Occlusion

Visibility determination for static and dynamic objects

(Al Hastings, Insomniac Games)

Static Occlusion

- Essentially it is an implementation of PVS (potentially visible set) concept
- From a given camera position, the occlusion database reports which objects are potentially visible
- Only works for static objects (ufrag, ties, shrubs, foliage, etc)
- The run-time is simple – all the complexity comes in generating the database

Dynamic Occlusion

- Implemented using the RSX hardware pixel counters
- Used for dynamic objects (mobys, real-time lights and particle effects)
- Visibility determination is computed based on bounding volumes (spheres, cones, AABBs) and not on the actual geometry
- There is a one frame lag – the PPU uses the results that the RSX computed on the previous frame. This leads to complications.

Static Occlusion Pipeline

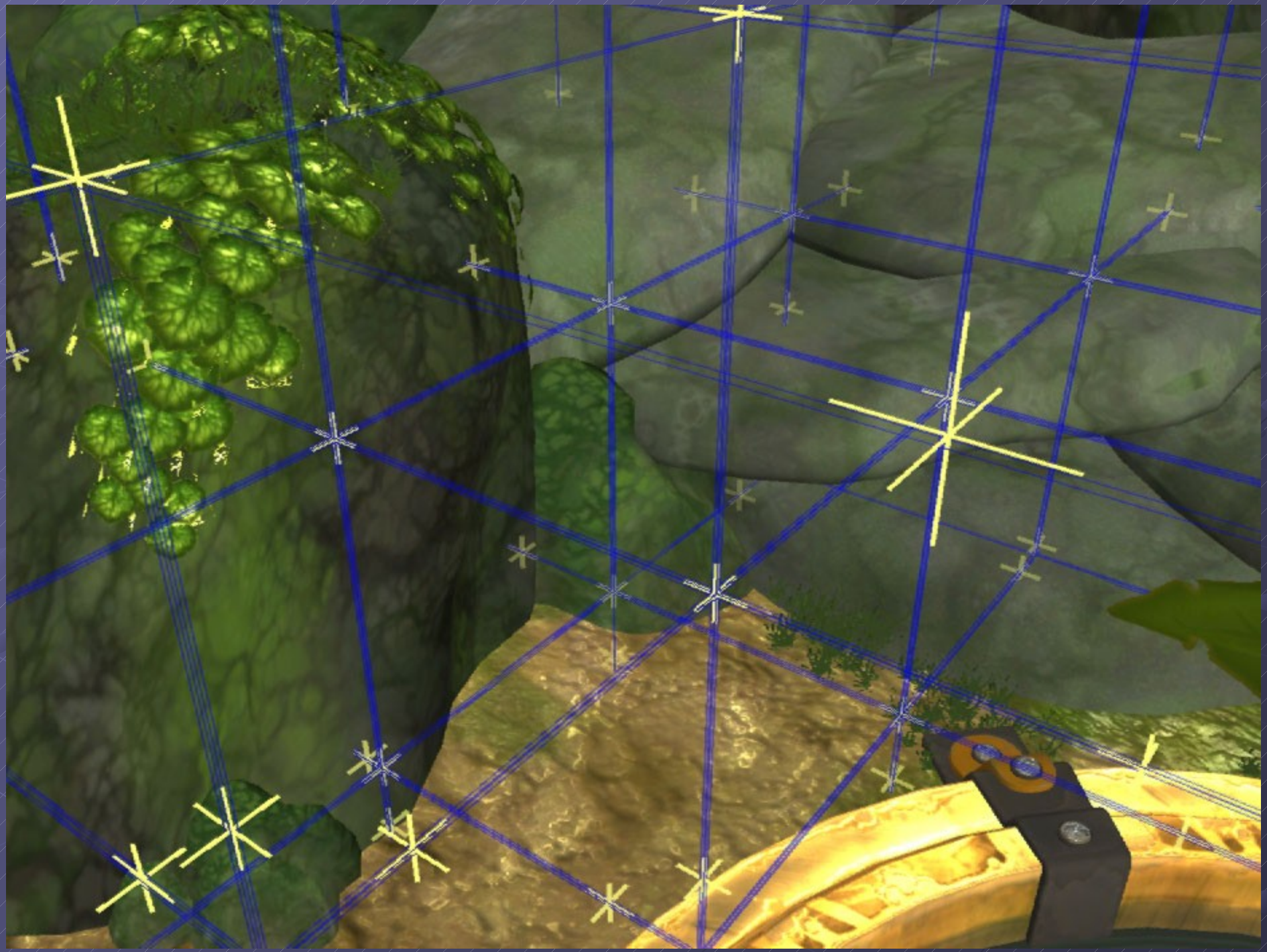
- 1) Paint grids where the camera can go
- 2) Auto-generate sample points from grids
- 3) Run ps3 utility to determine visibility
- 4) Group objects into 2048 “clusters”
- 5) Compress the visibility database
- 6) Match up the database to the latest assets in the level

