
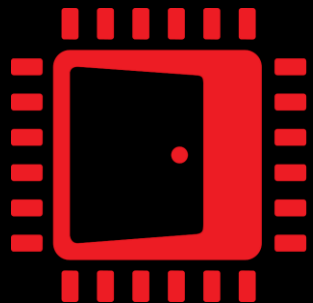


AMD 
EPYC

AMD 
RYZEN

AMD 
RADEON



AMD 
GPUOpen

REAL-TIME SPARSE DISTANCE FIELDS FOR GAMES

LOU KRAMER

AMD 
together we advance_

ACKNOWLEDGMENTS

Anton Schreiner

- Lead R&D Programmer of the presented project
- Member of the AMD Core Technology Group Game Engineering team

Guillaume Boissé – R&D

Dihara Wijetunga – Programmer

Jay Fraser – Programmer

Fabio Camaiora - Programmer

Lou Kramer

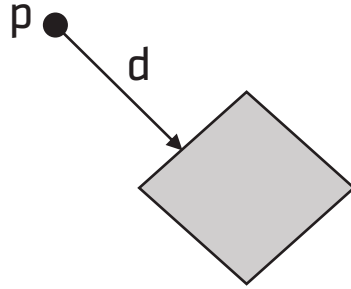
- Developer Technology Engineer
- Integration of the project into games

AGENDA

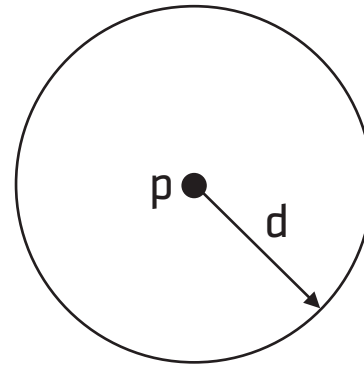
- Sparse Distance Fields recap
- Brixelizer: Real-time Sparse Distance Fields for Games
 - Algorithm
 - Optimizations
- Integration of Brixelizer into game engines
- Brixelizer Global Illumination

DISTANCE FIELDS RECAP

- A distance field denotes the distance for every point in space to its closest surface



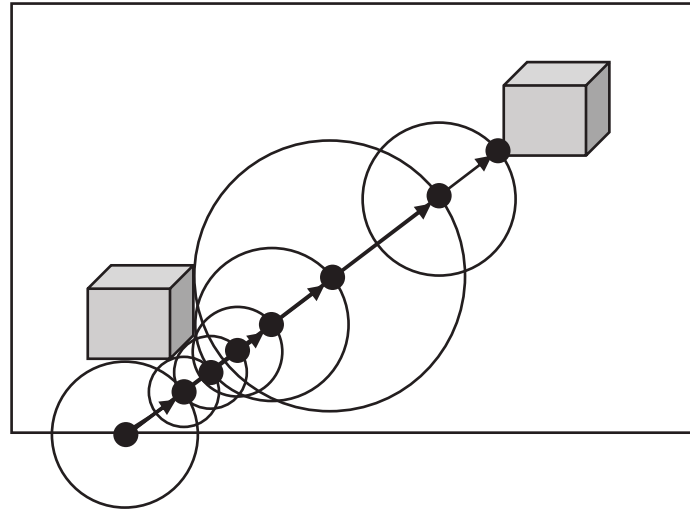
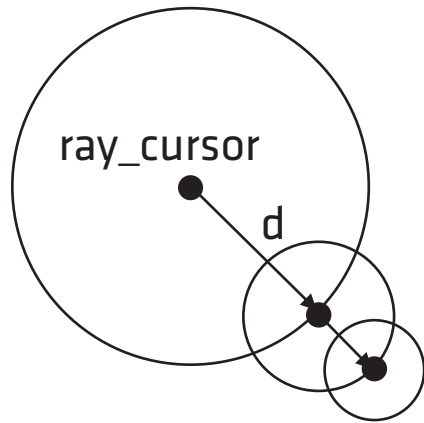
- $d = \text{sdf}(p)$
- The space around p with a distance $< d$ is empty



- We don't know at which direction the closest surface is as we only get the distance

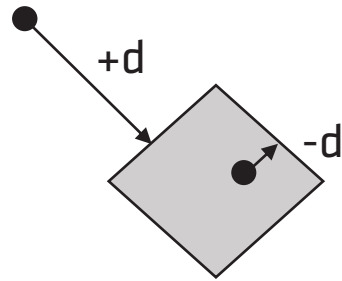
DISTANCE FIELDS RECAP

- We can use distance fields to find the hit point of a ray with the scene
- Usual distance field traversal is: $\text{ray_cursor} += \text{ray_direction} * \text{sdf}(\text{ray_cursor})$

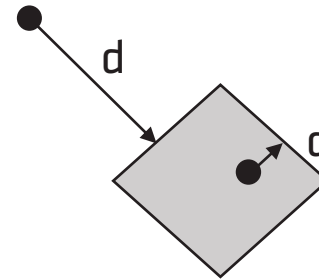


SIGNED & UNSIGNED DISTANCE FIELD

- In a signed distance field, the distance is positive if the point in space is outside of a mesh
- Respectively, the distance is negative if the point is inside a mesh



Signed

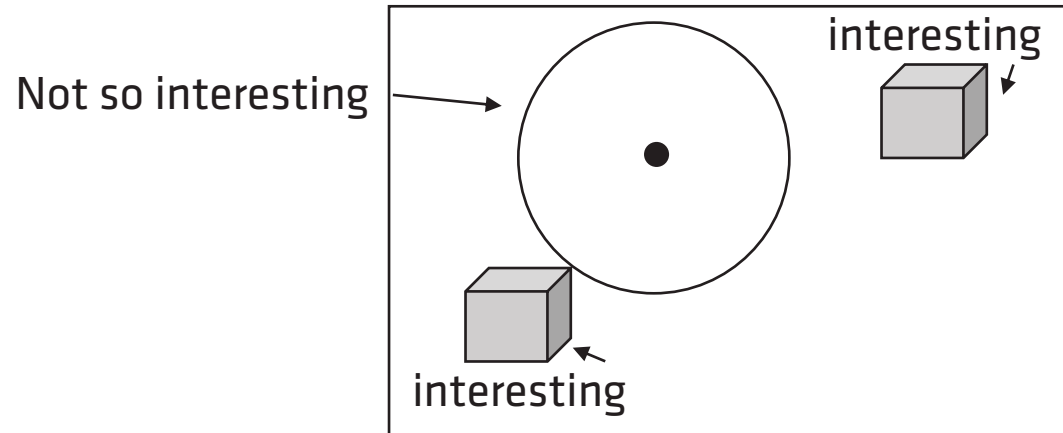


Unsigned

- In an unsigned distance field, we always have $\text{abs}(d)$, so both sides of a surface are treated equally
- We do not know if a point is inside or outside a mesh

DENSE & SPARSE DISTANCE FIELDS

- Distance Fields are a dense scene representation
 - For every point in space we store the distance information
- This is a lot of data
- The majority of the space is actually empty, so the distance to the closest surface is very large
 - Neighbouring points in empty space don't contain much more information



- Can we 'group' empty space to save memory?
 - Sparse distance fields only have distance field information in areas close to surfaces
- Empty space is encoded differently

DISTANCE FIELDS RECAP

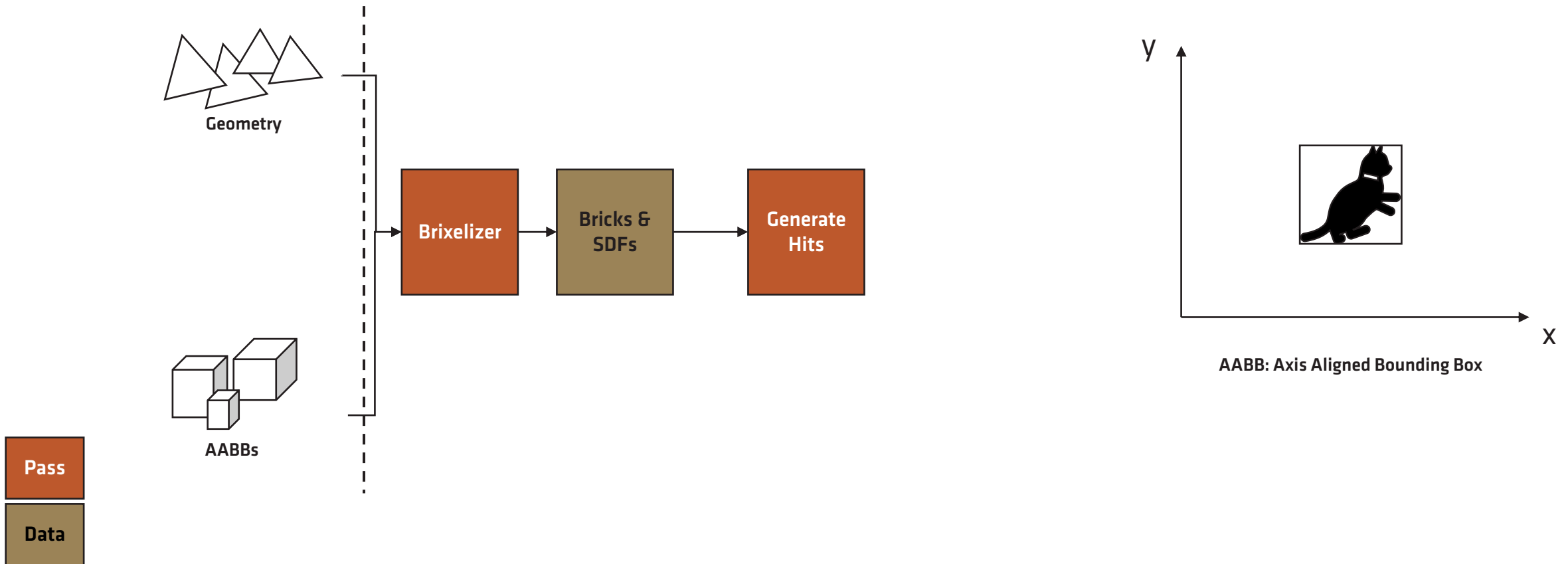
- Distance fields are either represented analytically or via a 3D texture
- An analytic representation can become very complex quickly
- Distance fields stored in 3D textures can represent any kind of meshes
- Traversing results in texture lookups

- What about the generation of distance fields?
- Current solutions are often pre-baked and thus, work on static geometry

In the following slides, distance refers to Euclidean distance

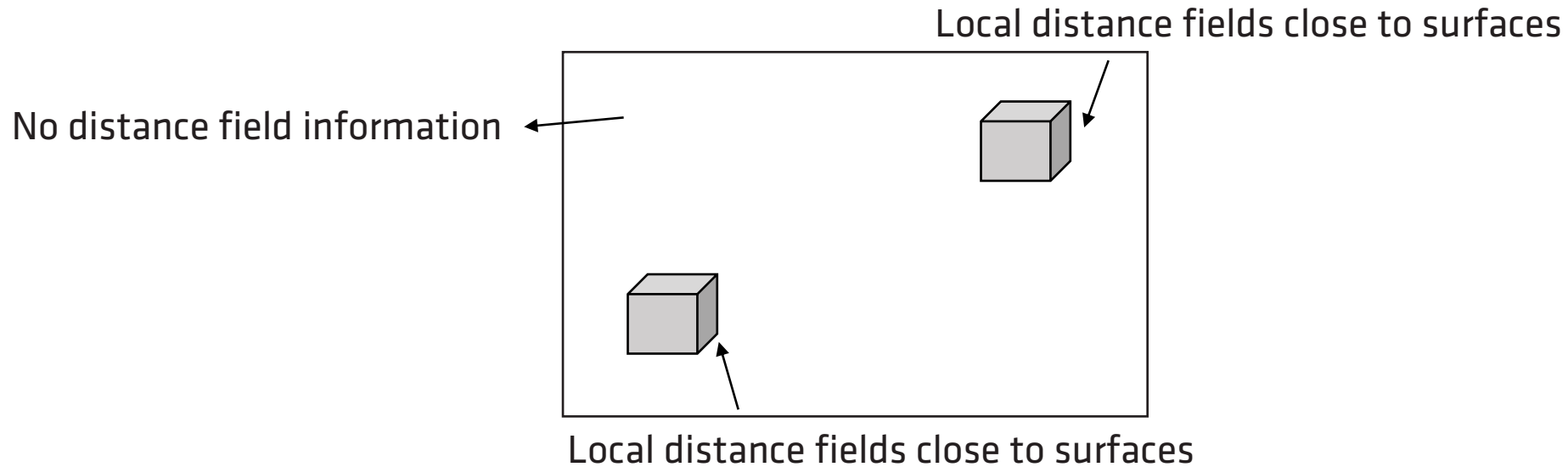
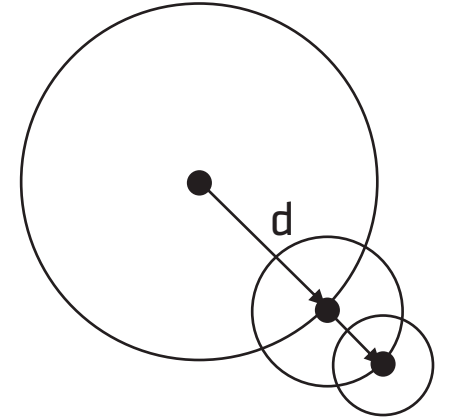
BRIXELIZER

- Brixelizer is a library that generates sparse distance fields in real-time for any given triangle-based geometry input



