




Brink Preferred Rendering with OpenGL



Mikkel Gjøel
Graphics Programmer

@pixelmager 

outline



- PC rendering overview
- state, shaders, occlusion queries
- virtual texturing
- bindless vertex attributes
- debugging OpenGL
- lessons learned



Entirely PC specific pipeline, our console setup is a lot different, so this presentation only covers the PC

It's a AAA game with OpenGL (zomg!1)

Focus on API usage, will not go too much into rendering-algorithm details

splash damage

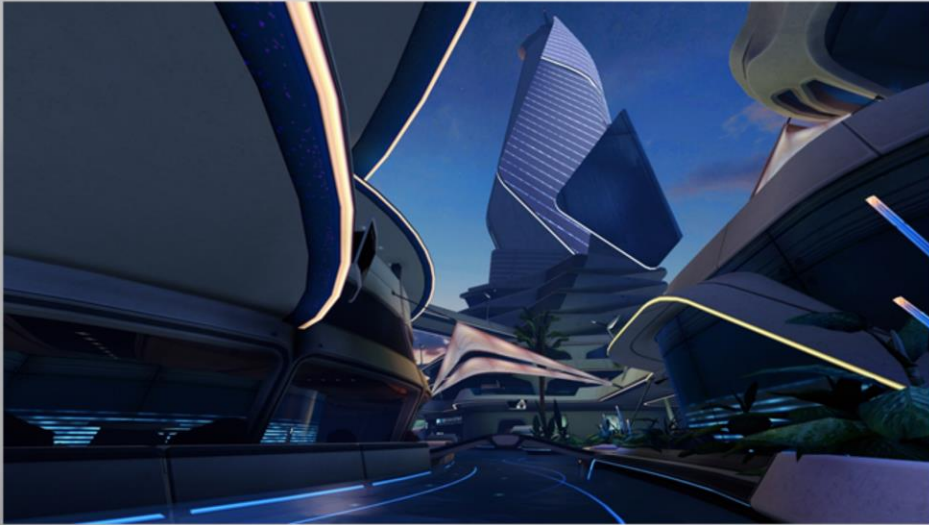


- Founded 2001
- Mod team gone pro
- Developed Enemy Territory games with id Software
- Just finished Brink with Bethesda Softworks

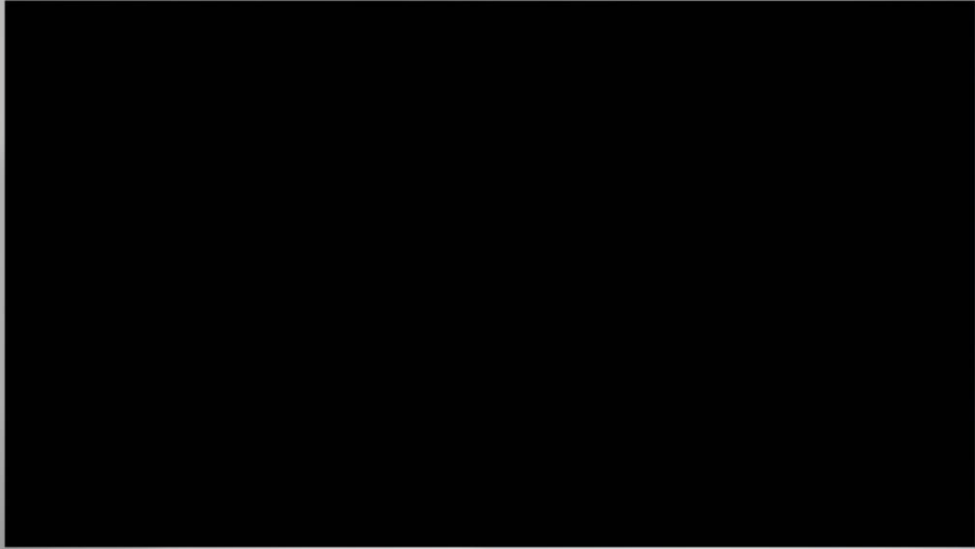


UK based games company

Started as a mod-team, Wolfenstein enemy territory -> doom3 multiplayer -> qw:et -
> Brink on PC/xbox360/ps3



BRINK 



brink pc-rendering overview



- 👉 thin, lightweight renderer, data-driven
- 👉 mostly GL2.x pipeline with GL3.x shaders
 - Vertex Buffer Objects, Frame Buffer Objects
 - Vertex- and Fragment-programs
 - Pixel Buffer Objects for asynchronous VT-readback



Thin layer in the sense that we try to move as many responsibilities to the game-code as possible – e.g. Interpolation of post-process values

Not bitsquid-engine level datadriven, but we have a lot of logic in text-files that can be hot-reloaded at runtime.

Adding features at very end of project is viable (e.g. FXAA, though that says more about fxaa... try it today)

No sampler-objects for compatibility reasons.

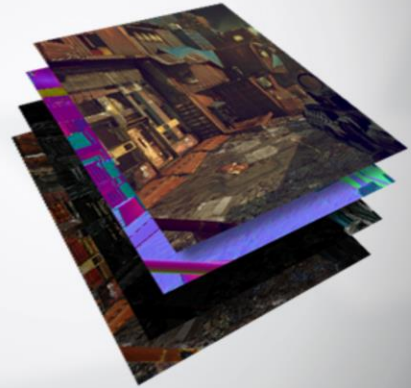
Target hw nv8800+ / amd2900+

brink pc-rendering overview



👉 fully dynamic lighting, deferred, fat attribute buffers

- fast content-iterations on lighting
- fast shader-development
- fast dynamic lighting
- ...more work for static lighting



depth	normal (xyz)	specCol (rgb)	albedo (rgb)
	skinflag	specExp (log)	specMult

Most of this is pretty standard stuff.

