lighting. First, k Bricks requiring radiance updates in the current frame are identified. Then, the indices of the effective faces of these Bricks are compacted into a Buffer to facilitate subsequent GPU thread group and thread allocation. For each effective face, n rays are emitted in hemispherical directions, and the collected results are weighted and averaged to calculate the Irradiance and store it.



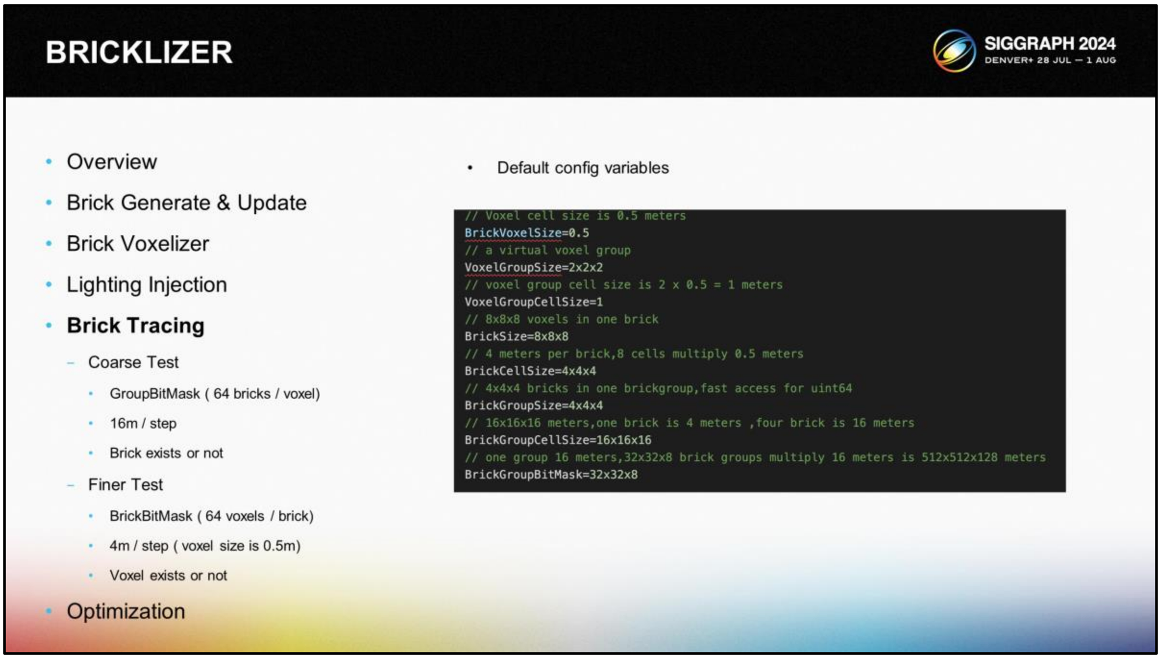
BRICKLIZER

- Overview
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- **Brick Tracing**
  - Coarse Test
    - GroupBitMask (64 bricks / voxel)
    - 16m / step
    - Brick exists or not
  - Finer Test
    - BrickBitMask (64 voxels / brick)
    - 4m / step (voxel size is 0.5m)
    - Voxel exists or not
- Optimization

Utilizing BrickGroupBitMask and BrickBitMask, the HDDA algorithm can be implemented, enabling fast raycasting and intersection detection. The specific processing logic is as follows:

Starting Point Offset: When a ray originates from a starting point, the starting point needs to be translated outward along the ray direction to the surface of a voxel for tracing, avoiding self-intersection.

BrickGroupTracing: Based on the starting point's position, the corresponding BrickGroup is located. If the Group exists, the corresponding 64-bit BitMask is retrieved.