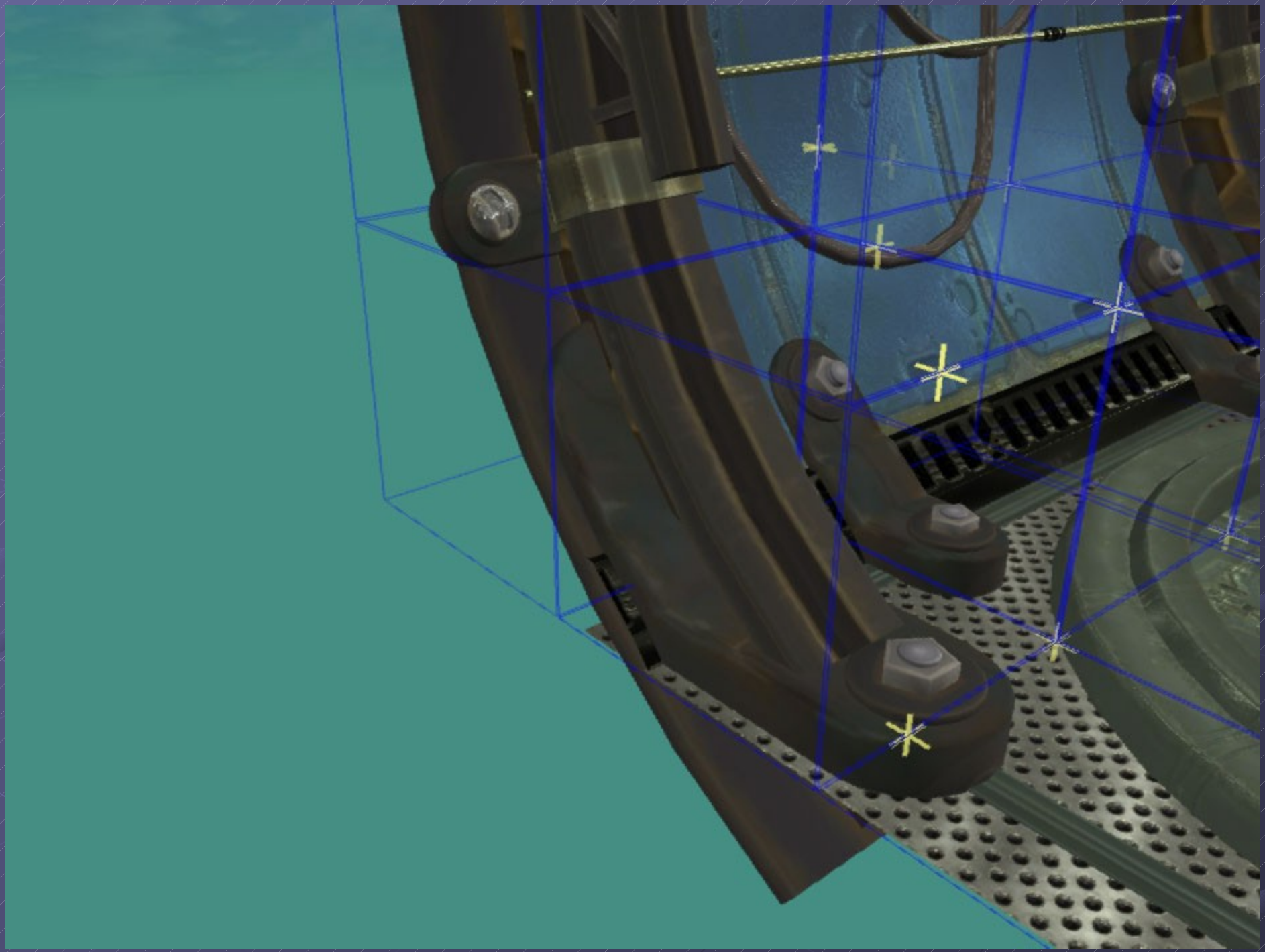
Painting the grids

- Grids are uniform and non-hierarchical
- Usually 2m on a side, sometimes larger
- Typically 20,000 – 100,000 on a level
- Painting is done in-game by running around with the character.
- It's a very manual process – it can take a day or more to populate a large level with grids. And there is ongoing maintenance work as the level changes.
- But the manual process has some advantages:
  - Adaptable to any type of gameplay
  - If there's a problem with the grids, it's easy to find and easy to fix.

