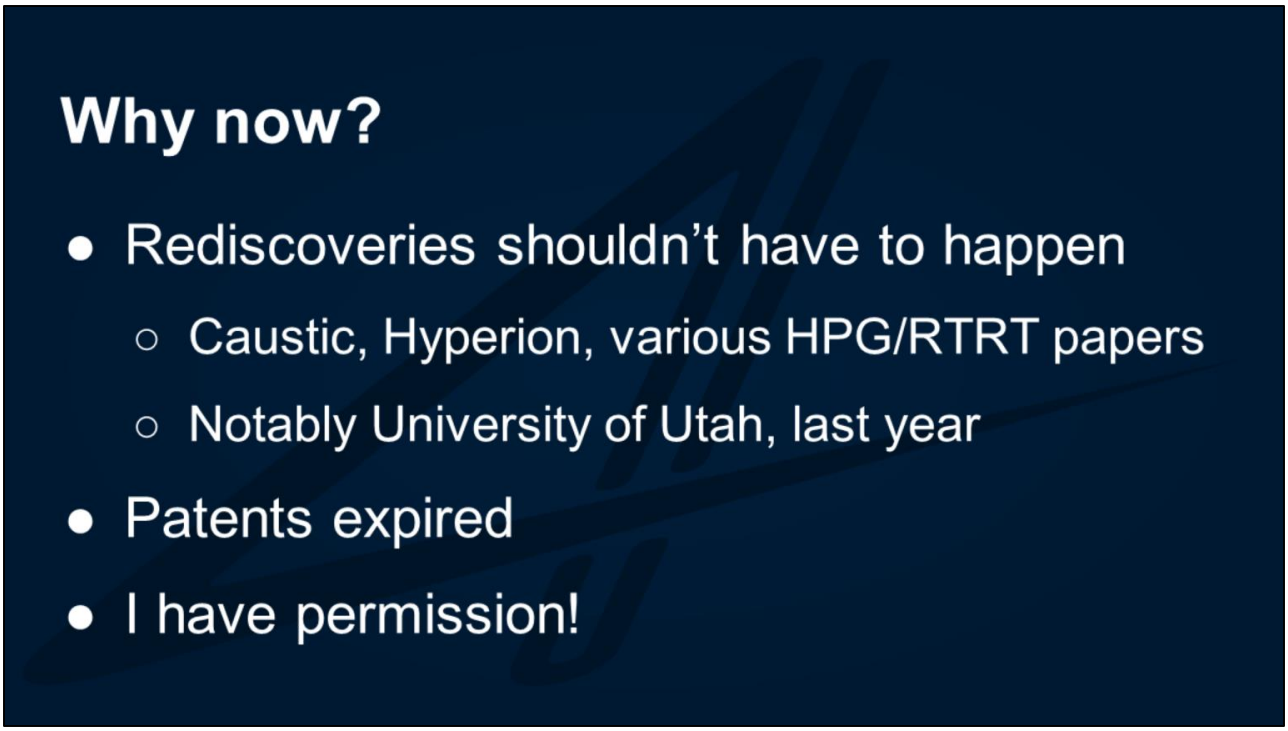
HPG and Hot3D are normally about the latest and greatest hardware.
This isn't that – this is about some hardware from before NVIDIA started calling their products GPUs.



Why is this being presented at HPG?
In many recent conferences, someone has presented a good rendering technique, and I've gone up and said it was interesting, but did they know we did it some years before?
An obviously they didn't, because we didn't publish it.
We can't be alone in this (although others might not actually tell the presenters), so this is a call for people to describe old technology as the patents start expiring and people stop caring about protecting the IP.

Advanced Rendering Technology

- Founded 1995
  - Based on Adrian Wrigley's PhD

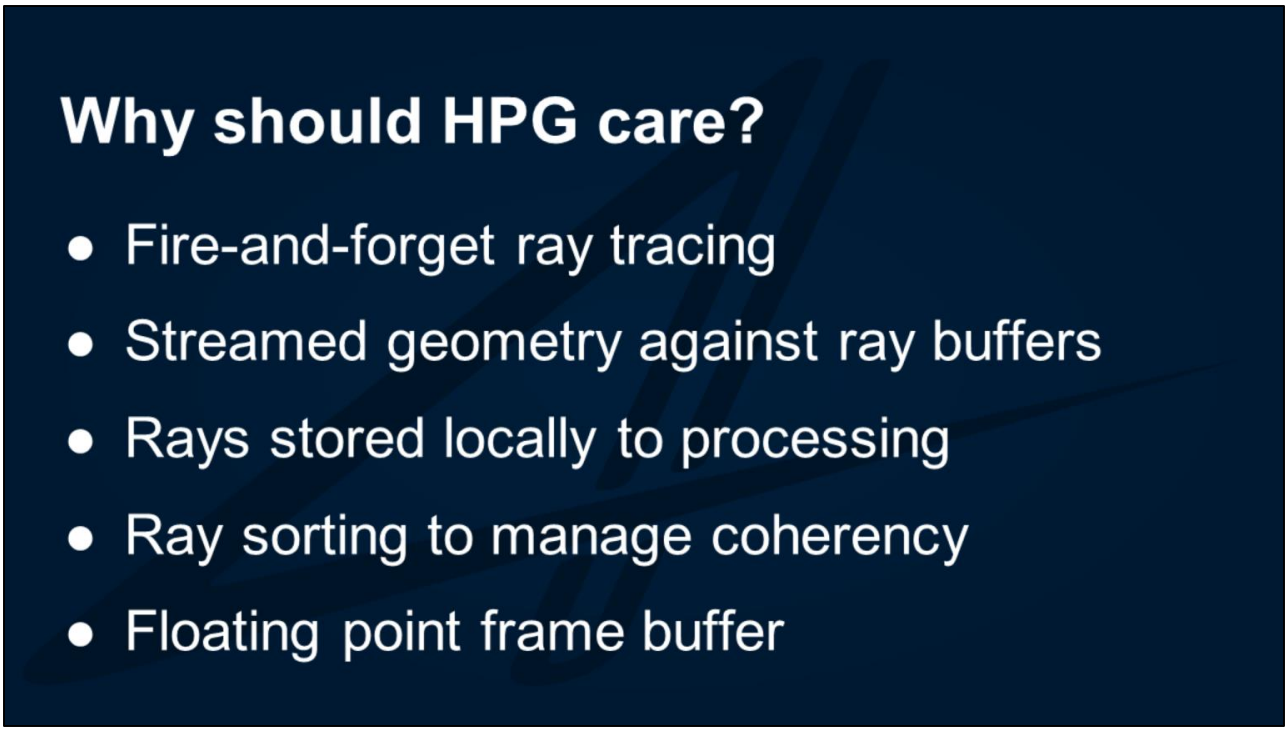
- Networked "Render farm in a 4U rack"
  - Custom ray tracing ASIC (AR250, AR350)
- Based on RenderMan (RIB, RSL)
  - Plug-ins for 3D Studio Max, Maya

ART was founded in Cambridge, England in 1995, based on Adrian Wrigley's 1994 PhD thesis in the Rainbow Graphics Group in the computer laboratory.
(See http://www.lib.cam.ac.uk/collections/departments/manuscripts-university-archives/subject-guides/theses-dissertations-and-1 for information on ordering the thesis; it does not appear to be online. The corresponding patent is available at https://patents.google.com/patent/US5933146A/en.)
The idea was to make a rendering appliance that could be used like a render farm (and shared between artists).
Absent anything else resembling a standard for photorealistic ray tracing, it was based on the RenderMan specification.

Customers

- Dyson (transparent plastic)
- Car firms (lighting clusters)
- Game covers, jewellery
- Digital Dimension (Superbowl XXXIII)
- Nearly Jim Henson Studios (Farscape)

Obviously not enough customers existed to make the company highly successful, but those who were interested included those who could use ray tracing on transparent surfaces or liked depth of field.
Jim Henson didn't buy one, but caused us a lot of confusion as to how we could be used for Fraggles until Farscape came out.



Why was ART's hardware relevant to HPG?
HPG and the RTRT conference have discussed fire-and-forget tracing, streaming geometry over rays stored locally, sorting rays for coherency, etc.
One oddity was the existence of a floating point frame buffer (on the market six months before a patent got filed for it, which caused some difficulties to the graphics industry).

What's a RenderDrive?

RD2000 (1998)

- Network-attached dedicated rendering computer
- 100Mbit Ethernet connection
- Driven from PC (etc.)
- MSRP: $19,995 USD

The RenderDrive 2000 (first product) was launched in 1998 as a networked ”rendering appliance”, as a 4U rackmount device.
In the style of a render farm, it was intended to be used by multiple artists, connected over an ethernet connection via RenderPipe software (typically from a Windows NT PC).
Launch price was roughly $20,000 – adjusting for inflation, roughly $30,000 by today's talk.

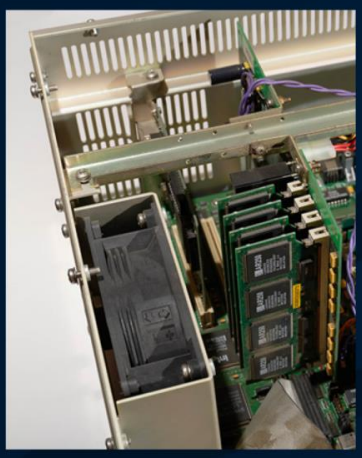
What's *in* a RenderDrive?

