Flexible Rasterizer in OpenCL

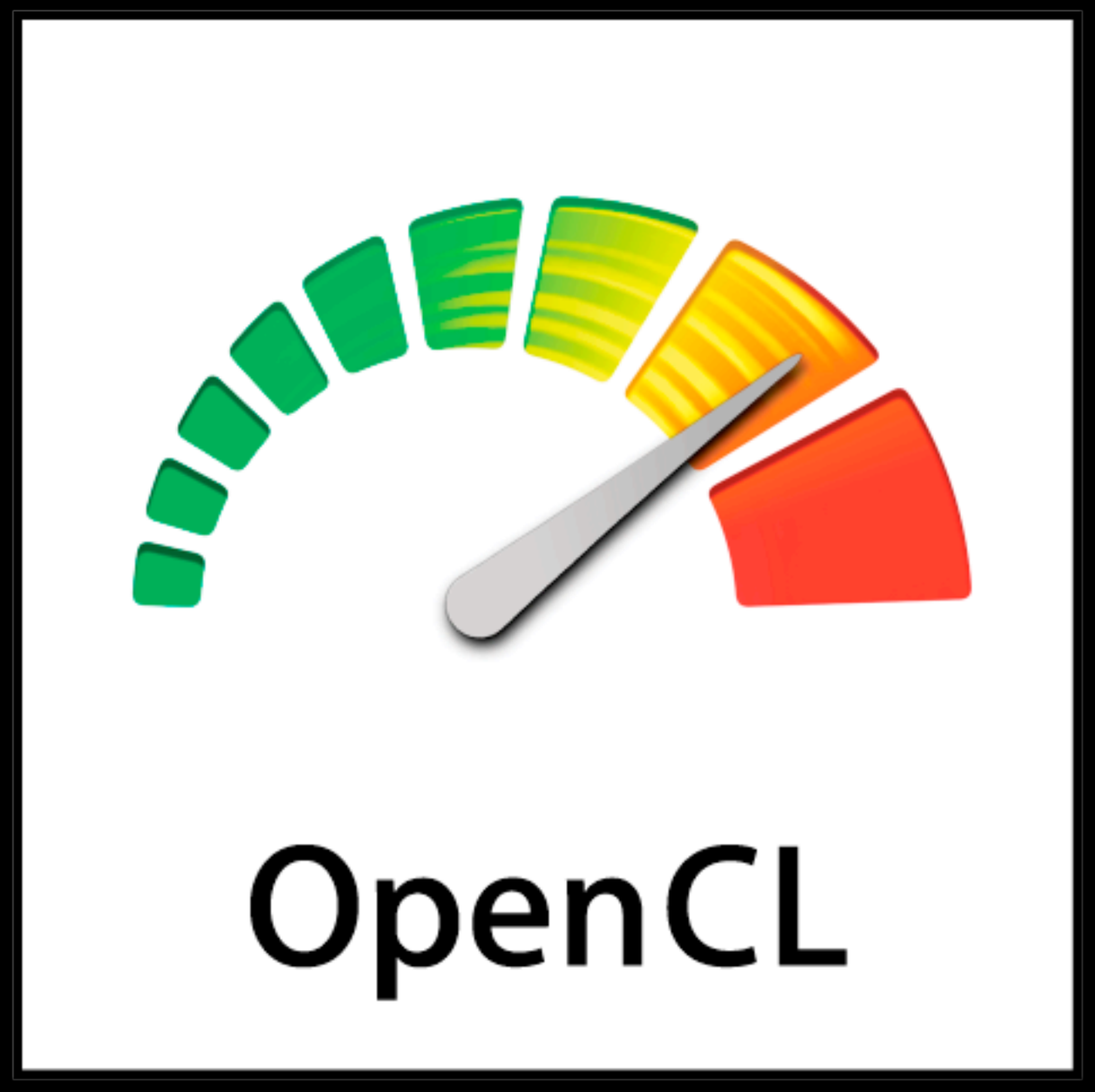
Florian Ziesche

TOC

- Why?
- OpenCL
- Pipeline
- User Frontend (Flexibility)
- Hardware
- Future

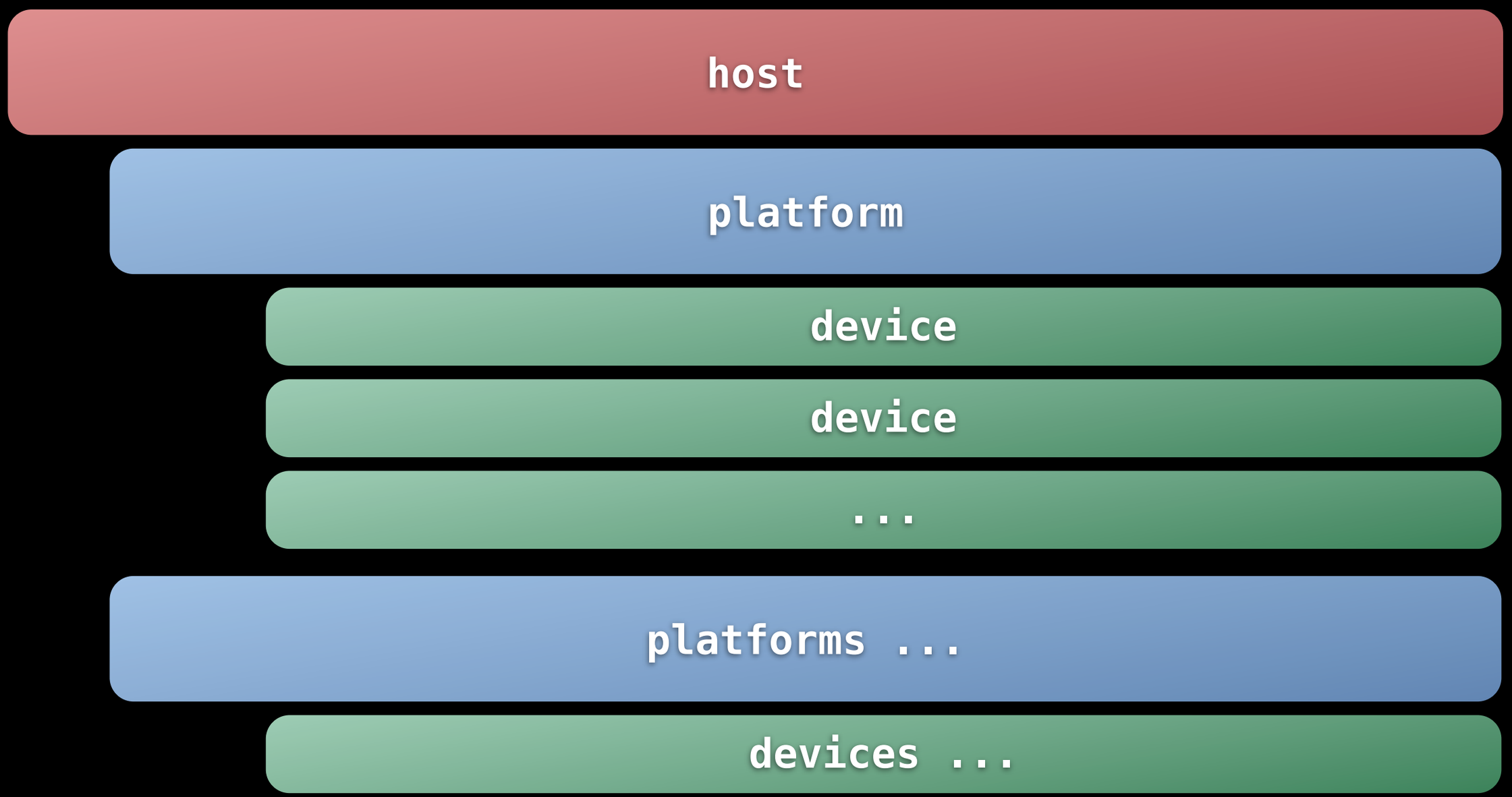
Why?

