SIGGRAPH2011

Practical Occlusion Culling in Killzone 3

Michal Valient

Lead Tech, Guerrilla B.V.

SIGGRAPH2011

Talk takeaway

SIGGRAPH2011

- Occlusion culling system used in Killzone 3
- The reasons why to use software rasterization
- (Some) technical details
- How to pick good occluders
- Q&A

Wed, October 12, 2011

I'll first **describe** the **occlusion** system in Killzone 3
and the reasons why we chose it and why should you

Then I'll talk about some **technical details**
and **heuristics** for picking **good** occluders.

And I hope we'll have some time for questions.

... but let's **first** take look at what **Killzone** 3 actually is.

Killzone 3 is a **first person shooter** released
earlier this year **exclusively** for Playstation 3.

And it's essentially about **space marines** trying to escape
from the planet full of **space Nazis**.

Wed, October 12, 2011

The game is set in **huge detailed outdoor** environments…

Where you can fly around with an armed jetpack...

And you get to fight giant spider robot.

Twice.

Killzone 3 is a big game

and we needed good object visibility solution

that works well with these diverse settings.

Wed, October 12, 2011

The game is set in huge detailed outdoor environments…
Where you can **fly** around with an **armed jetpack**...
And you get to fight giant spider robot.
Twice.

Killzone 3 is a big game
and we needed good object visibility solution
that works well with these diverse settings.

Wed, October 12, 2011

The game is set in huge detailed outdoor environments…
Where you can fly around with an armed jetpack...
And you get to fight **giant spider** robot.
Twice.

Killzone 3 is a **big game**
and we needed **good** object **visibility solution**
that **works** well with these **diverse** settings.

Killzone 3 visibility solution

SIGGRAPH2011

- Software rasterization running on SPUs
- Render occluders into depth buffer
  - Use simplified version of the scene geometry
- Conservatively scale down
  - To make it fit into SPU memory
  - To make it faster to test against
- Test all objects against small depth buffer
  - Test bounding boxes

We chose to try **software rasterization** running completely on **SPUs**.

We render the **simplified** version of the level **geometry**
into a depth buffer (these are the occluders)

And we then use **scaled down** version of this **depth buffer**
to test visibility of all **objects** or **lights** in the current view frustum.
The test itself uses **bounding box** of the objects.

Why software rasterization

SIGGRAPH2011

- Previous solution did not scale well
  - Manually placed portals

Wed, October 12, 2011

I'll try to **summarize** the reasons why we chose this **solution**
and I'm sure you **recognize** some of your own experiences here.

First and foremost we saw that our solution based on **portals**
does **not scale** well with the big **outdoor levels**.

Especially since our portals were hand **placed by artists** and we were
running into **production stalls**.

Very important issue if you try to create the game **in two years**.

Why software rasterization

SIGGRAPH2011

- Previous solution did not scale well
  - Manually placed portals
- Works automatically
  - Can be enabled early in production

Wed, October 12, 2011

The process of **occluder creation** can be made largely **automatic** and can be enabled from the early days of production.

The whole **concept** of occluders is very **similar** to building of regular geometry and it's easy to **understand by artists**.

This makes it very **easy to step in** and manually create occluders **where needed**.

Why software rasterization

SIGGRAPH2011

- Previous solution did not scale well
  - Manually placed portals
- Works automatically
  - Can be enabled early in production
- Completely dynamic solution
  - Any object can become an occluder

Unlike most other approaches, this one is **completely dynamic**.

Any **sufficiently large object** on screen can serve as a **good occluder**.
If you're **hiding** behind a destructible barrel or a **metal plate** in our game, it **is** an **occluder**.

Doors can open and close and they **perfectly block** the visibility **without** you having to write **special code** for such case.

Why software rasterization

SIGGRAPH2011

- Previous solution did not scale well
  - Manually placed portals
- Works automatically
  - Can be enabled early in production
- Completely dynamic solution
  - Any object can become an occluder
- Maps well to SPUs
  - No sync issues
  - No GPU costs related to visibility testing

Software rasterization maps well onto SPUs - it's **easy** to distribute
and **SPUs are generic** enough to allow us to run **complicated object culling** logic.