Host computer:

- DEC Alpha 21164
- 768MB-1.5GB RAM
- 4GB SCSI drive for images
- 40MB PATA SSD for boot

Inside, there's effectively an AlphaStation motherboard with a 500MHz DEC 21164 CPU, and either 768MB or 1.5GB of RAM.
The reason for the Alpha was to allow enough room for multi-million polygon geometry sets with complex shaders – at the time a PC workstation might have had 32MB of RAM.
The 4GB HDD was actually 4GB (not 4,000,000 bytes) since it predated the convention of making disks look larger by rounding.
There was a 40MB boot SSD – the system actually booted minimally and installed in RAM, then booted there, so it could be turned on more quickly.

What's *in* a RenderDrive?

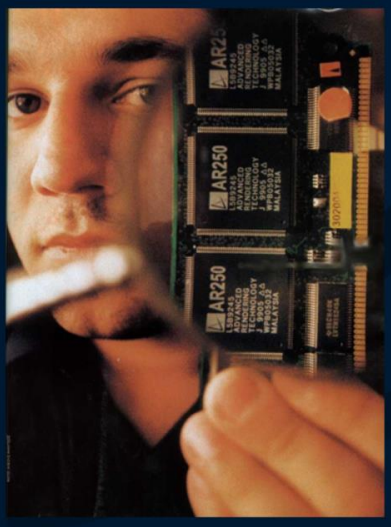


Custom hardware:

- PCI-X daughterboard
- "RD4" : 1 SIMM / 4 AR250 (5.1 GFlops)
- "RD16" : 4 SIMM / 16 AR250 (20.4 GFlops)

The more interesting part of the contents is the custom hardware.
There's a PCI-X daughterboard – PCI-X being a 66MHz, 64-bit version of PCI, not PCI-e.
Plugged into that via some repurposed SIMM sockets were 1-4 cards, each of which contain four custom AR250 chips.

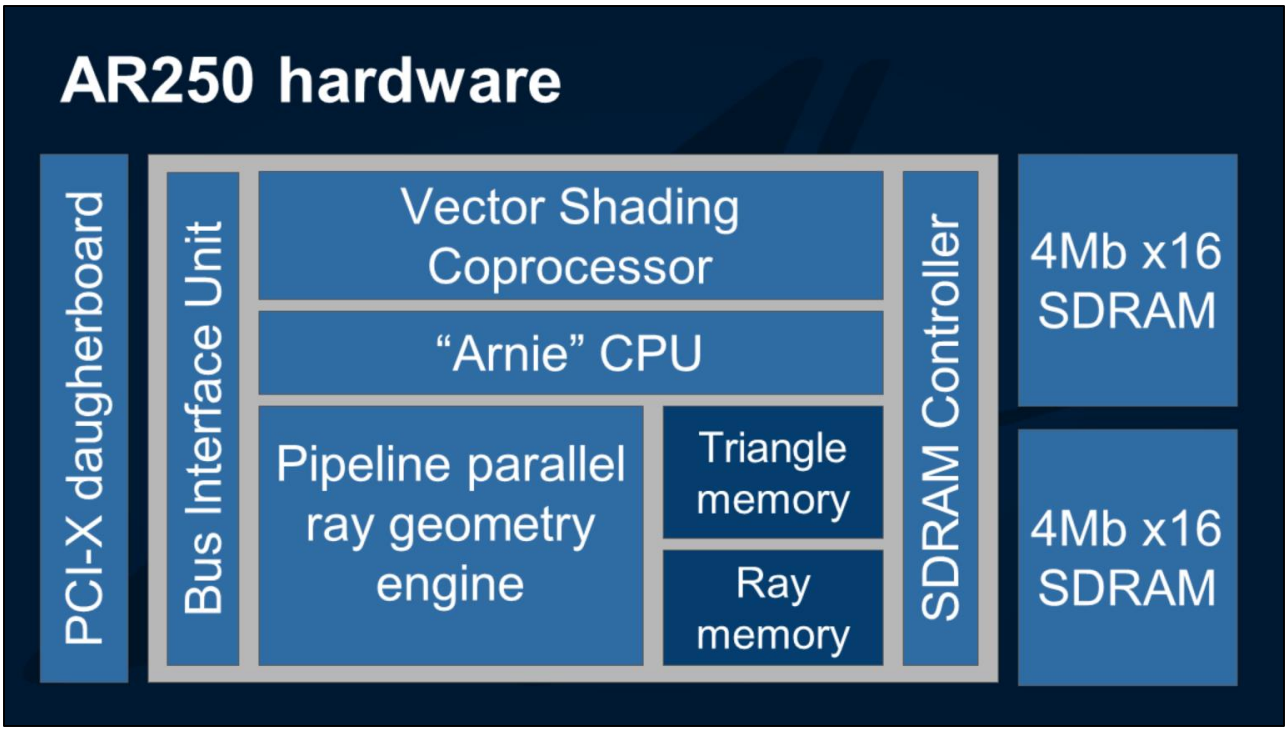
AR250 4-SIMM Module



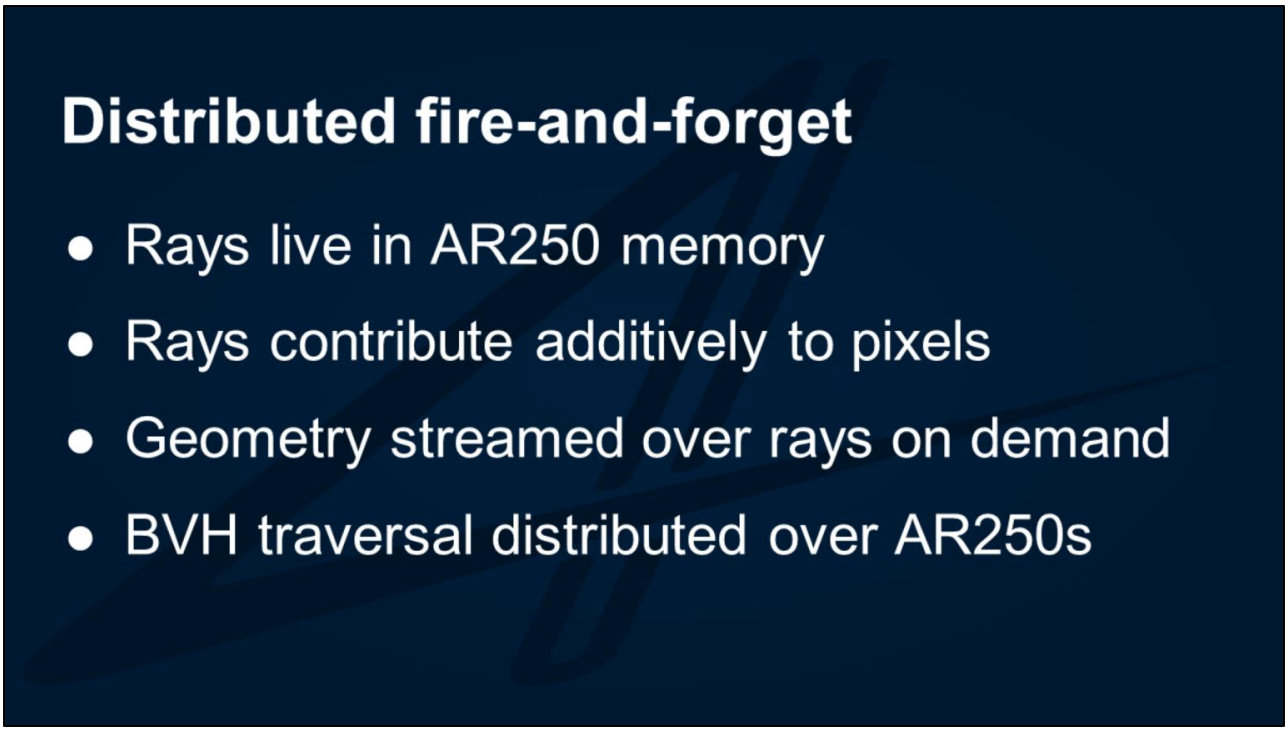
AR250	AR250	AR250	AR250
RAM	RAM	RAM	RAM
RAM	RAM	RAM	RAM

- 4 AR250s per SIMM
  - 16MB SDRAM per AR250
  - Blinkenlights!

Each board with four AR250s had 16MB of 32-bit SDRAM attached to each AR250 (as 2x16 bit chips).
Importantly there were LEDs, which simplified debugging – when the flashing lights stopped, the chip had crashed.



Inside the AR250 was a custom Arnie CPU core.
This was originally to be an ARM7, then moved to an in-house design apparently because it simplified the coprocessor interface; the instruction set was still very similar to ARM, including conditional instructions.
There was a SIMD shading coprocessor (of which more later), and a custom ray intersection coprocessor.
The chip ran at 40MHz, and included 32 IEEE754 units, giving a peak of 1.2GFLOPS.
Originally the numerical representation was to be a log format, on the basis that multiplies and power operations were common – adds and subtracts would incorporate a log/exponentiation step. This was eventually seen as too costly, but is still mentioned in ART’s patents.