BRICKLIZER

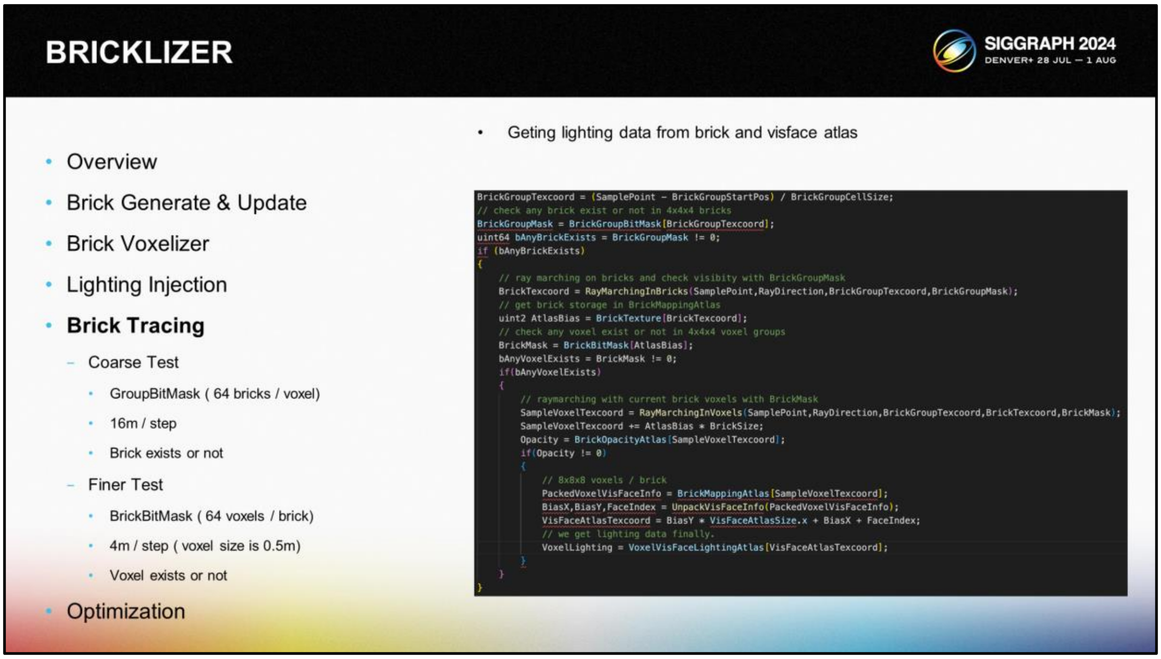
SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Overview
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- **Brick Tracing**
  - Coarse Test
    - GroupBitMask (64 bricks / voxel)
    - 16m / step
    - Brick exists or not
  - Finer Test
    - BrickBitMask (64 voxels / brick)
    - 4m / step (voxel size is 0.5m)
    - Voxel exists or not
- Optimization

- Default config variables

```
// Voxel cell size is 0.5 meters
BrickVoxelSize=0.5
// a virtual voxel group
VoxelGroupSize=2x2x2
// voxel group cell size is 2 x 0.5 = 1 meters
VoxelGroupCellSize=1
// 8x8x8 voxels in one brick
BrickSize=8x8x8
// 4 meters per brick,8 cells multiply 0.5 meters
BrickCellSize=4x4x4
// 4x4x4 bricks in one brickgroup,fast access for uint64
BrickGroupSize=4x4x4
// 16x16x16 meters,one brick is 4 meters ,four brick is 16 meters
BrickGroupCellSize=16x16x16
// one group 16 meters,32x32x8 brick groups multiply 16 meters is 512x512x128 meters
BrickGroupBitMask=32x32x8
```

Here are the default config variables.The default voxelsize is 0.5 meters,there are 8x8x8 voxels in one brick,thus one brick covers the range of 4 meters.There are 4x4x4 bricks in one brickgroup,so the brickgroup covers the range of 16 meters.We have the number of 32x32x8 brickgroups ,so the we can cover 512x512x128 meters.