And **unlike GPU** based solutions, you get **exact results**
within the same frame **without** complicated **synchronization** logic.

Wed, October 12, 2011

Let’s look at the **example** of how the occlusion **system works**.

Wed, October 12, 2011

Let's look at the **example** of how the occlusion **system works**.

Wed, October 12, 2011

If we look from **the side**, you see we don't render anything behind the **closed door**.

But as soon as the **door open**...

Wed, October 12, 2011

...we start to render the rest of the **visible scene**.
As I mentioned earlier, this
happens entirely **automatically**.

And since I **forgot** to record the **occlusion**
depth buffer, you'll have to trust on this one.

Implementation overview

We implemented the system as series of **SPU jobs**.
Most run in **parallel** to do the heavy lifting.

Occluder setup

SIGGRAPH2011

- First stage, not parallel
- Outputs clipped + projected triangles
  - One list of triangle data
  - One “index” DMA list per rasterizer job
- Caches are important at this stage
  - 2KB vertex array cache (90% hit rate)
  - 32-entry post-transform cache (60% hit rate)
  - Various double-buffered output caches

The **first** SPU job loads **visible occluder primitives** and
outputs **clipped** and **projected** triangles for rasterization.

A single **list of triangles** is shared between the **rasterizer jobs**, and each job
has its own **DMA list** pointing to the subset of triangles it needs to draw.

The **occluder** primitives are **identical** to visual meshes, with **vertex and index arrays**.
Therefore we introduced several caches to
reduce bandwidth and vertex transformation costs.
This setup is very similar to what you find in a GPU.

Rasterization

SIGGRAPH2011

- Split 640x360p depth buffer into 16 pixel-high strips
  - Rasterize in parallel, one SPU job per strip
  - Load triangles using the prepared DMA list
- Traditional scanline rasterizer
  - Fill internal 640x16 floating point depth buffer
- Vectorization is the key
  - Set up three or four edges at once
  - Generate 4x1 pixels at once
  - Optimize in assembly

We **split** our depth buffer into **16-pixel-high strips** and run one rasterize job per strip in parallel.

Each rasterize job **loads** the **list of triangles** that intersect with its strip using the **DMA list** prepared by the setup job.

We perform **standard scanline rasterization** into an internal floating point buffer.

The rasterize jobs are **compute-bound**, and the code is **extensively vectorized** to improve throughput. **Inner loops** are written in SPA **assembler**.

Rasterization

SIGGRAPH2011

- Compress depth buffer
  - One output pixel is maximum depth of 16x16 block.
    - But patch single pixel holes first.
  - Encode as uint16, reserve 0xffffu for infinity
- Output single scanline of 40x23 occlusion buffer

For output, we **conservatively compress** the depth buffer.
Each **output pixel** is the **maximum** depth value of a **16x16 pixel** tile in the depth buffer.

Before compression, we **patch single pixel holes** to avoid leaks
when the occluders are **not water-tight**.
This is a bit of a **cheat**, but it's **necessary** otherwise
such hole **pushes** the tile way into the background.

The last step **encodes depth** into **16 bits**,
Occluder frustum is shorter than visual frustum so we
reserve one **bit** for points **behind** occluder **far** plane.

Occlusion tests

- Tests happen in parallel
- Each object consists of one or more parts
- First test object bounding box
  - Skip for objects visible last frame
- Then test individual parts
- Continue with submesh culling
  - Small Spatial Kd-Tree inside most meshes
  - Allows for culling arbitrarily small mesh chunks

The **last step** is the actual visibility **testing**.
We have **one job** that gathers **all objects** in the camera frustum
and then **spawns** an occlusion **test job** for each batch of objects.

We have a **two level hierarchy** of objects and meshes
objects live in the scene, and meshes are what we send to the GPU.
We test objects first to avoid testing meshes.