The rendering model was distributed fire and forget:
Rays were stored only in the SDRAM attached to the AR250s.
Rays have a weighting and calculated a value to be accumulated into pixels.
Geometry is streamed over the rays by the AR250s, and the BVH traversal is streamed out to the AR250s from the host processor – only the Alpha held the full scene.

AR250 ray intersection

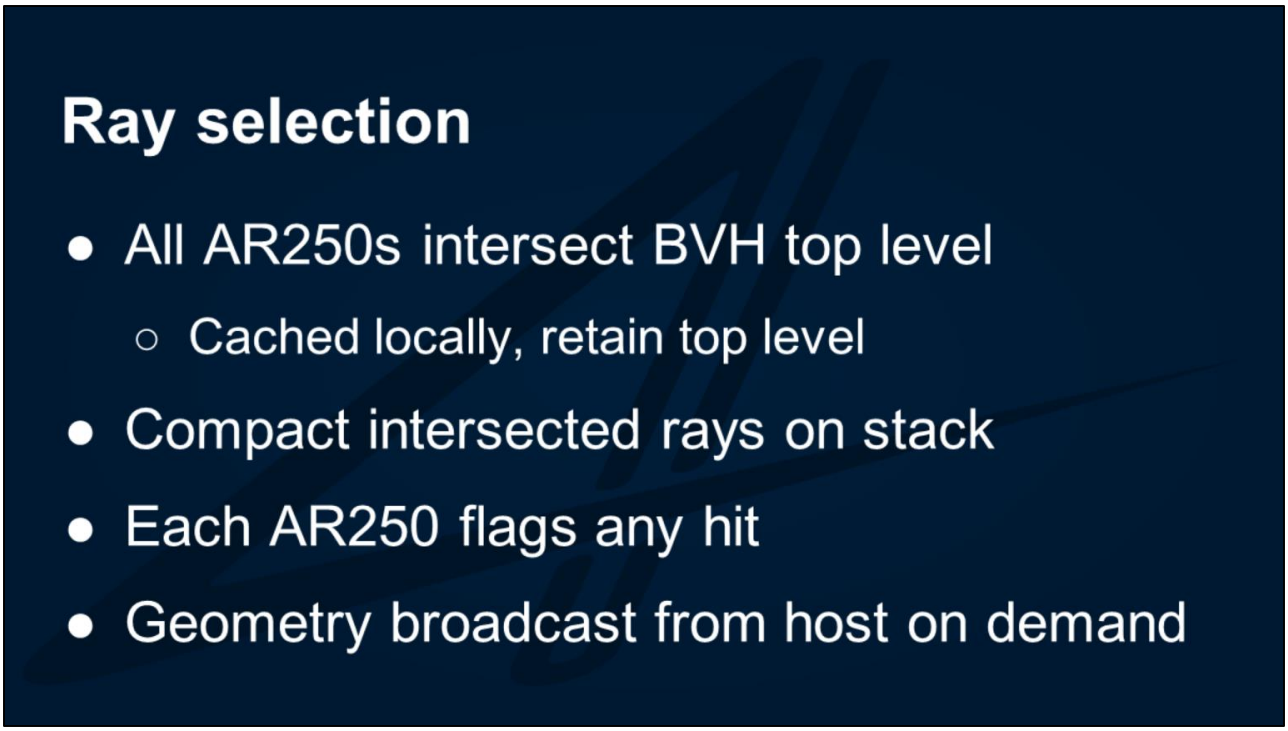
- Rays checked against 6 primitives in //
- Triangle (geometry)
- Parallogram (BVH)
- Segment (useless)

The ray intersection unit could intersect rays with 6 primitives at a time, deeply pipelined – one result per cycle.
The hardware could treat the three vertices as a triangle (for final geometry), a parallelogram (used to define a bounding box for spatial subdivision), or an elliptical segment – which was unused.

AR250 ray representation

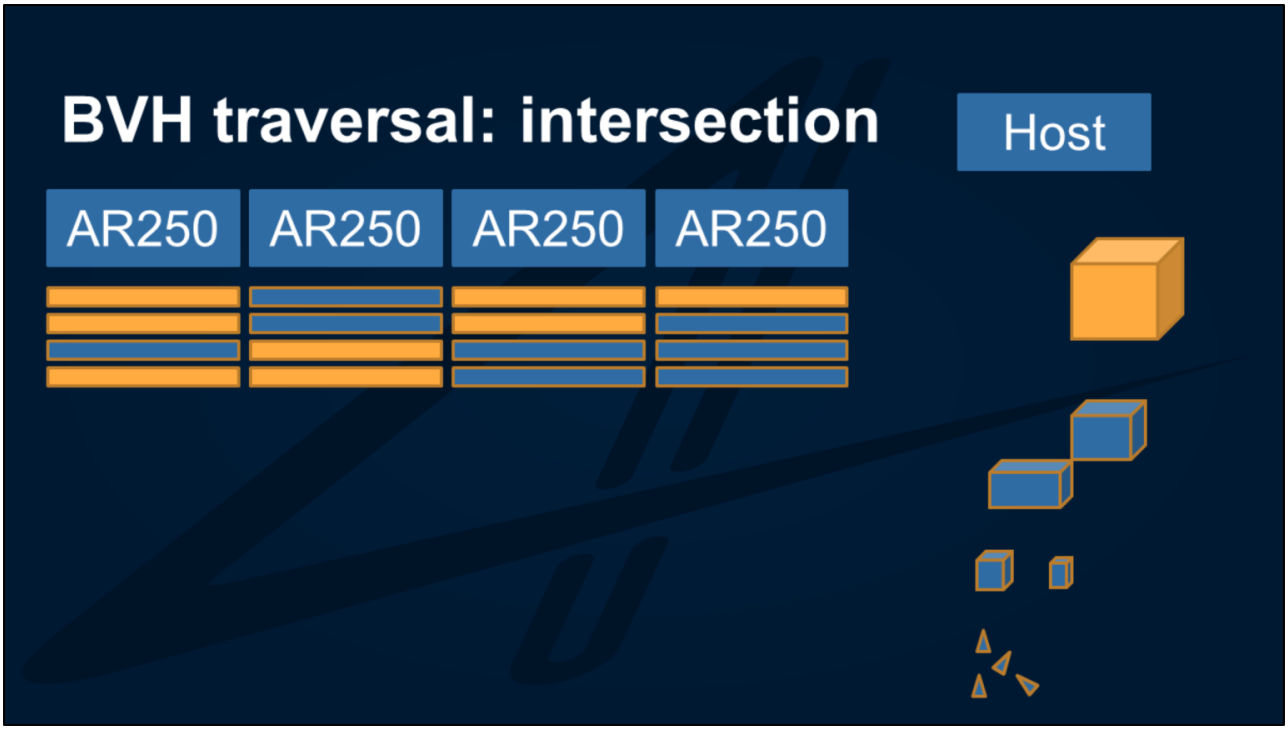
- Ray id (origin pixel)
- Colour weighting (RGB)
- Vector origin, direction
- Float ***lambda***, ***hitU***, ***hitV***
- Id ***hitSurface***, fromSurface

AR250 rays held an ID, which connected the ray back to the originating pixel.
There was a colour (float 3-tuple) ray weight, and a vector origin and ray direction.
Lambda, U, V and the id of the surface hit were set during intersection.
There was a ”from surface” identifier to limit self-intersection.
Rays also had an arbitrary payload used during shading – the shading process didn’t have access to the geometry, so uniform and varying parameters attached to the surface needed to be stored in the ray at intersection time.