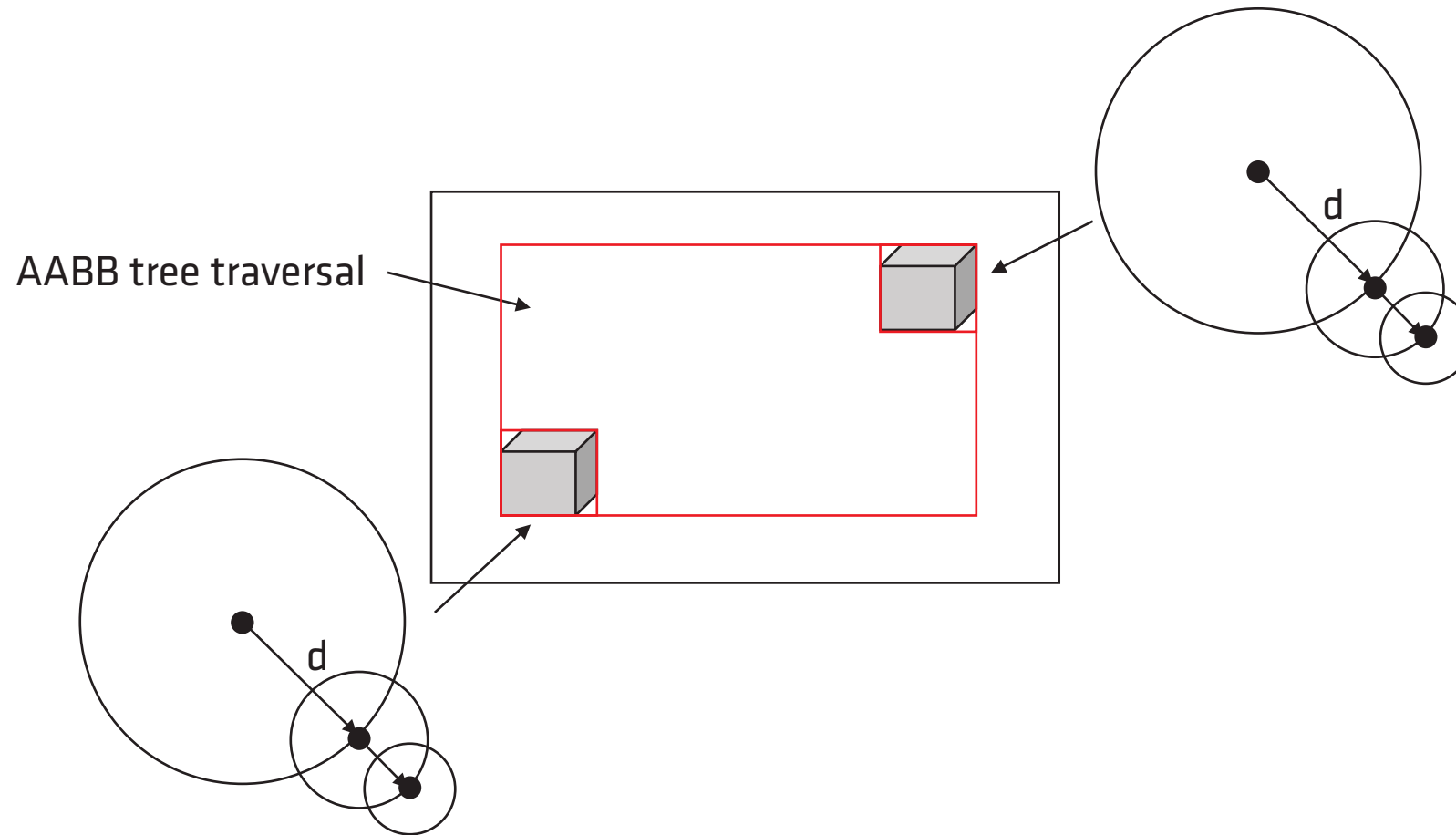
BRIXELIZER - OUTPUT

- The output of Brixelizer is a sparse distance field
- Generally, the output of Brixelizer is used to find the ray hit point with the scene



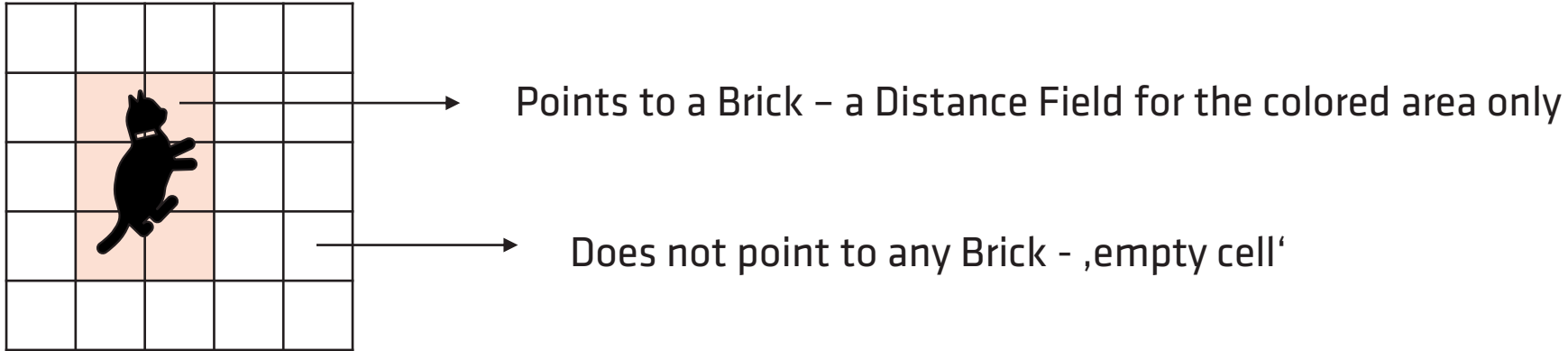
BRIXELIZER - OUTPUT

- The basic idea is to first traverse down an AABB tree
- The leaf nodes of the AABB tree point to the distance fields



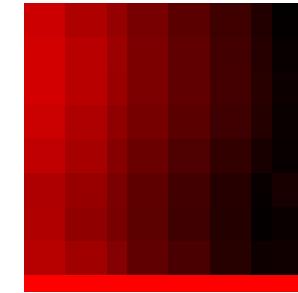
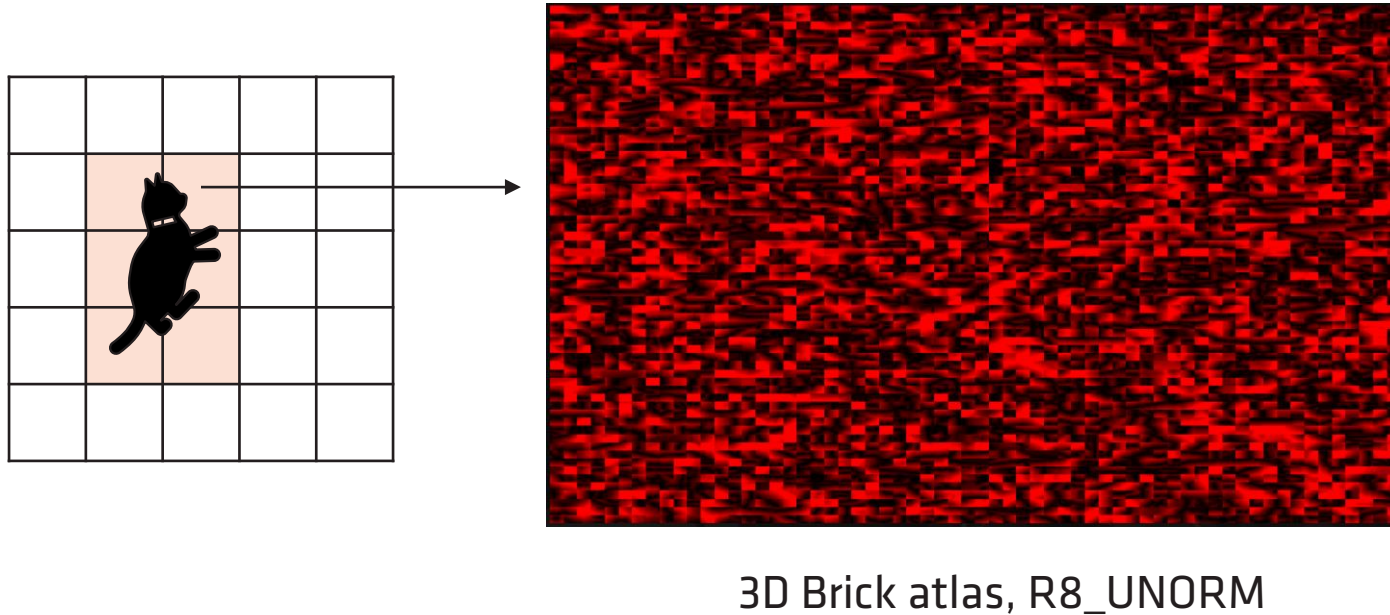
BRIXELIZER - OUTPUT

- The sparse distance field is represented in the form of a collection of local distance fields, called Bricks
- Bricks are generated close to surfaces

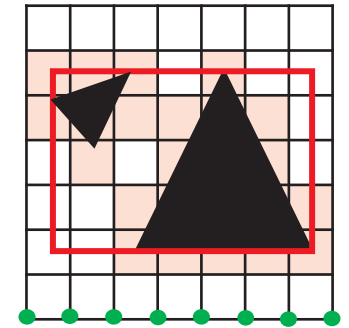


- Each grid cell that intersects with a surface stores a pointer to a 3D Brick atlas

BRIXELIZER - OUTPUT



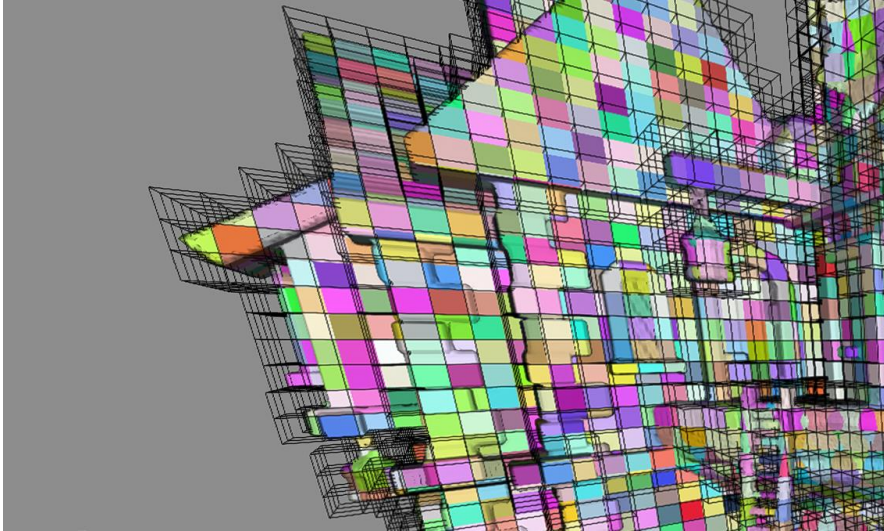
Brick



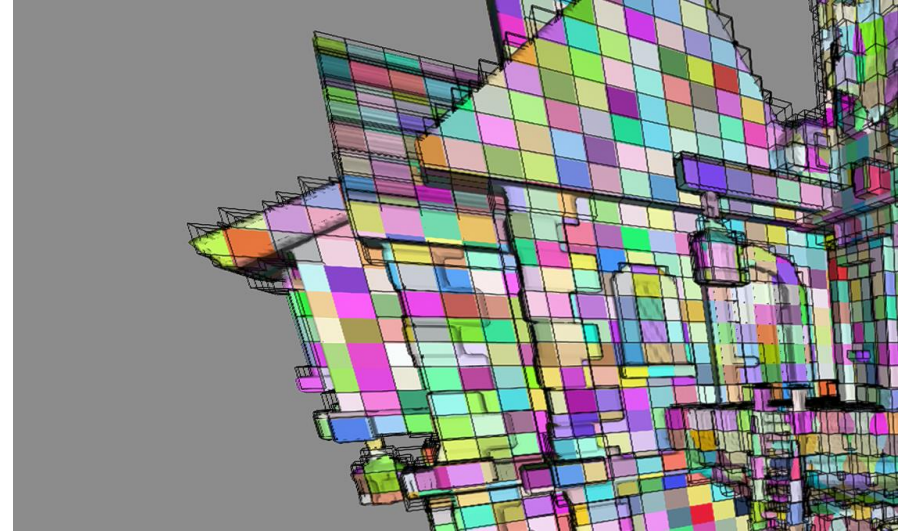
Brixel

- A Brick is of resolution 8x8x8 and contains the local distance field of a grid cell, that intersected with a surface
- Encodes 7x7x7 region