* full pipeline customization / do things you simply can’t do with OpenGL
* “I want to render Quads again” -> “hybrid rasterization / ray-tracing rendering”
* performance testing / profiling
* run on any OpenCL hardware
* same source code!
* relieve the graphics hardware of some work
* do computations that are possibly faster on a CPU



* “open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform”
* cross platform
* devices: Intel/AMD/ARM/PPC/... CPUs, Nvidia/AMD/Intel/... GPUs, FPGAs, ...

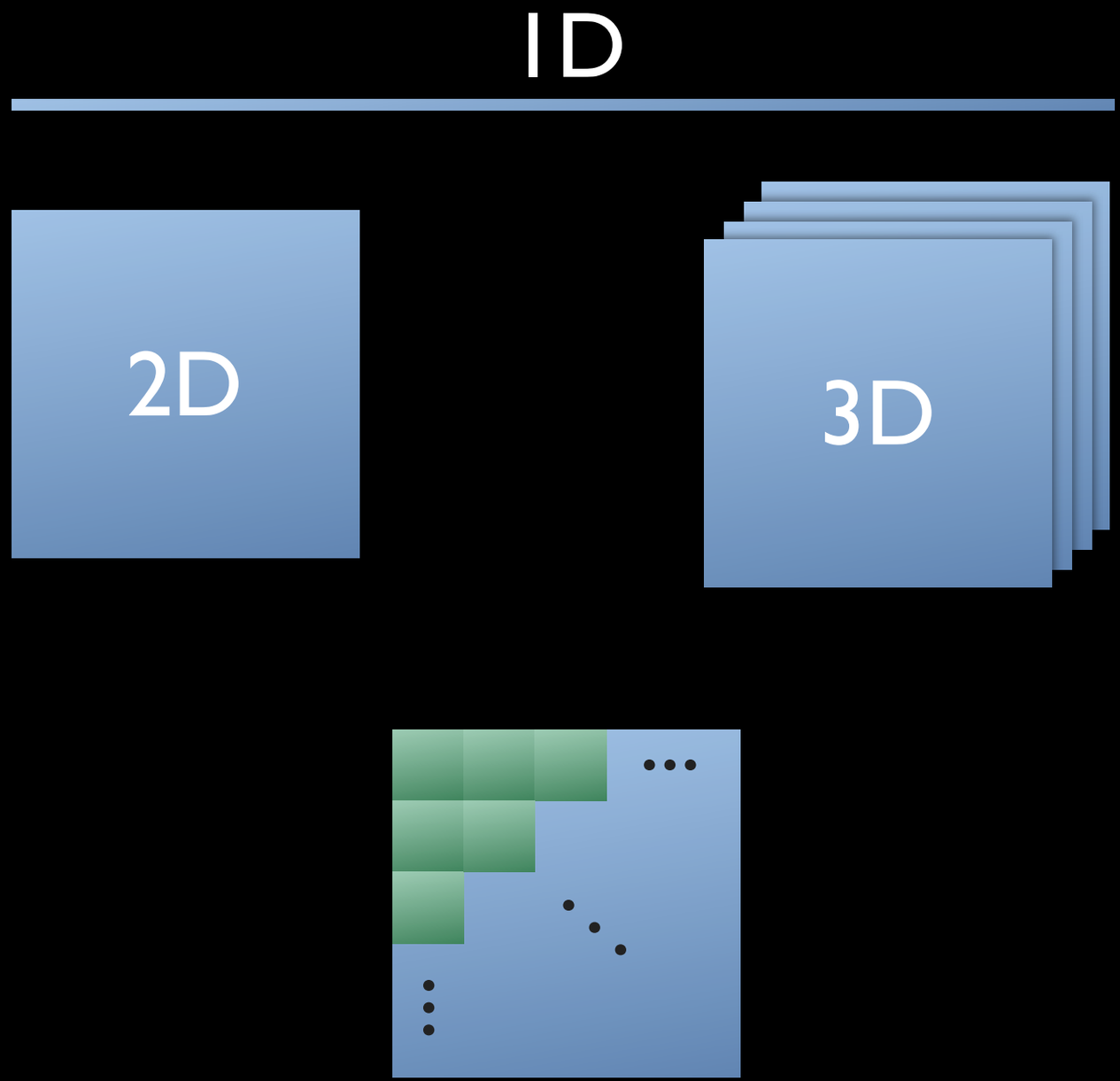
OpenCL Platform



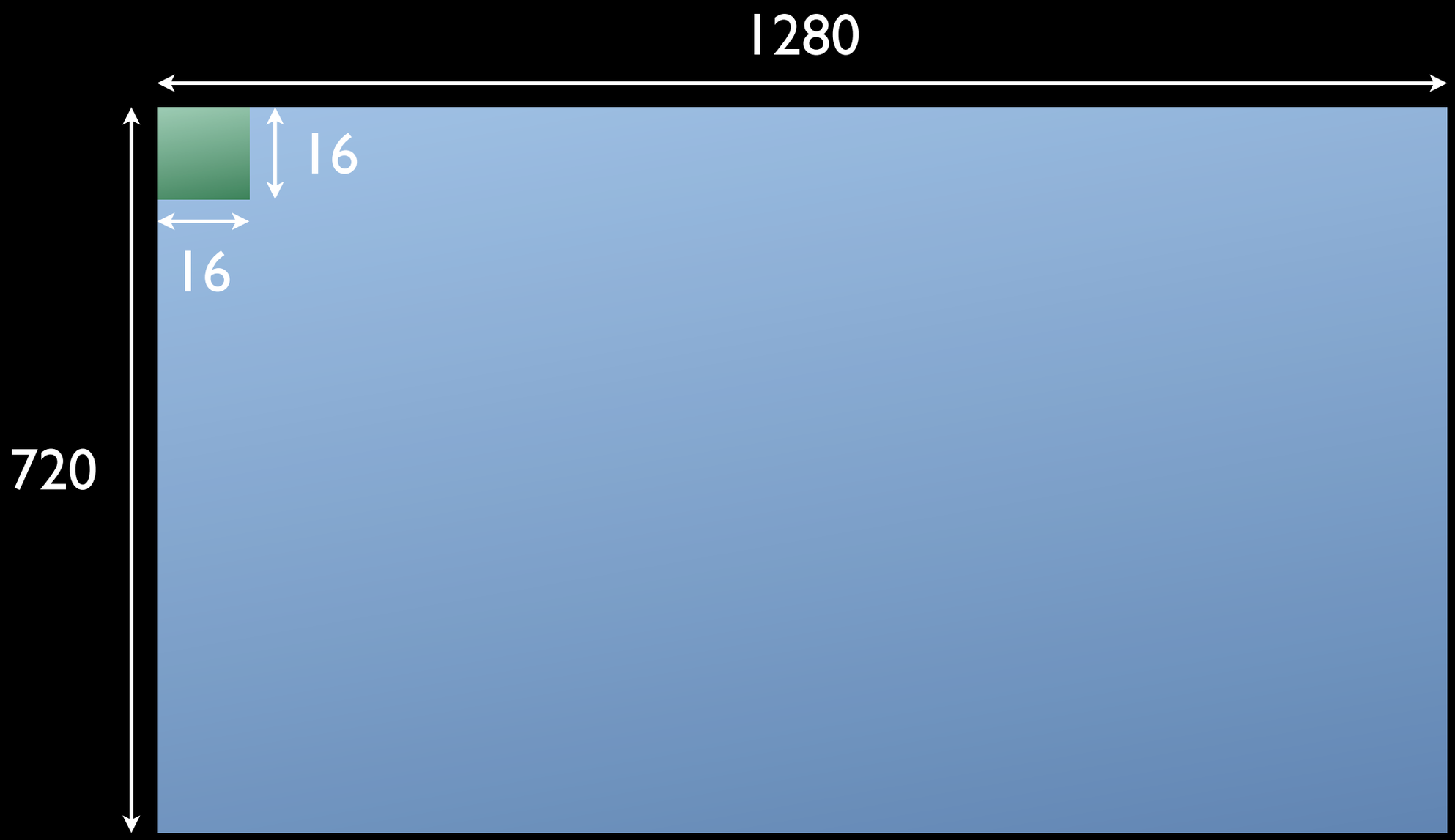
- 1 host, 1+ platforms, 1+ devices per platform

* host ~ platform (at least according to spec)
* device = compute device
* device consists of multiple compute units (“cores”, “compute clusters”, “compute groups”)
* compute units can consist of multiple “processing elements” (“ALUs”)

Execution Model



* SIMD or SPMD (Single Program Multiple Data) depending on device
* program contains kernels (but generally 1:1)
* data split into/accessed in 1D, 2D or 3D ranges (“problem”)
* execution split into work-groups
* work-groups split into work-items
* work-item executes kernel for 1 data element
* work is distributed automatically