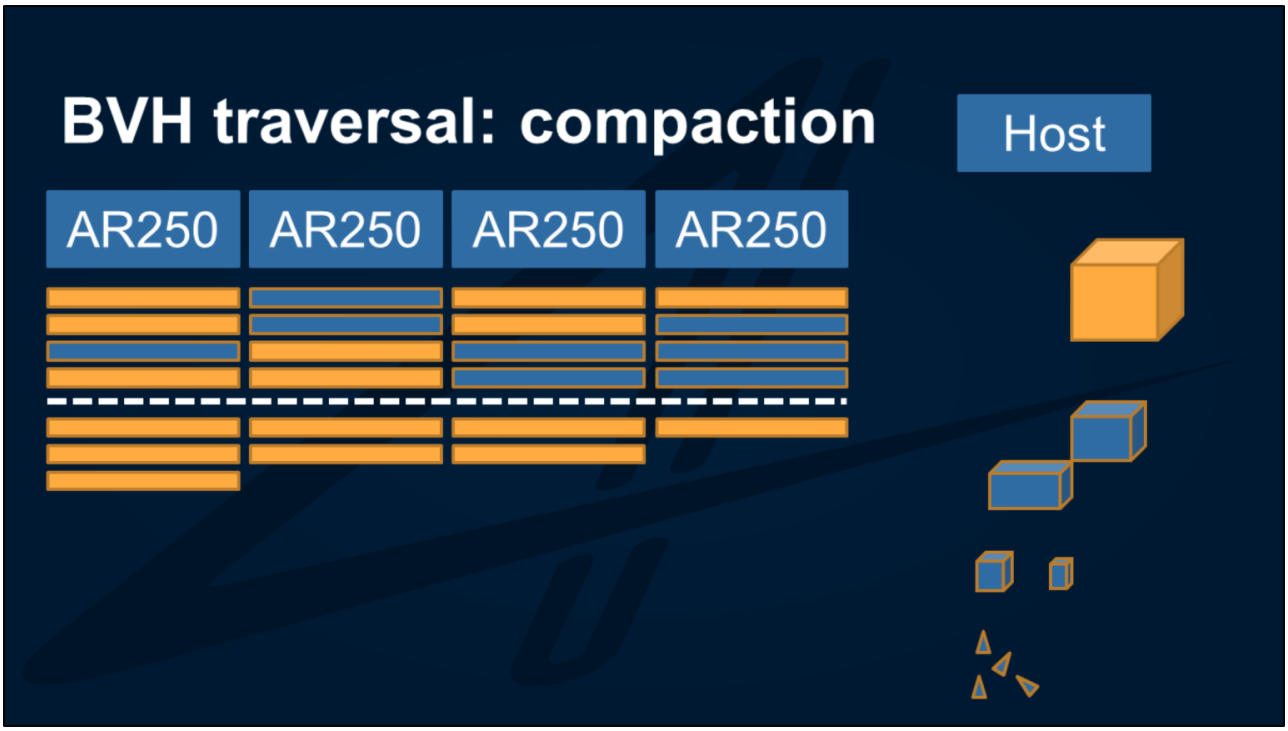


During processing, the rays stored on the AR250s were run against the top level of the BVH.
The top level was typically kept cached, since all new rays would have to intersect against it.
After intersection, the rays which hit are collected on a stack.
Each AR250 which had a hit would flag this, with the results being multiplexed for host access, and the host processor would broadcast geometry and the BVH only if any AR250s still had rays which hit it.

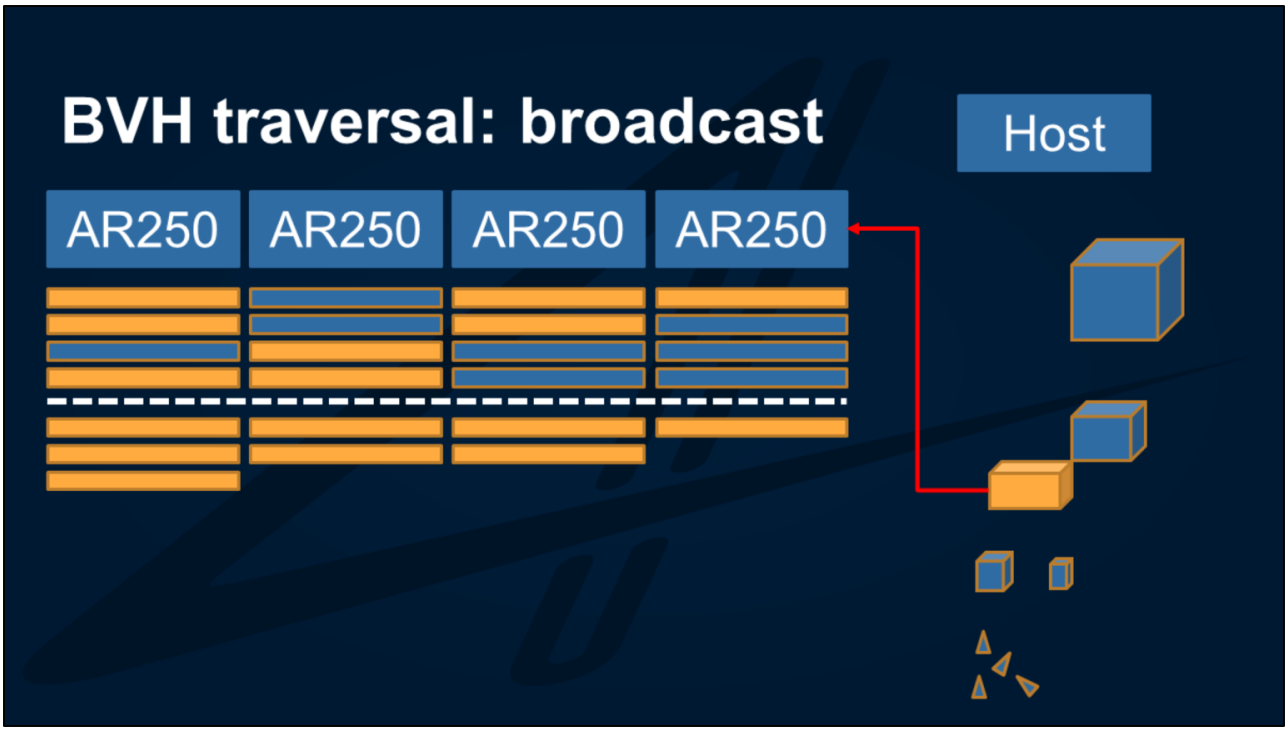
In more detail: All rays on the AR250s (orange) are intersected with top level of the BVH (orange).



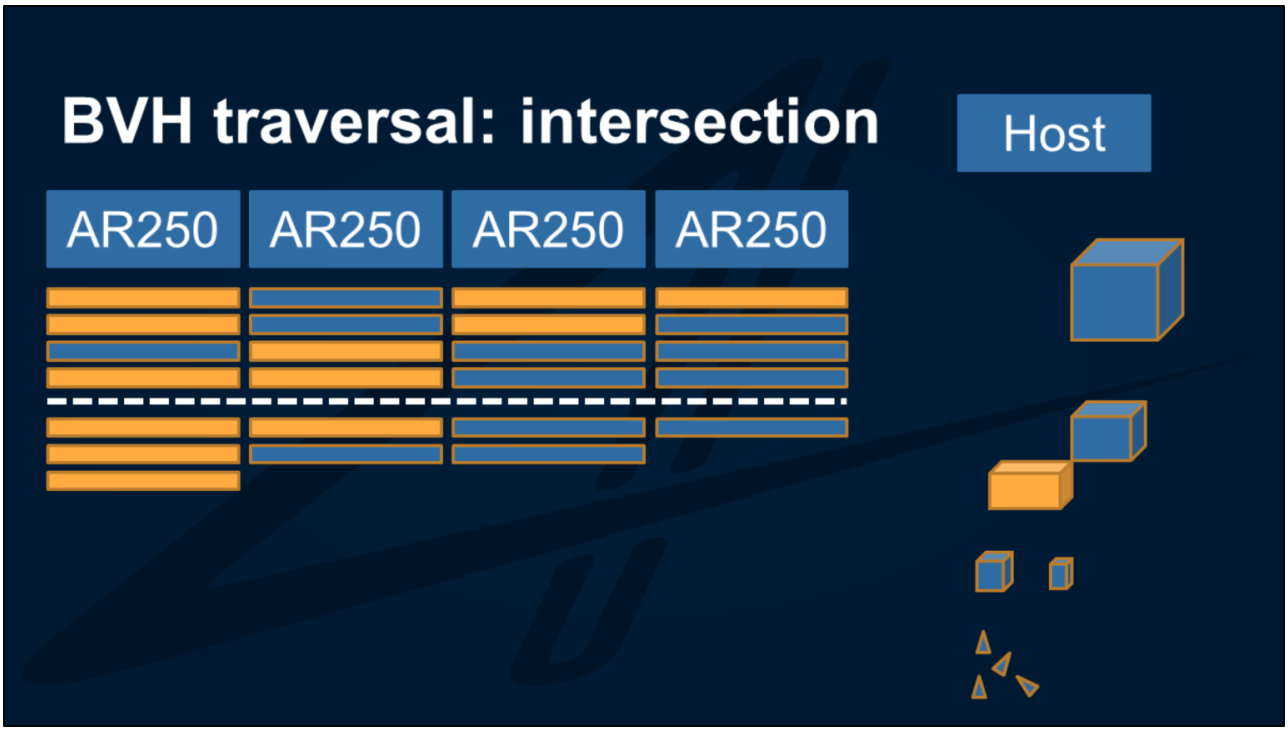
The AR250s intersect the top level geometry with their local rays – some of which miss (blue).