BRICKLIZER

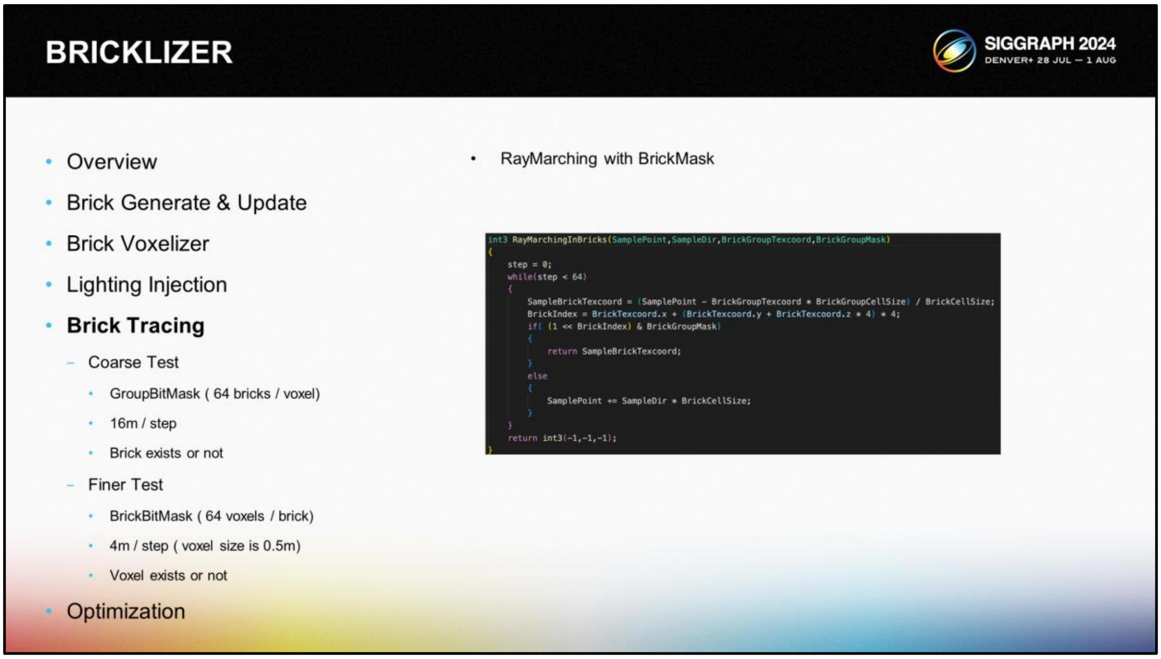
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Overview
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- **Brick Tracing**
  - Coarse Test
    - GroupBitMask (64 bricks / voxel)
    - 16m / step
    - Brick exists or not
  - Finer Test
    - BrickBitMask (64 voxels / brick)
    - 4m / step (voxel size is 0.5m)
    - Voxel exists or not
- Optimization

- Geting lighting data from brick and visface atlas

```
BrickGroupTexcoord = (SamplePoint - BrickGroupStartPos) / BrickGroupCellSize;
// check any brick exist or not in 4x4x4 bricks
BrickGroupMask = BrickGroupBitMask[BrickGroupTexcoord];
uint64 bAnyBrickExists = BrickGroupMask != 0;
if (bAnyBrickExists)
{
    // ray marching on bricks and check visibity with BrickGroupMask
    BrickTexcoord = RayMarchingInBricks(SamplePoint,RayDirection,BrickGroupTexcoord,BrickGroupMask);
    // get brick storage in BrickMappingAtlas
    uint2 AtlasBias = BrickTexture[BrickTexcoord];
    // check any voxel exist or not in 4x4x4 voxel groups
    BrickMask = BrickBitMask[AtlasBias];
    bAnyVoxelExists = BrickMask != 0;
    if(bAnyVoxelExists)
    {
        // raymarching with current brick voxels with BrickMask
        SampleVoxelTexcoord = RayMarchingInVoxels(SamplePoint,RayDirection,BrickGroupTexcoord,BrickTexcoord,BrickMask);
        SampleVoxelTexcoord += AtlasBias * BrickSize;
        Opacity = BrickOpacityAtlas[SampleVoxelTexcoord];
        if(Opacity != 0)
        {
            // 8x8x8 voxels / brick
            PackedVoxelVisFaceInfo = BrickMappingAtlas[SampleVoxelTexcoord];
            BiasX,BiasY,FaceIndex = UnpackVisFaceInfo(PackedVoxelVisFaceInfo);
            VisFaceAtlasTexcoord = BiasY * VisFaceAtlasSize.x + BiasX + FaceIndex;
            // we get lighting data finally.
            VoxelLighting = VoxelVisFaceLightingAtlas[VisFaceAtlasTexcoord];
        }
    }
}
```

Here is the Pseudocode how we get lighting data from brick and visfacelightingatlas.First raymarching is in bricks levels with brickgroupbitmask.Then the voxels level raymarching.Each raymarching process use 64bits to check 64 bricks or voxels is presence or not.