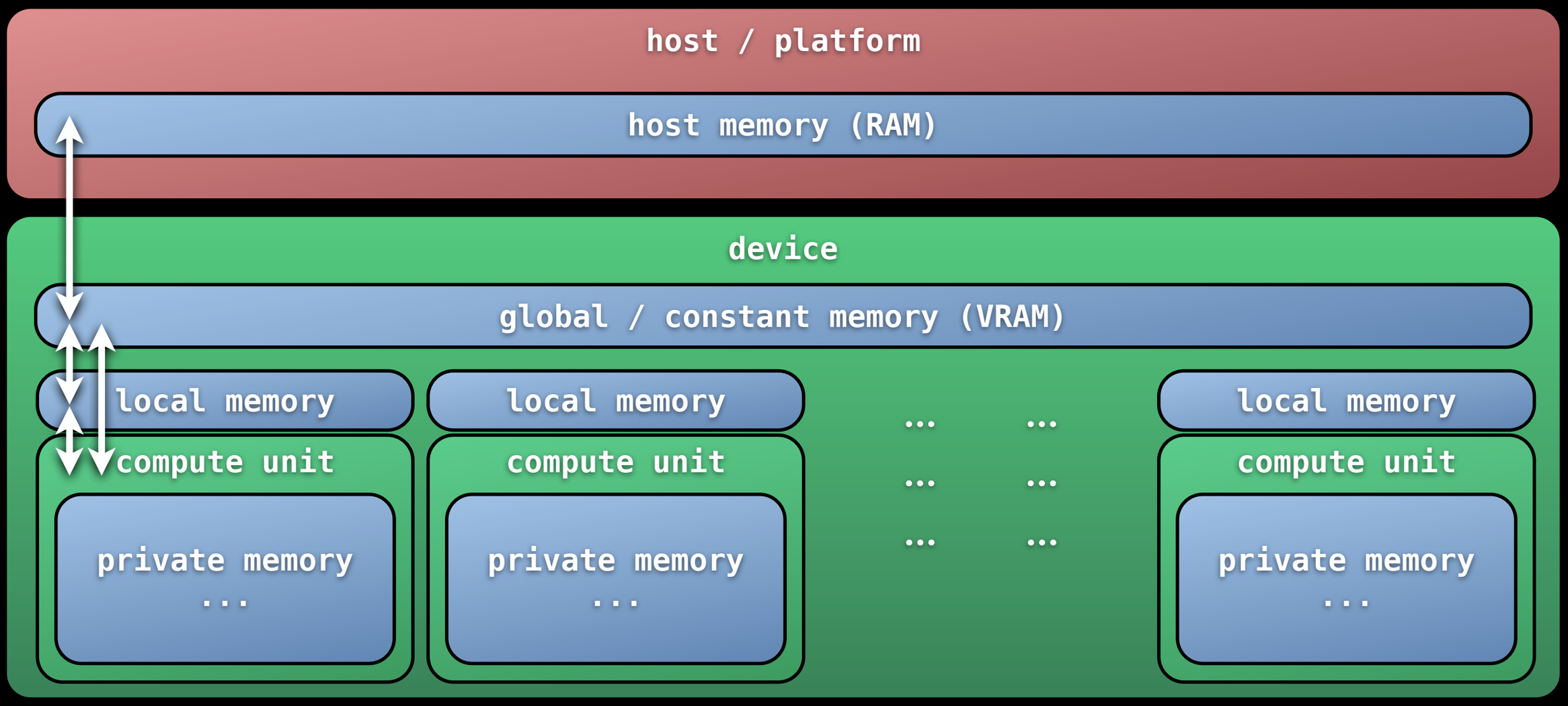
80 * 45 = 3600 work-groups

1280 * 720 = 921600 work-items

* example: 1280 * 720 px framebuffer
-> global 2D range: (1280, 720)
-> local 2D range: (8, 8) or (16, 16) or ...
* device specific!
* for a local range of (16, 16):
-> 80 * 45 = 3600 work-groups
-> 1280 * 720 = 921600 work-items

Memory Model

- address spaces: global, local, private, constant
- host: dynamic allocation, device: static/no allocation



* also: host mapped global memory
* unified address space possible
* global: accessible by both host and device (big, but slow)
* local: accessible only by a work-group (small, mostly fast)
* private: accessible only by a work-item (smaller, fastest)
* constant: usually global memory, but read-only and probably cached

OpenCL Programming

* C99 subset + extensions
* no standard library (headers), but lots of built-in functions (math, image, atomics, ...)
* built-in vector types (2, 3, 4, 8, 16 wide)
* GLSL is very similar (could mostly be regarded as a subset of OpenCL C)
* no dynamic memory allocation
* no recursion
* no variadic macros
* sporadic printf support (CPU only)
* kernels (source code) compiled at runtime
* “binaries” can be retrieved (LLVM byte code, PTX, X86 ASM, ...)
* possible in the future: SPIR (intermediate format, highly influenced by LLVM IR)
* compiler provided by the platform

```
// simple OpenCL kernel
kernel void simple_kernel(global const float4* input,
                          global float4* output) {
   const size_t id = get_global_id(0); // work-item id for dimension #0
   output[id] = clamp(input[id], 0.0f, 1.0f); // clamps input to [0, 1] and writes to output
}
```

```
kernel void interpolate(global const float3* v0,
                        global const float3* v1,
                        global const float3* v2,
                        global float3* dst,
                        const float2 uv) {
   const size_t id = get_global_id(0);
   const float3 v[3] = { v0[id], v1[id], v2[id] };
   const float w = 1.0f - uv.x - uv.y;
   float3 interp_vec = (float3)(0.0f, 0.0f, 0.0f);

   // scalar code
   interp_vec.x = (v[0].x * uv.x) + (v[1].x * uv.y) + (v[2].x * w);
   (&interp_vec)->y = (v[0].y * uv.x) + (v[1].y * uv.y) + (v[2].y * w);
   ((float*)&interp_vec)[2] = (v[0].z * uv.x) + (v[1].z * uv.y) + (v[2].z * w);

   // vector code
   interp_vec = (v[0] * uv.x) + (v[1] * uv.y) + (v[2] * w);

   dst[id] = interp_vec;
}
```

* compile program -> program object -> create kernel object for each kernel in a program object (-> last sample)
* create buffer objects (with appropriate flags)
* set as kernel parameter
* run kernel (and read back results)

```
kernel void histogram(read_only image2d_t image,
                      global unsigned int* buckets) {
   const size_t idx = get_global_id(0);
   const size_t idy = get_global_id(1);
   const sampler_t point_sampler = (CLK_NORMALIZED_COORDS_FALSE | // in [0, texture size - 1]
                                    CLK_FILTER_NEAREST | // nearest sampling
                                    CLK_ADDRESS_NONE); // no address checking
   const float value = read_imagef(image, point_sampler, (int2)(idx, idy)).x;
   const unsigned char uchar_value = convert_uchar_sat(value * 255.0f); // saturated convert
   atomic_inc(&buckets[uchar_value]); // atomically increment value at address
}
```

- init buckets: 0-ed 256*4 byte buffer
- when done: read back results on the host