One of the reasons for preferring fat gbuffers to "light prepapss" is our virtual texturing pass.

Doing the VT indirection from more than one pass takes long (accessed for normals, and again for albedo/spec etc.)

a frame of brink



VT PrePass
depth PrePass
OCQ
ambient+attributePass
decals
lightPass
translucency
post, aa, resolve
GUI

SSAO



depth is straight copy from depth-buffer (possible in OpenGL)

Two sets of decals: "Regular", emissive or multiplicative decals in transl, albedo-blended in "decals"

FXAA

HBAO

a frame of brink



VT PrePass
depth PrePass
OCQ
ambient+attributePass
decals
lightPass
translucency
post, aa, resolve
GUI

SSAO



a frame of brink



VT PrePass
depth PrePass
OCQ
ambient+attributePass
decals
lightPass
translucency
post, aa, resolve
GUI

SSAO



a frame of brink



VT PrePass
depth PrePass
OCQ
ambient+attributePass
decals
lightPass
translucency
post, aa, resolve
GUI

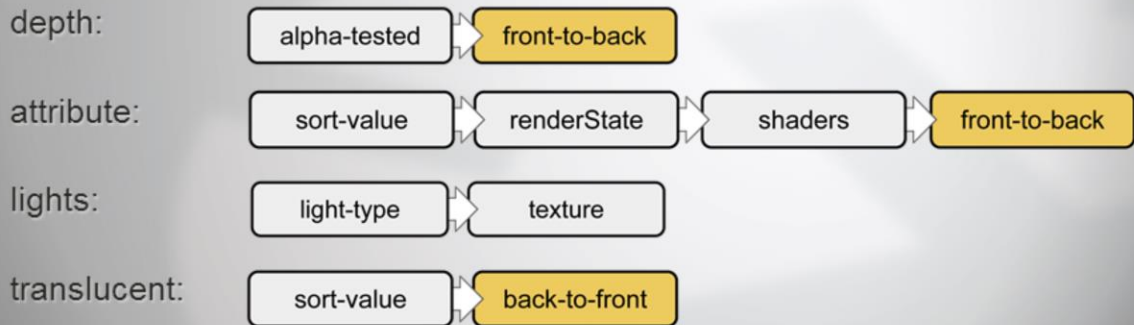
SSAO



sorting for state-change



- each pass is a list of drawCmds
 - sorted to reduce statechange



A list of objects to be rendered into every renderpass is build and sorted to reduce state-change. No state-“caching” required as only minimal state is set.

A single initial block of state per render-pass (e.g. no depth-test for translucent, no color-write for depth etc.)

In attr-pass, most environment will be static and have the same shaders. Few changes between drawcalls (due to vt)

shaders



- 👉 static set of handwritten shaders available to artists.
 - written by gfx-programmers and tech-artists together with artists
 - python-scripts to generate variations with different **\$defines**
- 👉 ~150 files, ~350 shader-combinations, ~3sec compile-time

```
void main() {  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Artists will usually team up with a gfx-programmer or tech-artist to create new shaders.

We have been very happy with this model. The artists too. Few shaders: Less places to optimize – and optimizations done affect more areas.

Low number of shaders largely due to deferred rendering

Low compile-time means not necessary to cache binary shaders (using extension)

shader pre-processor



- Regular preprocessor functionality (some now natively in glsl)

```
$include "inc/preferred_glsl.inc"  
$define PREFERRED_RENDERING  
$ifdef PREFERRED_RENDERING  
    return bestValue();  
$else  
    return fuglyHack();  
$endif
```

Pre-processor does the usual stuff – a lot of it now available, but was not at the time of writing

shader pre-processor



- Regular preprocessor functionality (some now natively in glsl)

```
$include "inc/preferred_glsl.inc"
#define PREFERRED_RENDERING
#ifdef PREFERRED_RENDERING
    return bestValue();
#else
    return fuglyHack();
#endif
// vertex-attributes looked up by name
vec4 vertexColor = $colorAttrib;