BRICKLIZER

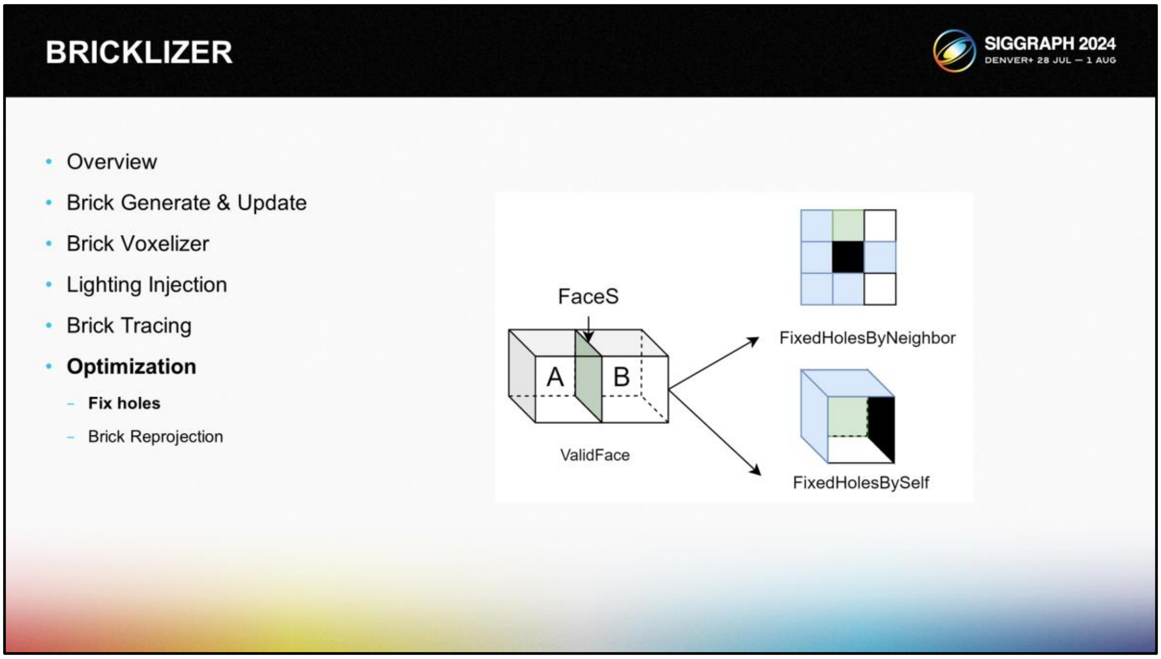
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Overview
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- **Brick Tracing**
  - Coarse Test
    - GroupBitMask (64 bricks / voxel)
    - 16m / step
    - Brick exists or not
  - Finer Test
    - BrickBitMask (64 voxels / brick)
    - 4m / step (voxel size is 0.5m)
    - Voxel exists or not
- Optimization

- RayMarching with BrickMask

```
int3 RayMarchingInBricks(SamplePoint,SampleDir,BrickGroupTexcoord,BrickGroupMask)
{
    step = 0;
    while(step < 64)
    {
        SampleBrickTexcoord = (SamplePoint - BrickGroupTexcoord * BrickGroupCellSize) / BrickCellSize;
        BrickIndex = BrickTexcoord.x + (BrickTexcoord.y + BrickTexcoord.z * 4) * 4;
        if( (1 << BrickIndex) & BrickGroupMask)
        {
            return SampleBrickTexcoord;
        }
        else
        {
            SamplePoint += SampleDir * BrickCellSize;
        }
    }
    return int3(-1,-1,-1);
}
```

the Pseudocode how we get valid brick from brickmask.and raymarching on voxel level is similar.