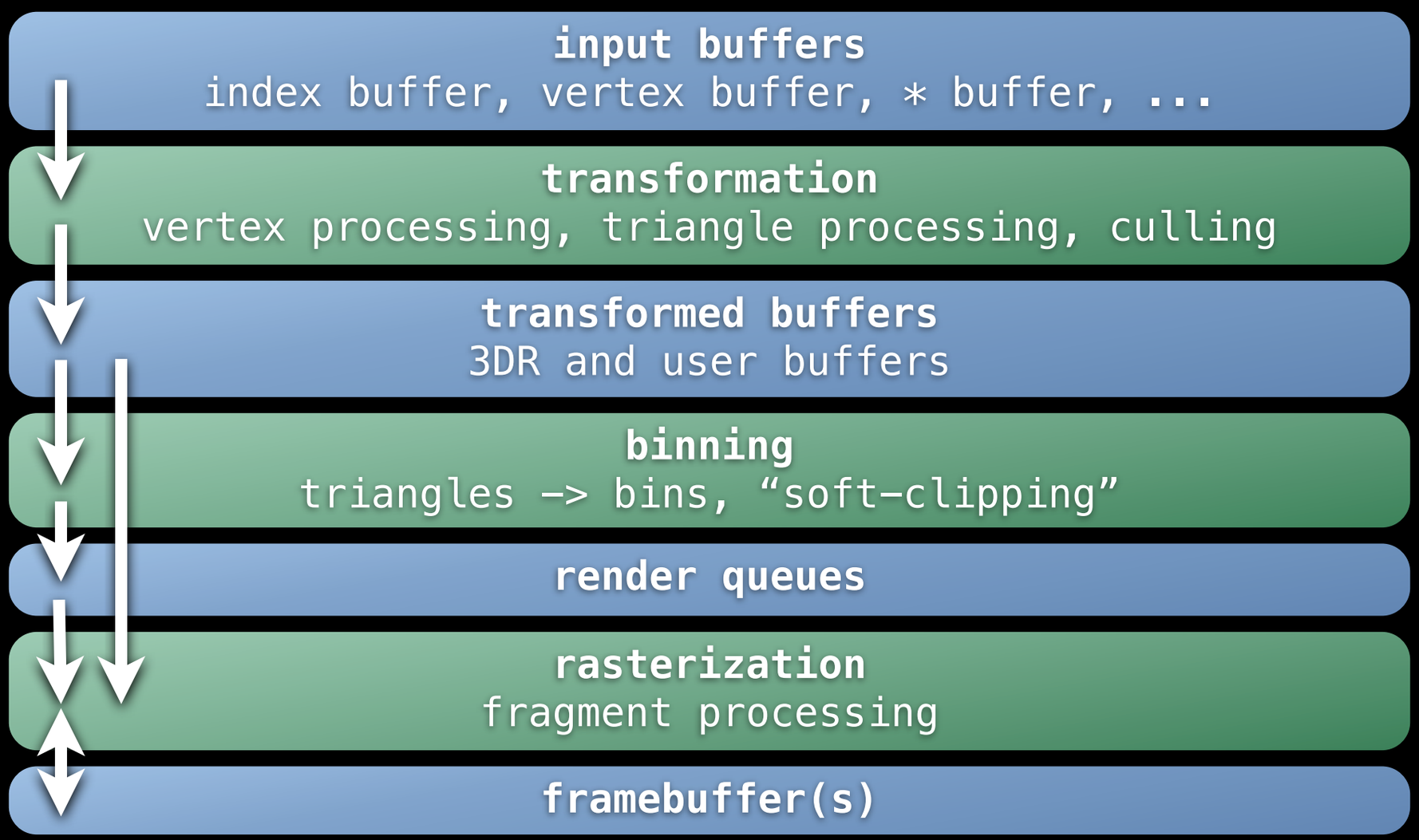
```cpp
#include <iostream>
#include <vector>
#define __CL_ENABLE_EXCEPTIONS
#include "cl.hpp"
using namespace std;

static constexpr char kernel_code[] { u8R"(
   kernel void nop_kernel(const unsigned int dummy_val) {
         printf("absolutely nothing: %d\n", get_global_id(0));
   })"
};

int main(int argc, char* argv[]) {
   cl_int err = CL_SUCCESS;
   try {
         vector<cl::Platform> platforms;
         cl::Platform::get(&platforms);
         if(platforms.size() == 0) {
             cerr << "no opencl platform found" << endl;
             return -1;
         }
         cl_context_properties properties[] { CL_CONTEXT_PLATFORM, (cl_context_properties)platforms[0](), 0 };
         cl::Context context(CL_DEVICE_TYPE_CPU, properties);

         vector<cl::Device> devices { context.getInfo<CL_CONTEXT_DEVICES>() };

         cl::Program::Sources source(1, { kernel_code, sizeof(kernel_code) });
         cl::Program program(context, source);