//uniforms / textures mapped on shader-load
vertexColor = vertexColor * $colorModulate + $colorAdd;
vec4 diffuse = texture( $diffMap, texCoord ) * vertexColor;
```

Pre-processor also picks up uniforms, textures and vertex-attributes (essentially just defines again), making it easy to add new uniforms that are then picked up from code.

(added bonus: It's easy to spot uniforms in code)

occlusion queries



↘ area and frustum-culled before OCQ



Game does area, portal and frustum-culling before any occlusion queries

occlusion queries



- ↘ area and frustum-culled before OCQ
- ↘ render world-space bboxes / light-frustum
 - if predicted visible, issue OCQ during rendering
- ↘ ~60 avg, ~200 max OCQ per frame
~0.5ms



Dicing of static geometry based on geometry-density – roughly 8x8x8m chunks

If lights are visible, we wrap the actual render-call in an OCQ (could do the same for geometry, but we don't....)

occlusion queries



- 👉 initial version stalled on frame $n+1$
 - proved to be a bottleneck

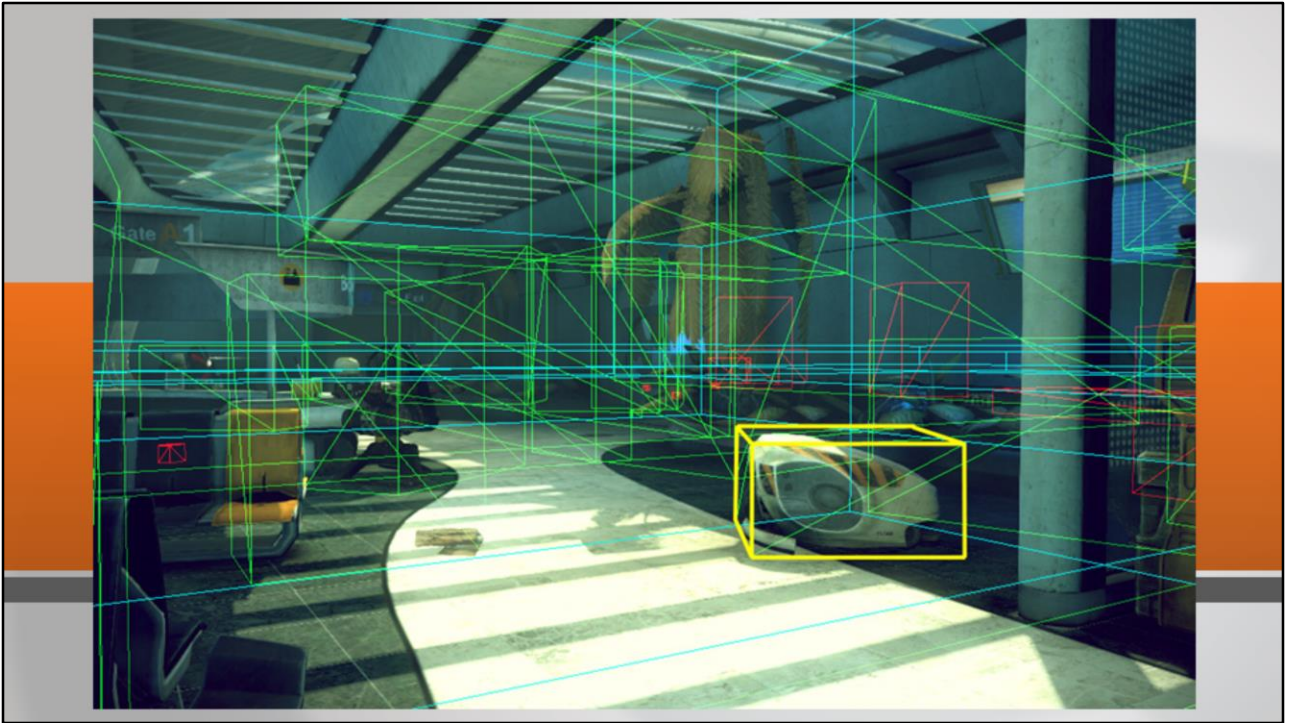


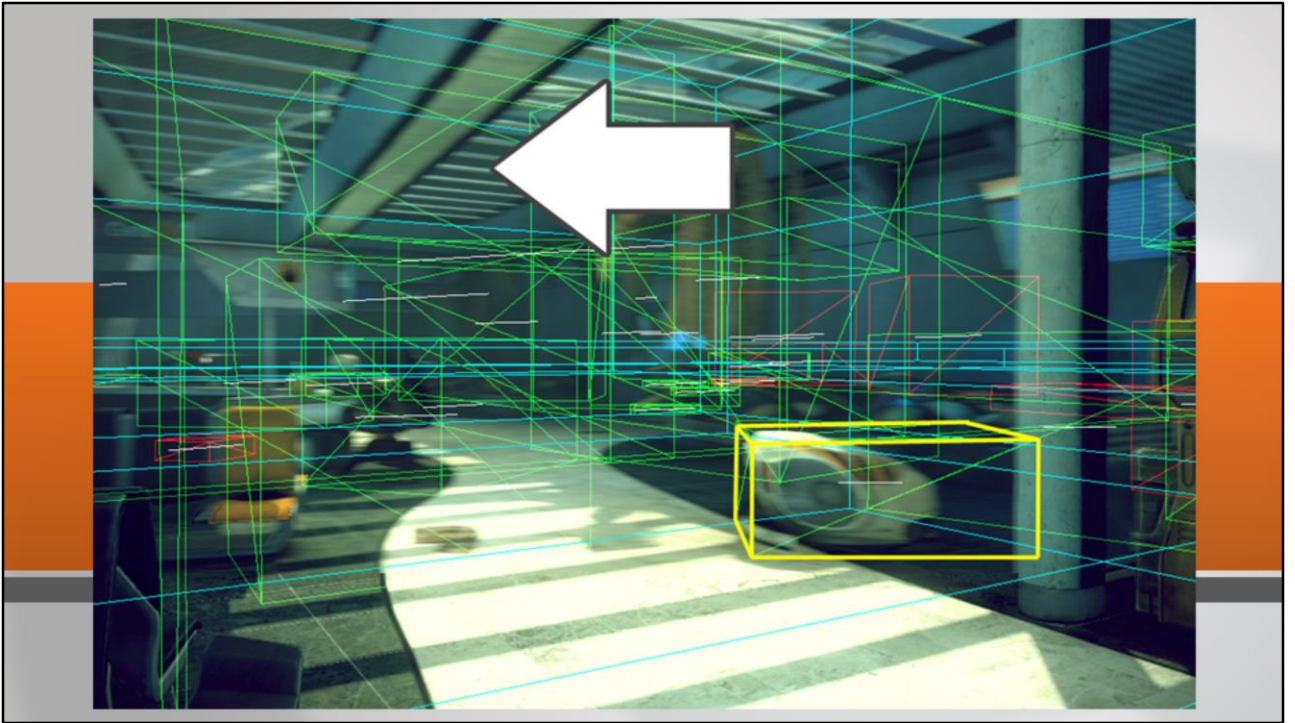
occlusion queries



- 👉 if not ready, assume nothing changed
 - do not issue new OCQ
- 👉 expand bbox in view-movement-direction to predict visibility

Bbox expansion per box, along world-major axis is fine – could do better but no gain as coarse approx anyway





virtual texturing

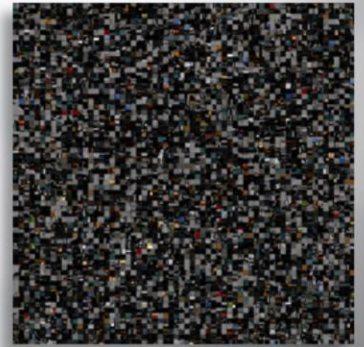


Multiple sources

virtual texturing



- 👉 only load the texels you need
 - same texel-density everywhere on screen

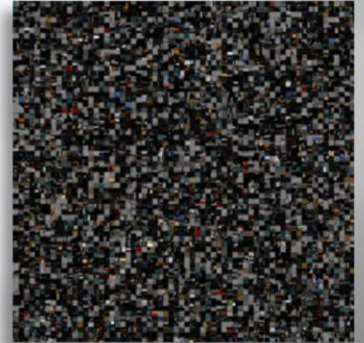


Essential idea is to only load the data needed – same goal as any other streaming system.

virtual texturing



- 👉 only load the texels you need
 - same texel-density everywhere on screen
- 👉 reduces complexity for artists
 - "texture however you want, as long as it fits"



One of the main benefits is the reduced complexity for artists: They can use as much texture as they want, as long as it fits on disk.
(they still need to worry about varying texel-density, as we don't have a step that unifies this.... It is possible though).

A build-step takes all textures and packs them into an atlas, then modifies the vertex-data uv-coordinates to point to the new UV.
(it also saves original texture-size to allow for texture-wrapping).

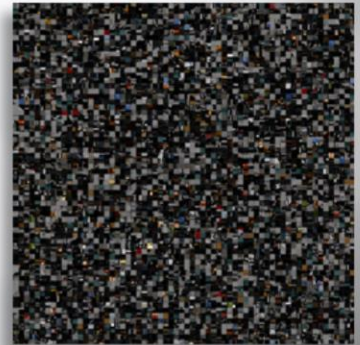
Brink is not limited to entirely unique textures, we allow for reuse of textures, and texture-wrapping.

Unique is slightly faster in shader, but requires more memory... Allowing re-use is why we can ship on one dvd, and not three....

virtual texturing



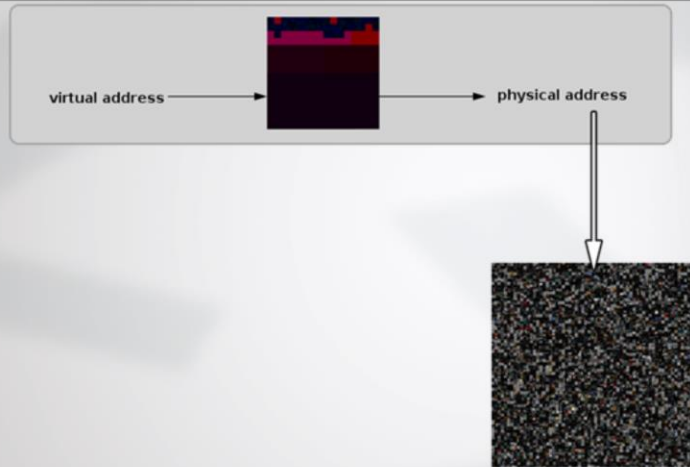
- 👉 only load the texels you need