BRIXELIZER - BRICKS



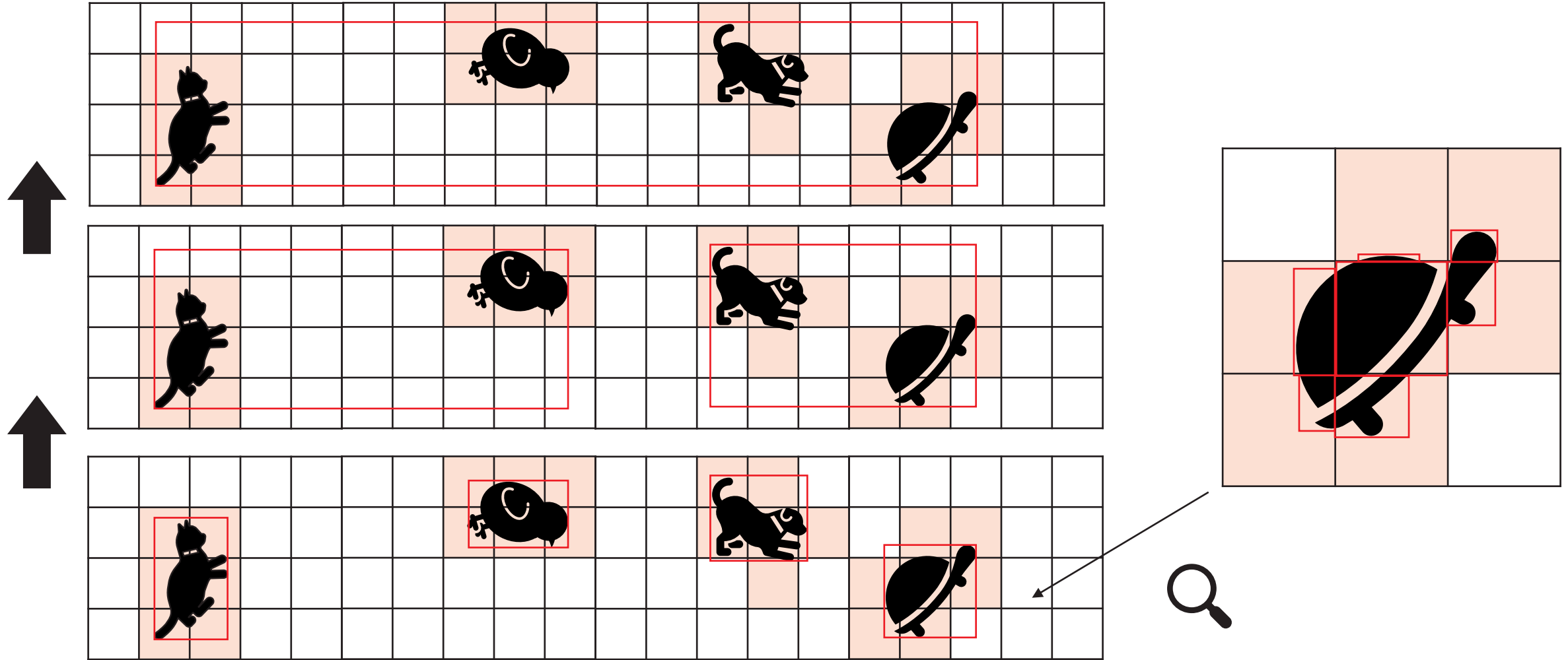
Voxels with Bricks



AABBs of the Bricks

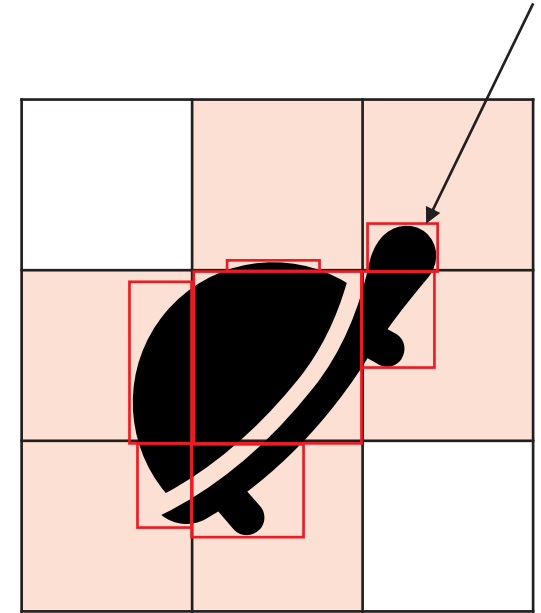
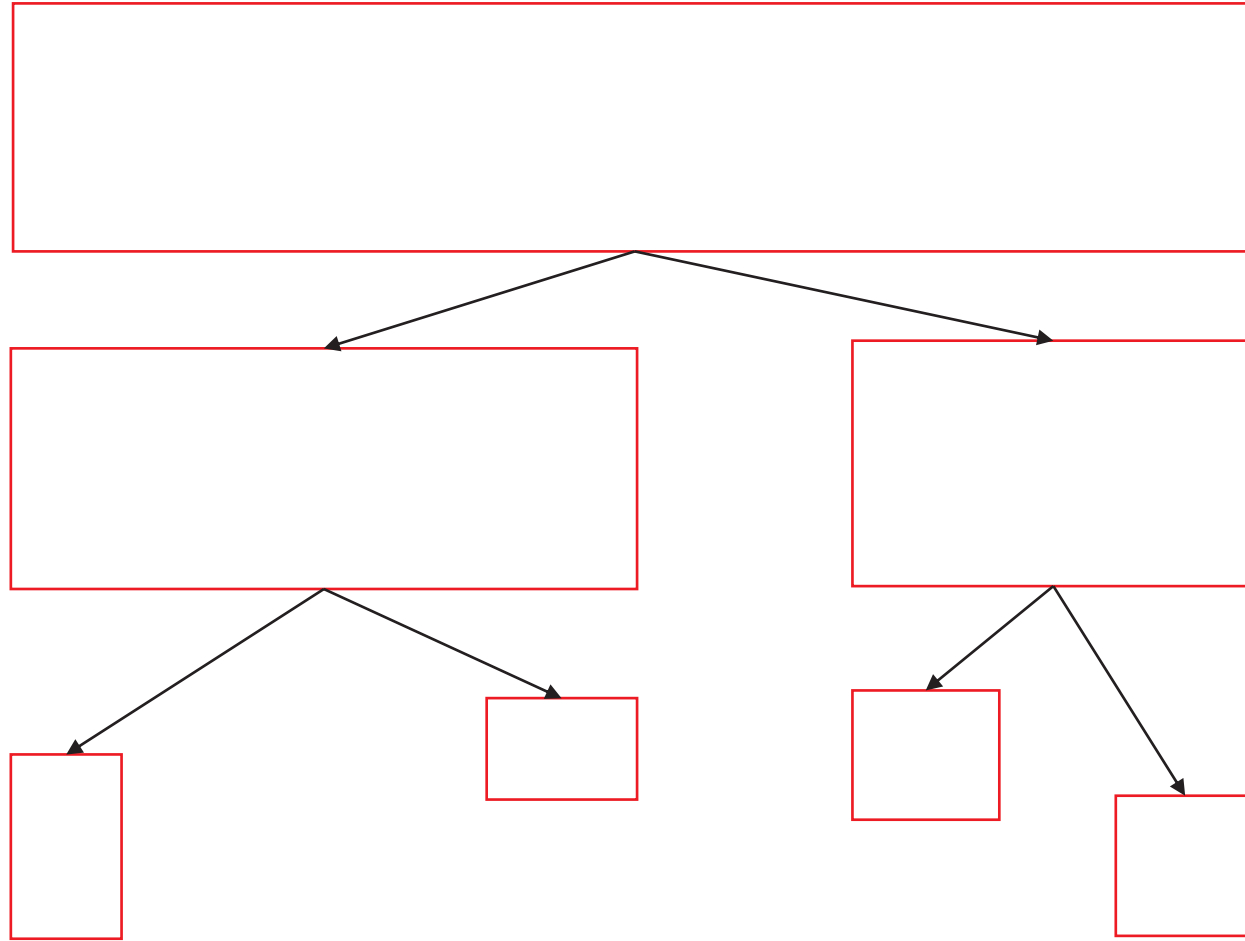
BRIXELIZER - OUTPUT

- Once the Bricks are generated, they are used to build bottom up a 3-level AABB tree



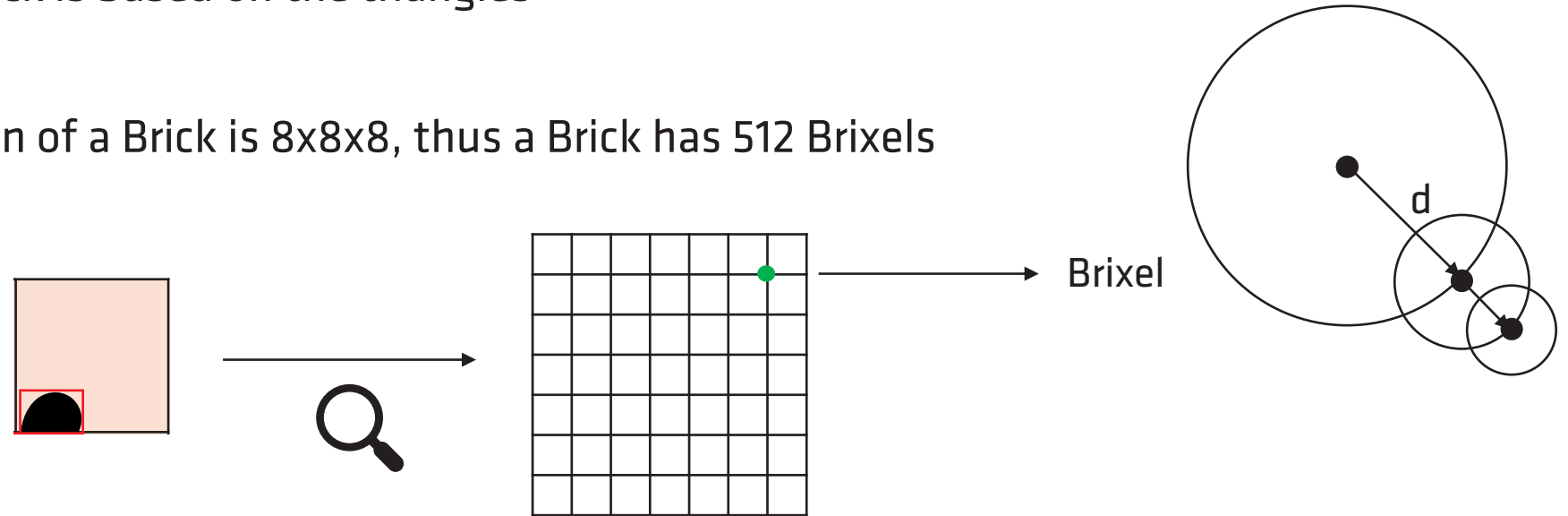
BRIXELIZER - OUTPUT

- The AABB tree is used for ray-scene traversal



BRIXELIZER - OUTPUT

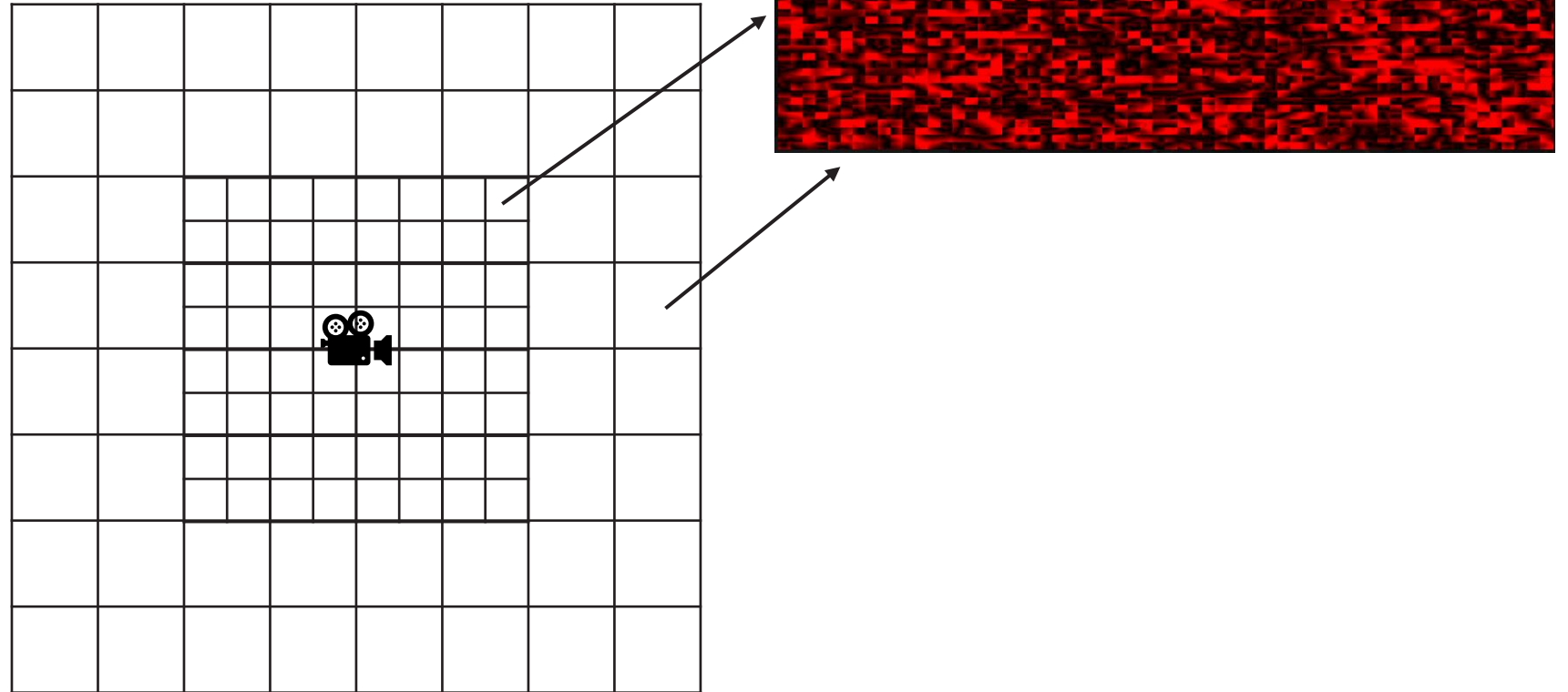
- Once we traversed down the tree to hit the AABB within a Brick, we evaluate its distance field
- The distance field of a Brick is based on the triangles
- Remember: The resolution of a Brick is 8x8x8, thus a Brick has 512 Brixels



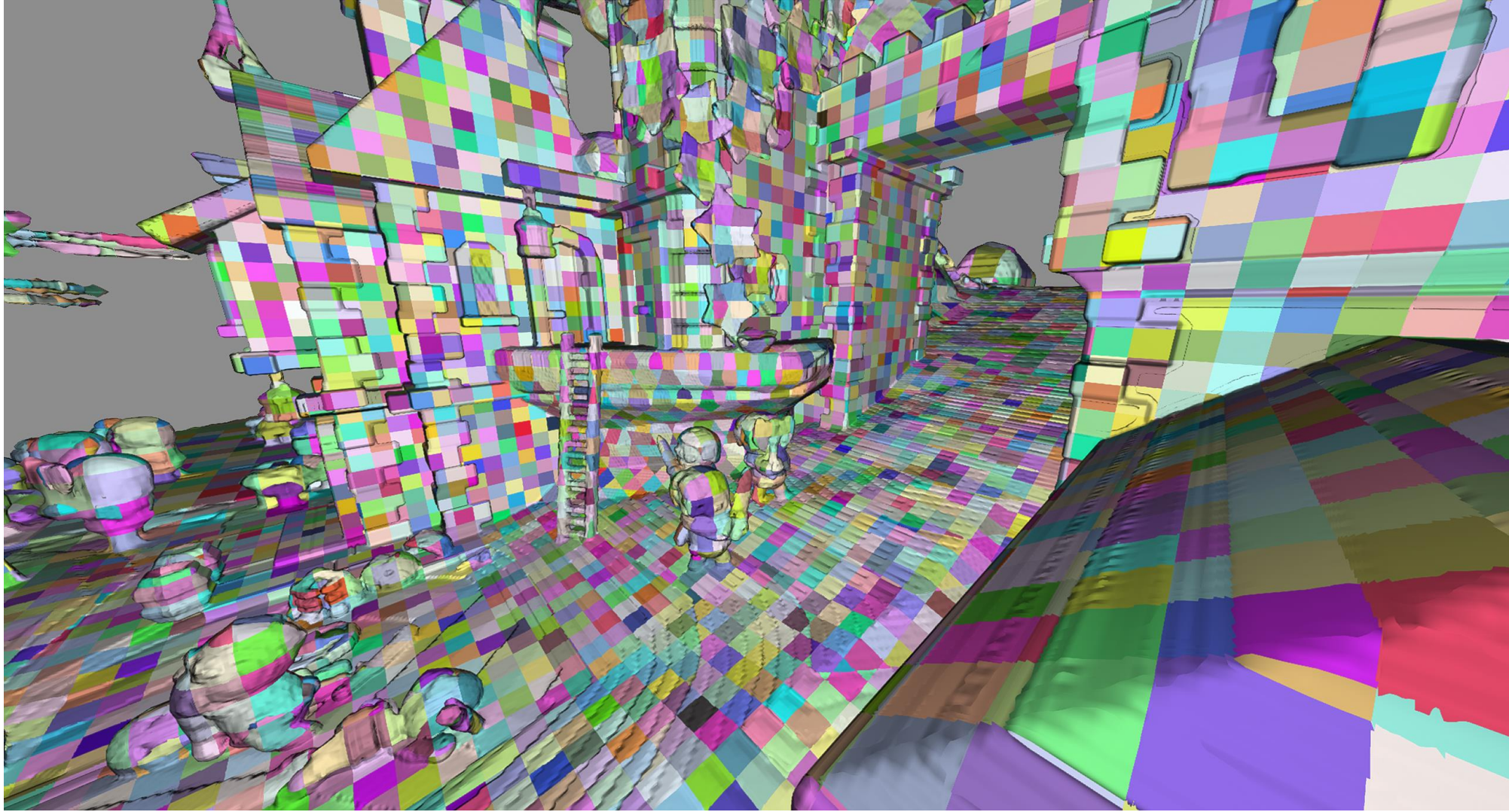
- Each Brixel stores the distance to its closest surface
- The region the Brick encodes depends on the cascade and its resolution