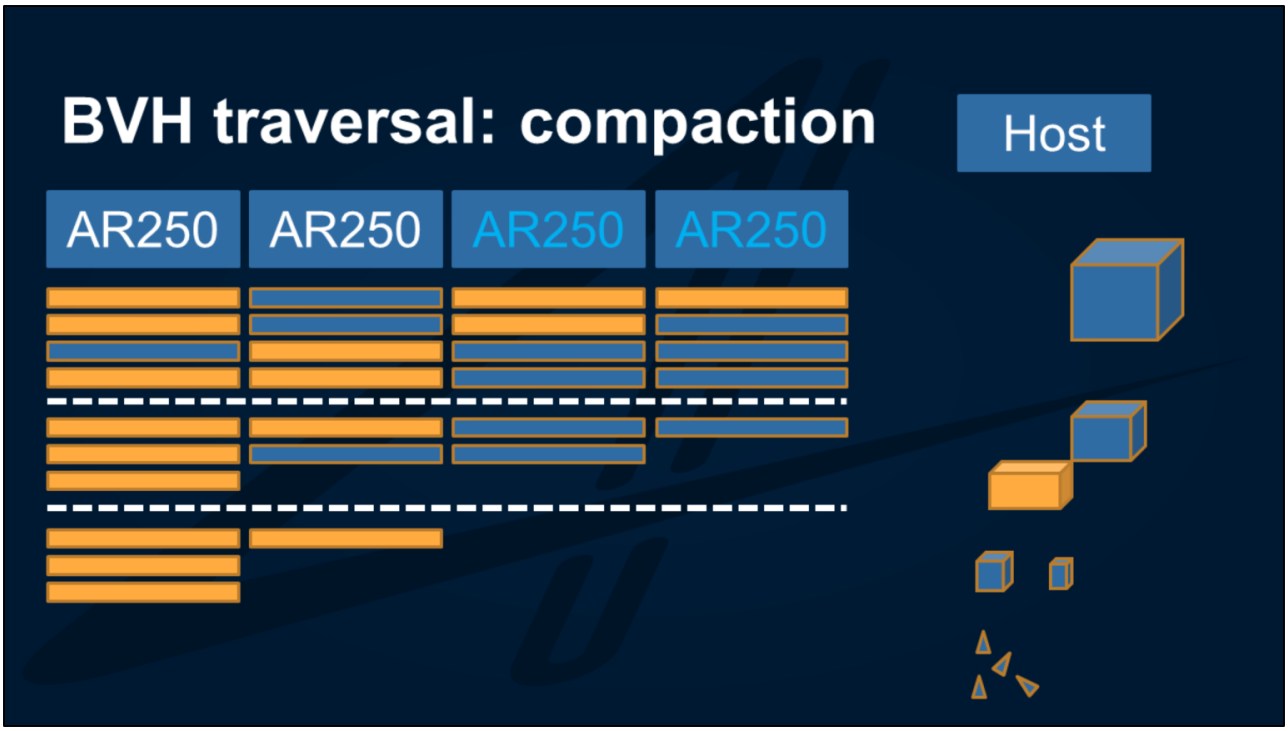
Rays which hit are collated.
Unfortunately, I haven't been able to confirm whether the ray structures themselves were copied (which would improve memory coherency) or whether there was indirection via a pointer (which would have made the copy mechanism faster).



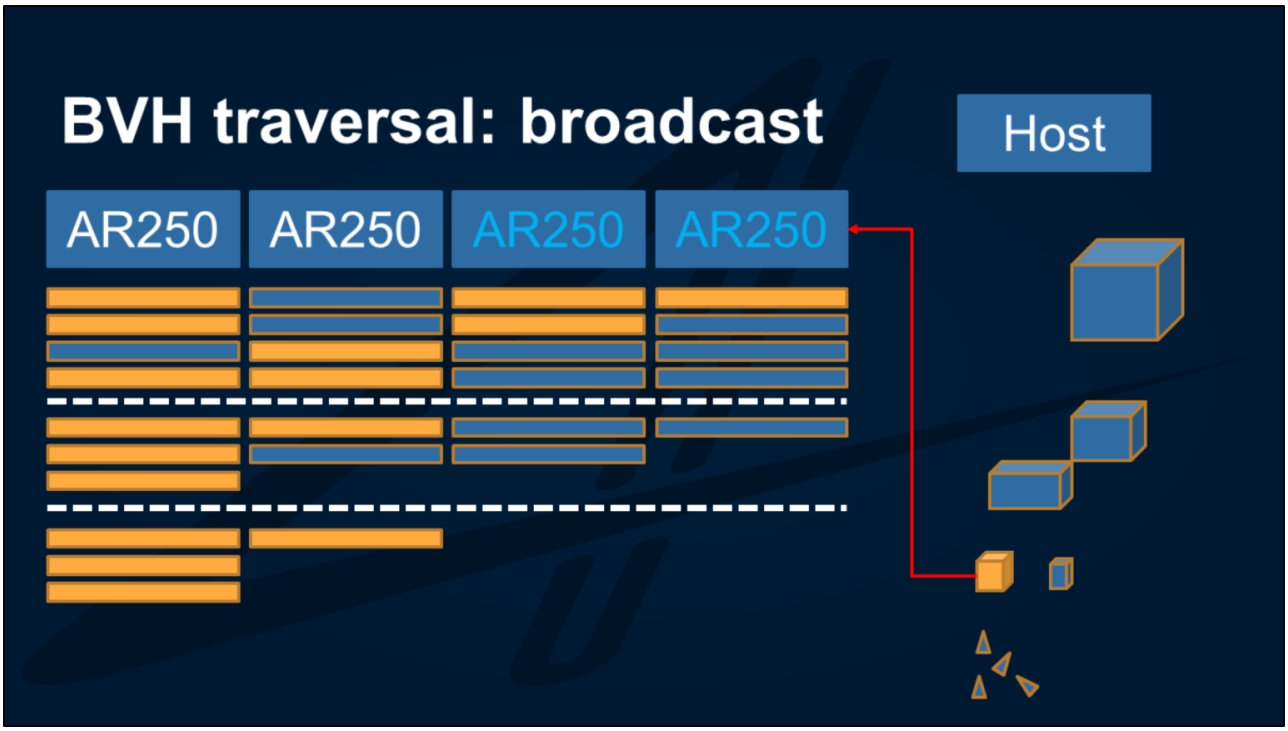
Since the AR250s report hits, geometry for the next level of the BVH is broadcast from the host to the AR250s.



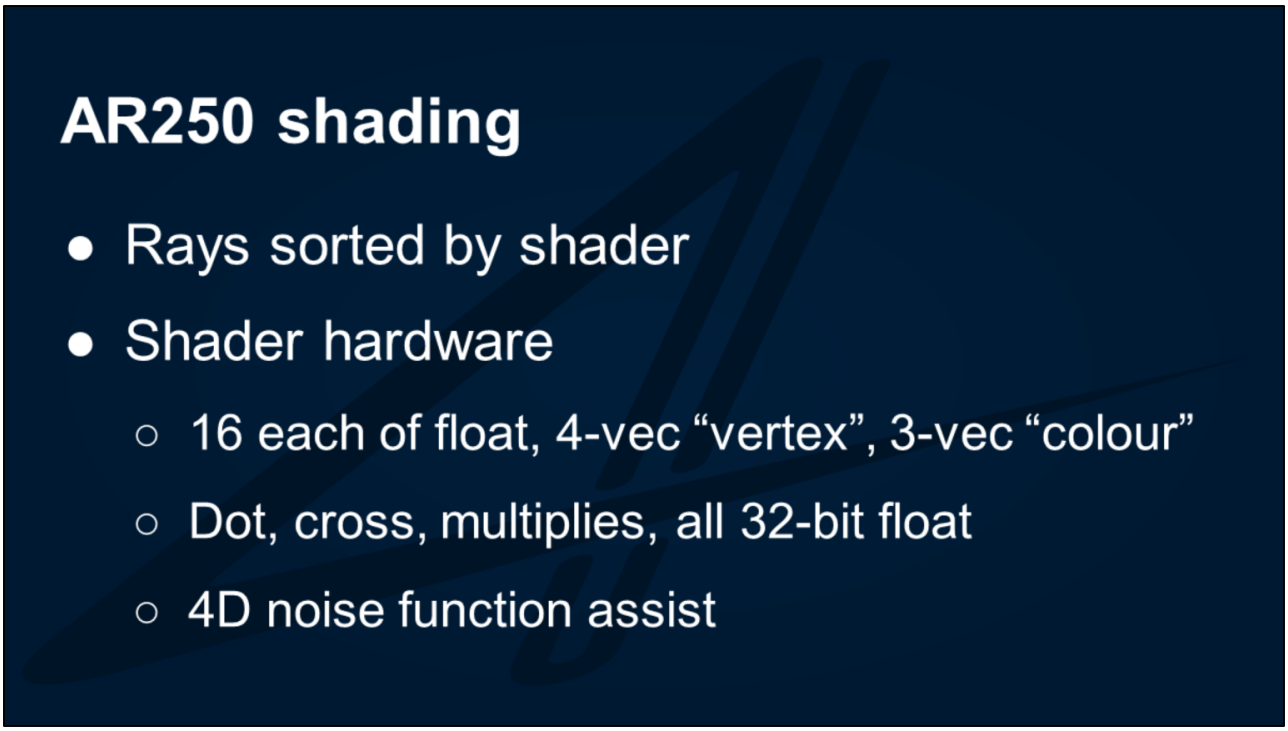
The live set of rays is intersected against the new geometry – on this occasion, only the first two AR250s contain rays which hit.



The live rays are again collated. On this occasion, two of the AR250s have nothing to do.



Two of the AR250s are still active, so the next level of the BVH is broadcast.
If none of the AR250s were active, BVH traversal would stop at this point, and we would move on to the next unvisited node.



For shading, the rays were sorted by shader – so the same shader could be run over a consecutive sequence of rays.
Shading progressed one ray at a time on each AR250.
The shader acceleration hardware consisted of 16 registers each of 32-bit float, 4x32-bit vector and 3x32-bit colour.
Instruction set varied by register – you could “dot” a vector or colour but not a float, you could “cross” only a vector, etc.
Unusually, there was a helper opcode for evaluating Perlin noise in up to 4 dimensions.