If a mesh has **many triangles**, we can continue with **submesh culling**.
This uses the mesh's **Kd-tree** to cull away whole ranges of **mesh triangles**.
Visible meshes form **new primitive** sent to GPU.

The **output** of these jobs is the **final result** of the occlusion query, and is sent for rendering.

Occlusion tests

- Accurate tests
  - Bounding box rasterization and depth test
  - Working on small depth buffer, be conservative
- Fast bounding sphere tests
  - Precomputed hierarchical reject data
  - Constant time test for small spheres
  - Only used for fast reject

We have **two kinds of visibility test** we can use to cull objects and parts.

The basic test is the most **accurate**, but also the most **expensive**.
It **rasterizes a bounding** box with **depth testing** against the small occlusion buffer.

We also have **constant-time tests** for **small objects**.
We **precompute** several versions of the **occlusion buffer** by
conservatively dilating the depth values.
This allows us to perform very **quick bounding** sphere **tests**.

If an object passes the fast test, or if it is too large, we do the accurate test.

Generating good occluders

Wed, October 12, 2011

In the final part of the presentation I'd like to explain how we create occluders.

Where to get occluders

SIGGRAPH2011

- Aiming for automated solution
- Originally wanted to use scene geometry
  - Reduced polygon count
  - Too many errors, in general does not work
- Now using physics mesh
  - Closed, low polygon meshes
  - Not always conservative in the right sense
    - Visual mesh can be inside physics mesh causing drops

We didn’t want artists to hand-make occluders
for the entire level, so we looked for an automatic solution.

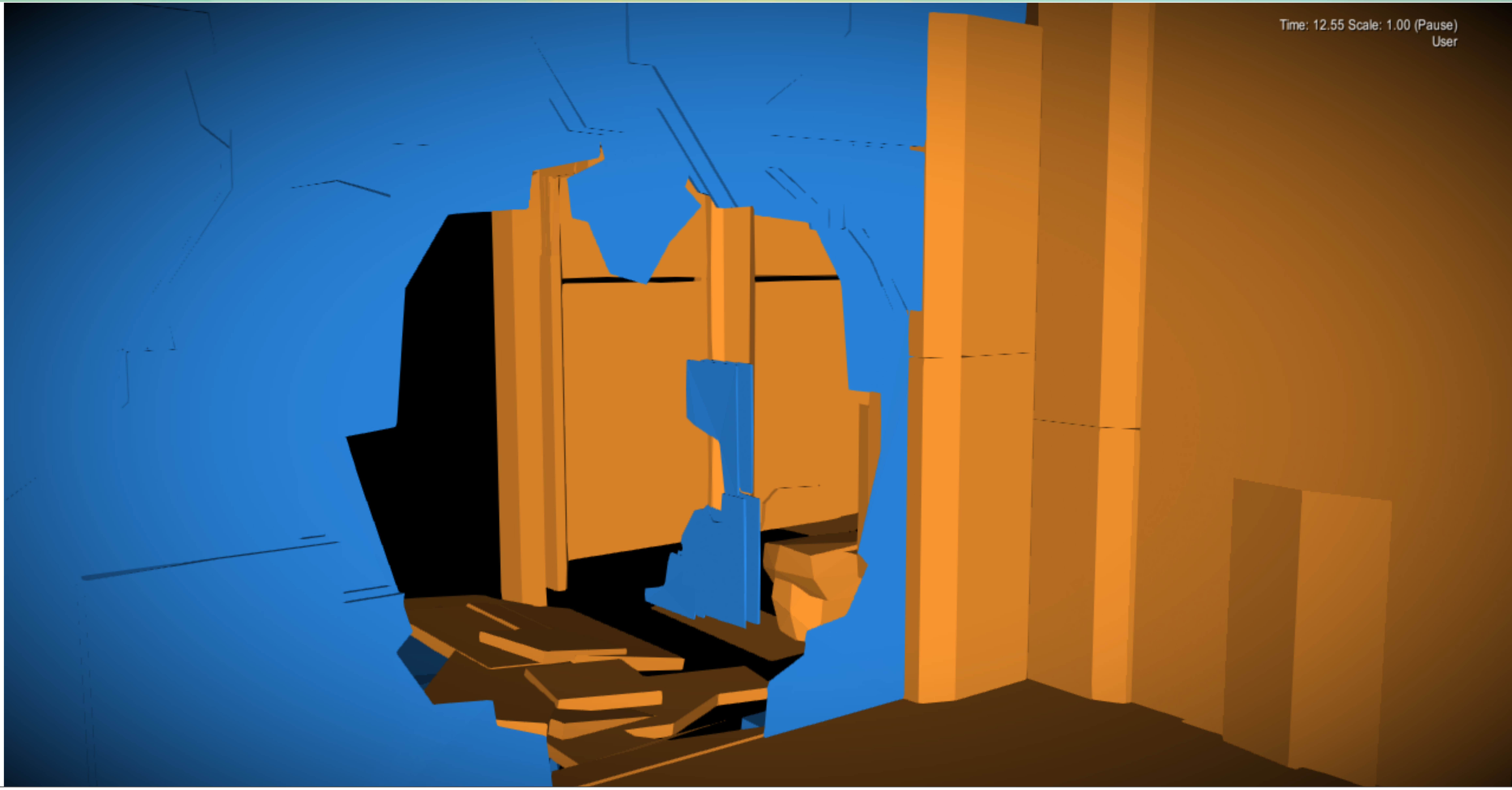
We experimented with using visual meshes,
but the good polygon reduction proved
to be difficult to get right.