         program.build(devices);

         cl::Kernel kernel(program, "nop_kernel", &err);
         cl::Event event;
         cl::CommandQueue queue(context, devices[0], 0, &err);
         kernel.setArg(0, 42);
         queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(23), cl::NullRange, nullptr, &event);
         event.wait();
   }
   catch(cl::Error ex) {
         cerr << "ERROR: " << ex.what() << "(" << ex.err() << ")" << endl;
   }
   return 0;
}
```

* uses the C++ CL wrapper (cl.hpp)
* _really_ basic setup and kernel execution
* global range: 23 (1D)
* local range: automatic (NullRange)
* raw string literals are awesome

Pipeline

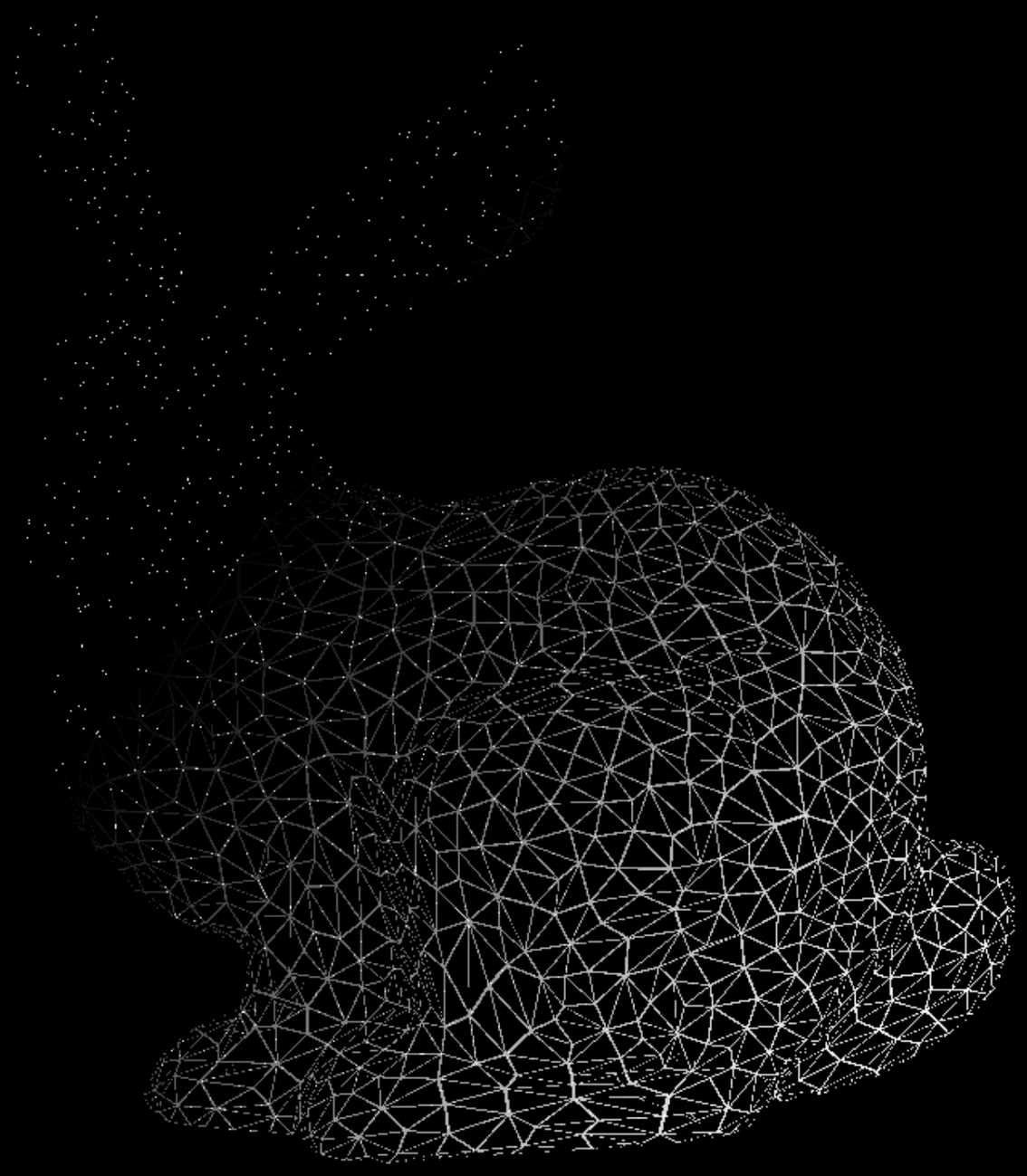
- papers and other references:
  - Laine and Karras "*High-Performance Software Rasterization on GPUs*"
  - Dachsbacher, Slusallek, Davidovic, Engelhardt, Georgiev "*3D Rasterization: A Bridge between Rasterization and Ray Casting*"
  - Fabian "ryg" Giesen "*A trip through the Graphics Pipeline 2011*"