Generating the sample points

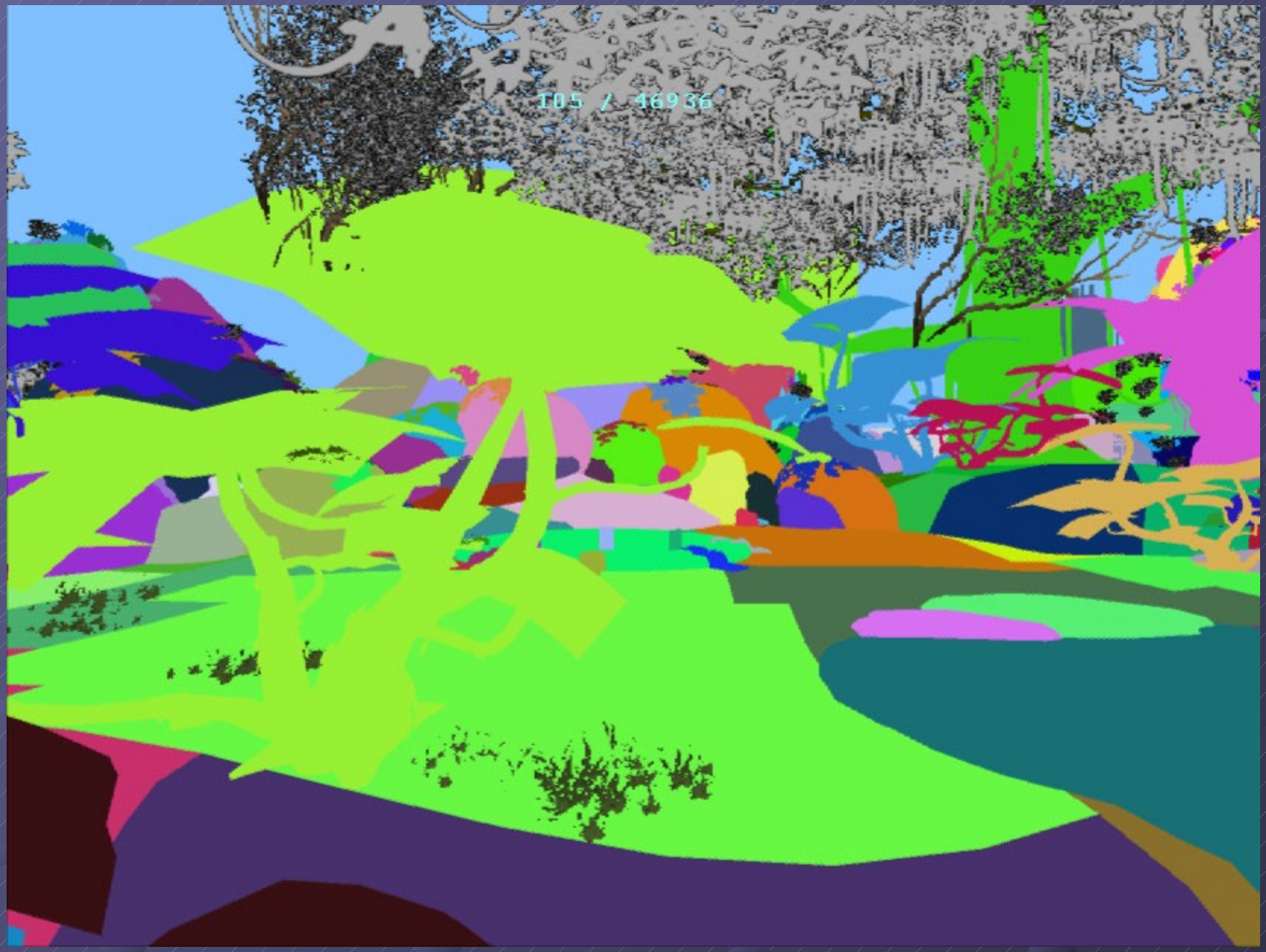
- Automated process on ps3 that runs whenever the grids are edited.
- Uses run-time collision tests to (try to) avoid putting samples inside geometry (which can lean to a lot of over-inclusion)
- Relies on the collision geometry being a close match to the render geometry.
- Attempts to place samples at the corners of each grid, or as close as the collision will allow.
- Typically a level ends up with 1.5x more sample points than grids.