 - same texel-density everywhere on screen
- 👉 reduces complexity for artists
 - "texture however you want, as long as it fits"
- 👉 reduces dependency between drawcalls
 - everything uses the same texture
 - ...actually we have multiple pagefiles



As all objects are using the same texture (the virtual one), there is a significantly reduces dependency between drawcalls. this allows us to 1. dice up the world across objects, and 2. never change textures while rendering vt objects

We have multiple page files which leads to multiple address spaces. E.g. dynamic objects are in a separate address-space. Each level has a separate file. Support 256 concurrent address-spaces – DLC accesses 3.

virtual texturing

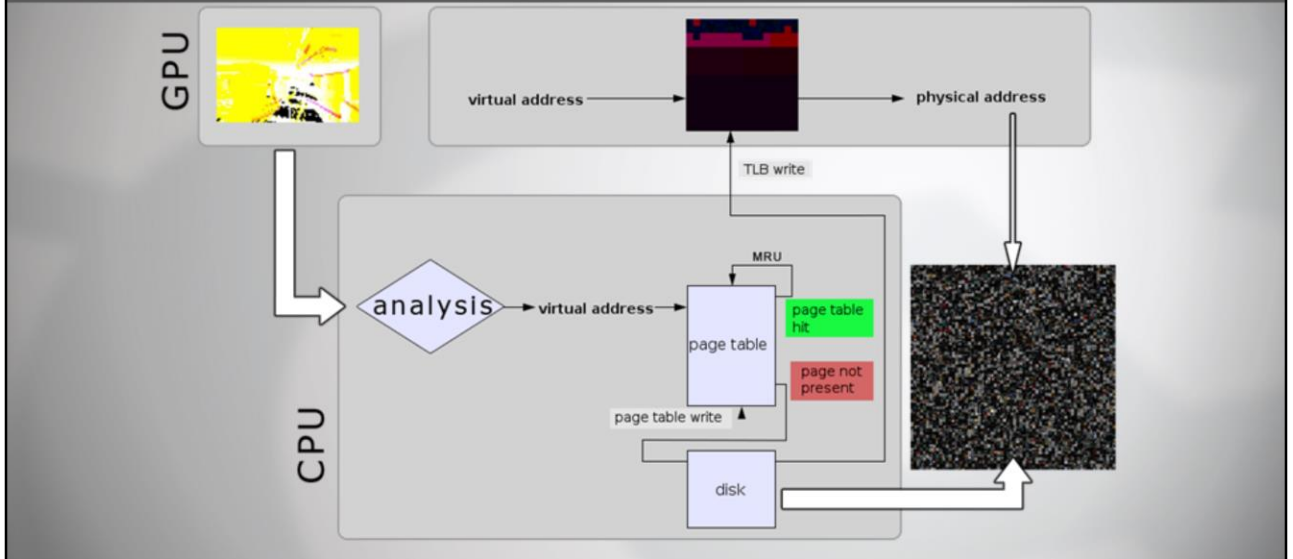


Rendering uses the page table in the shader to convert from virtual to physical coordinates to look up the correct texture mapping

3 dependant texture-reads into virtual textures (diffuse / normal / specular). All using the same UV – so essentially a single TLB with multiple address-spaces.

Never fails completely – always loads lowest 3-4 mip levels on level-load (only a couple hundred kb)

virtual texturing



low-res packed rgba32 ID-buffer

- page_u, page_v, mip-level, <virtual address-space index>

ID buffer gets rendered and read back

Analysis determines a page miss (in case of page-hit, MRU table gets updated)

Page gets reserved in the physical address space (8192x8192, storage is DXT textures, 4096x4096 on consoles)

Page data gets loaded from storage, then transcoded from DCT to DXT

“TLB” (2D texture with 1 texel/page in the address space) gets updated with mapping information from virtual to physical coordinates

virtual texturing



- 👉 processing mapped memory from separate thread
 - PBO mapped pointer valid in all threads



The PBO being valid in all threads makes it easy to utilize other threads for the analysis

virtual texturing



- 👉 processing mapped memory from separate thread
 - PBO mapped pointer valid in all threads
- 👉 minimum 1 frame delay for async readback
 - 2+ frames delayed for multi-GPU



With alternate-frame-rendering using SLI, you need to add a frame latency for an async texture readback per GPU in order to not stall a gpu during rendering

virtual texturing



- 👉 processing mapped memory from separate thread
 - PBO mapped pointer valid in all threads
- 👉 minimum 1 frame delay for async readback
 - 2+ frames delayed for multi-GPU
- 👉 anisotropic mip-selection, bi-linear filtering



Using the hw anisotropic mipmap-selection, but does plain bi-linear filtering

bindless vertex attributes



1. decouples vertex-format from vertex-pointers
2. provides direct GPU-pointer, reducing indirection-overhead in driver

