Implementing old-school graphics chips with Vulkan compute

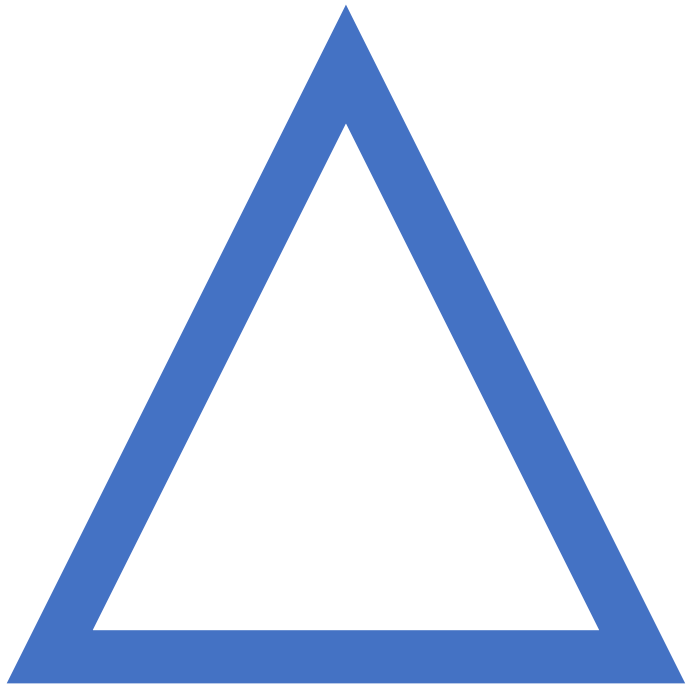
Hans-Kristian Arntzen

Khronos Meetup – Munich

2019-10-11

Content

- The problem space
- Compute shader rasterization
- Optimizing with Vulkan subgroups
- Test implementation



Problem space

It’s just triangles, how hard can it be, right?

Old-school?

- Late 90s, early 2000s console graphics hardware is quirky
  - Does not look anything like a modern GPU
- Understanding how legacy tech works is fun
  - Nintendo 64 RDP as an example
  - Reimplementing them is also “fun”
- Goal here is accurate software rendering
  - … but on Vulkan compute!
  - … because why not

High-level emulation

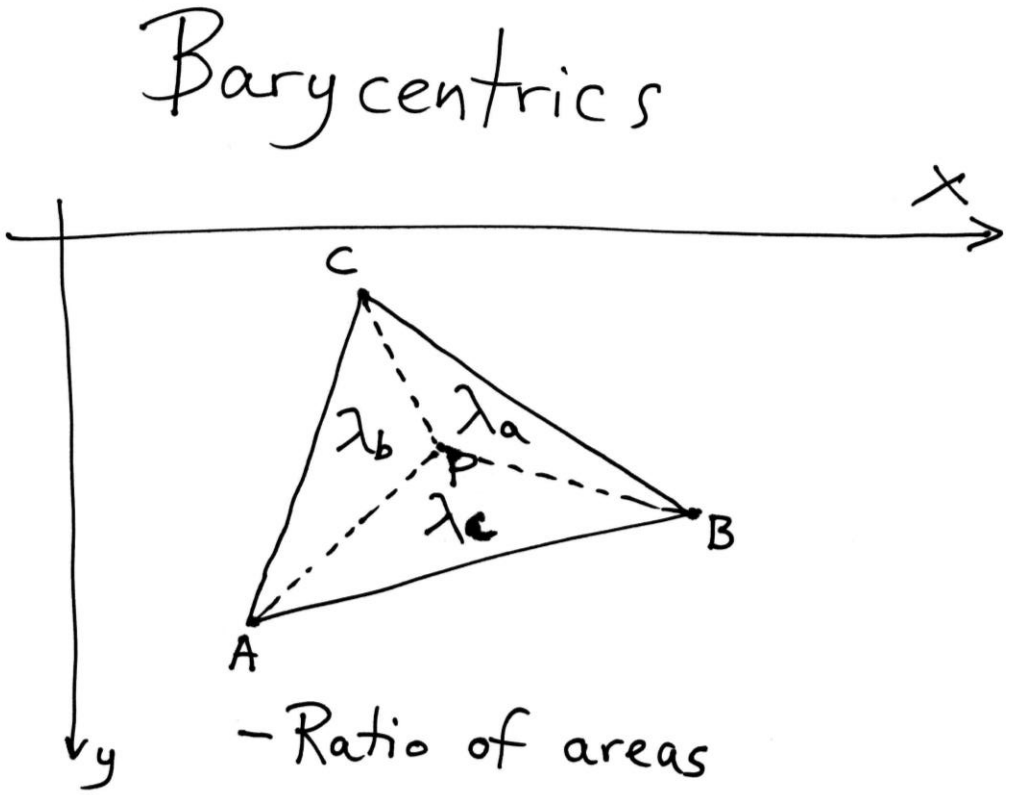
- Reinterpreting the intent of an application
- Almost exclusively used for N64 and beyond
- Higher internal resolution
- Rasterization and fragment pipeline
- Sacrifices accuracy for speed and “bling”
- Many challenges, but not what this talk is about