Determining Visibility

- This is done using a ps3 stand-alone app called ‘occlgen’.
- Eventually we could move this back to run on a PC. Speed is a toss-up currently.
- 6 renders are done for each sample point to get 360º visibility.
- Large levels can require upwards of a half million renders. This process could take as long as 10 hours on Resistance.
- End result is a bit-vector for each grid indicating which objects are visible. This data can be several hundred Megs.


105 / 46936

Grouping objects into ‘clusters’

- Processing moves back to the PC now
- 2048 clusters on a level. Typically 4-8 objects in a cluster.
- Clusters are formed from objects that are visible from similar sets of grids. This introduces less error than basing the groupings on spatial proximity.
- This stage can take several more hours (uses a brute-force algorithm which could be improved)
- Clustering introduces over-inclusion error (typically 10% to 20%, but worse on larger levels)

Compressing the database

- At run-time, each grid stores a 256-byte array indicating the on/off visibility for each of the 2048 clusters
- Further compression is done by combining grids with very similar visibility lists.
- This stage can take up to an hour of processing
- This also introduces over-inclusion error. Typically it adds less error than clustering.
- Run-time budget for occlusion data is 2 Megs.

Syncing the database

- Visibility is re-computed once per day at most. But the level art changes much more quickly.
- Occlusion data should decay gracefully as the art changes. But in the past this is the stage that has proven most problematic.
- In principle, adding or moving an object will only invalidate occlusion on that object while keeping it intact on the rest of the level.
- The new tools should make this more robust because we now have proper UIDs for each object.

What's good about it

- Very fast to query at run-time – almost free
- Relatively low memory footprint
- Doesn't impose any restrictions on level construction techniques

What's bad about it

- Labor intensive and time intensive process
- Hard to keep it up-to-date during production
- Can't handle moving geometry, deforming geometry, destructible geometry
- Suffers from some over-inclusion (large ties are the biggest problem, compression error is a secondary problem)
- Monolithic database (hard to stream, etc)

Dynamic Occlusion

- No pre-computed data is involved.
- Potentially visible objects render a bounding volume then check on the next frame if any pixels were visible.
- Things that use dynamic occlusion:
  - Mobys
  - Real-time Lights
  - Effects

Problems caused by one-frame lag

- It’s possible to see a one-frame dropout when an on-screen object becomes un-occluded.
- Special-case code is required to handle camera cuts, mode transitions, etc.
- On-screen, occluded objects need to re-test their visibility every frame.

What's good about it

- Relatively lightweight system (rendering the bounding volumes usually takes a fraction of a millisecond)
- Works with any level design
- Typically it manages to cull that vast majority of objects that should be culled.

What's bad about it

- Basing occlusion on bounding volume visibility works poorly for some objects (long, thin mobys in particular)
- One frame lag issues
- Visibility spikes can occur after camera cuts and mode transitions

Future Work

- Explore using RSX conditional rendering to remove one-frame lag in certain situations.
- Extend system to work well for large-scale moving objects (use actual geometry for visibility test instead of bounding volume)
- Avoid visibility spikes with better handling of mode transitions and camera cuts

Questions?