```
GL_NV_vertex_buffer_unified_memory  
GL_NV_shader_buffer_load
```



This extension is all about helping the driver out, reducing the amount of work it is required to do
Lower-level api than usually available.

bindless vertex attributes



```
glVertexAttribPointerARB( ATTR_COL, 4, GL_UNSIGNED_BYTE,  
                          GL_TRUE, vertsiz, ofs_col );
```

```
glBufferAddressRangeNV( GL_VERTEX_ATTRIB_ARRAY_ADDRESS_NV,  
                       ATTR_COL, vbo_gpu_addr+ofs_col,  
                       vbo_gpu_siz-ofs_col );
```

```
glVertexAttribFormatNV( ATTR_COL, 4, GL_UNSIGNED_BYTE,  
                       GL_TRUE, vertsiz );
```

On top the usual GL-setup

On bottom using the bindless extensions.

The format is split out to a separate call. Looking at Brink, this is significant as most of our vertex-data is of the same format.

This means that we never change vertex-formats during our main render-passes.

bindless vertex attributes



- 👉 implementation revealed the slack in our attribute-code, drivers are forgiving
 - this was the main work - writing the code was easy...
- 👉 use debug-context to spare crashes (thanks Jeff Bolz!)

The bulk of the work was to fix the places in our code where we relied on magic in the drivers. We were being slack without knowing it.

Dealing with direct gpu pointers means you will hang your gpu every time you do something wrong – and doing the initial conversion, that is every time.

Thanks to some quick work from the nvidia-guys, the debug-context will now not only not-hang, it will tell you what call would have hung, and what you were doing wrong. Greatly eases the implementation.

bindless vertex attributes



- 👉 implementation revealed the slack in our attribute-code, drivers are forgiving
 - this was the main work - writing the code was easy...
- 👉 use debug-context to spare crashes (thanks Jeff Bolz!)

- 👉 practically got rid of vertex-attribute cpu-overhead
 - 5-10% of CPU frame-time
- 👉 ...not practical for frame-temporary allocations
Getting GPU-address takes time.