BRIXELIZER - CASCADES

- Brixelizer uses cascades around the camera
- Each cascade has its own 3-Level AABB tree
- All cascades will allocate their Bricks in the same 3D Brick atlas



BRIXELIZER – DEBUG VIEW

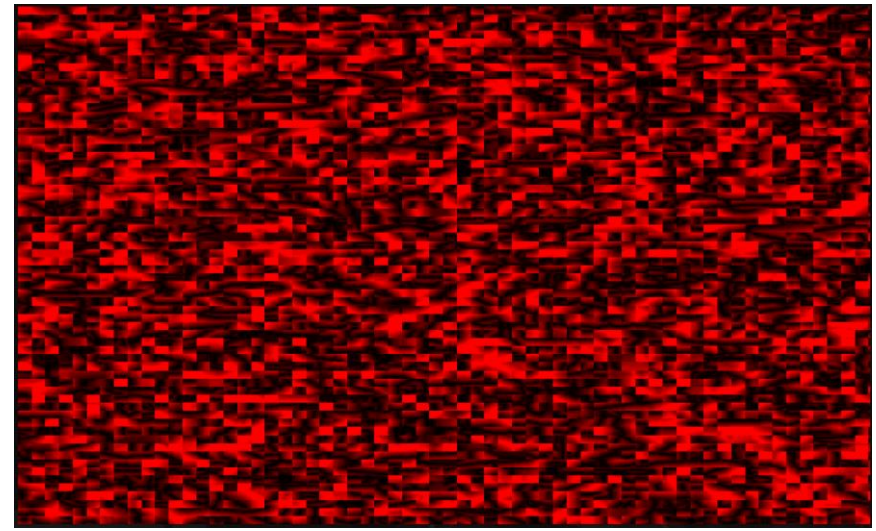
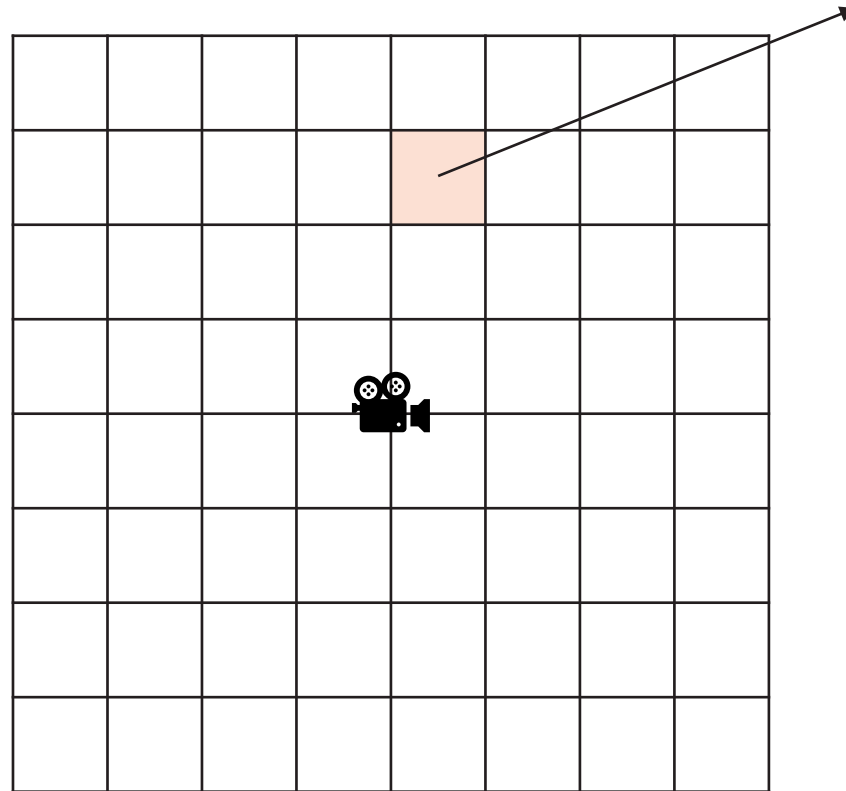


ALGORITHM

BRIXELIZER - ALGORITHM

- How do we generate the Bricks for each cascade?

→ Voxelization Pipeline



VOXELIZATION PIPELINE

Voxelize

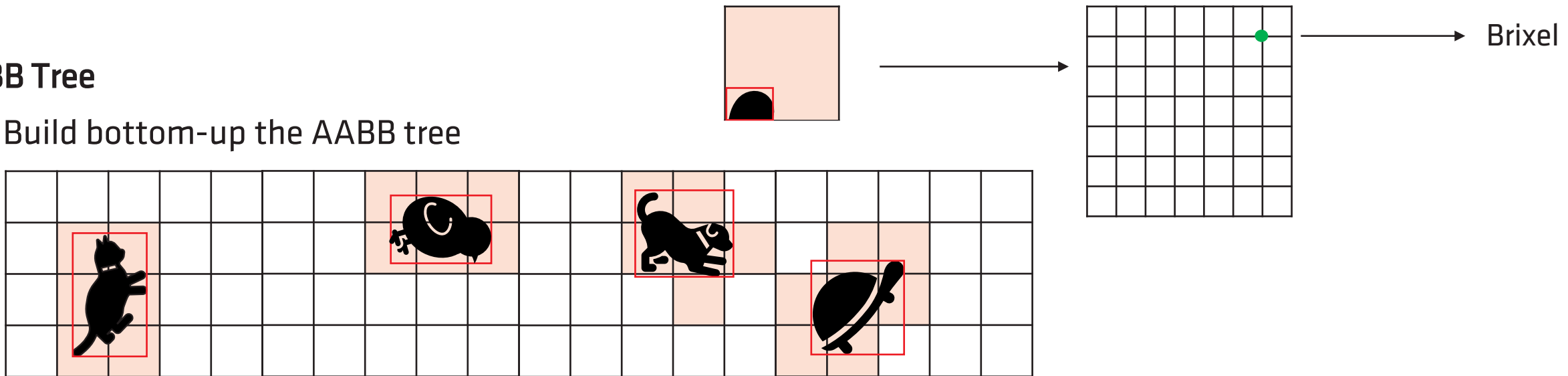
1. Identify all grid cells that intersect with triangles: Each grid cell is called a voxel
2. For each voxel that contains triangles, create a Brick

Emit SDFs

3. For each Brick, determine which Brixels intersect with triangles and store the minimum distance
4. Fill in the remaining empty Brixels

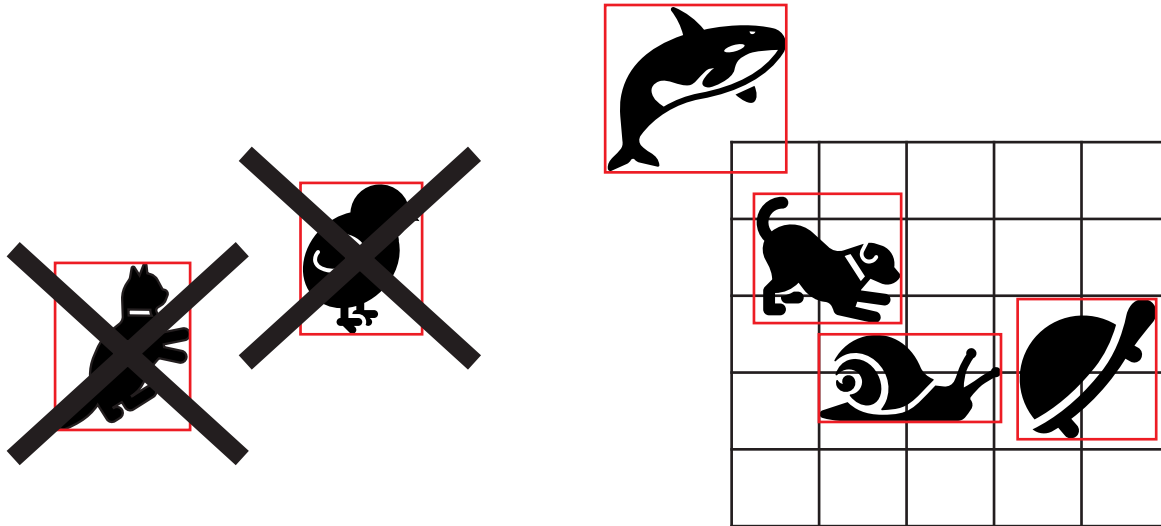
AABB Tree

5. Build bottom-up the AABB tree



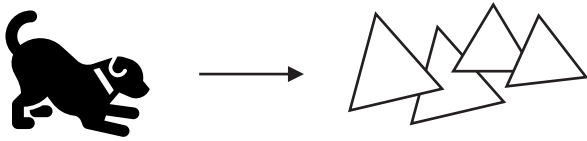
VOXELIZE

- Input: List of geometries + AABBs
- Cull geometries if their AABBs do not intersect the cascade grid



VOXELIZE

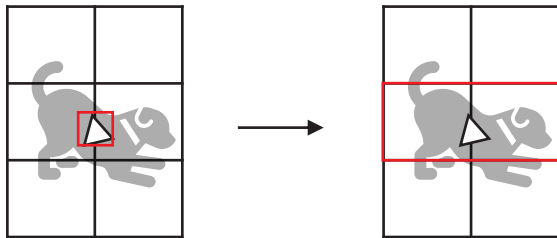
- Scan the geometries



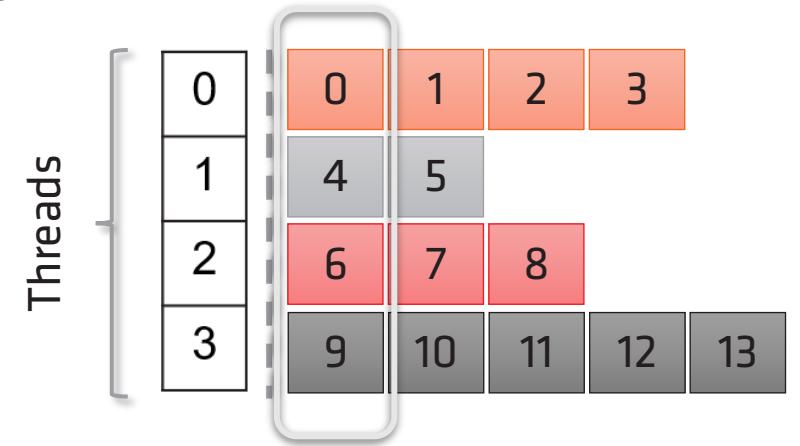
- Count all the triangles for each geometry and sum them up
- The voxelizer spawns 1 thread for 1 triangle

VOXELIZE

- Scale the triangle AABB to voxel size unit
→ Gives the maximum number of voxels the triangle possibly intersects
- Each such triangle and voxel pair is called a reference

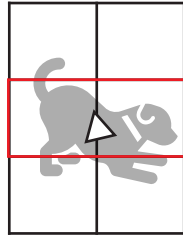


= maximum of 2 references

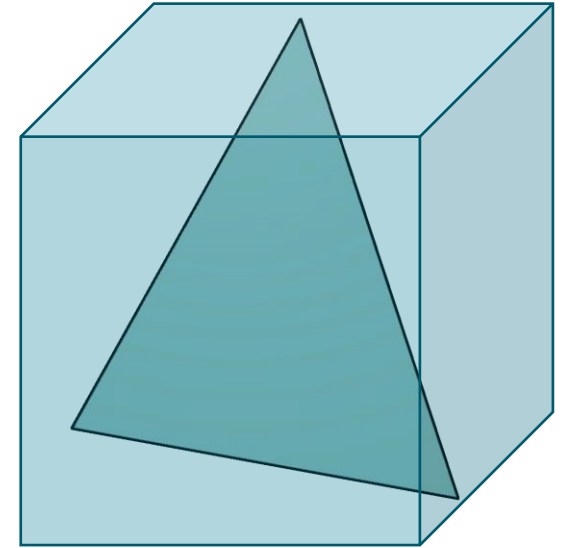
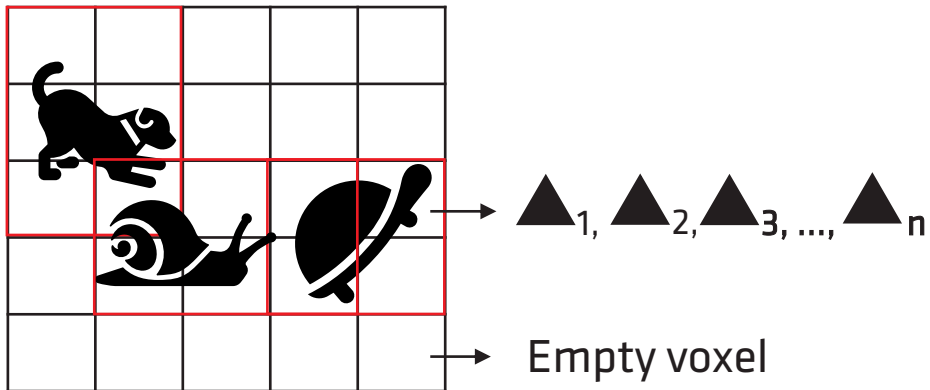