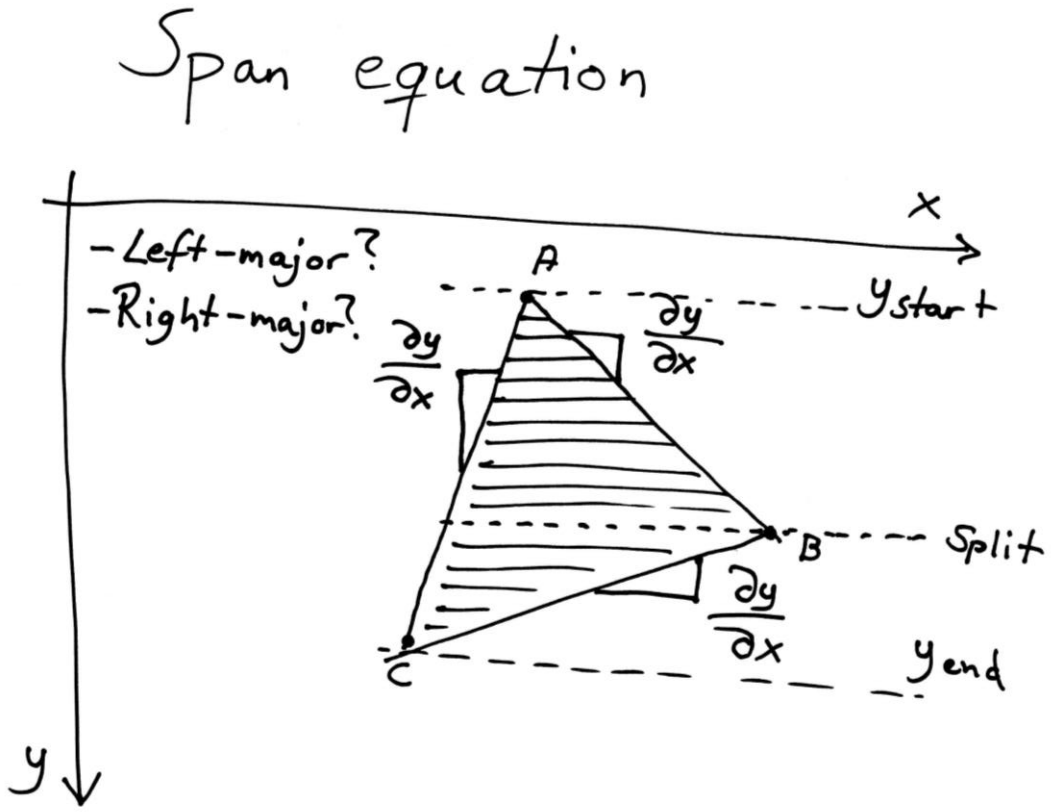
Low-level emulation

- Focus on recreating exact behavior
- Emulate what the GPU is doing in detail
- Usually only reserved for a CPU reference renderer
  - Slow!
- Very specific “look and feel”

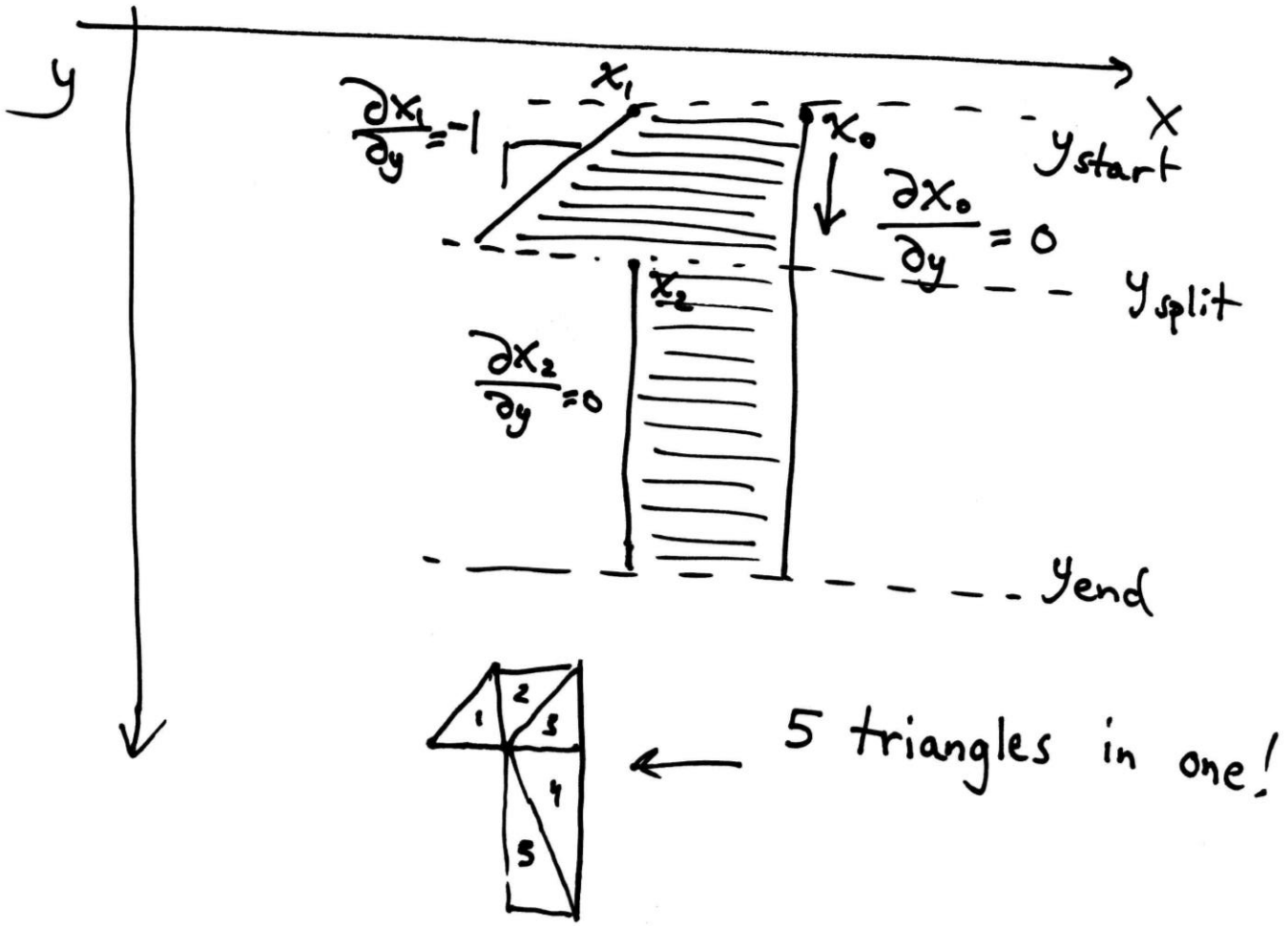
N64 was known to be blurry for a reason …