Physics mesh is much better choice.
It’s available for each mesh in the game
and it’s cleaned and sufficiently low polygon count.

Unfortunately the physics meshes can be slightly larger
than visual meshes causing objects to disappear.
Worst offenders have to be fixed manually.

Where to get occluders

SIGGRAPH2011



Wed, October 12, 2011

Here's an example of occluders
generated automatically from physics mesh.
You can notice there's some unnecessary detail,
but in general the quality is pretty good.

How to select good occluders

- Simple heuristics to identify good occluders
  - Discard anything which is small
  - Discard by meta data
    - clutter, set dressing, foliage, railings…
  - Discard if surface area is significantly smaller than bounding box surface area
- Artists can override the process
  - Still creating the best occluders by hand

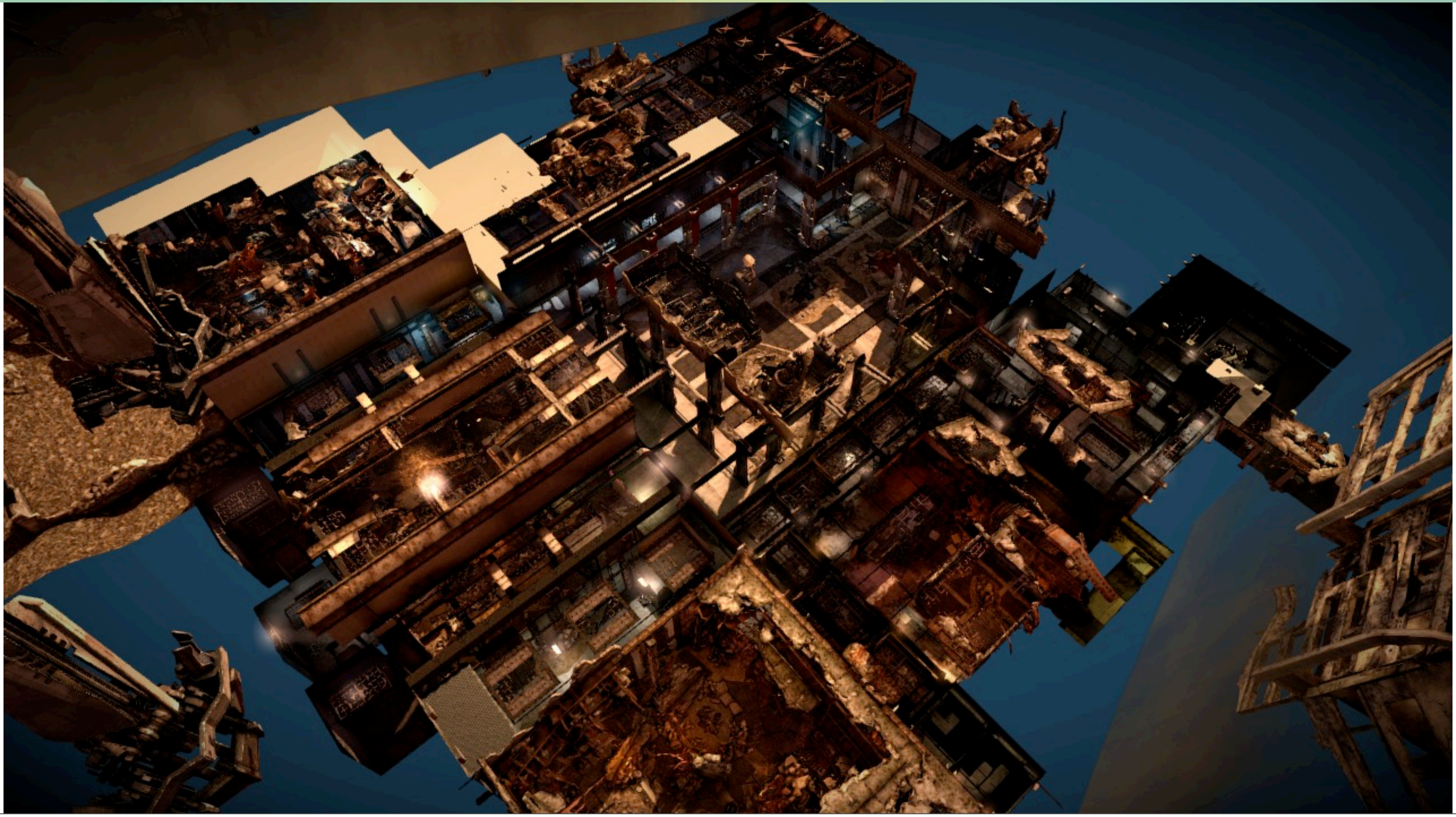
Even using physics mesh there was too much occluder geometry.

We needed to reject occluders that were unlikely to
contribute much to the occlusion buffer.
Our simple heuristics rejects all small objects or
objects where the name suggests that they are not good occluders.

We also reject meshes whose surface area suggests
that they are thin or with too many holes.