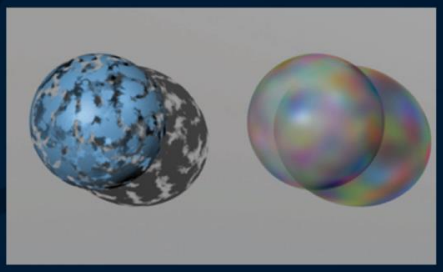
AR250 assembler

```
@@ Nab = dPda ^ dPdb
@@ Np = dPdu ^ dPdv = (1/det) * Nab
@@ Ng = orientation * Np = (1/abs(det)) * Nab
@@ orientation = sign(1/det) = sign (det)
@@ det = 0 => use Ng = Nab (doesn't help calculatenormal)
vcross   vr3, surface_dpda, surface_dpdb  @ Nab
fabs     fr0, fr5             @ abs(det)
vscali   surface_geo_normal, vr3, fr0
fsign    fr0, fr5             @ orientation
adrl     ir0, sl_orientation @ used by calculatenormalaux
sl_fstrb fr0, 0, ir0
```

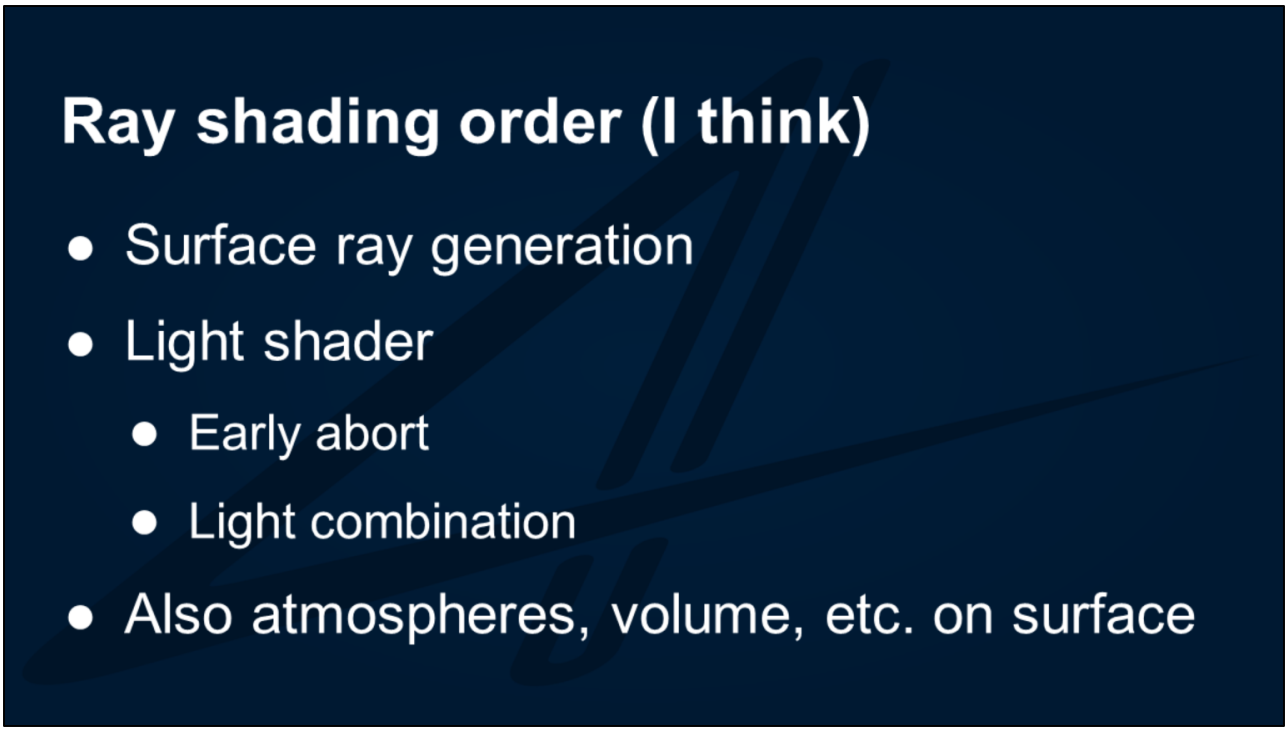
Here’s some example AR250 code.
Note that there are vector cross and scale instructions, and float abs and sign instructions.
Colour operations were similarly prefixed by ”c” (cadd, cmul, etc.).

AR250 shader outputs

- RGB Opacity
  - Shadow / transparency
- Emission
- Illuminance callback
- Framework light management



What were these shaders evaluating?
A non-obvious thing is the opacity of the surface (which is an explicit output of a RenderMan shader).
This was used both for transparency (via a continuation ray – continuing in the same direction as the current incoming ray) and for shadows.
I don't recall there being an "any hit" optimisation for shadows; I suspect shaders were the same for shadow and conventional rays, but secondary ray generation for shadow rays was suppressed.
Output also have to include an emission amount that contributes directly to the pixel value and a callback for illuminance (which generated shadow rays – more on this later).
Lights were handled explicitly by the framework, so individual shaders didn't need hard-wired knowledge of lights.



I believe the shading order was that the surface shader would generate rays, including light rays.
Light rays would then get shaded by the light shader, which meant that they could be culled if outside a spotlight cone, if too dim due to distance, or blocked by a projector.
Light rays could be combined if there were many low-contribution lights in roughly the same direction.
Other shader types such as atmosphere and volume shaders were effectively run alongside surface shading.

Post-processing

- Adaptive sampling/quality
  - ¼ rpp first pass
- Ray buffer kept full
- Lens flare and colour management
- Compositing

The host processor was responsible for ray management.
Ray processing used adaptive sampling, throwing more rays if a discontinuity was detected.
Initial and final quality were configurable; at the lowest quality, one ray was emitted by default every four pixels, relying on the adaptive sampling to increase resolution.
The ray buffer was kept full – if an AR250 was running low on rays, more work could be dispatched.
Processing was one tile at a time, mostly to improve texturing performance during shading.
The Alpha was responsible for lens flare (using the floating point format) – there's a patent on emulating the effect of lenses, including the human iris.
Compositing was supported with rays that missed geometry (including with partial transparency).

API: Geometry

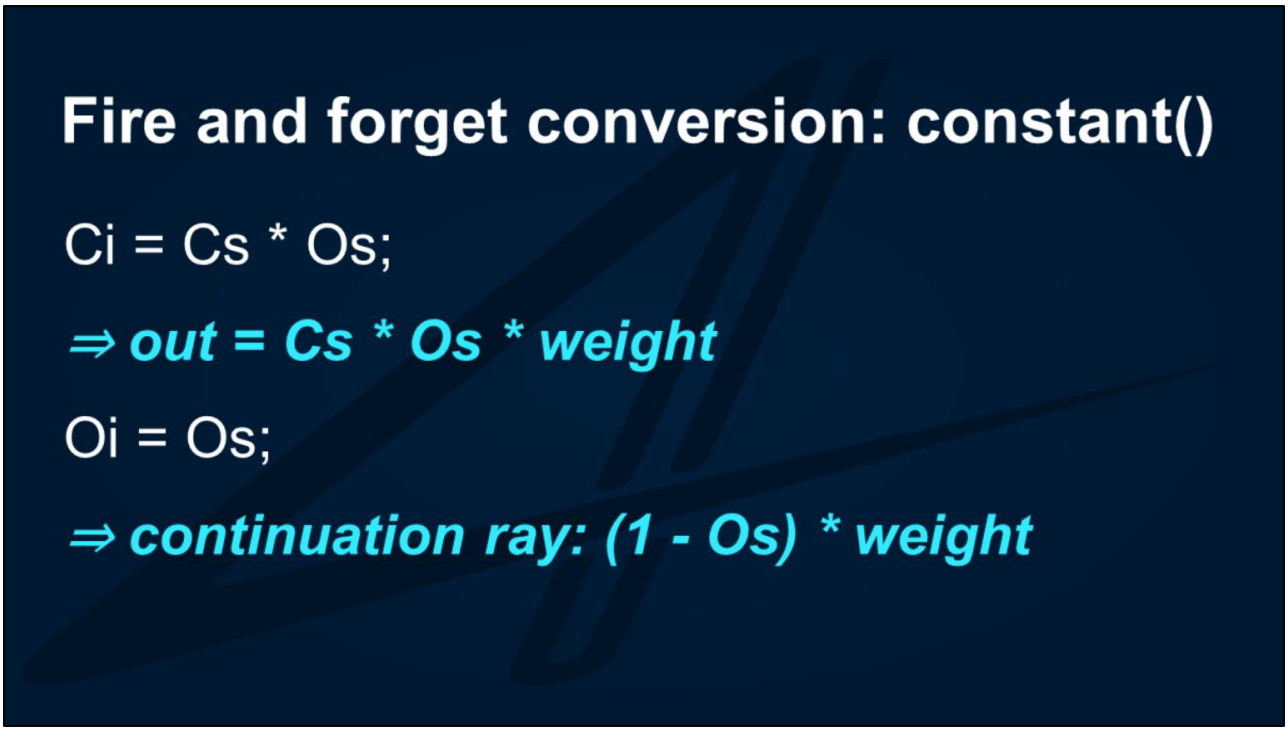
- RIB
  - Displace/transform (somehow)
- BVH management, dynamic tessellation
- Animation interpolation
  - Ray time ranges

Geometry was submitted to the RenderDrive using a RIB file, which was compressed over the ethernet interface.
The RenderDrive supported displacement and transformation shaders. I have no recollection of this being done using the AR250s, and the transformed geometry data would have needed to be back on the Alpha in order to generate the BVH, so it's currently a mystery how this was supported (but it was).
Plug-ins for 3DS Max and Maya sent triangles without further shading, post-tessellation – this was more precise than trying to emulate higher-order surfaces.
The BVH was built on the Alpha (with no custom hardware), using viewpoint-aware tessellation.
Animation was supported in two ways: Rays had a time field, and these could be used to intersect instances of geometry which were moved in the style of an accumulation buffer.
A moving camera was supported directly, generating rays along the path of the camera.