Old-school rasterization





Complex span equation shapes



Triangle setup and CPU vertex processing

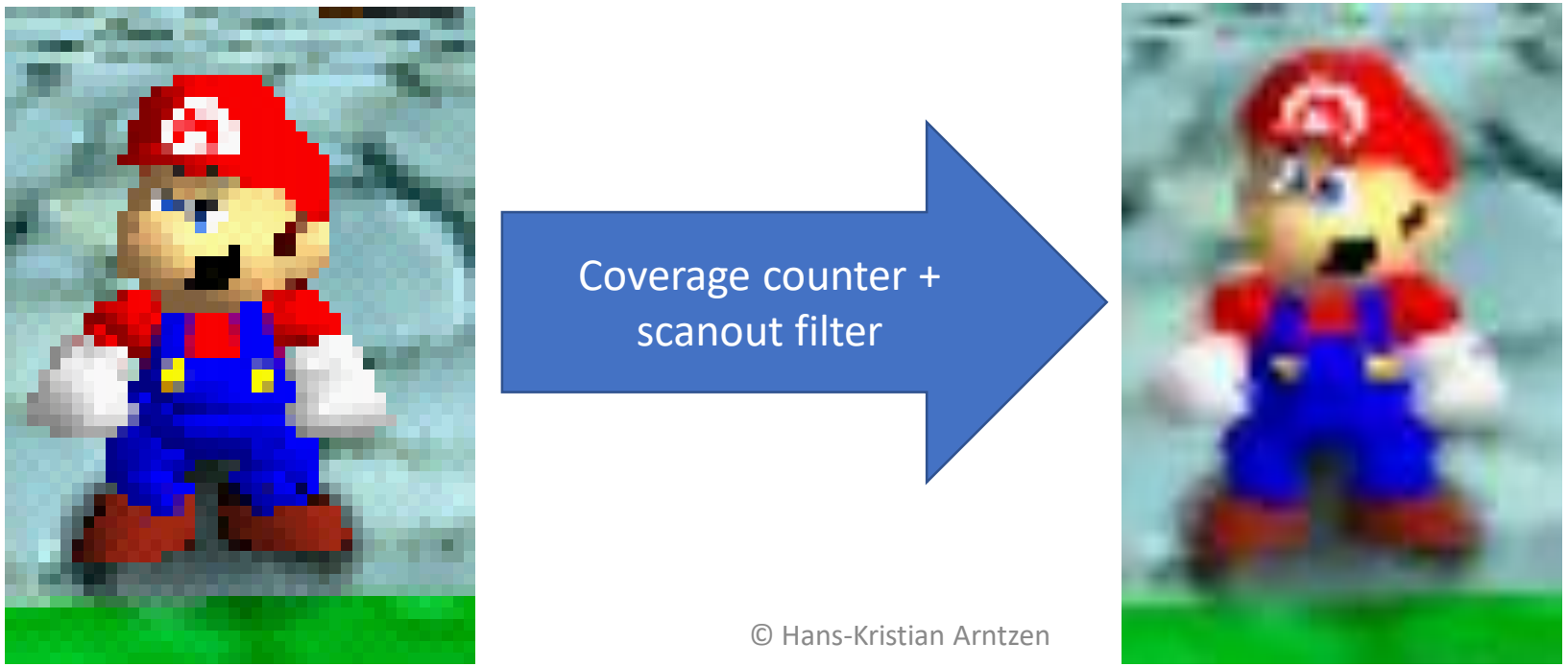
- Poly-counts were generally low
- Good use case for programmable co-processor / DSP
  - GTE on PS1, RSP on N64, VU0/1 on PS2, etc …
  - SW lighting, Gouraud shading
- A low-level emulation will usually consume triangle setup
  - Precomputed interpolation equations
  - Usually fixed point