VOXELIZE

- Process the references
- Check if the triangle and not just its AABB intersects with the voxel



- Count the remaining number of references on a per-voxel basis
→ For each voxel we store the number of triangles it intersects with

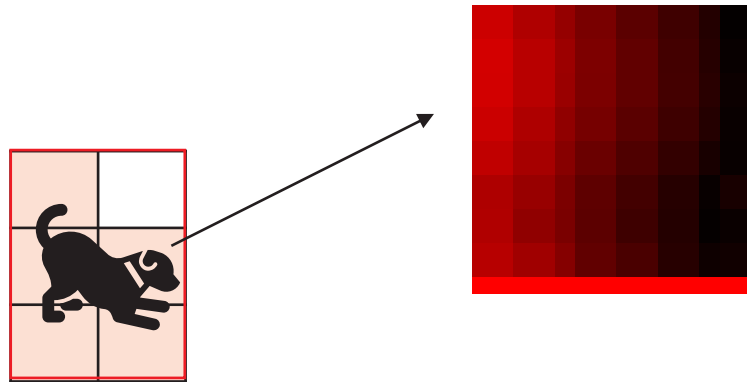
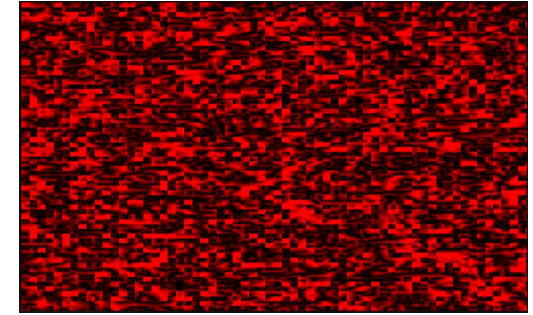


Triangle AABB

EMIT SDF

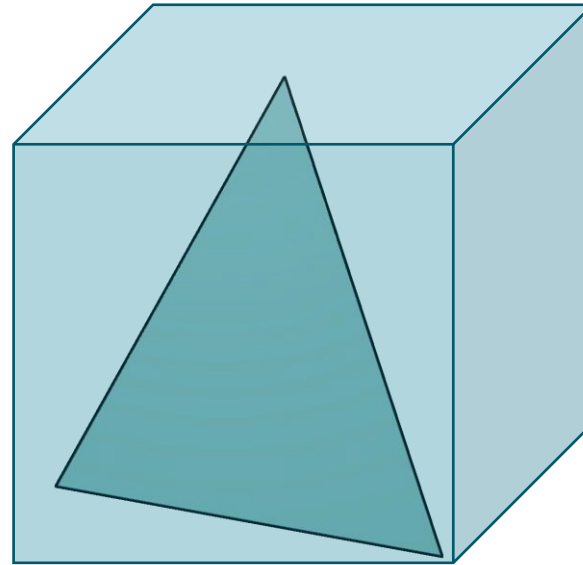
For each voxel that intersects at least one triangle:

- Allocate a new Brick
 - Each Brick is 8x8x8
 - Each Brixel is 8 bits (R8_UNORM)
 - It encodes 0 ... 1 distance that is normalized to the linear Brick size
- 512 Bytes per Brick

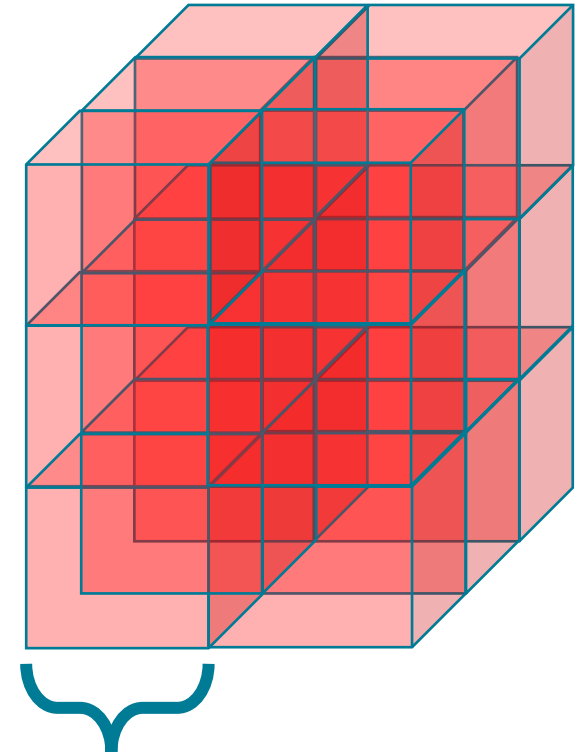


EMIT SDF

- Finally, fill the Bricks with the distance fields
- Each thread group processes one Brick
- Each thread processes one triangle
- Check for triangle-Brixel intersections
- Store the **minimum** distance



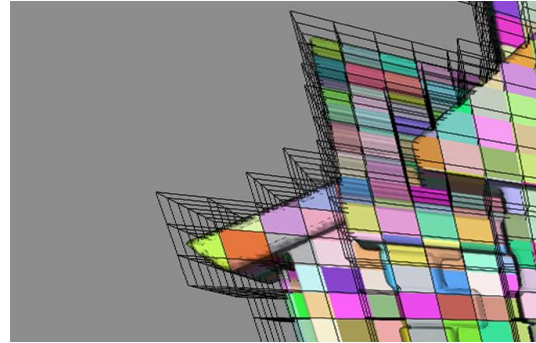
Triangle AAB



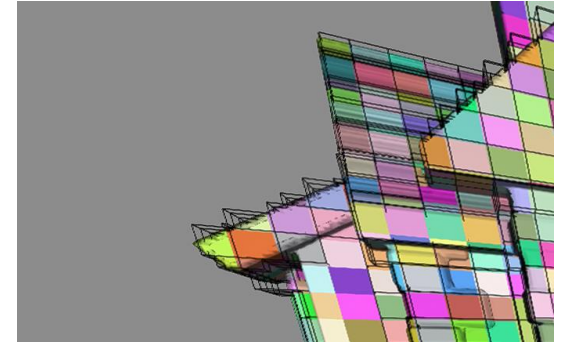
Brixel

EMIT SDF

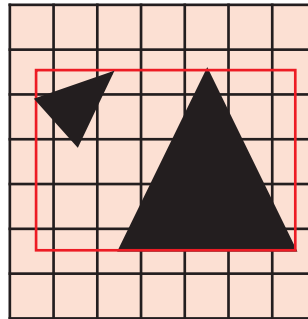
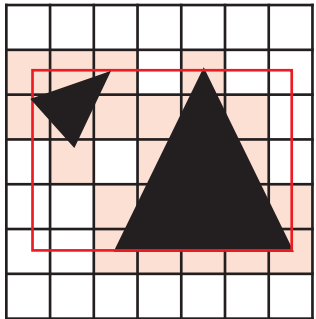
- Each Brixel that intersects a triangle stores the minimum distance
- Store the AABB of all the triangles in the Brick
 - used to build bottom up the AABB tree
- Some Brixels are still empty
- Use Eikonal / Jump flooding to fill in the gaps



Voxels with Bricks



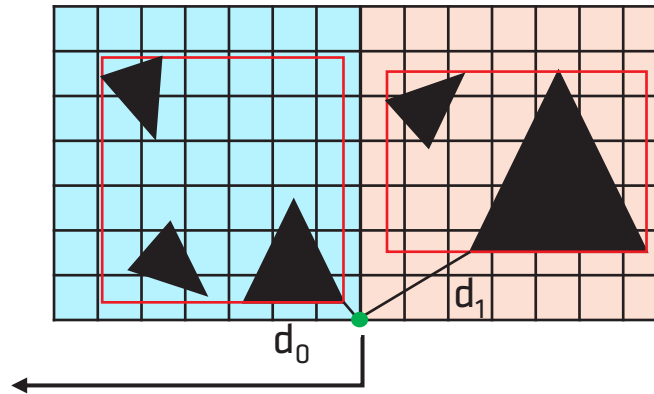
AABBs of the Bricks



EMIT SDF

- Neighboring Bricks have the same border Brixel values
- They use the same triangles for baking
- Triangle vs Brick AABB is expanded by $(1/7) * \text{brick_size}$

→ C_0 continuity



Brixel stores d_0 for both Bricks

ALGORITHM OVERVIEW

Local Distance Fields

Voxelize

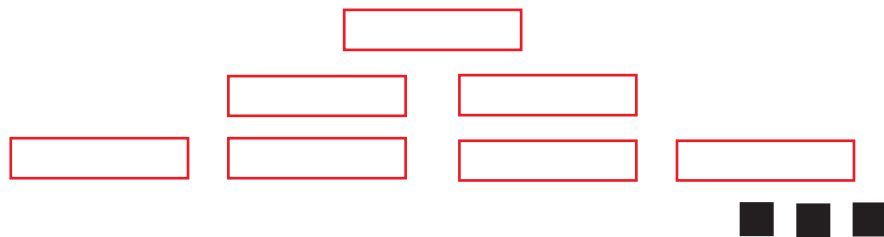
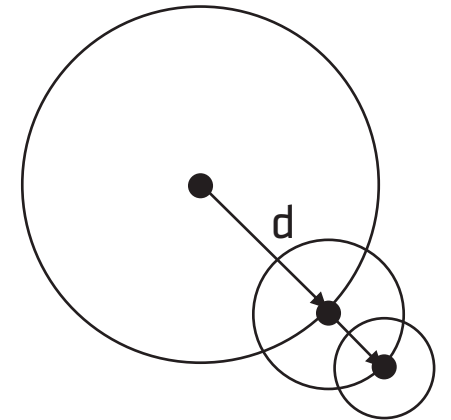
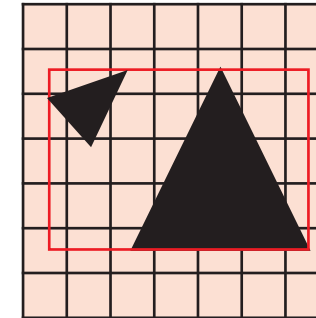
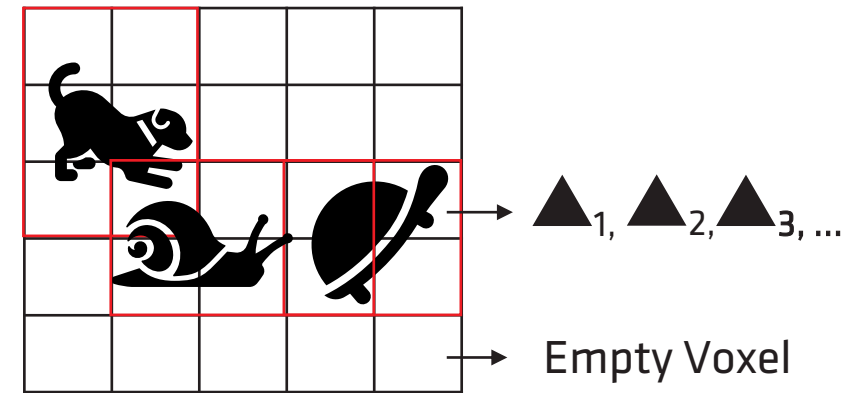
1. Generate for each voxel a list of triangles it intersects
2. Create a Brick for non-empty voxels

Emit SDFs

- For each Brick, determine which Brixels intersect with triangles and store the minimum distance
- Fill in the remaining empty Brixels of each Brick

AABB Tree

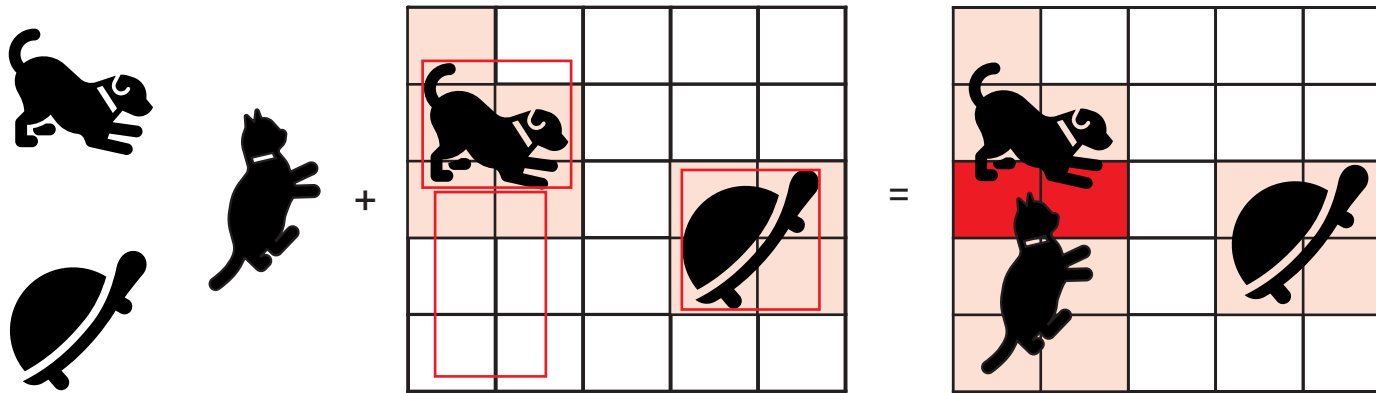
5. Use the AABB of the Bricks to build bottom-up the AABB tree



OPTIMIZATIONS

STATIC GEOMETRY

- Static geometry does not need to be re-voxelized each frame
- Brixelizer only creates Bricks for voxels, that do not point to any Brick yet
- If a voxel already points to a Brick, this voxel is skipped for the whole voxelization + emit sdf pipeline