Software: Shaders

- RenderMan's nested shaders
  - Surface, atmosphere/volume, etc.
- Fire and forget conversion
- Illuminance conversion

Non-geometric shaders were run on the AR250s.
RenderMan's nested surface model including atmosphere and volume shaders were supported.
RenderMan's shading model looks like a recursive traversal, so transformations were needed to support the fire-and-forget shading model and to separate illuminance.



Starting with a simple constant shader:
In RSL, Ci is output surface colour, Cs is input (default) surface colour, Oi is output surface opacity (a colour version of alpha), and Os is input surface opacity.
The shader compiler produced a constant output value (to be accumulated back into the ray) scaled by the incoming ray weight.
For opacity, a further ray was cast continuing in the same direction as the original, with weighting scaled by 1-opacity.

Fire and forget conversion: tracing

Ci = Os * (Cs + refl * trace());

⇒ ***out = Os * Cs * weight***

⇒ ***new ray: Os * refl * weight***

Oi = Os;

⇒ ***continuation ray: (1 - Os) * weight***

New rays were generated with similar tracking of the weighting contribution to the final output.

Illuminance callback conversion

```
color c = 0; point unitNorm = normalize(norm);
illuminance(p, unitnorm, PI/2) {
    c += Cl * normalize(L).unitNorm;
}
Ci = ambient() + c;
```

RenderMan had an illuminance() construct which effectively defined a lambda function, with the code inside the illuminance() block executed for each light. Cl was the lighting contribution from the light, and L was the light direction.

Illuminance callback conversion

```
color c = 0; point unitNorm = normalize(norm);
illuminance(p, unitnorm, PI/2) {
    weight = oldwt * normalize(L).unitNorm;
}
Ci = ambient() + c;
```

This was converted to a ”call-back” function which was evaluated on each light by the system.
With with other traced rays, a weight contribution was evaluated in the shader.

Ray management

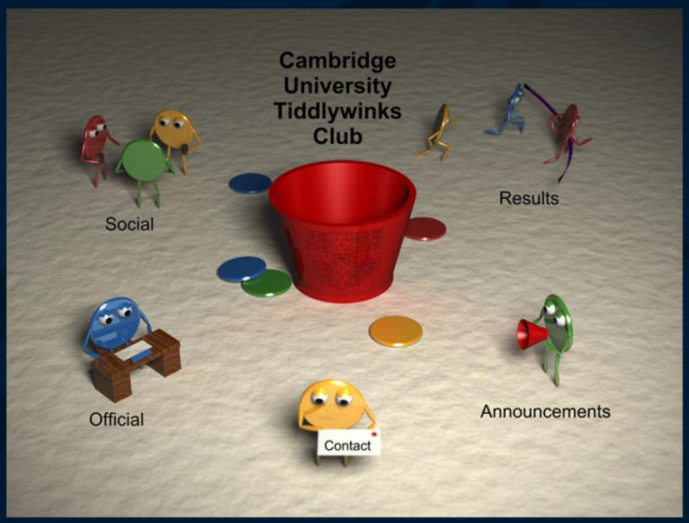
- Shading can generate multiple rays
  - Not just shadow rays
- Russian roulette based on weighting
- Kill and restart primary ray
- Eye rays and reflections in parallel

Unlike some ray traversal schemes, we supported arbitrary numbers of rays being generated in the shader, and these could proliferate (unlike some schemes that only generate shadow rays or one additional traced ray at each intersection).
A large number of rays could exist in the AR250 memory, but to avoid running out of space a “Russian roulette” scheme was used to kill rays with low weighting probabilistically, trading ray count for noise.
Primary eye rays could be killed at restarted in extremis.
New eye rays could be started alongside ongoing intersection of reflection rays.

Gallery: Tiddlywinks!

I’m currently number 12 in the world at tiddlywinks, so the important thing for me was to be able to ray trace a logo for Cambridge University Tiddlywinks Club (the other images in this presentation are taken with permission from the archives of the ART web site).
This scene has indirect illumination on the mat, produced by a large number of ”trace()” calls at the surface.
This render at one point caused a RenderDrive to die due to stress (it may have been a dodgy power supply); this is the most expensive thing I’ve blown up.

Gallery: Tiddlywinks!



At one point, the front page of CUTwC looked like this, which was a very expensive way of producing a web site.
Today, the mat background effect on the web site is still one generated by tracing with 4D Perlin noise, with polar coordinates to get a repeating pattern.

Without ray bundles, derivatives weren't handled automatically – we had to re-run sections of the shader as required with offsets in order to produce the appropriate values.
No ray length was preserved, and there was no pencil tracing – the ray footprint was evaluated by back-projecting to screen space, allowing texture filtering (including mip maps) to work.
Obviously the scheme did not directly support non-physical shader effects, where lighting was not a weighted linear combination of values.
This was a problem when we tried to emulate 3D Studio Max's built-in shaders for the plug-in: lacking HDR, 3D Studio used to modify the opacity of a surface when a light fell on it in order to preserve highlights on transparent surfaces.
ART was working on a "continuation shader" scheme (tagging rays for later processing) to support this kind of thing after I left.

RenderMan oddities

- Blending overlapping geometry
- Grammar and constant folding
  - color c1 = 2.0 * noise();
  - color c2 = 1.0 * noise();
- PRMan vs BMRT

Some oddities were RenderMan behaviour.
In PRMan, overlapping objects are blended – if you put two differently-coloured polygons in exactly the same place, you get the average of their colours.
We didn't support that.
Another oddity was that some shading functions like noise() evaluated differently in different contexts (polymorphic by return value).
For example, noise() in a float context returned a single-channel float; in a colour context it returned a RGB noise value.
2.0 * noise() is a float context, so it generates a single channel value.
Floats could be implicitly cast to colours (replicating channels), so c1 contains monochrome noise.
However, constant folding was performed before determining the context – so the "1.0 * noise()" converted to plain "noise()".
That meant that the noise() in c2 was evaluated in a colour context, and the result is colour noise.
Please hit the person responsible for this.
There were also some subtle differences between BMRT and PRMan.

How fast were we? The “Hebe mirror” scene was a statue of Hebe rendered with depth of field and a large area light, reflected in a mirror, with a large ray count. A RenderDrive took 13 minutes – which seems slow by modern standards. But a Pentium II took 22 hours to do the same.

Why didn't we all get rich?
There was an issue with the RAM – if its auto refresh clashed with shader writes, you could get random crashes.
This meant shaders had to stick to the small on-chip memory, limiting the use of look-up tables – which made some shaders very slow.
Some scenes had a very small number of rays that happened to have a lot of bounces; this killed parallelism.
Competitively, the RD2000 launched to compete with Sun workstations and Pentium PCs, but the appearance of SSE in fast consumer CPUs made the low clock of the AR250s limiting. The ray tracing stayed competitive with a 300MHz Pentium II a lot longer than shading performance.
RenderMan, while continuing as a product, was never really widely adopted as a standard; MentalRay was more widely adopted by various modelling packages, and the appearance of fast software like Arnold made a dedicated hardware renderer less competitive.
At the time of launch, there was no widely-adopted solution for global illumination – except possibly Radiance. Photon mapping appeared after the RenderDrive. But at least at the point I left, there was no global illumination solution integrated into the RenderDrive.