Rasterization pipeline problems

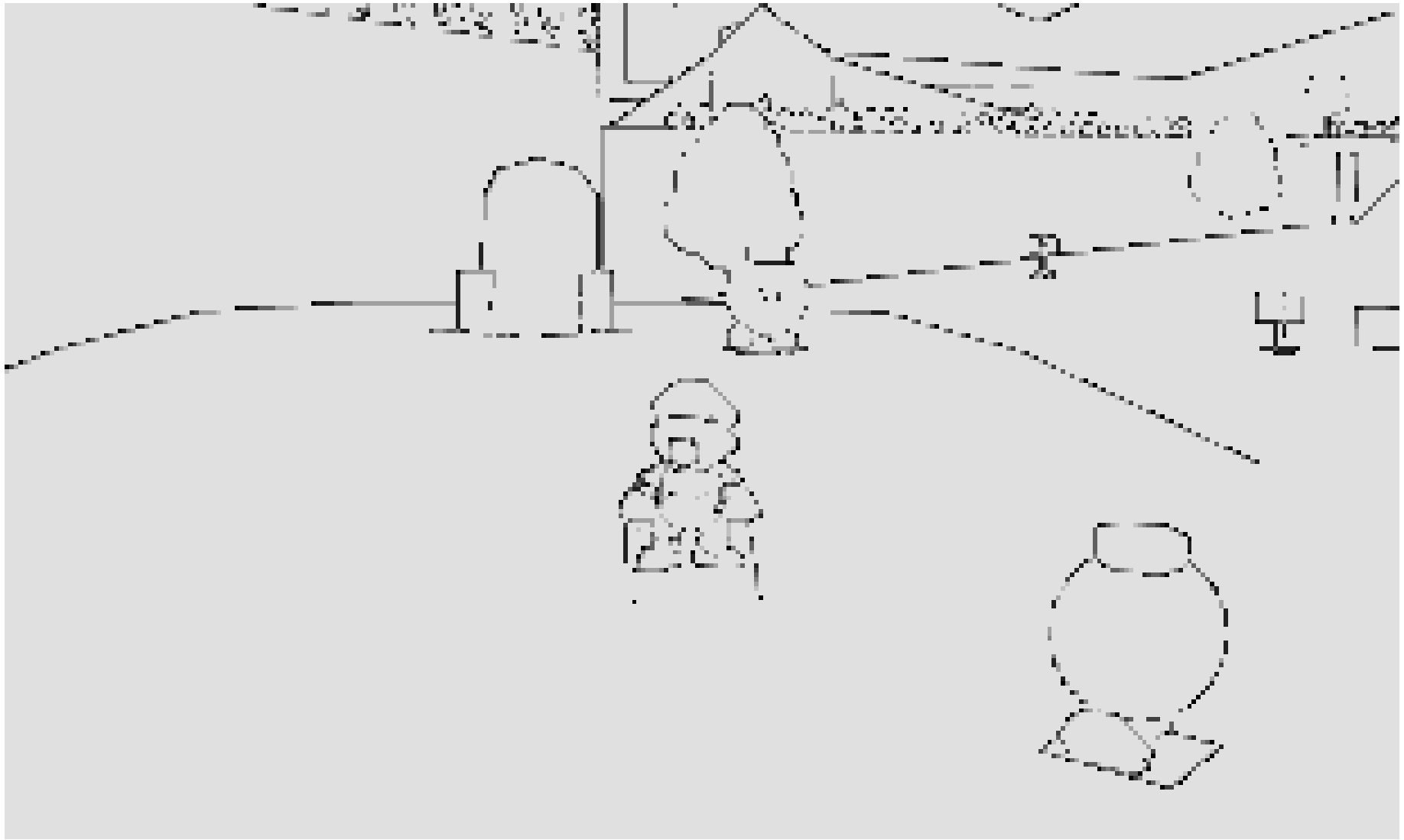
- Rasterization is one of the last major fixed function blocks in modern GPUs
  - Hi-Z and early-ZS is key for high performance, well suited for fixed HW
- Rasterization rules
  - Workaround -> manual test -> discard -> late-Z everything -> ☹
- Blending
  - Programmable blending is inefficient on desktop – fine on mobile though!
- Depth/stencil buffer
  - Programmable depth/stencil testing is even worse – and terrible on mobile as well
- Anti-aliasing
  - This gets extremely weird …
- Memory aliasing
- fragment_shader_interlock is the current “solution”
  - Throwing performance out the window is not fun ☹

Anti-aliasing problems

- AA in N64 is notoriously weird
- Fixed function MSAA does not map to anything
- N64 feeds coverage data into post-AA in video scanout



Correlated coverage guides post-AA



The fragment interlock hammer

- We can have programmable everything in fragment with interlocks
- Take a mutex per pixel with some hardware assistance
- Lock must be in API order => ordered interlock
- Can be extremely slow …
  - In order semantics + locks => recipe for performance horror
- Spotty support
  - Generally considered an extremely obscure feature
- Alternatives?
  - Atomic linked list of pixels, incredibly difficult to pull off in emulation use case

Rethinking the problem

- Fragment shader
  - Start with fast, fixed function API interfaces
  - Add a million hacks and workarounds to make it work
- Compute shader
  - Abandon fixed function restrictions
  - Build from ground up, full implementation flexibility
  - Tune for relevant constraints
  - More future looking?

Unsolvable problems?

- Any multi-threaded emulation will fail on these, not just GPU
- Cycle accuracy
- Texture cache behavior
  - N64 has a programmer-maintained 4 KiB texture cache though …
- Generally not important for correctness
  - Obscure cycle timing on CPU is a thing
  - Not really on GPU
- Arbitrary cross-pixel dependencies
  - Aliasing color / depth not 1:1
  - Is this a thing?

Compute shader rasterization

Gotta use those teraflops for something

The render loop

- Basic CPU software rendering is super easy
  - `foreach (prim in primitives) render_all_pixels_in(prim)`
- Need to feed compute with a massive number of threads
  - Naïve CPU loops are not MT friendly
- Common solution is going tile based
  - `foreach (tile) foreach (prim covering tile) render(tile, prim)`
  - Lots of techniques for compute in this domain

The naïve compute ubershader

```
int x = coord_for_thread().x;
int y = coord_for_thread().y;
Color color = load_color_framebuffer(x, y);
Depth depth = load_depth_framebuffer(x, y);

for (auto &setup : primitive_setups)
{
    if (!test_rasterization_coverage(setup, x, y))
        continue;
    Color rgba = interpolate_rgba(setup, x, y);
    UV uv = interpolate_uv(setup, x, y);
    Color samp = sample_texture(setup.texinfo, uv);
    Color combined_color = combiner(rgba, samp, setup.comb);
    depth_stage(depth, interpolate_z(setup, x, y));
    blend_stage(color, combined_color);
}

store_color_framebuffer(x, y, color);
store_depth_framebuffer(x, y, depth);
```