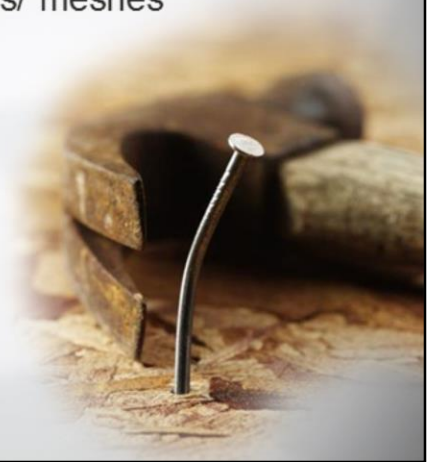
Result of this was a nice reduction in the CPU time taken up by setting vertex-attribs. This was around 5-10% of our render cpu-time.

Brink does "naïve" frame-temporary memory allocations, effectively allocating new VBOs for every object that requires memory (typically gui or particles). This works very well (we used to have a more elaborate setup, allocating a single chunk of memory, but the driver actually handles these allocations very well - may as well use that). Ironically, when then optimizing vertex-attr. code, the frame-temp alloc. gets in the way.

debugging OpenGL - tools



- 👉 gDebugger / glIntercept for cmdBuffer debugging
 - useful for checking correct textures/shaders/"meshes"



Debugging really is the #1 time-sink during development.

gDebugger / gl-intercept (which hopefully will continue to work on nv-hw) is very useful to make sure you are actually doing what you think you are doing – allows you to inspect command-buffer.

Wish: glIntercept with webgl preview? ...with gpad geometry-preview? ...and shaders? ...hotreloadable..? One can only dream...

debugging OpenGL - tools



- 👉 gDebugger / glIntercept for cmdBuffer debugging
 - useful for checking correct textures/shaders/"meshes"
- 👉 Nsight / GPU PerfStudio almost useful for timing.
 - NSight has actual start/end drawcall GPU-times!
Only way to get this. Try it today!
 - Neither is very mature for OpenGL



Nsight / GPU perfstudio are mostly useful for timing-purposes, but they require some more work to be truly useful for OpenGL AAA games.

(Nvidia released app profiler lib appears to be able to do this level of profiling on it's own).

We tried App Analyzer, which looks very promising, but it didn't produce . We are hopeful though.

(intel-tool is not targeted at working for opengl at all)

debugging OpenGL - tools



- 👉 gDebugger / glIntercept for cmdBuffer debugging
 - useful for checking correct textures/shaders/"meshes"

- 👉 Nsight / GPU PerfStudio almost useful for timing.