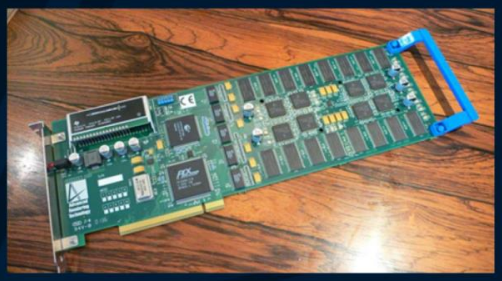
ARx50 Processor Roadmap

	AR250	AR350	AR450
Release	1998	2000	Never
Process	LSI 0.35um	TI 0.22um	TI 0.13um
Area	650 KGates	1.8 MGates	-
Frequency	40 MHz	80 MHz x 2	-
Performance	1.2 GFlops	5.1 GFlops	-
Notes	First Silicon	Bug fixes + performance	Radical Redesign

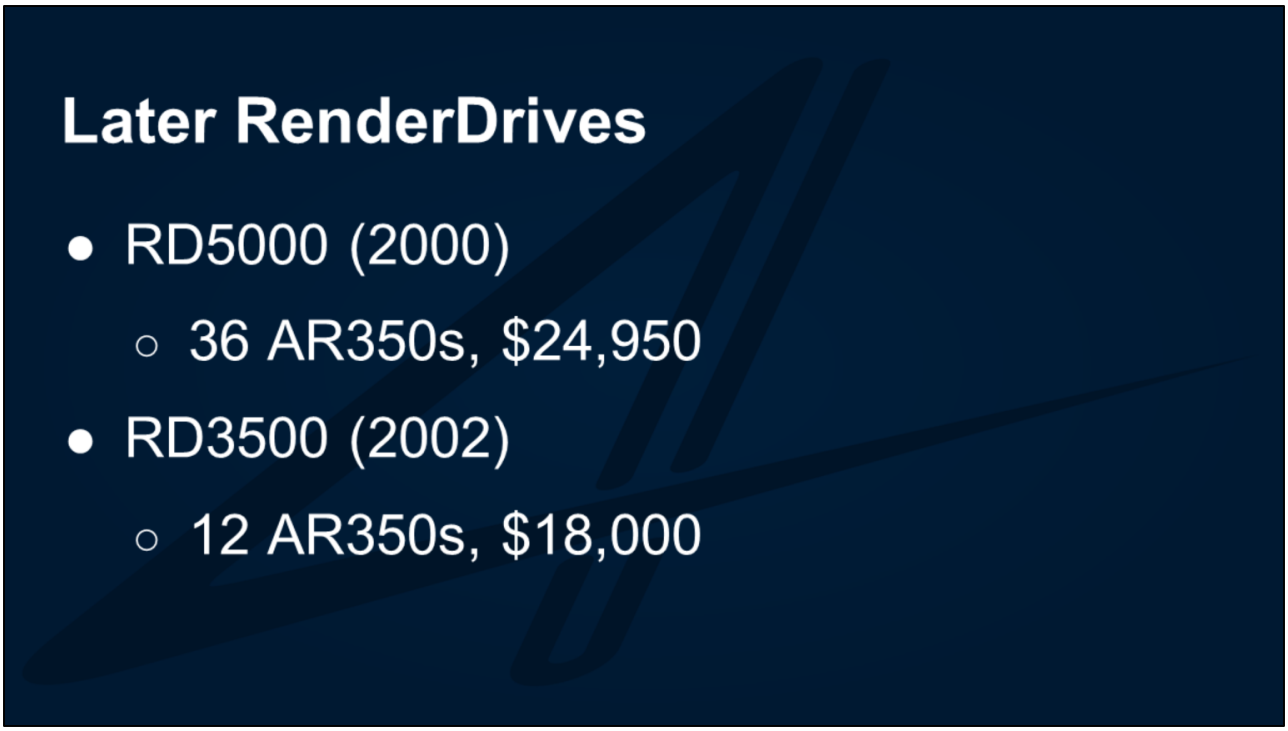
After the AR250, ART produced a successor AR350 chip with twice the clock and double the cores. This was produced on the process used for the UltraSPARC III. Later the AR450 was being designed, but the project was cancelled before production.

ARTvps PURE P1800

- 2001
- PCI-X
- 16 AR350 Processors
- $3,699



As PCs got more powerful and machines capable of managing large amounts of geometry became more affordable, having a dedicated host computer seemed sometimes unnecessary.
As a result, ART produced a “budget” card which could be driven directly from the artist’s workstation.



Especially for networked uses, there were still reasons to have a dedicated host processor for rendering.
Later machines moved away from Alpha processors, since AMD64 systems became an affordable alternative.
The RD5000 was launched as a high-end successor to the RD2000, with 72 cores at twice the clock speed of the RD2000.
The RD3500 was launched later as a budget alternative.

Over time, ART evolved as it attempted to be more global.
The company had difficulties around the time of the dot com crash, and was bought out by ”ART VPS” (“virtual photography solutions”), co-founded by one of the founders of ART, Daniel Hall.
ART-VPS continues, now producing the Shaderlight software ray tracing plug-in; RenderDrives are no longer part of the line up.

I need to thank the management of Shaderlight/ART-VPS, the IP holders, who gave me permission to discuss the details in this talk.
My ex-colleagues Iakovos (one of the chip designers) and Andrew (a system engineer) helped resolve some details.
I need to thank Adrian, who unfortunately died a few years ago. He was a brilliant engineer, if a bit hard to work with – he would never explain why he wanted you to do something, but 90% of the time he was right.
Finally, thanks to the other engineers responsible for the ART hardware, of whom this is a subset. And of course, the audience.

Supplemental

Texturing

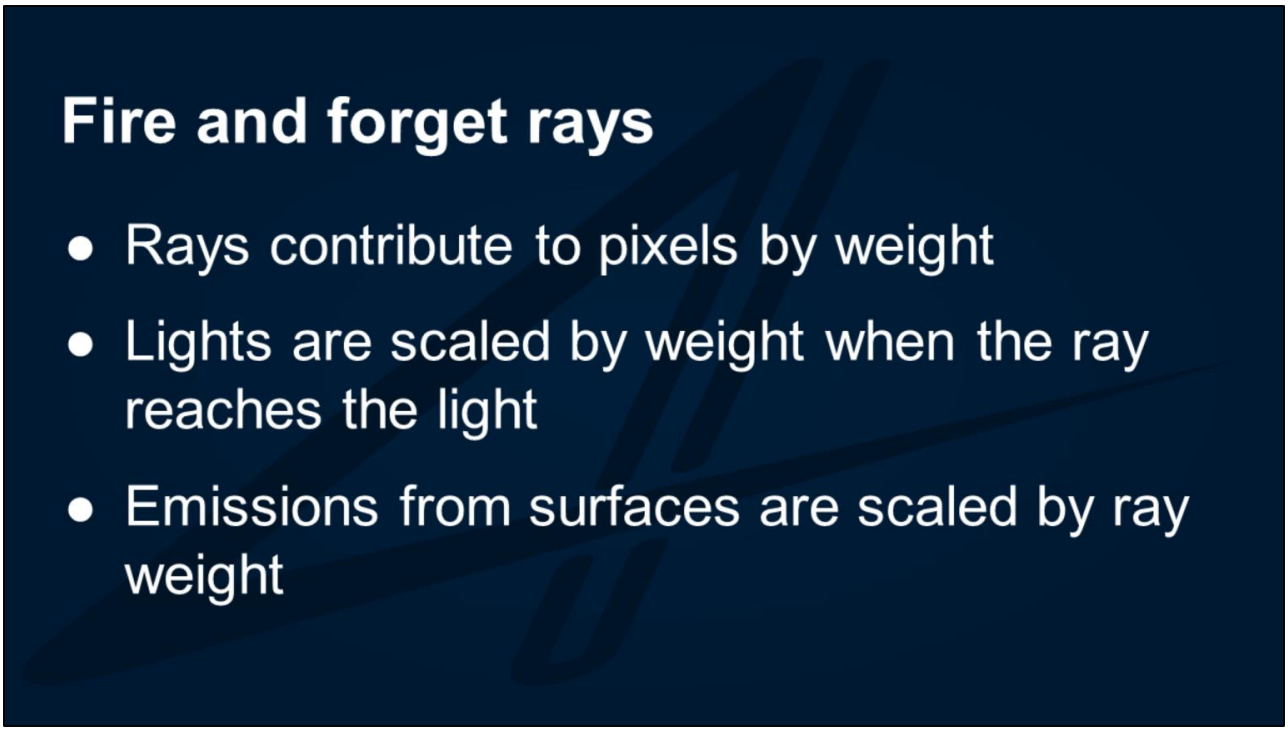
- Textures used a 3 x floating point format
- Filtering was entirely by shader
- Mip-maps were supported in software
- Texture sections were cached on the AR250s and distributed much like geometry

There was no dedicated texturing hardware, but textures were supported, converted to a standard triple-float format.
Shader utilities provided filtering and mip-mapping.
Tiled components of textures were broadcast on demand during shading from the host CPU, similarly to the scheme for BVH traversal.

Fire and forget rays

- Each ray has a colour weight
- Eye rays start with a weight of (1,1,1)
- Shading generates more rays with varying weights
- Ray weights are shared commutatively

Fire-and-forget may need a little more explanation.
A weight is associated with each ray, the weight having a float value per channel.
Rays start out with a unit weight (although this could be handled differently for antialiasing).
During shading, the ray weights may be scaled by another value, typically between 0 and 1.
For example, a surface may generate a reflection and refraction ray with weightings scaled by the Fresnel equation.
New rays are scaled by the weight of the parent ray.



The weight of each ray modulates the contribution of that ray to the final pixel.
Light values are scaled by the ray weight – so a ray heading to a light can be scaled according to the BRDF, and this can be used to modulate the lighting contribution.
This ray contributes to the pixel value when the ray reaches the light (if its weighting has not already reached 0).
A surface which emits light directly (such as via an ambient term) has that value written back to the pixel, again scaled by the ray weight.