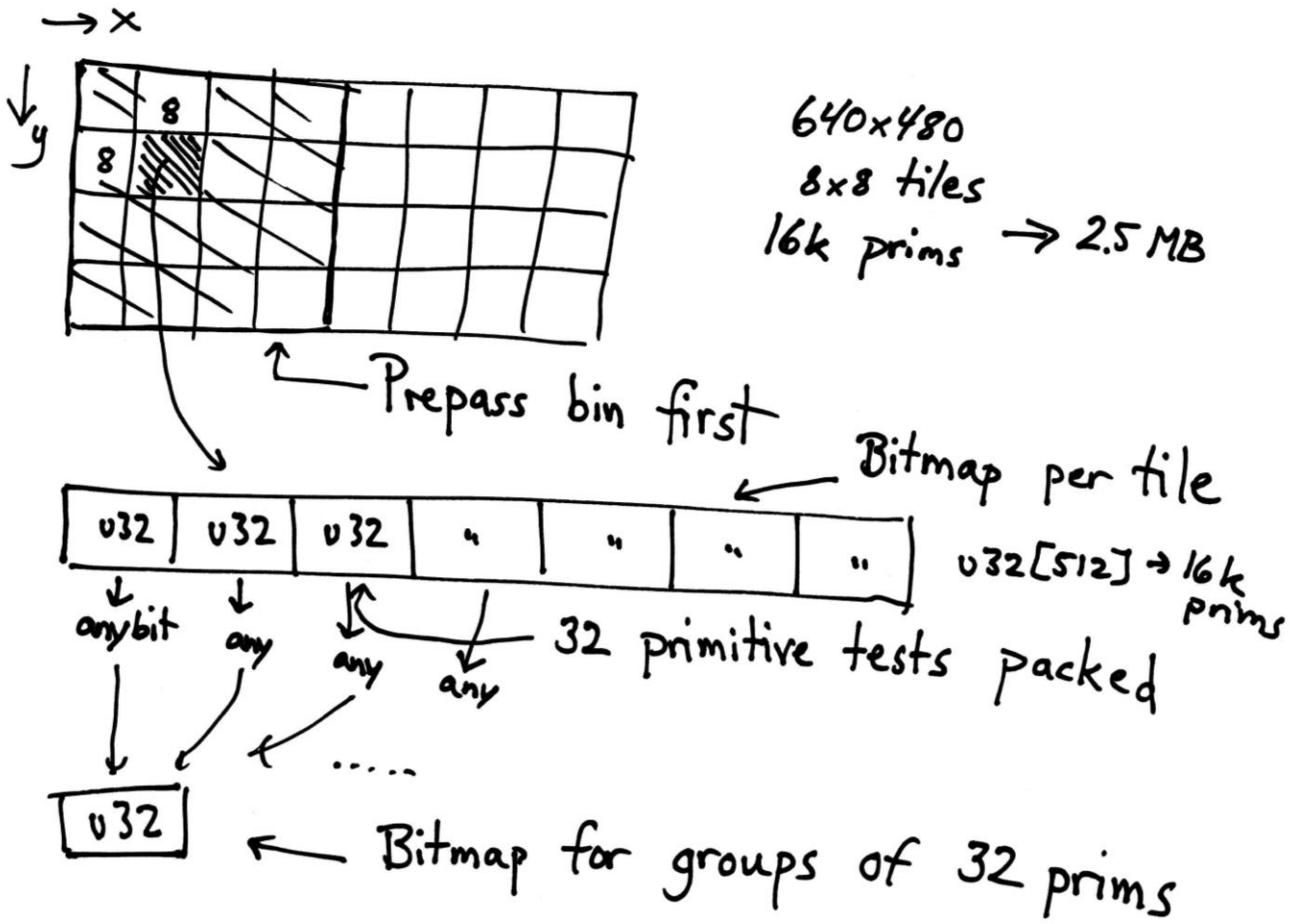
Very few pixels pass this test

Expect more like 2000 lines of code here

Fully programmable blending and ROP, in GPU registers!

HOST_VISIBLE buffer if CPU and GPU need to be coherent (N64 >_<) ...

Tile binning to bitmasks



Better bitscan loops

```
int num_prims_32 = (num_prims + 31) / 32;
int num_prims_1024 = (num_prims_32 + 31) / 32;

// All the loops are dynamically uniform. GPU is happy ☺
// Skip over huge batches of primitives.

for (coarse_index in num_prims_1024)
{
    foreach_bit(coarse_bit, coarse_mask[tile][coarse_index])
    {
        int mask_index = coarse_bit + coarse_index * 32;
        foreach_bit(bit, mask[tile][mask_index])
        {
            int primitive_index = bit + mask_index * 32;
            // Now we test and render.
        }
    }
}
```

findLSB / ctz / etc
Single instruction

Alternative: flat list of triangle IDs, but massive VRAM requirement in worst case scenario.

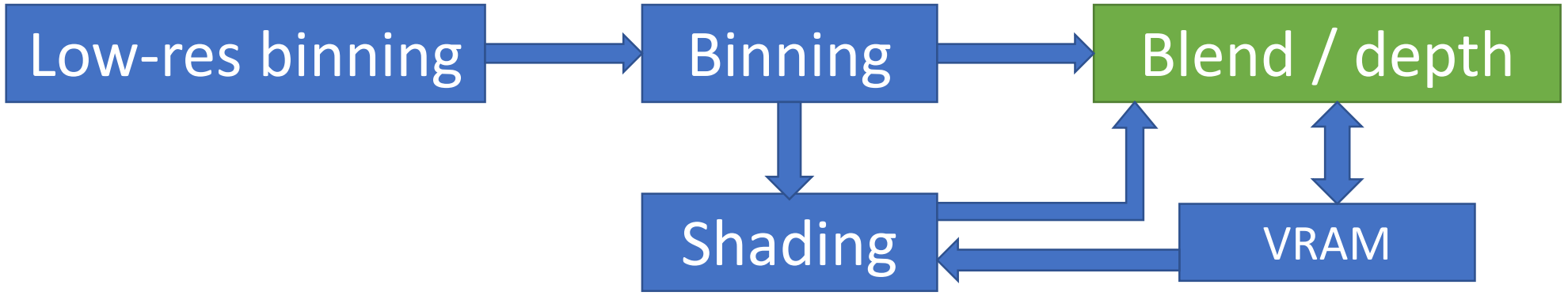
To ubershade or not to ubershade

- Even ancient GPUs have a lot of state bits
  - Primitive type?
  - Texture combiner state?
  - Blending modes?
  - Depth modes?
  - And more …
- Compute kernel to render a tile needs to handle **everything**
  - Insane register pressure => poor occupancy => poor performance
  - At least the branches are dynamically uniform ☺
  - Might be best solution if configuration space is small
  - Bindless resources