 - NSight has actual start/end drawcall GPU-times!
Only way to get this. Try it today!
 - Neither is very mature for OpenGL

- 👉 no tools currently shows FBO or geometry



None of the tools show content of FBOs. GPU perfstudio looks very promising for OpenGL though. App Analyzer is meant to show FBOs.

debugging OpenGL – custom code



👉 hot-reloadable assets

- shaders / textures / meshes / assets
- runtime-modifying shaders for debugging



Roll your own. Everyone wants 360pix + gpad to have a magical lovechild that does everything.

If you can hot-reload at runtime, and quickly iterate on code, that's a very viable alternative.

Hot-reloadable assets – main solution, very useful! Option to override packaged assets runtime.

shaders / textures / meshes / assets... this is the most important debugging-tool!
Typical debugging session is quickly modifying shaders to extract the necessary debug-data.

debugging OpenGL – custom code

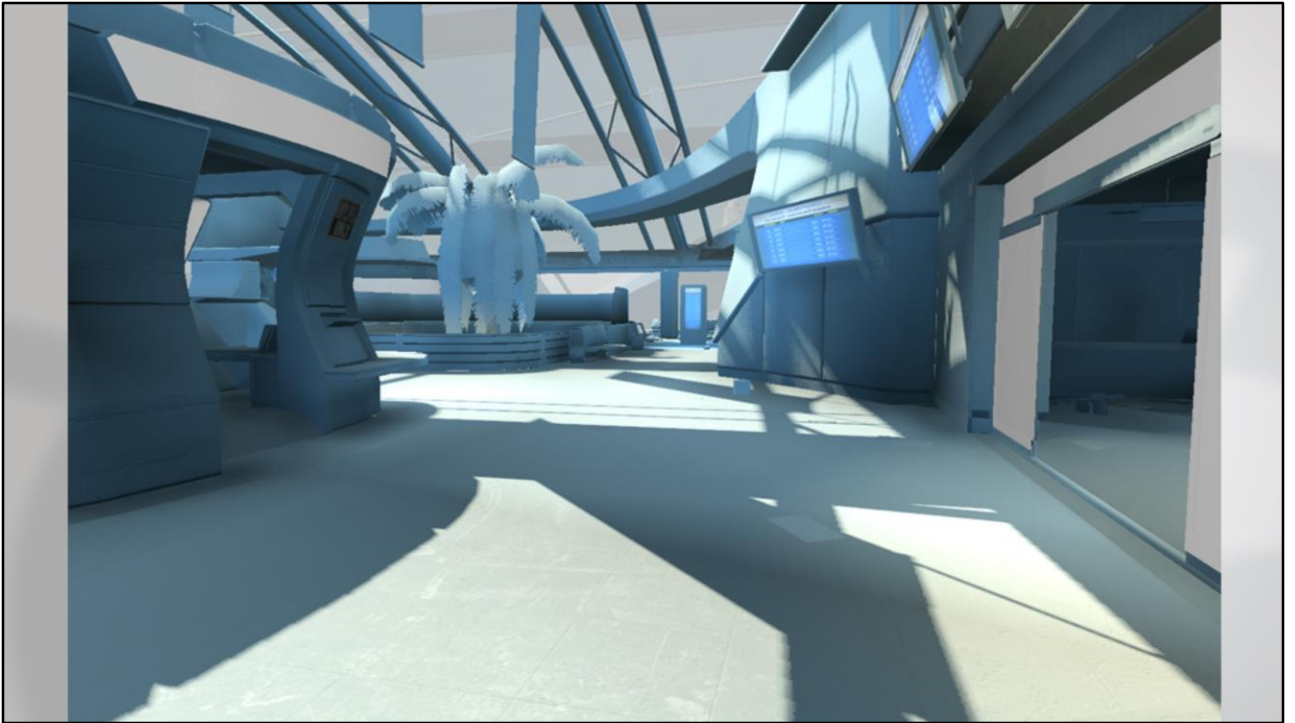


- 👉 hot-reloadable assets
 - shaders / textures / meshes / assets
 - runtime-modifying shaders for debugging
- 👉 “configurable” render-stages
- 👉 various visualization modes
 - wire, batches, info on obj, lights, ocq etc.
 - show textures / rendertargets, write to files

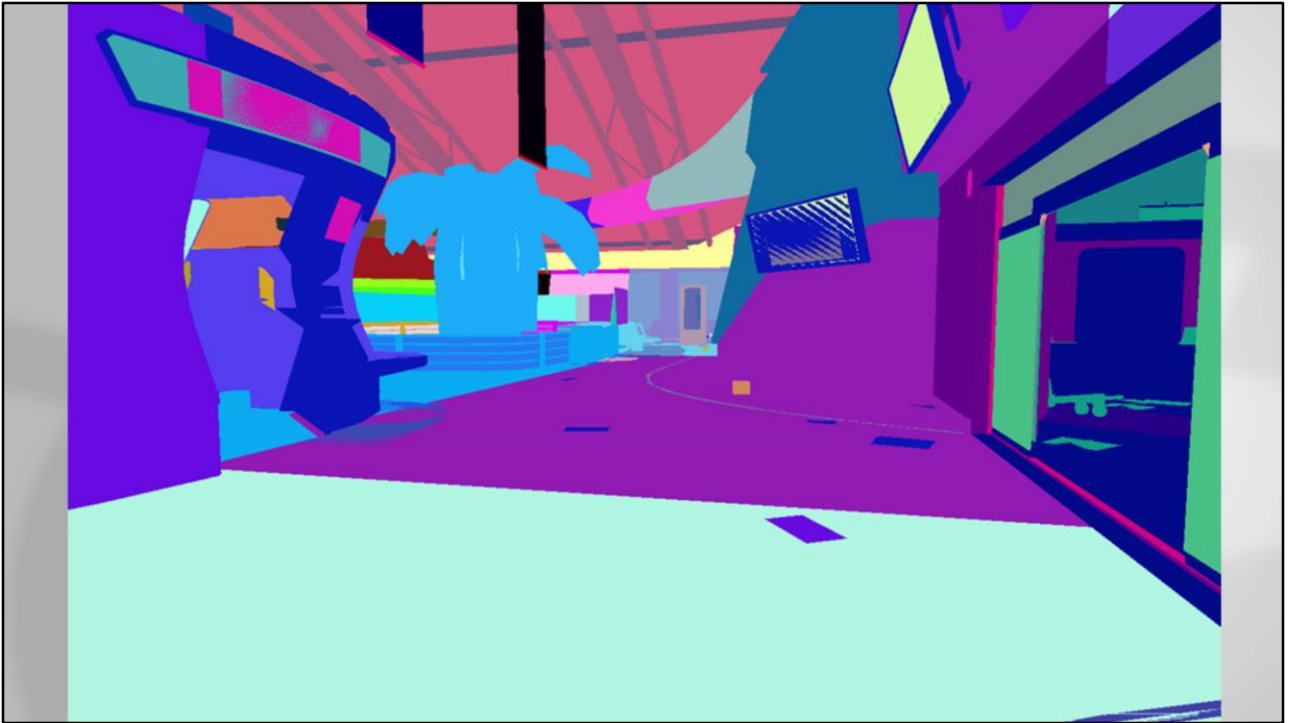


Also adds a number of debug render-modes to allow a level of debugging for non-coders. Very useful for artists to see e.g. tangent-spaces, overdraw etc.

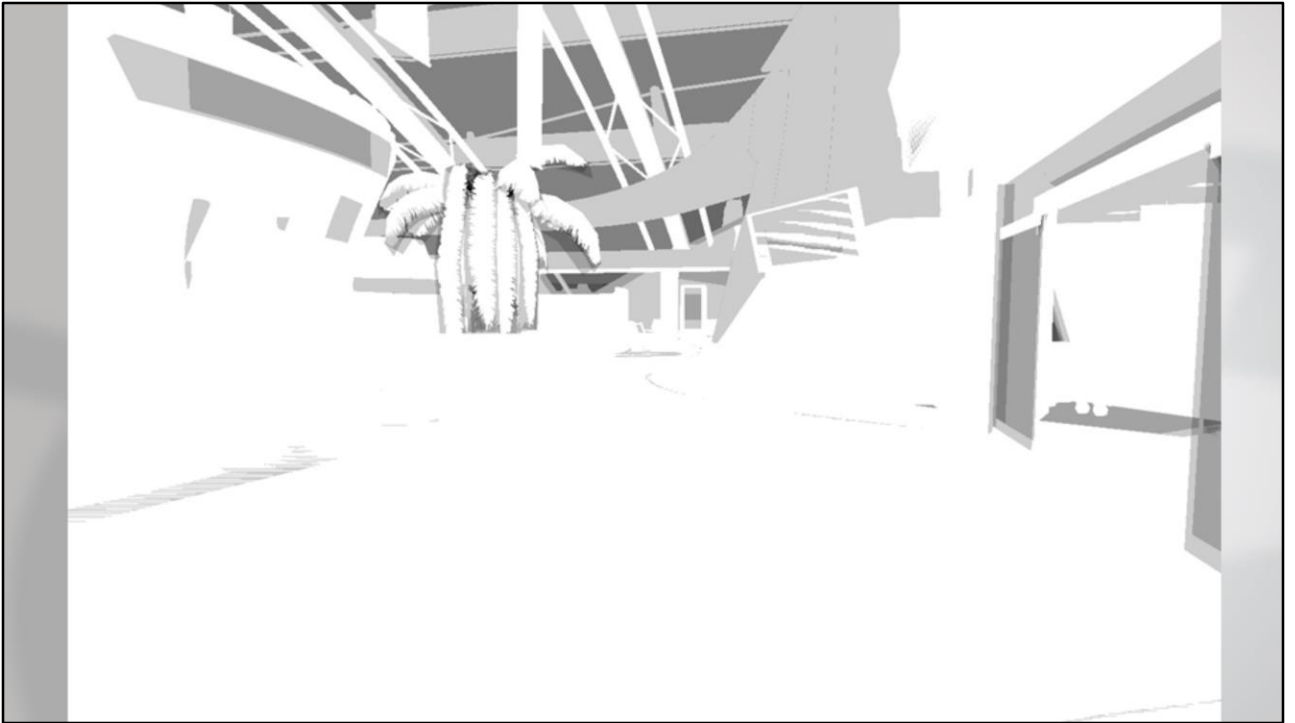




Selectively disabling different types of texturing: diffuse / specular / bump etc.



Dicing across objects, based on geometry density – builds BSP-tree



Overdraw – mostly for particles (valid to see alpha-tested overdraw as well)

debugging OpenGL – custom code



- full OpenGL-statedump
 - write to file and diff
- queryTimers for profiling
 - Make sure to pair with CPU-timings to find all bottlenecks



- Break when something is bad, dump state, return to normal, dump state – diff in favourite tool
- The Printf of gfx-debugging.