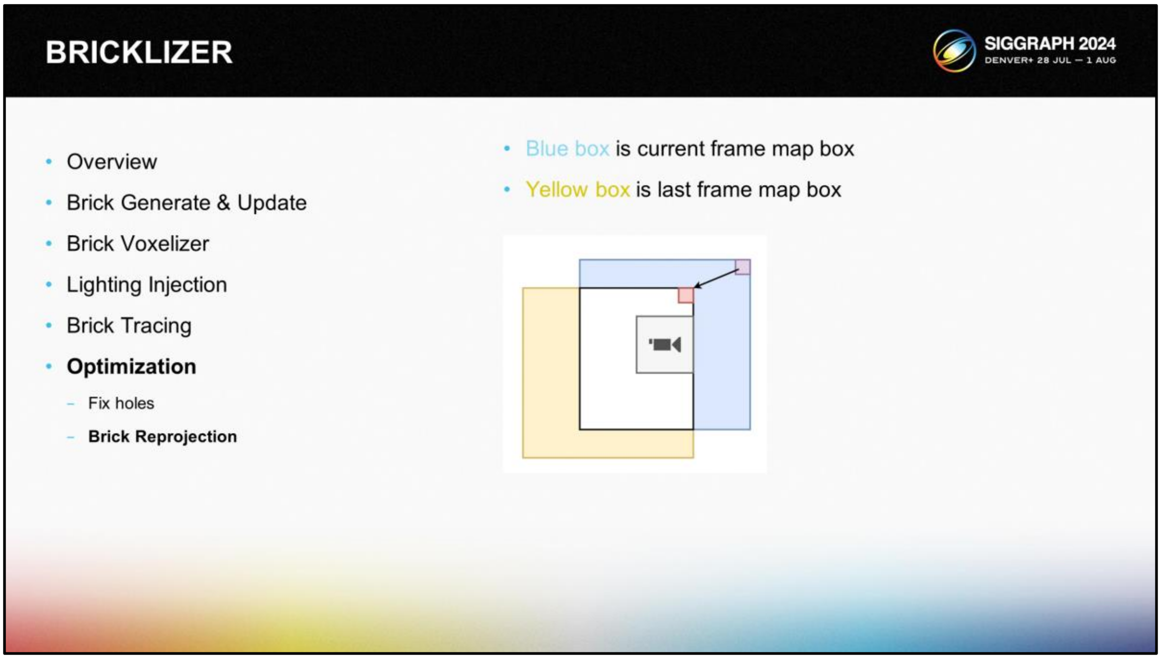
BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Overview
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- Brick Tracing
- **Optimization**
  - **Fix holes**
  - Brick Reprojection

Firstly, it is necessary to identify which faces need to be repaired. To briefly explain the concept of effective faces: simply put, only observable faces are considered effective. As shown in the figure below, FaceS is the overlapping face between Voxel A and Voxel B. If B is an empty voxel or a semi-transparent voxel (i.e., Opacity < 1), FaceS can be considered visible and is thus an effective face. For each non-empty voxel, if there are effective faces and some of these faces are empty, they are added to a buffer waiting for subsequent repair processing.

Specifically, there are two options for repair. Repairs will prioritize searching and patching within a 3x3 grid of faces based on the plane the face resides in, retrieving its Albedo and Normal for filling. Another simpler and more direct option is to directly take the weighted average of the Albedo of other faces belonging to the same voxel as this face's Albedo, and the Normal can be obtained by rotating the Normals of other faces (or simply selecting the orientation of the voxel face as the normal direction—the effect is basically correct and without light leakage).



BRICKLIZER

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Overview
- Brick Generate & Update
- Brick Voxelizer
- Lighting Injection
- Brick Tracing
- **Optimization**
  - Fix holes
  - **Brick Reprojection**

- Blue box is current frame map box
- Yellow box is last frame map box

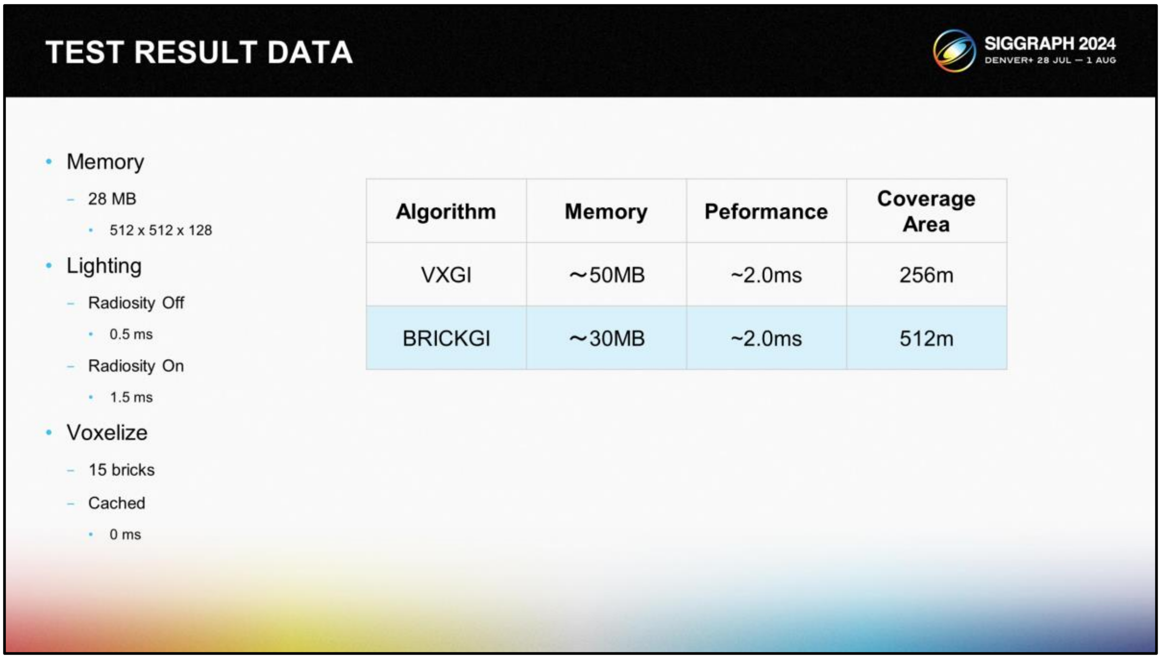
During camera movement, if it is detected in a frame that the updated bounding box exceeds the range of the mapped bounding box, the corresponding reprojection logic needs to be executed. On the CPU, all Bricks are traversed, and those that exceed the new mapped bounding box are deleted. At the same time, basic parameters such as AtlasOrigin need to be updated. On the GPU, only the corresponding BrickTexture and HDDAStruct need to be updated. An offset is calculated for the remaining Bricks and passed as a parameter to the ReprojectionPass, shifting the corresponding Bricks accordingly.