Splitting up the ubermonster

- What if the per-tile kernel consumed pre-shaded tiles?
  - Reduce the ubershader to only deal with blending and depth-testing
- Distribute chunks of work tagged with (tile + primitive index)
  - One vkCmdDispatchIndirect per variant
  - **Can assign different specialized shaders to deal with different render state**
  - Specialization constants are perfect here
- Need to allocate storage space for color/depth/etc
  - **Intense on bandwidth**, but gotta use that GDDR6 for something, right?
  - Could be a reasonable tradeoff for 240p / 480p content
- Callable shaders would be nice …

Split architecture



Position data (SSBO) – 16k entries

Attribute data (SSBO) – 16k entries

Primitive state descriptor index (UBO) – 16k entries

State descriptors (UBO) – 1k entires

Advanced Vulkan features

Now this is where it gets interesting

Subgroup operations

- New feature in Vulkan 1.1
  - Share data between threads efficiently
  - Peeks into a world where threads execute in lock-step in a SIMD fashion
  - Alternative is going through shared memory + barrier()

```
in int gl_VertexIndex;
void main()
{
    float v = float(gl_VertexIndex);
}
```

Isolated threads
- Shader code is nice and scalar

```
in intNx32_t gl_VertexIndex;
void main()
{
    floatNx32_t v = floatNx32_t(gl_VertexIndex);
}
```

Subgroups
- Represents a SIMD unit
- The ISPC paradigm

Wrapping your head around subgroups

- A “thread” -> “lane in a SIMD vector register”
- Branches -> “make lanes active or inactive”
- All values are the same in subgroup -> “scalar register”
  - Shading languages do not express this well
- Many recent graphics talks will mention it
  - Console optimization
- https://www.khronos.org/blog/vulkan-subgroup-tutorial
- https://www.khronos.org/assets/uploads/developers/library/2018-vulkan-devday/06-subgroups.pdf

subgroupBallot()

```
bool primitive_is_binned = test_primitive(thread_index);
// Reduce a boolean test across all threads to a bitmap, nice!
uvec4 ballot = subgroupBallot(primitive_is_binned);

if (subgroupElect())
{
    // Only one thread needs to write.
    if (gl_SubgroupSize == 32)
        write_u32_mask(ballot.x);
    else if (gl_SubgroupSize == 64)
        write_u32x2_mask(ballot.xy);
}

// Legacy way? atomicOr on shared memory or reduction passes <_<
```

```
uint index = gl_GlobalInvocationID.x;
uint value = indices[index];
indices[index] = subgroupBallot(value != 0u).x;
```

Navi ISA

```
s_inst_prefetch         0x0003
s_getpc_b64             s[0:1]
v_mad_u32_u24           v0, s3, 32, v0
s_mov_b32               s0, s2
s_load_dwordx4          s[0:3], s[0:1], null
v_lshlrev_b32           v0, 2, v0
s_waitcnt               lgkmcnt(0)
buffer_load_dword       v1, v0, s[0:3], 0 offen
s_waitcnt               vmcnt(0)
v_cmp_ne_i32            vcc, 0, v1
s_mov_b64               s[4:5], vcc
v_mov_b32               v1, s4
buffer_store_dword      v1, v0, s[0:3], 0 offen glc
s_endpgm
```

VK_EXT_subgroup_size_control

- Not all GPUs have a fixed subgroup size
- Compilers can vary subgroup sizes
  - Intel (8, 16 or 32) - May even vary within a dispatch
  - AMD pre-Navi (64 only)
  - AMD Navi (32 or 64)
  - NVIDIA (32 only)
- gl_SubgroupSize builtin is fixed, but lanes might disappear
  - This is totally fine for many use cases of subgroups, but …
- VARYING_SIZE_BIT, FULL_GROUP_BIT and subgroupSizeRequirement
- Critical extension to use subgroups well on Intel

Subgroup atomic amortization

- Distributing work on GPU typically means atomic increments
- Atomics are expensive
- Amortize atomics overhead
  - Most compilers do this when incrementing a fixed address by 1
  - May or may not when incrementing by != 1