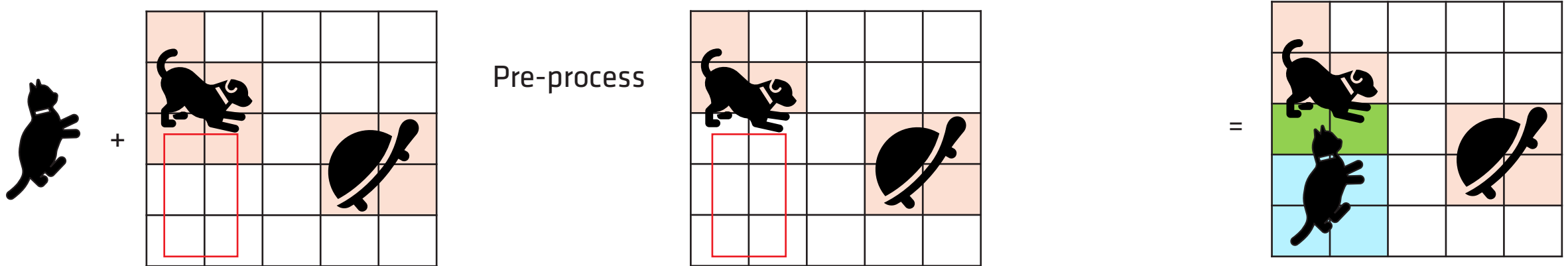


The red voxels only contain dog data

The distance field represents a cat without head → which is clearly wrong 😊

STATIC GEOMETRY

- Add a pre-process step to mark any voxels that possibly intersects with new geometry
- Include marked voxels in the voxelization step



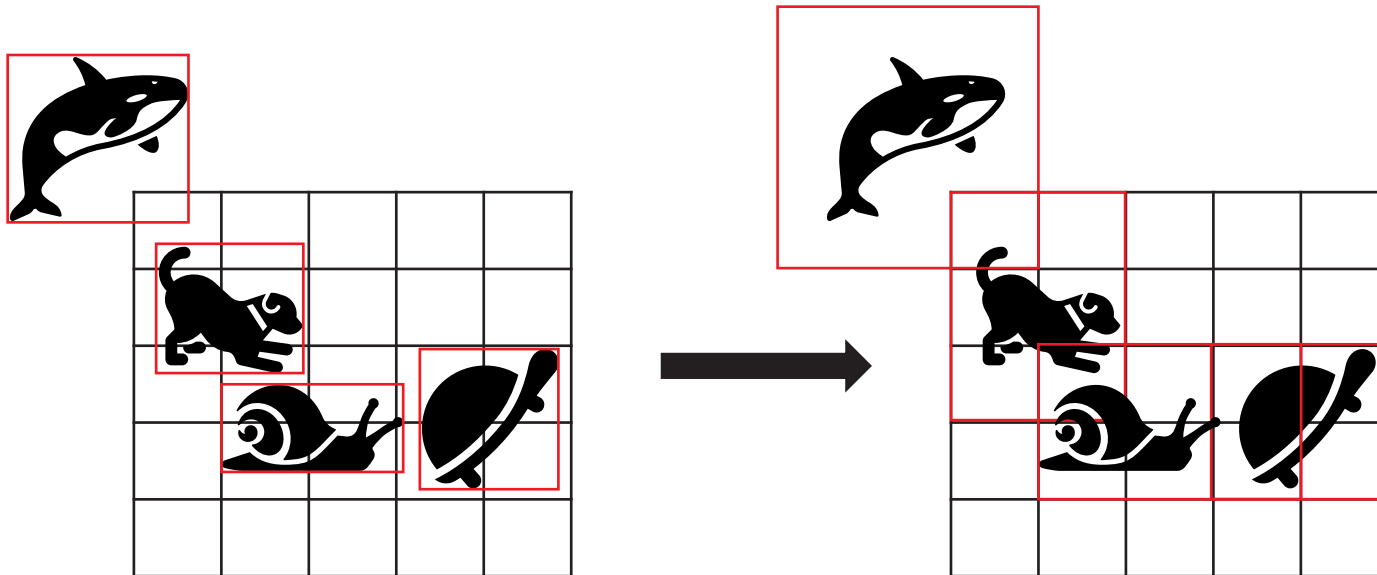
- Pre-processing is done for new and deleted geometry
- The cat got its head back! The green voxels contain dog and cat data

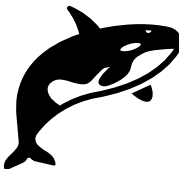



STATIC GEOMETRY – PRE-PROCESS

- The AABB of each geometry is scaled to be in voxel-size unit
- Use the volume of the AABB to get the **maximum** number of possible voxels the geometry intersects

→ Geometry reference

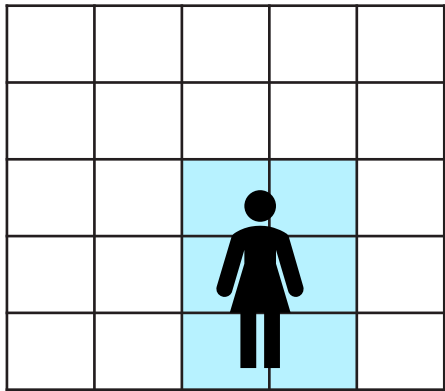
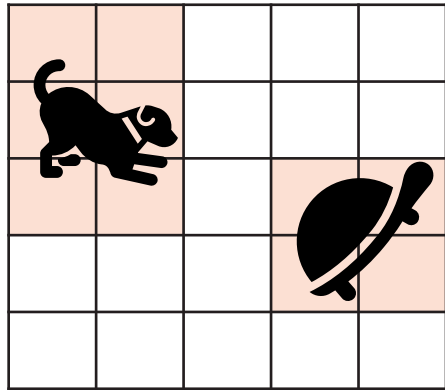
- Mark voxels of all geometry references as empty



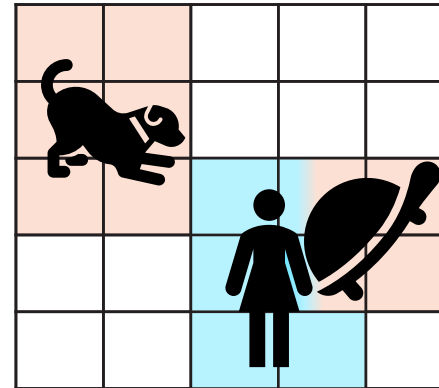
	= 9 → 1
	= 6
	= 6
	= 4

DYNAMIC GEOMETRY

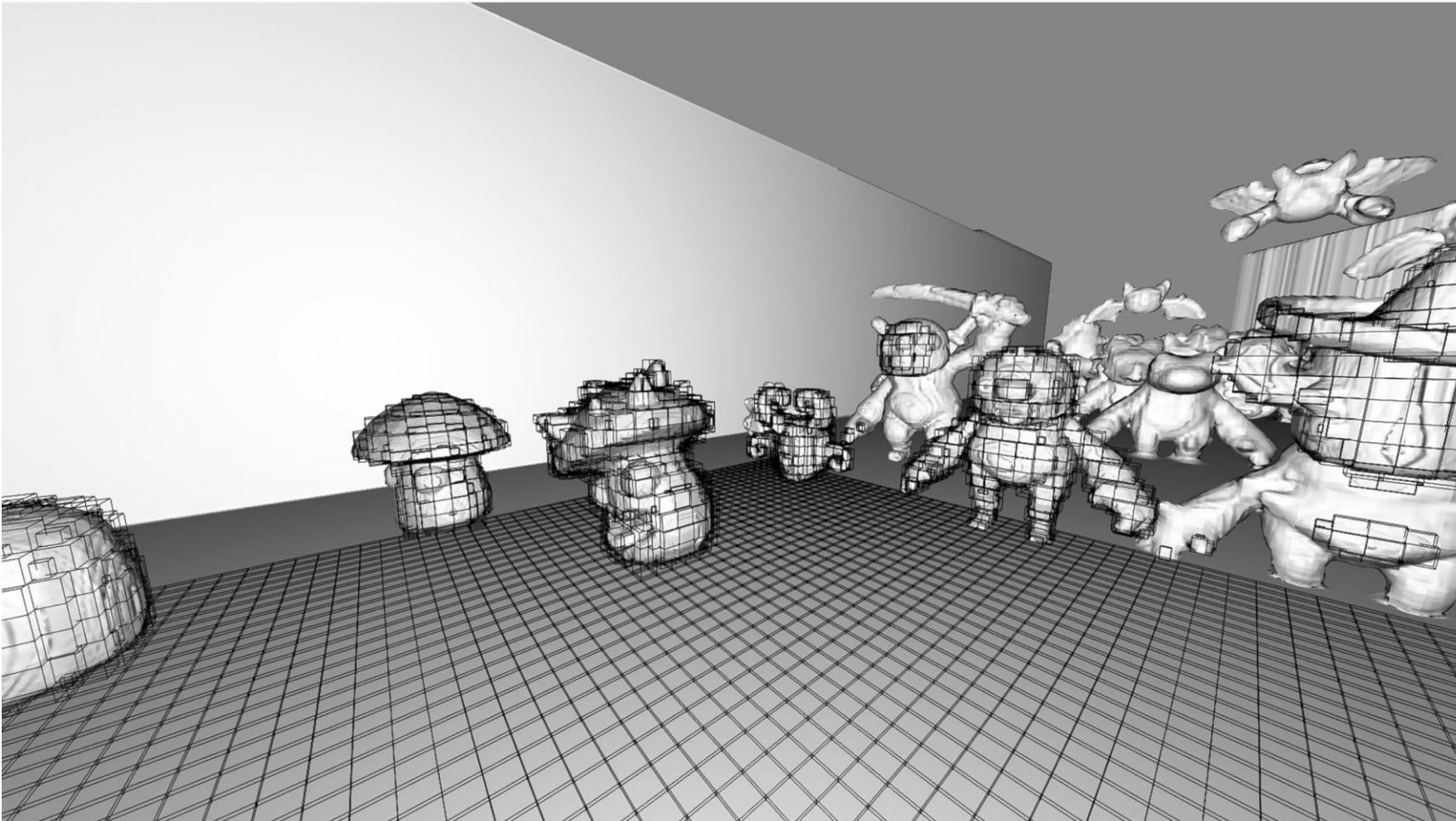
- Dynamic geometry is voxelized in a separate cascade
- Cascades with dynamic geometry are cleared and re-build from scratch on update
- We merge the static cascade with the dynamic cascade before the Eikonal / Jump Flooding stage



$\min(\text{orange box}, \text{blue box}) =$
per Brixel



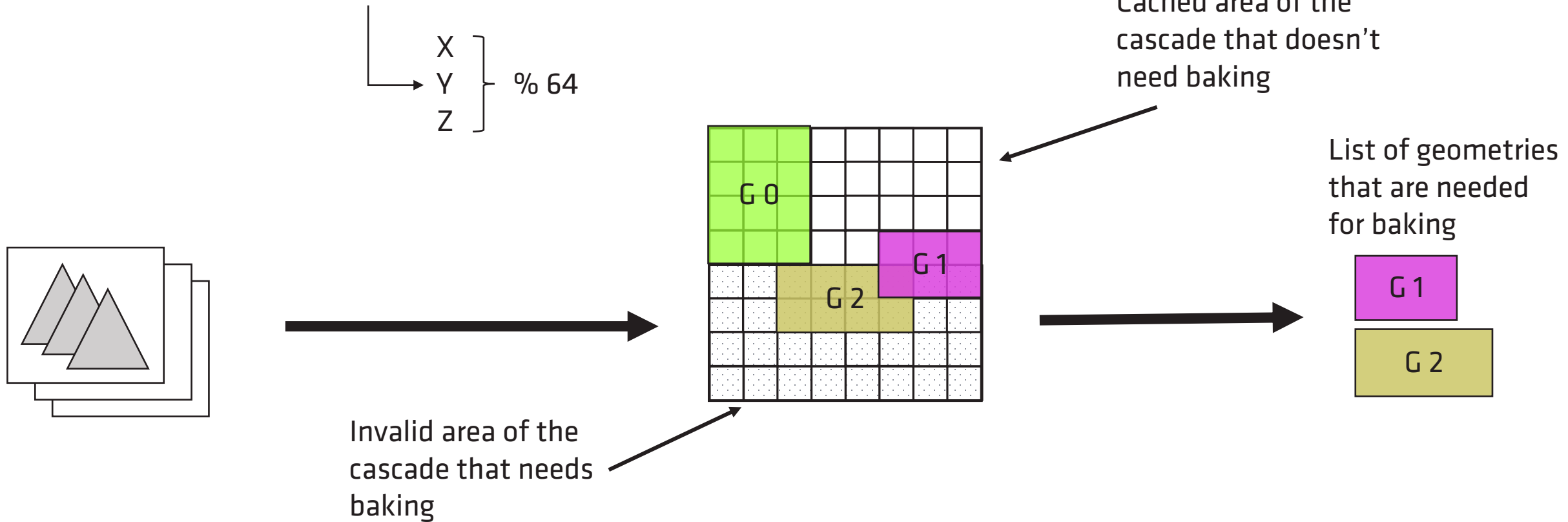
DYNAMIC GEOMETRY – CASCADE LEVEL 0



<https://gpuopen.com/video/GDC-2023-Sparse-Distance-Fields-Video.mp4>

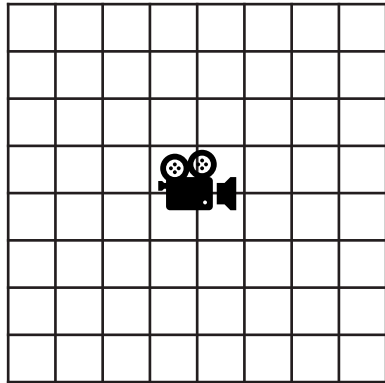
PARTIAL CASCADE UPDATE

- Brixelizer supports partial cascade updates
 - Voxelize only updated slices
 - No need to copy the data around – move the virtual center of the cascade
 - Data is addressed toroidally w.r.t. the center