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

RESULTS

HOW RESULTS TURNED OUT

I am done with the detailed introduction of our algorithm, now let's take a look at how the test result in the real world may look like.



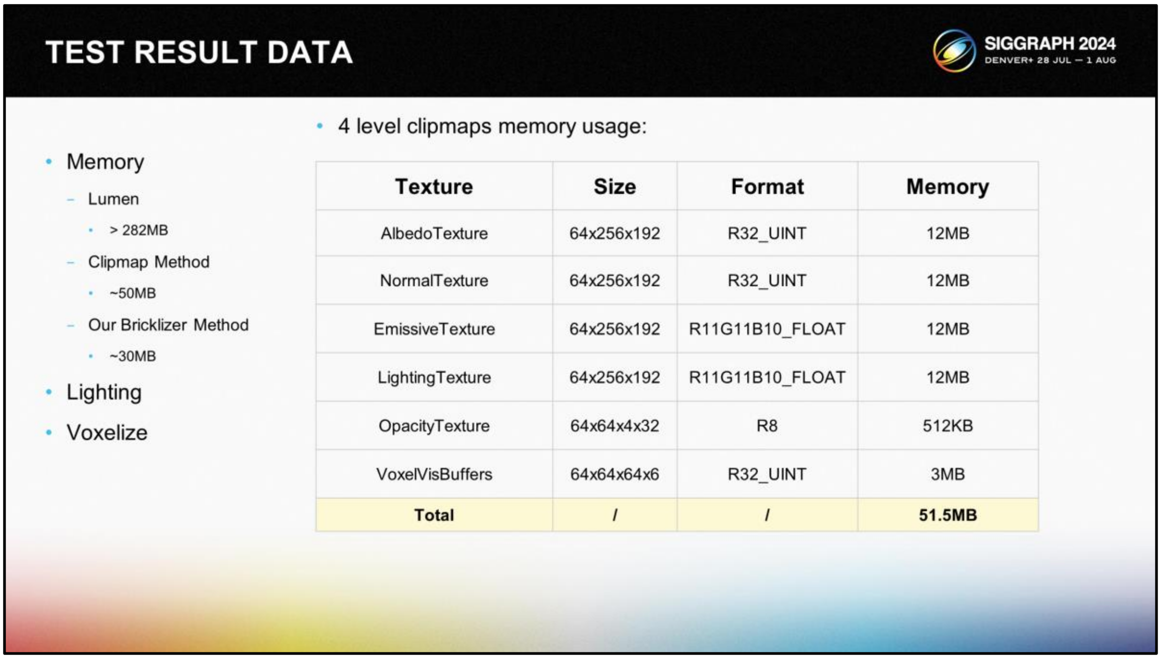
TEST RESULT DATA

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Memory
  - 28 MB
    - 512 x 512 x 128
- Lighting
  - Radiosity Off
    - 0.5 ms
  - Radiosity On
    - 1.5 ms
- Voxelize
  - 15 bricks
  - Cached
    - 0 ms

Algorithm	Memory	Peformance	Coverage Area
VXGI	～50MB	~2.0ms	256m
BRICKGI	～30MB	~2.0ms	512m

Here, we have used the 2023 flagship mobile devices to test VXGI and BrickGI. As you can see, our BrickGI only needs less than 30MB of memory for a spherical volume with a radius of 512 meters. In contrast, the traditional clipmap approach take more than 50MB. Furthermore, in terms of performance, we can complete all the GI calculations within 2ms.