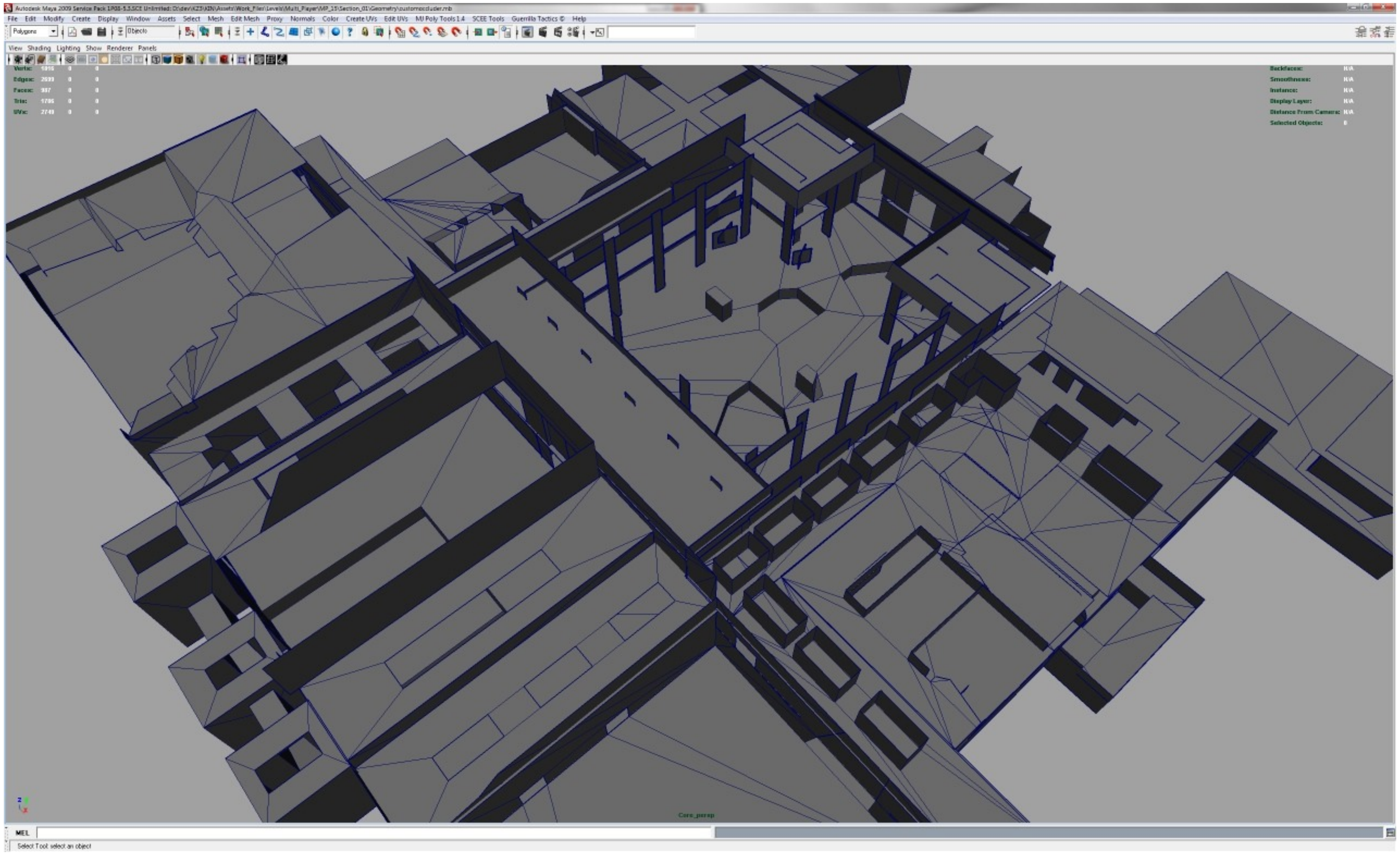
Artists can of course step in and override the heuristics
or provide their custom occluders for difficult cases and optimization.

Artist generated occluders



Here's an example of KZ3 multiplayer level where the automatic heuristics did not work well.

Artist generated occluders



Wed, October 12, 2011

And here’s the highly optimized occluder mesh created by artist.

Conclusion

- Software rasterization is great
  - Fast on SPUs
  - Easy to integrate
  - Very accurate (if occluders are accurate)
- Creating occluders is hard
  - Automatic system was not enough
  - Plan for this in content creation
  - Define workflow for finding and fixing leaks
- Voxelization anyone?

We like this system, it's simple, efficient and fits well with our pipeline.

Unfortunately the content creation proved to be a problem.
The automated solution did not work well enough in some cases
and artists had to create custom occluders for entire levels.
Luckily the geometry is easy to create.

Using scene voxelization might be a good way
to generate simple, robust occluders automatically.

Conclusion

SIGGRAPH2011

- Special thanks to Will Vale (Second Intention Ltd) for implementing this system for us.

Wed, October 12, 2011

I'd like to thank Will Vale for implementing the system for us and help with this presentation.

Statistics

- 100 occluders, 1500 triangles
- Test 1000 objects, 2700 parts
- Timings
  - Setup job: 0.5ms
  - Rasterize job: 2.0ms (on 5 SPUs)
  - Query job: 4.5ms (on 5 SPUs)
  - Overall latency: ~2ms

Questions?