Arithmetic subgroup operations

```
// Merge atomic adds

uint bit_count = bitCount(binned_mask));

// Reduce in registers
uint total_bit_count = subgroupAdd(bit_count);
uint offset = 0u;
if (subgroupElect())
    offset = atomicAdd(counts, total_bit_count);
offset = subgroupBroadcastFirst(offset);
offset += subgroupExclusiveAdd(bit_count);

// Legacy?
// Prefix sum in shared memory and then atomic.
// Write result to shared, then broadcast.
```

```
uint index = gl_GlobalInvocationID.x;
uint offset = subgroupExclusiveAdd(index);
list[offset] = index;
```

```
s_inst_prefetch         0x0003
s_getpc_b64             s[0:1]
v_mad_u32_u24           v0, s3, 32, v0
v_mov_b32               v2, v0
s_orn2_saveexec_b64     s[4:5], 0
s_mov_b32               s6, lit(0xffff0000)
s_mov_b32               s7, lit(0xffff0000)
s_mov_b32               s0, s2
s_movk_i32              s8, 0x0000
s_movk_i32              s9, 0xffff
s_load_dwordx4          s[0:3], s[0:1], 0x000020
v_mov_b32               v2, 0
s_nand_b64              exec, 0, 0
v_add_nc_u32            v2, v2, v2 row_shr:1
v_add_nc_u32            v2, v2, v2 row_shr:2
v_add_nc_u32            v2, v2, v2 row_shr:4
v_add_nc_u32            v2, v2, v2 row_shr:8
s_and_saveexec_b64      s[6:7], s[6:7]
s_movk_i32              s6, 0xffff
v_permlanex16_b32       v3, v2, -1, s6 fi:1
v_add_nc_u32            v2, v2, v3
v_readlane_b32          s6, v2, 31
s_nand_b64              exec, 0, 0
s_and_saveexec_b64      s[8:9], s[8:9]
v_add_nc_u32            v2, v2, s6
s_mov_b64               exec, s[4:5]
v_sub_nc_u32            v1, v2, v0
v_lshlrev_b32           v1, 2, v1
s_waitcnt               lgkmcnt(0)
buffer_store_dword      v0, v1, s[0:3], 0 offen glc
s_endpgm
```

8-bit and 16-bit storage

- Storing intermediate data in 32-bit all the time would be wasteful
- Color might fit in uint8 * 4 (or pack manually in uint)
- Depth might fit in 16 bpp
- Coverage/AUX state in 8 bpp
- YMMV

Mip-mapping / derivatives

- Simple
  - Each thread works on a pixel group
  - Good luck with that register pressure …
- Shared memory and barriers?
  - Ugh
- subgroupShuffleXor
- NV_compute_shader_derivatives
  - Widely supported on desktop, should be EXT!
  - dFdx/dFdy/fwidth and implicit LOD all in compute
  - Enables subgroupQuad operations
  - Linear ID or 2x2 ID grid

Quad operations in compute is kinda nice

```
uint index = gl_GlobalInvocationID.x;
vec2 width = fwidthFine(uvs[index]);
widths[index] = width;
```

```
s_inst_prefetch         0x0003
s_getpc_b64             s[0:1]
v_mad_u32_u24           v0, s3, lit(0x00000100), v0
s_mov_b32               s0, s2
s_load_dwordx8          s[0:7], s[0:1], null
v_lshlrev_b32           v1, 3, v0
s_waitcnt               lgkmcnt(0)
buffer_load_dwordx2     v[2:3], v1, s[0:3], 0 offen
s_waitcnt               vmcnt(0)
v_mov_b32               v5, v2 quad_perm:[0,0,2,2] bound_ctrl:1
v_mov_b32               v0, v3 quad_perm:[0,0,2,2] bound_ctrl:1
v_mov_b32               v4, v2 quad_perm:[0,1,0,1] bound_ctrl:1
v_mov_b32               v6, v3 quad_perm:[0,1,0,1] bound_ctrl:1
v_sub_f32               v5, v2, v5 quad_perm:[1,1,3,3] bound_ctrl:1
v_sub_f32               v0, v3, v0 quad_perm:[1,1,3,3] bound_ctrl:1
v_sub_f32               v2, v2, v4 quad_perm:[2,3,2,3] bound_ctrl:1
v_sub_f32               v3, v3, v6 quad_perm:[2,3,2,3] bound_ctrl:1
v_add_f32               v2, abs(v5), abs(v2)
v_add_f32               v3, abs(v0), abs(v3)
buffer_store_dwordx2    v[2:3], v1, s[4:7], 0 offen glc
s_endpgm
```

Async compute

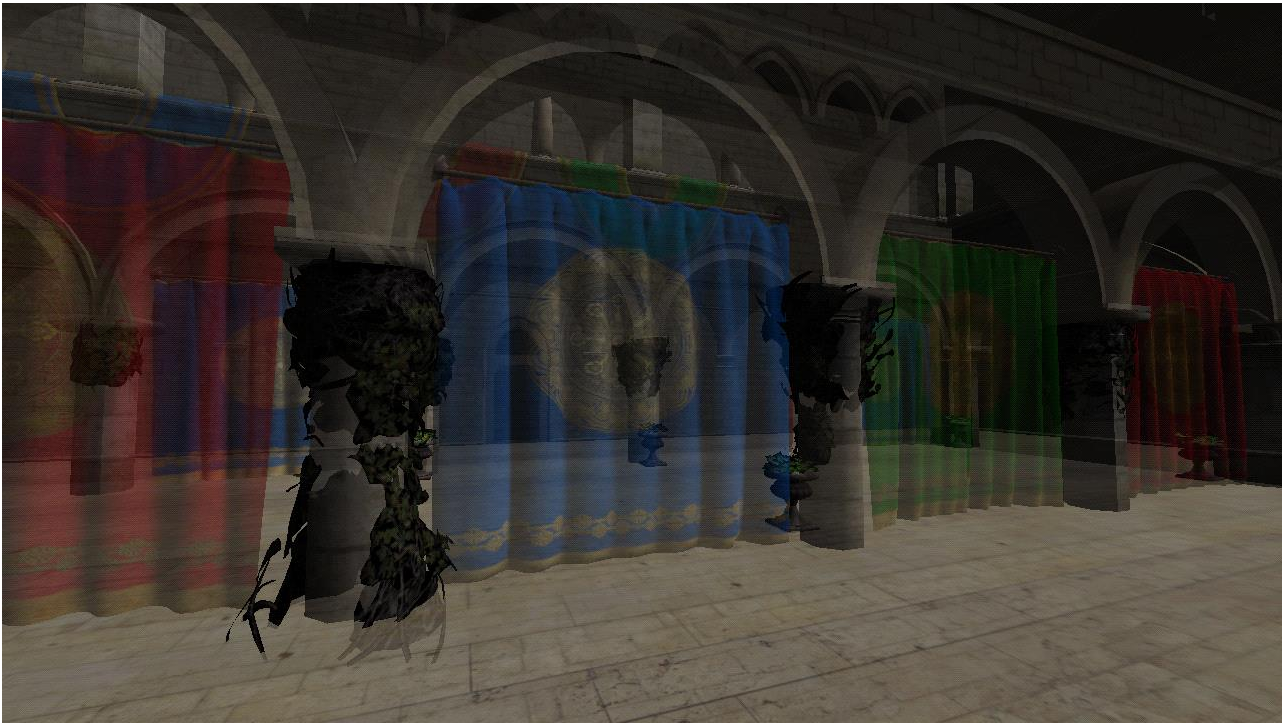
- Pipeline is split in two
- Binning (+ shading if not using ubershader)
  - COMPUTE queue
- Final ROP stage doing depth / blending
  - GRAPHICS queue
  - Only in-order part of pipeline
- Overlaps bandwidth intense work with ALU intensive
  - Need to serialize if using off-screen rendering and consuming result next pass