Pipeline



* pipeline that will be implemented
* transform and rasterize are easily user-programmable
* partial or complete replacement of the pipeline will be possible
* big challenge: make it fast and easily programmable!

Transform Stage

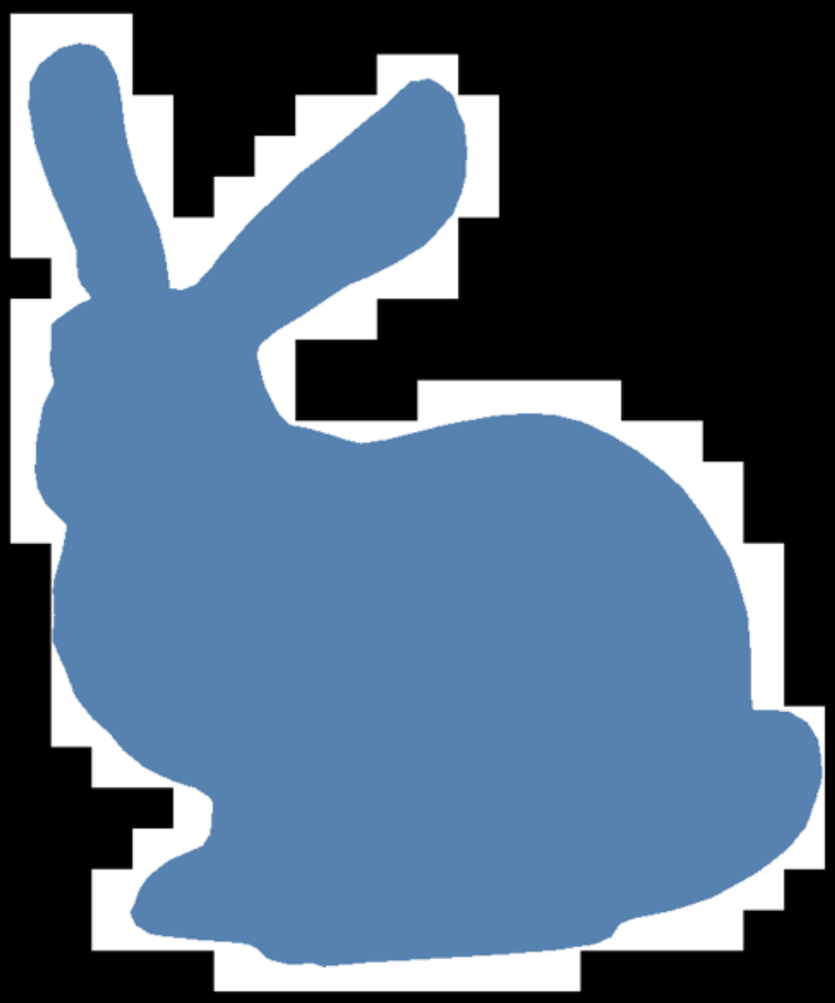


* combination of all primitive processing programs of a regular hardware graphics pipeline
* input data can be accessed by-vertex and/or by-primitive
* frustum culling + backface culling + removal of invalid/degenerate triangles
* input order will be maintained, unless otherwise specified (possibly faster)
* max transformed buffer size must be known beforehand
 * future extension: allocate huge device buffers and handle memory (output) manually (of course still problematic)
* clipping (part 1):
 * if triangle is fully visible/invisible (doesn't clip any frustum plane or all are clipped)
 * -> can already cull invisible triangles

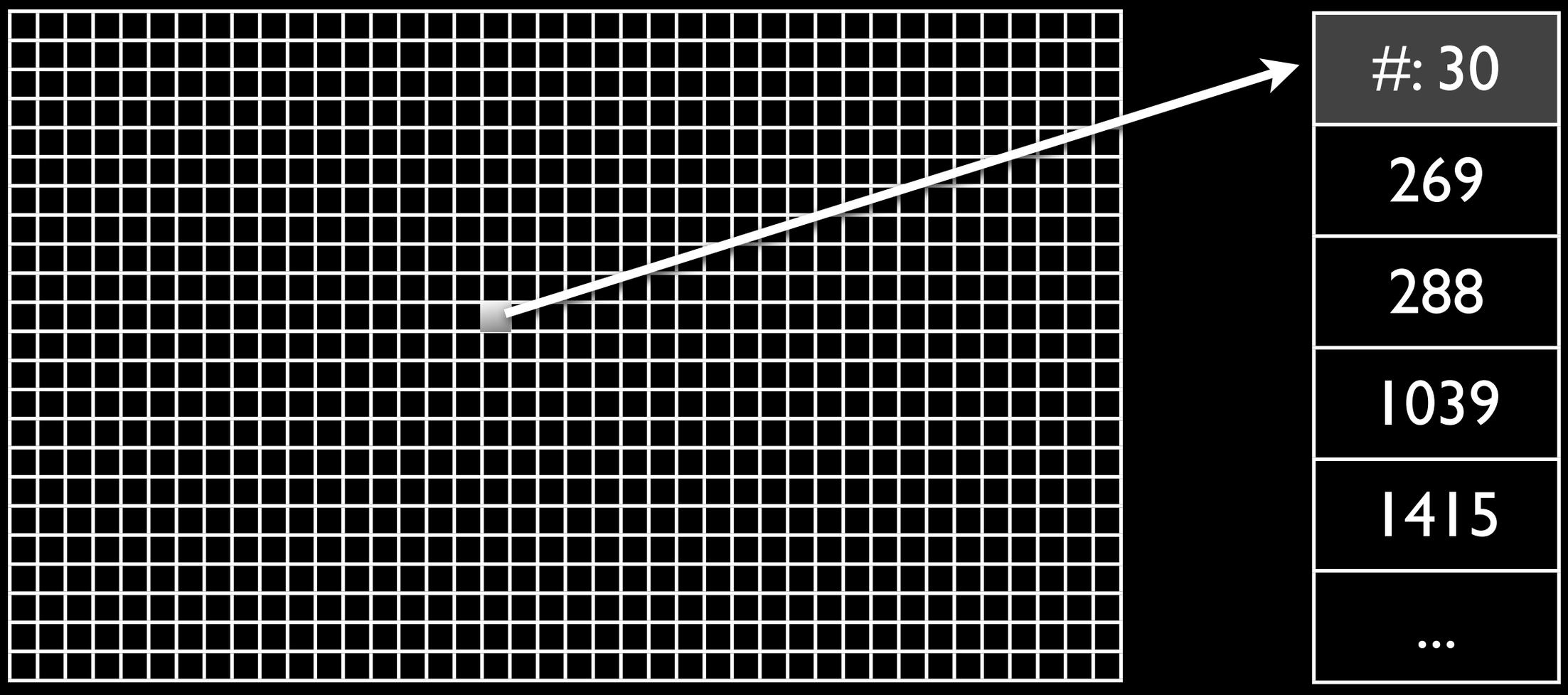
- output:
  - 3D rasterization variables (V0, V1, V2, depth)
  - transformed user data