TEST RESULT DATA

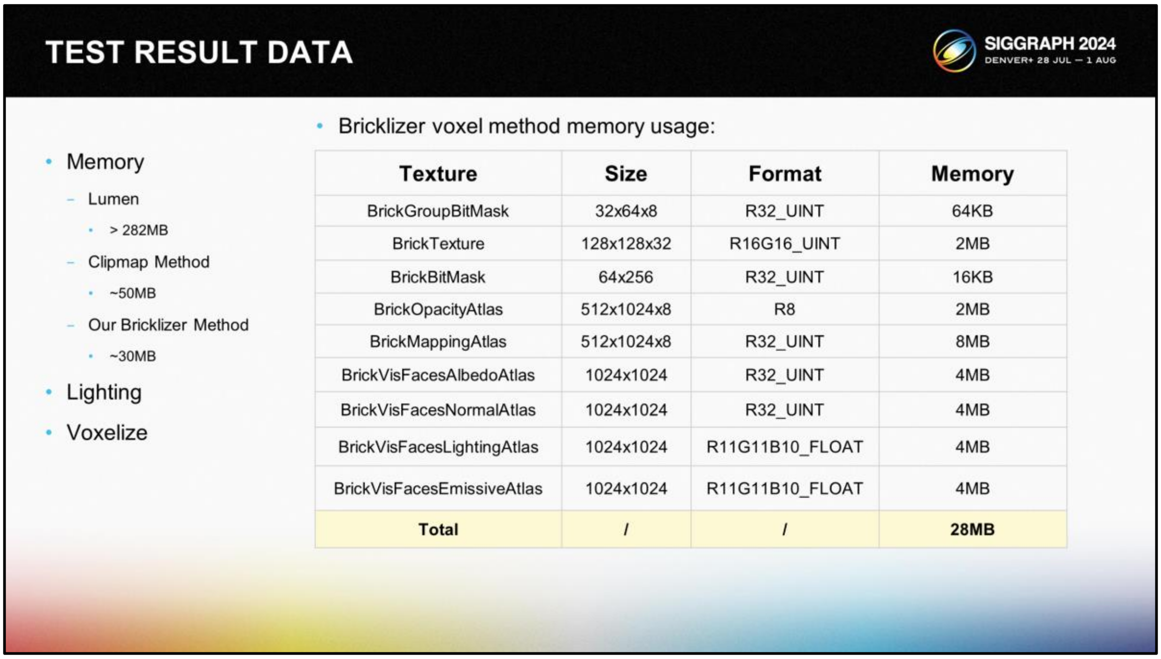
SIGGRAPH 2024
DENVER+ 28 JUL – 1 AUG

- Memory
  - Lumen
    - > 282MB
  - Clipmap Method
    - ~50MB
  - Our Bricklizer Method
    - ~30MB
- Lighting
- Voxelize

- 4 level clipmaps memory usage:

Texture	Size	Format	Memory
AlbedoTexture	64x256x192	R32_UINT	12MB
NormalTexture	64x256x192	R32_UINT	12MB
EmissiveTexture	64x256x192	R11G11B10_FLOAT	12MB
LightingTexture	64x256x192	R11G11B10_FLOAT	12MB
OpacityTexture	64x64x4x32	R8	512KB
VoxelVisBuffers	64x64x64x6	R32_UINT	3MB
Total	**/**	**/**	**51.5MB**

Here is the detailed memory usage of the voxelization method based on clipmap, which is approximately 50MB.



TEST RESULT DATA

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Memory
  - Lumen
    - > 282MB
  - Clipmap Method
    - ~50MB
  - Our Bricklizer Method
    - ~30MB
- Lighting
- Voxelize

- Bricklizer voxel method memory usage:

Texture	Size	Format	Memory
BrickGroupBitMask	32x64x8	R32_UINT	64KB
BrickTexture	128x128x32	R16G16_UINT	2MB
BrickBitMask	64x256	R32_UINT	16KB
BrickOpacityAtlas	512x1024x8	R8	2MB
BrickMappingAtlas	512x1024x8	R32_UINT	8MB
BrickVisFacesAlbedoAtlas	1024x1024	R32_UINT	4MB
BrickVisFacesNormalAtlas	1024x1024	R32_UINT	4MB
BrickVisFacesLightingAtlas	1024x1024	R11G11B10_FLOAT	4MB
BrickVisFacesEmissiveAtlas	1024x1024	R11G11B10_FLOAT	4MB
Total	**/**	**/**	**28MB**