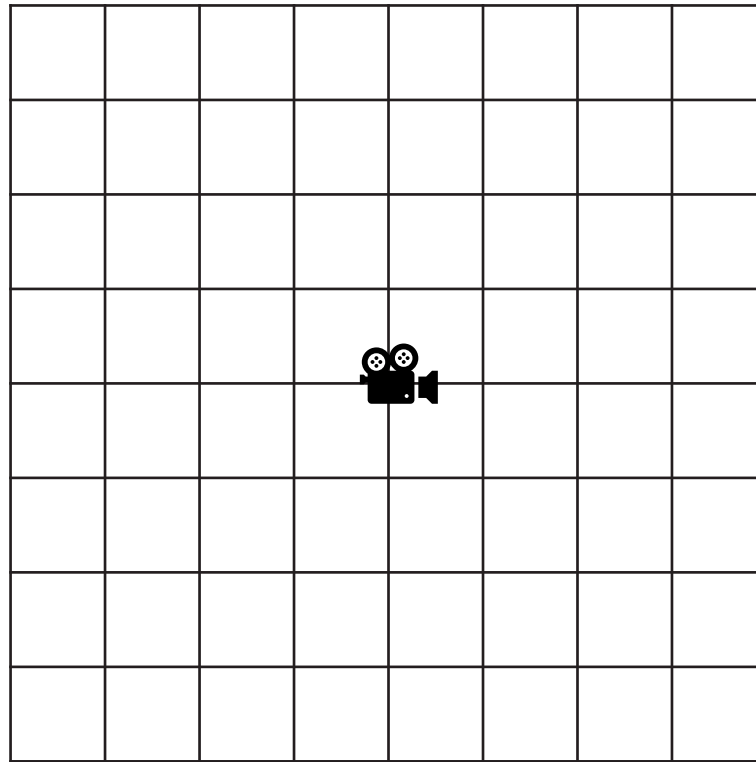


TIME SLICING

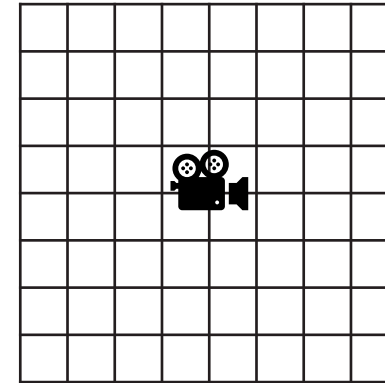
- Limit the cost of updating the structure with many cascades
 - Improves update performance at the cost of some latency
 - Could even limit to only partially update a single cascade per frame
- Can choose different strategies for time slicing prioritization based on scene and requirements



Frame n



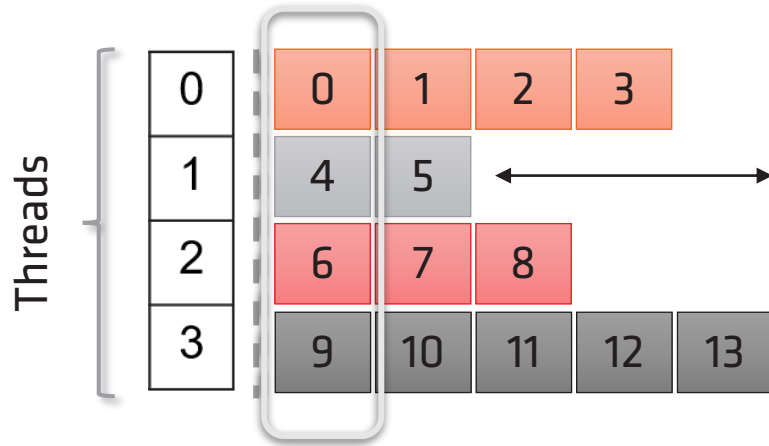
Frame n + 1



Frame n + 2

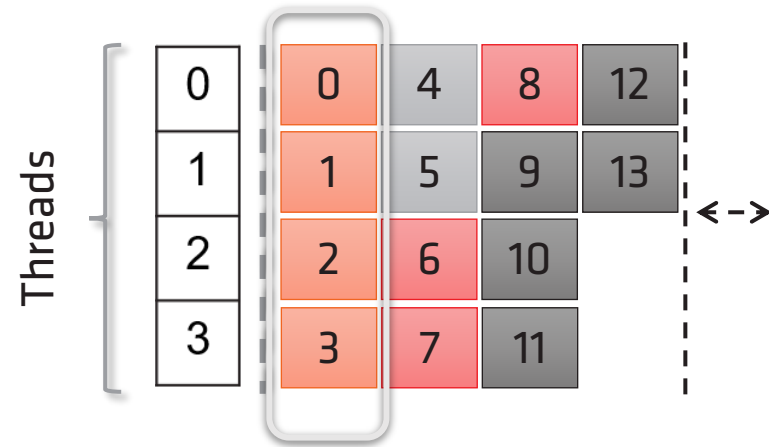
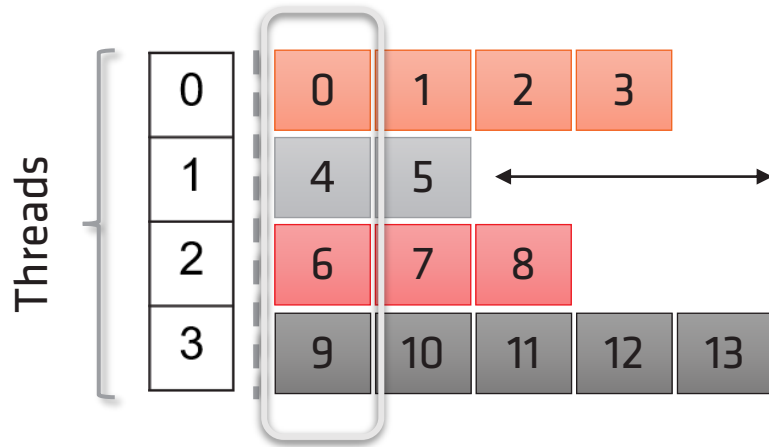
VOXELIZE – BALANCE THE WORKLOAD

- One thread per triangle is invoked to determine the number of references per triangle
→ So how many voxels this triangle possibly intersects with
- Each thread could go on to process all the references of its triangle
- However, this can lead to unbalanced workload



VOXELIZE - BALANCE THE WORKLOAD

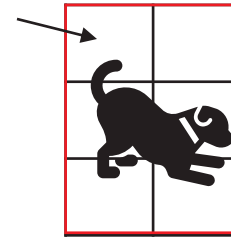
- Store all triangle references of the thread group in group shared memory
- Redistribute the workload based on reference id



EMIT SDF – BALANCE THE WORKLOAD

- When computing the distance field, one thread group works on a single Brick
- Some Bricks intersect many triangles, others very few
- Split up the work for Bricks with many triangles in sub-voxels
- Each sub-voxel processes 32 triangles
- One thread groups works on a single sub-voxel

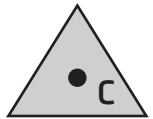
Few triangles



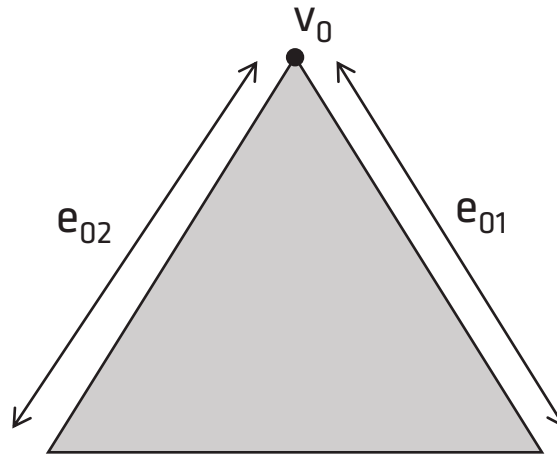
Many triangles

SMALL TRIANGLES

- Small triangles are handled differently to save memory and speed up their processing
- A small triangle is smaller than a given threshold, dependent on the resolution of our Brick
 - $1/10^{\text{th}}$ of a Brick \rightarrow smaller than a single Brixel
- Only the center of the triangle is stored and used to do a point approximation to determine the distance from the triangle to the voxel
- A small triangle needs 12 Bytes, while otherwise 24 Bytes are used



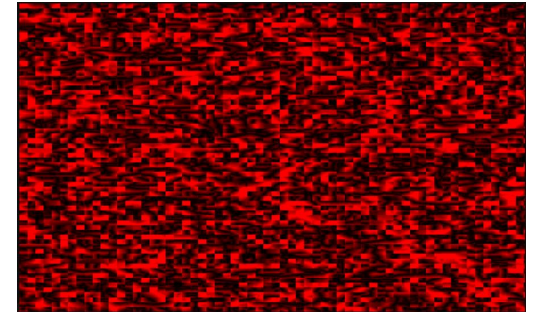
Store the center



Store one vertex and two edges

UPPER LIMITS

- There is a limit on how many references are processed per frame
- Any reference that exceeds this limit is processed in a later frame
- There is also a maximum number of Bricks we can allocate per bake
- If we exceed this limit, an invalid Brick ID is returned
- The cascade size and resolution must be adjusted properly for the scene



3D Brick atlas has a fixed size!

We can't allocate more Bricks than the atlas has space for

PERFORMANCE

- Performance is influenced among others by
 - how many voxels are touched per update
 - how many triangles need to be processed
 - how many Bricks need to be baked
- All of these depend on the scene and its geometries and the cascade
- There is also a small overhead involved in state tracking and clean-up for every cascade update

→ It is important to tune the parameters of Brixelizer appropriately for the given scene, as they can affect performance substantially

- Preliminary results* show, that Brixelizer
 - Can process several hundreds of thousands of voxels in under a millisecond
 - Can process hundreds of thousands of triangles in under a millisecond
 - Can bake several thousand new Bricks in under a millisecond
- If any of these go into the millions, the processing time can take a few milliseconds.

*Test System: RX 7900 XTX, Driver 23.2.1 – Based on a pre-release version of Brixelizer. Performance is expected to change for the released version.

CURRENT LIMITATIONS

- Memory management
 - Although memory consumption is reduced by using a sparse distance field, the fundamental problem remains
 - Trade-off between precision and memory consumption
- Precision
 - The distance field generated out of a triangle-based scene is only an approximation. Using directly the triangle-based scene is more accurate
 - This can get visible for near-field objects
- No global distance field – only a collection of local distance fields
 - Brick ignores the rest of the geometry during distance field build
 - Neighbour values match $\rightarrow C_0$ continuous
 - The gradient is not continuous by default on the Brick boundary
- Distance fields only provide a point in space as a hit point
 - Possibly also a gradient
 - Anything else needs to be added manually on top of Brixelizer

INTEGRATION

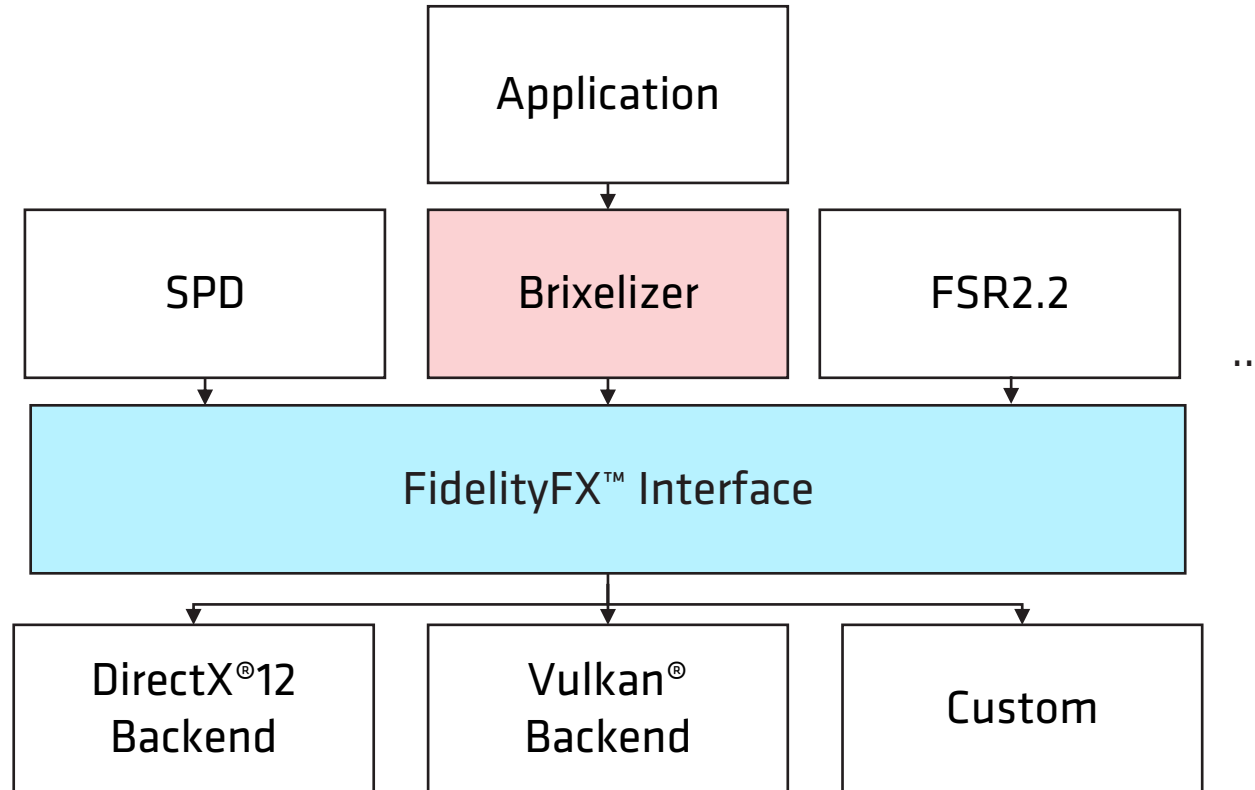
INTRODUCING FIDELITYFX BRIXELIZER

- Brixelizer will be part of the new AMD FidelityFX™ SDK. This means it's
 - Open source
 - Cross platform
 - Highly optimized
- Runs on any DX12-compatible GPU
- Provided via library, with full C++ and HLSL source with API documentation to enable custom integrations
- Sample provided for DirectX® 12



INTEGRATION – BACKEND

- Brixelizer will be part of the new AMD FidelityFX™ SDK
- Check out the FidelityFX™ SDK presentation for more details!