```
static bool dumpstate = false;  
if(dumpstate) { dumpstate = false; glStateToFile(); }
```

lessons learned



👉 do not trust state: Reset per frame



You're not the only one rendering. Steam and friends that render overlays, grabbing your context.
paranoid-mode sanity checks of hw-state, making sure it is what we think it is before a dra

lessons learned



- 👉 do not trust state: Reset per frame
- 👉 use the debug context. Add it today! ~1 hour



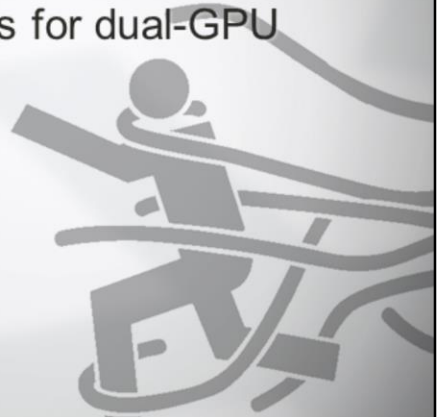
Cort Stratton / naughty dog altdevblogaday post on debug-context

Using the debug-context removes the tedious glerror bisection of code, trying to find the villain-functioncall

lessons learned



- 👉 do not trust state: Reset per frame
- 👉 use the debug context. Add it today! ~1 hour
- 👉 no vendor-independent solutions exists for dual-GPU laptops

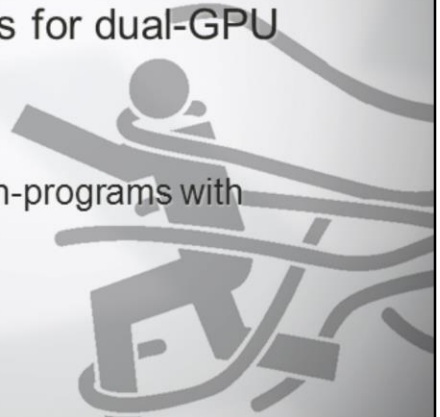


For laptops with two GPUs, a low-power one and a high-power one, there is no "standard" for how to pick the right GPU to run on. There are vendor-dependant ways to do this (e.g. Nvapi), but a unified solution should be developed.

lessons learned



- 👉 do not trust state: Reset per frame
- 👉 use the debug context. Add it today! ~1 hour
- 👉 no vendor-independent solutions exists for dual-GPU laptops
- 👉 keep a small code-setup for tests
 - driver teams appreciate small reproduction-programs with source-code too



Small isolated code-setups are very handy to figure out api-quirks.

Driver-teams respond very well when sent small code-setups that show errors and reproduce claimed driver-bugs (that mostly are not).

...a lot better than "my game doesn't work, here's a nightly snapshot, you need to run over to the dragon on level 4 and stab it with the banana.... the particle-effect should be red!! Fixit!"

conclusions



- ▶ OpenGL works for AAA games
 - OpenGL-support was never an issue
 - d3d10 on Windows XP

OpenGL allowed for direct3d 10 level functionality on all the platforms our customers were using.

- 👉 OpenGL works for AAA games
 - OpenGL-support was never an issue
 - d3d10 on Windows XP
 - responsive IHVs
 - AAA can “force” updated drivers

Hardcore shooters have it easy. Our customers are mostly invested enough in the game that they will update their drivers to get the game to run (they are also used to having to do this).

IHVs are very responsive and the great relationship with them means any kinks will be ironed out before shipping.

- 👉 IHV to use the debug-context more
 - enable a lower-level api
 - `GL_PARANOID_LEVEL = INT_MAX;`
 - performance warnings



The dbg-context is great, and should be used more – e.g. for performance warnings. The debug-context allows for a lower level of api. Rather than have the driver-teams write tons of fixup-code, show us what we are doing wrong so we can fix it. Add a “paranoid-mode” that will “treat warnings as errors”

OpenGL wishlist



👉 light-weight Display Lists

- this is our current console setup!
- ideally, we would still like to be able to set state... sorry.

👉 GPU self-feed would be great!



There is a very large overlap of work between frames. On consoles we are able to take advantage of this, build commandbuffers and simply patch these up with changed data (typically matrices) per frame. Being able to do this on PC would be great. We really want something like Display Lists, but restricted to make them meaningful for hw.

GPU selffeed for gpu-based occlusion etc. would be great. The best exact form is not entirely clear.

references



- <http://brinkthegame.com/>
- http://origin-developer.nvidia.com/object/bindless_graphics.html
- <http://altdevblogaday.com/2011/06/23/improving-opengl-error-messages/>
- <http://glintercept.nutty.org/>
- <http://developer.amd.com/tools/gDEBugger/Pages/default.aspx>
- <http://paralleInsight.nvidia.com/>

acknowledgements



- Arnout van Meer / Splash Damage
- Romain Toutain / Splash Damage

- Simon Green, Phil Scott, Jeff Bolz of NVIDIA fame

- Nicolas Thibieroz, Kevin Strange of AMD fame

All images are copyright of their respective owners



Teh Jobs

We're hiring: www.splashdamage.com/jobs



@splashdamage



Splash Damage



mig@splashdamage.com