* 3DR output (-> paper)
* user data is handled automatically (just a copy, except for the transformed data)

Binning



* screen: areas/bins that are white have at least 1 triangle (overdrawn by the actual bunny to put it into relation)
* clipping (part 2):
 * compute vertex screen coordinates and clip coordinates (reverse to rasterization):
 * line/viewport intersection
 * determine valid coordinates and compute min/max bound (all bins in that range)
 * future extension: compute slope from coordinates and compute affected bins perfectly

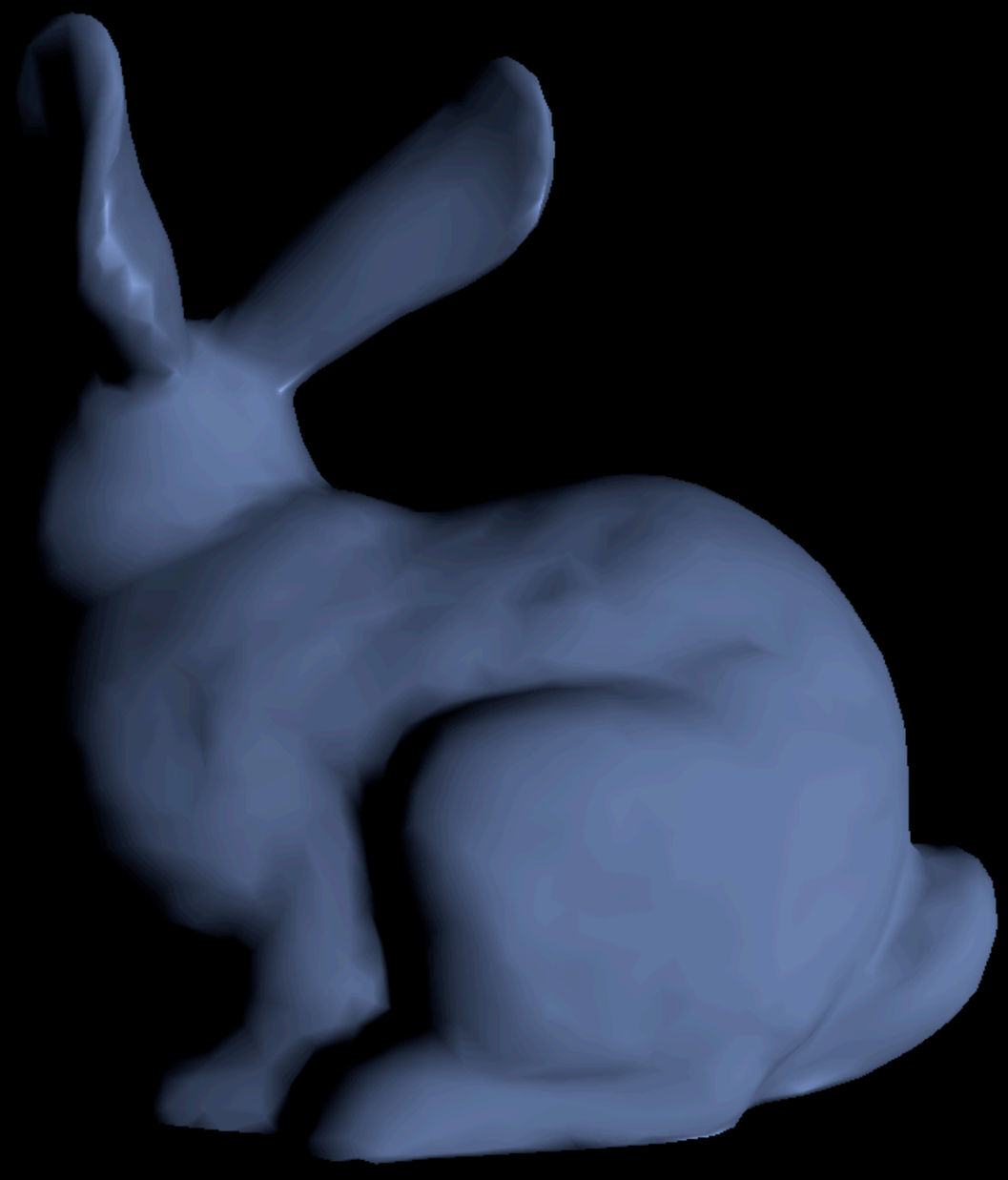


* Binning:
 * atomically add triangle id to appropriate queues/bins
 * queue/bin size is pre-determined (16*16px, 32*32px, ...)
 * size should match the used hardware (full utilization prefered)
  * should be larger on devices with many processing elements per compute unit (e.g. 192 ALUs -> at least 16*24px, better 2 or 4 times as much)
* alternative: only schedule each compute unit once and “allocate”/distribute/request work in kernel dynamically (always full utilization, but sync overhead)

- also done in this stage:
  - final clipping
  - scissor test
  - (really early) Z-Culling possible

* Z-Culling: if depth is not written by the user in the fine rasterizer and a non-empty depth buffer is bound, triangles that wouldn't pass the depth test could already by culled here (needs min/max values in depth buffer -> check against vertex depth)
* scissor test: if scissor-testing is enabled, the non-passing tile bins are not “scheduled”/ created in the first place (overlap -> rasterization stage)