Test implementation results

A fake retro GPU - RetroWarp

- Sandbox environment
  - https://github.com/Themaister/RetroWarp
  - Uses Granite as Vulkan backend
- Span equation based
- Bilinear / trilinear texture filtering
  - Implemented manually, no texture()
  - Perspective correct
- Trivial texture combiner
- 16-bit color w/ dither, 16-bit depth
- Fully fixed point
  - ... except attribute interpolation (might need 64-bit ints)

Test scene



- Sponza
- 1280x720
  - Overkill, 640x480 is more plausible
- 97309 primitives
  - Post clipping
  - Post back face culling
  - N64 had ~5k primitive budget
- Everything is alpha blended
  - Why not
  - Didn’t sort triangles, looks funny

Results – GTX 1660 Ti

Options	Ubershader (16x16 tiles)	Ubershader (8x8 tiles)	Split shader (16x16 tiles)	Split shader (8x8 tiles)
Subgroups ON Async compute OFF	7.7 ms	9.5 ms	9.0 ms	9.9 ms
Subgroups ON Async compute ON	7.2 ms	8.8 ms	8.2 ms	9.5 ms
Subgroups OFF Async compute OFF	10.6 ms	10.2 ms	9.3 ms	10.5 ms
Subgroups OFF Async compute ON	9.6 ms	9.2 ms	8.4 ms	9.5 ms

- Async compute helps
- Subgroup ops help a lot with ubershader
- The ubershader is not uber enough to trigger issues

Results – RX 5700 XT - Windows

Options	Ubershader (16x16 tiles)	Ubershader (8x8 tiles)	Split shader (16x16 tiles)	Split shader (8x8 tiles)
Subgroups ON Async compute OFF	5.5 ms	5.3 ms	6.3 ms	6.2 ms
Subgroups ON Async compute ON	4.6 ms	4.1 ms	5.0 ms	5.3 ms
Subgroups OFF Async compute OFF	7.8 ms	5.7 ms	6.3 ms	5.9 ms
Subgroups OFF Async compute ON	7.0 ms	4.5 ms	5.0 ms	5.0 ms

- Very similar story here, but 8x8 tiles win here.
- Gain in perf over GTX 1660 Ti correlates well with peak TFlops.

Results – RX 5700 XT – RADV (LLVM)

Options	Ubershader (16x16 tiles)	Ubershader (8x8 tiles)	Split shader (16x16 tiles)	Split shader (8x8 tiles)
Subgroups ON Async compute OFF	11.6 ms	9.9 ms	6.3 ms	5.9 ms
Subgroups ON Async compute ON	11.3 ms	9.3 ms	5.2 ms	5.1 ms
Subgroups OFF Async compute OFF	14.6 ms	10.6 ms	6.4 ms	6.4 ms
Subgroups OFF Async compute ON	14.3 ms	9.9 ms	5.5 ms	5.6 ms

- Here we see catastrophic failure when ubershader is too large.
- Happens eventually on any compiler …
- LLVM compiler got some catching up to do.

Results – UHD 620 – Mesa ANV

Options	Ubershader (16x16 tiles)	Ubershader (8x8 tiles)	Split shader (16x16 tiles)	Split shader (8x8 tiles)
Subgroups ON Async compute OFF	175 ms	155 ms	115 ms	113 ms
Subgroups ON Async compute ON	No change	No change	No change	No change
Subgroups OFF Async compute OFF	493 ms	216 ms	136 ms	131 ms
Subgroups OFF Async compute ON	No change	No change	No change	No change

- This is too much for integrated.
- More realistic resolution and geometry complexity improves it a lot.
- Subgroup ops help a **lot**

Other implementations?

- **High-Performance Software Rasterization on GPUs (2011)**
  - https://research.nvidia.com/sites/default/files/pubs/2011-08_High-Performance-Software-Rasterization/laine2011hpg_paper.pdf
  - CUDA, highly optimized
  - Has similar idea of coarse-then-fine binning
- paraLLEl-RDP (2016)
  - Earlier attempt to emulate N64 RDP in Vulkan compute
- PCSX2 (2014)
  - OpenCL
  - https://github.com/PCSX2/pcsx2/pull/302

Conclusion

- Compute shaders are a viable alternative
  - Allows accurate rendering in real-time
- Subgroup operations can be useful in unexpected places
  - Data sharing without barriers is very nice
- Async compute + graphics queue compute is a thing
- Radeon GPU Analyzer is useful
  - Verifying assumptions with ISA is great

Thanks!

@Themaister
themaister.net/blog
arntzen-software.no