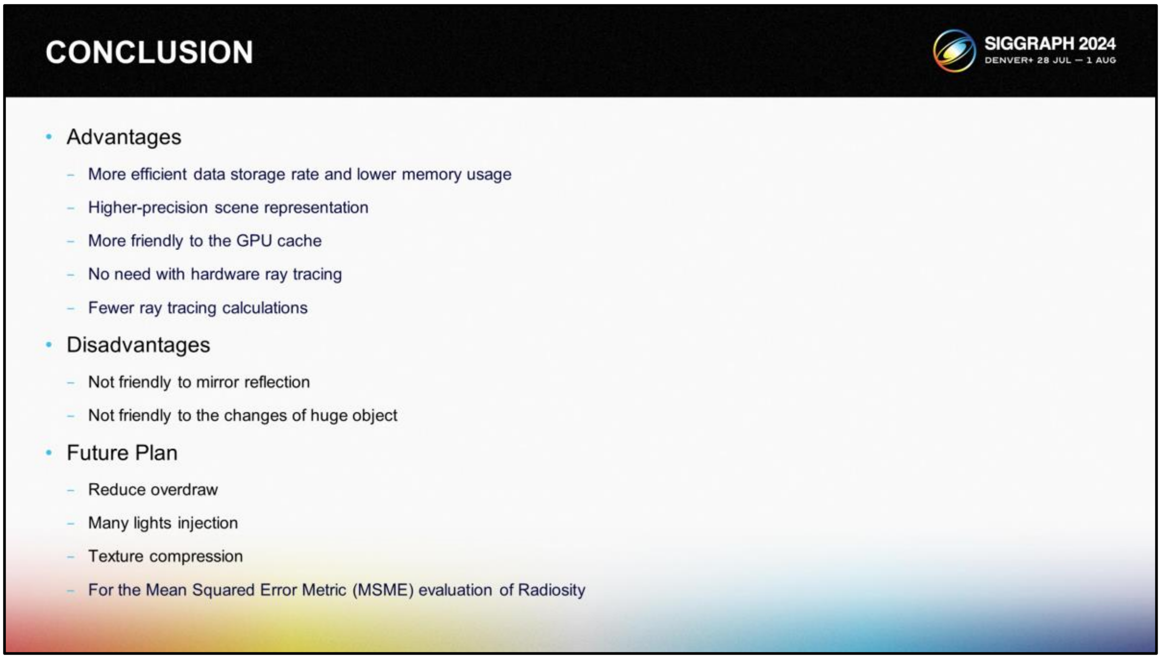
Here is the detailed memory usage of our voxelization method based on the latest bricklizer technology. It consumes approximately 30MB of memory, and the scene coverage is twice that of a 4-level clipmap, reaching up to 512 meters.

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

CONCLUSION

IS THIS THE END OF OUR TECH ?

Let‘s wrap up the advantages and possible future improvements of our algorithm.



CONCLUSION

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

- Advantages
  - More efficient data storage rate and lower memory usage
  - Higher-precision scene representation
  - More friendly to the GPU cache
  - No need with hardware ray tracing
  - Fewer ray tracing calculations
- Disadvantages
  - Not friendly to mirror reflection
  - Not friendly to the changes of huge object
- Future Plan
  - Reduce overdraw
  - Many lights injection
  - Texture compression
  - For the Mean Squared Error Metric (MSME) evaluation of Radiosity

The advantages of the system include a more efficient data storage rate that leads to lower memory usage, enabling a higher-precision representation of the scene. Furthermore, it is designed to be more compatible with the GPU cache, eliminating the need for hardware-based ray tracing and thus reducing the number of ray tracing calculations required.
However, there are also some disadvantages, such as its inability to handle mirror reflections effectively and difficulties in adapting to changes with huge objects.
Looking ahead, our future plan involves implementing strategies to reduce overdraw, incorporating multiple light injections, and compressing textures. These enhancements will be particularly useful when evaluating Radiosity using the Mean Squared Error Metric (MSME), aiming to improve both performance and accuracy.

SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

THANKS

SHUNCAO@TENCENT.COM

Thank you for your time. Feel free to reach out to me if you have any further questions.