INTEGRATION – BASICS

- Add ffx_bx.h or ffx_bx_simple.h to your codebase and link to the static library
 - The FFX Brixelizer API is offered through a Windows® library to link against
 - Full source code including shader and C++ code for the library will be made available on GPUOpen
- There are a number of function entry points related to
 - FFX Brixelizer Context management
 - FFX Brixelizer Cascade management → the generation of the distance field
 - FFX Brixelizer Tracing & Debugging → Tracing and Debugging utilities to help you accelerate integration

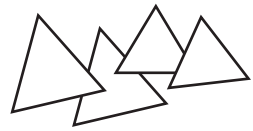
INTEGRATION - CONTEXT

Usually, you have one context per scene

```
typedef struct FfxBxCreateContextDesc
{
    float                sdfCenter[3]; // all cascades share the same center.
    uint32_t            numCascades; // the number of levels. Per level you can have a
                                maximum of 2 cascades: static and dynamic.
    FfxBxCascadeDesc     cascadeDescs[FFX_BX_MAX_CASCADES]; // specifies for each level if a static
                                and/or dynamic cascade should be created
    FfxInterface         backendInterface;
} FfxBxCreateContextDesc;
```

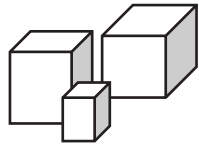
- Dynamic cascades ignore static geometry
- Static cascades ignore dynamic geometry
- Please refer to the official documentation for details

INTEGRATION – INSTANCES



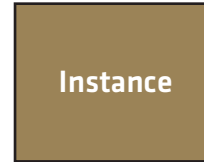
Geometry:

- Vertexbuffer
- Indexbuffer
- Transform



AABBs

`ffxBxSimpleCreateStaticInstance`
`ffxBxSimpleCreateDynamicInstance`



- Creates an instance, managed internally by Brixelizer
- Returns an instance ID for static instances

- An instance ID is used to uniquely identify a geometry + AABB input from the application
- Instance IDs are persistent
- It is recommended for an application to keep track of the IDs to verify whether a geometry has been already processed by Brixelizer or not

INTEGRATION – INSTANCES



ffxBxSimpleDeleteInstance



- Removes the instance from Brixelizer's internal instance list

- It's up to the application to stream in and out geometry such that it makes sense
- The maximum number of supported instances at a time is 64K
- Example reasons to remove instances:
 - Too far away
 - Destroyed geometry
 - Semi-static geometry
- Dynamic instances are deleted automatically every frame

INTEGRATION – CASCADE UPDATE

`ffxBxSimpleBakeUpdate` → fills in `FfxBxBakedUpdateDesc` for you

```
typedef struct FfxBxBakedUpdateDesc
{
    FfxBxCascadeUpdateDescription cascadeUpdateDesc;
    uint32_t      numStaticJobs;
    FfxBxJobDescription staticJobs[3 * FFX_BX_MAX_INSTANCES];
    uint32_t      numDynamicJobs;
    FfxBxJobDescription dynamicJobs[FFX_BX_MAX_INSTANCES];
} FfxBxUpdateDesc;
```

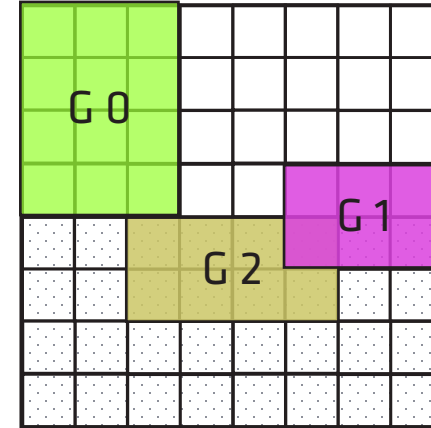
- Queues the instances to the correct cascades
 - Static instances are processed in the static cascade: static job list
 - Dynamic instances are processed in the dynamic cascade: dynamic job list
 - Creates invalidation jobs for the static cascade to clear voxels for new and destroyed instances
- Only empty voxels will be touched during cascade update!
- If there are more invalidation jobs than twice static instances, the whole cascade will be cleared

INTEGRATION – UPDATE CASCADES

- `FfxBxCascadeUpdateDescription` is filled on `ffxBxSimpleBakeUpdate`
 - Updates the center of the cascade

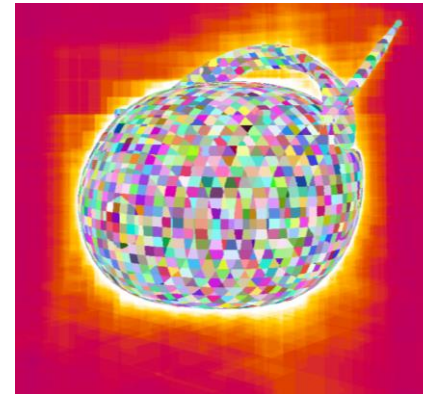
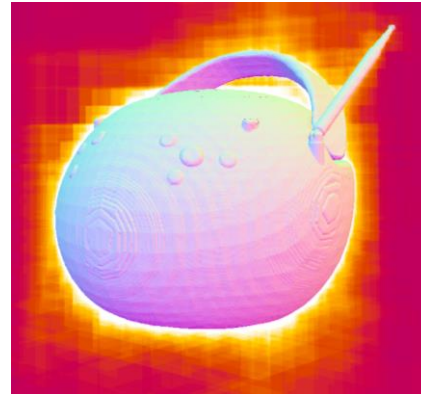
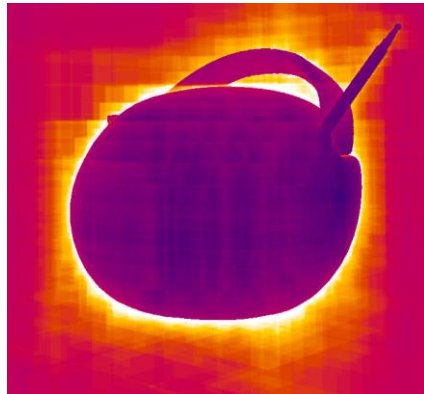
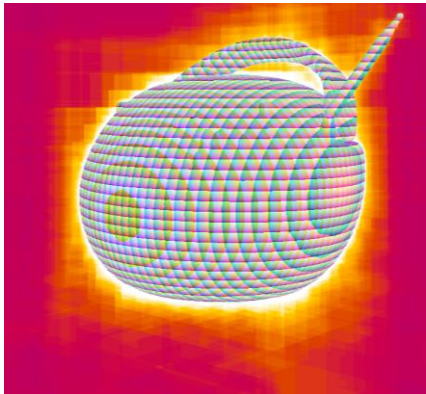


- Partial cascade updates have to be specified manually
- Allocate a local memory scratch buffer
- Update the cascade using `ffxBxSimpleUpdate`
- Brixelizer's built-in time slicing strategy:
 - Update the 1st cascade every even frame
 - The others every odd frame



INTEGRATION – TRACING & DEBUGGING

- Brixelizer library comes with debug visualization to help verifying successful integration
- Different debug modes
 - UVW
 - Iterations
 - Gradient
 - Brick ID
 - Cascade ID
- Select a subset of available cascades



BRIXELIZER – USE CASES

- Brixelizer offers an efficient way in finding the hit point of a ray with the scene
- This can be used in different ways – it's up to you!
- Some examples:

Global Illumination



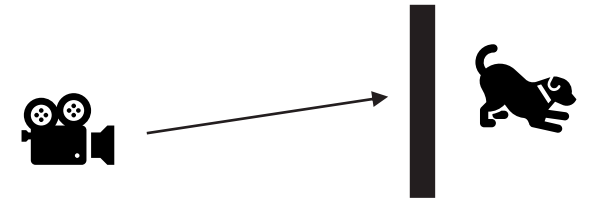
Ambient Occlusion



Volumetric Effects



Occlusion Testing



BRIXELIZER GLOBAL ILLUMINATION

BRIXELIZER GLOBAL ILLUMINATION



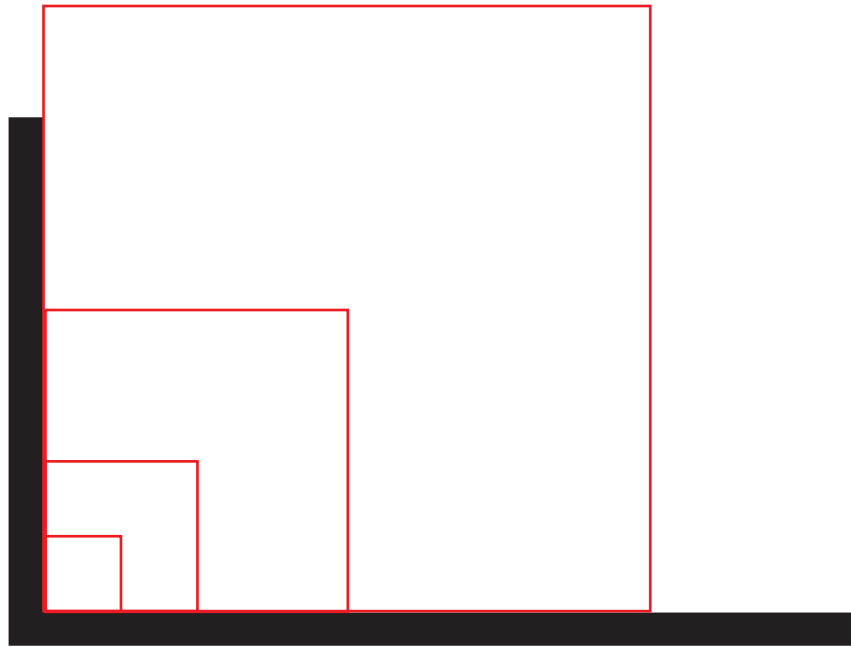
Resolved diffuse GI



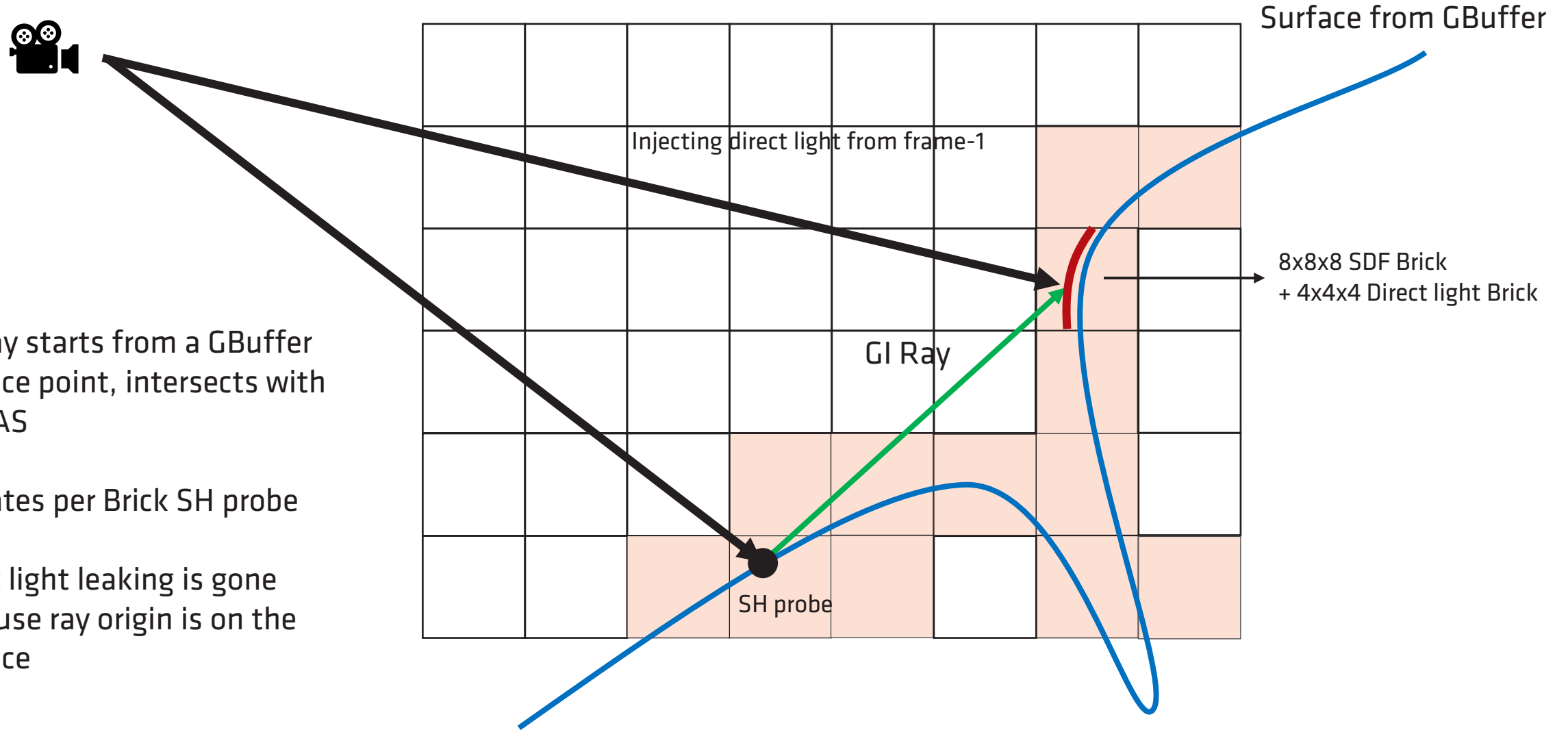
Final direct and indirect lighting

BRIXELIZER GLOBAL ILLUMINATION

- Brixelizer Global Illumination is a probe-based approach
- Only voxels that touch surfaces have probes
- All voxels that point to a Brick also point to a probe
 - 4x4x4 Direct Light Probe
 - Spherical Harmonic Probe
- For volumetric effects, a probe from a bigger cascade further from the surface will be used



BRIXELIZER GLOBAL ILLUMINATION



- GI Ray starts from a GBuffer surface point, intersects with Brix AS
- Updates per Brick SH probe
- Most light leaking is gone because ray origin is on the surface

CURRENT LIMITATIONS

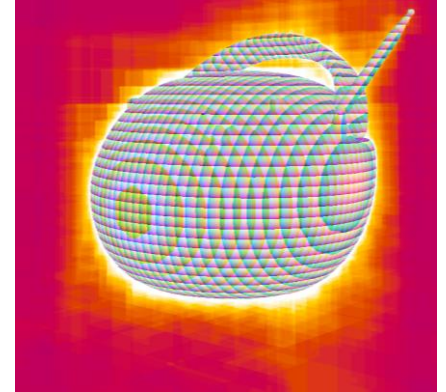
- It's basically screen space+
 - The direct light probe is updated from screen space
 - Updating out-of-screen direct light probe is very scene dependent
- Convergence over time can be visible
- Suffers from overshadowing for tiny geometry

→ Brixelizer Global Illumination is still under development!

CONCLUSION

- Brixelizer is a library that generates a sparse distance field in run-time for any triangle-based geometry
- It provides several options to tweak the distance field in terms of update performance and precision

→ It offers an **efficient way** in finding a ray hit point with the scene



- Brixelizer can be used for various effects such as GI, AO, volumetric effects and occlusion testing
 - This talk gave a quick glimpse in how one can compute GI on top of Brixelizer
- Brixelizer Global Illumination is still work in progress 😊

DISCLAIMER

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

Use of third-party marks / products is for informational purposes only and no endorsement of or by AMD is intended or implied. GD-83

© 2023 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Radeon and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Windows and DirectX are registered trademarks of Microsoft Corporation in the US and other jurisdictions. Vulkan and the Vulkan logo are registered trademarks of the Khronos Group Inc. Other names are for informational purposes only and may be trademarks of their respective owners.



together we advance_