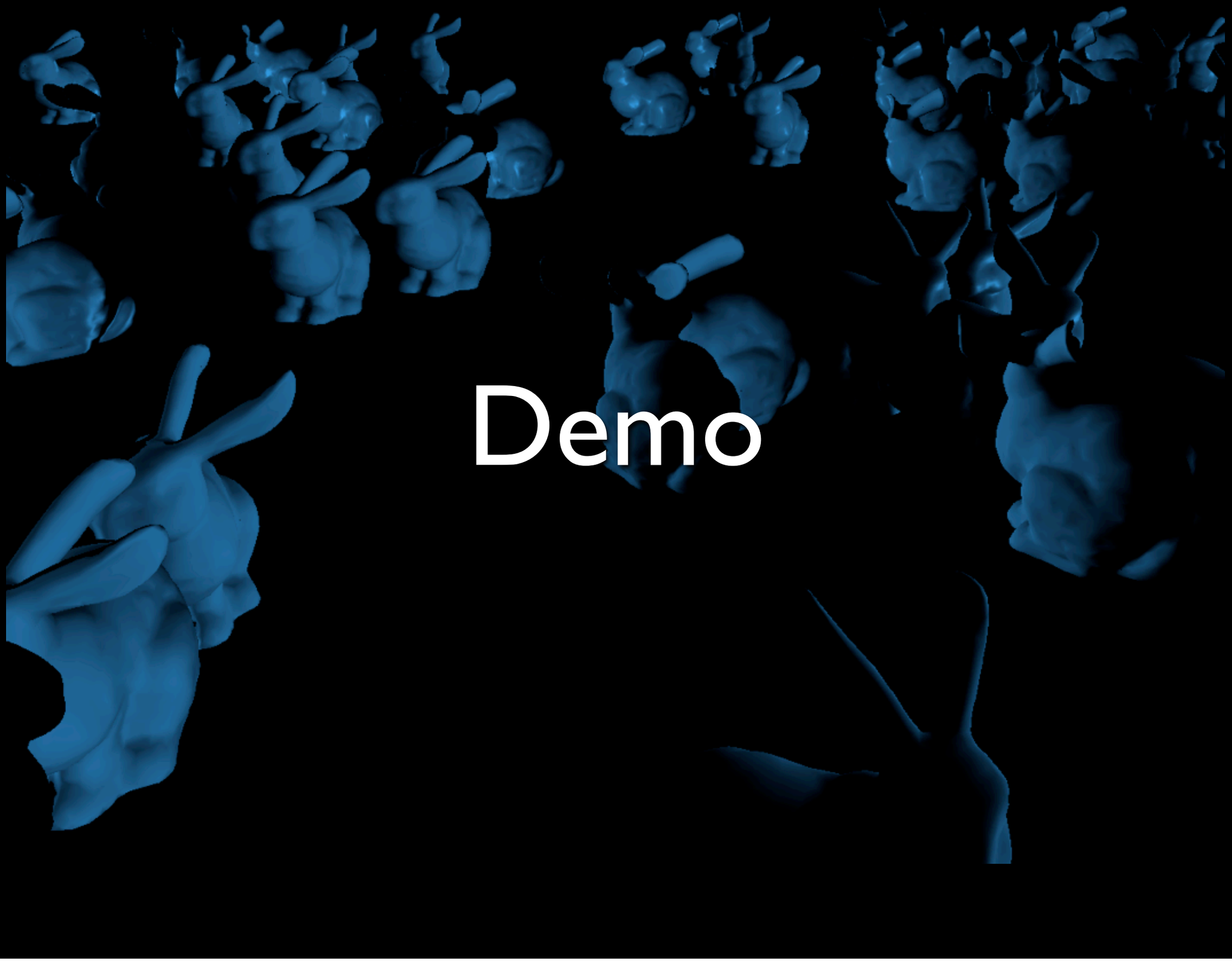
Rasterization Stage



* “fragment shader”
* 3DR: compute barycentric coordinates and depth
* depth testing / culling
* interpolate user data and call user code
* read and write fragments from/to local memory and store final fragment in global memory (framebuffer)

- read previous framebuffer data
- depth testing
- call user program
- when queue is done: write color + depth

* previous data: color and depth (prev. color and depth are always! available)
* depth testing:
 * linear depth and no (forced) far-plane
 * future: user specified depth test (for now: will add GL style depth testing)
 * cull non-passing fragments before calling the user code (if possible)
* again: (fine) scissor test
* framebuffer and texture handling:
* limits imposed by OpenCL (max image parameters, max sizes, formats, ...)
* for now: keeping it simple - only 2D
* otherwise: use global memory buffers


Demo

User Frontend

3 options:

* plain old vertex / fragment shaders
* partial pipeline customization
* complete customization / direct access

Vertex / Fragment Shaders

or: transform and rasterization stage programs

* simple option: plain old vertex and fragment shaders
* every graphics programmer knows what these are and what you can do with these
* easy migration or testing
* language and interface similar to GLSL
* OpenCL C, shader input/output declarations, uniforms, images, ...

```
// vertex program
oclraster_in simple_input {
   float4 vertex;
   float4 normal;
   float2 tex_coord;
} input_attributes;

oclraster_out simple_output {
   float4 vertex;
   float4 normal;
   float2 tex_coord;
} output_attributes;

oclraster_uniforms transform_uniforms {
   mat4 modelview_matrix;
} tp_uniforms;

void transform_main() {
   output_attributes->normal = input_attributes->normal;
   output_attributes->tex_coord = input_attributes->tex_coord;

   float4 mv_vertex = mat4_mul_vec4(tp_uniforms->modelview_matrix,
                                    input_attributes->vertex);
   output_attributes->vertex = mv_vertex;
   transform(mv_vertex);
}
```

input: user data, index, untransformed vertex

* projection is not done by the user -> transform has to be called!
* output buffer is created automatically, user can just write to it (pointer is given)
* later: possibility to call this per triangle or per primitive -> tessellation, triangle transformation, anything* is possible

```
// fragment program
oclraster_out simple_output {
   float4 vertex;
   float4 normal;
   float2 tex_coord;
} output_attributes;

oclraster_uniforms rasterize_uniforms {
   float4 camera_position;
   float4 light_position; // .w = light radius ^ 2
   float4 light_color;
} rp_uniforms;

void rasterize_main() {
   // phong lighting
   float3 light_dir = rp_uniforms->light_position.xyz - output_attributes->vertex.xyz;
   light_dir /= rp_uniforms->light_position.w;
   const float attenuation = 1.0f - dot(light_dir, light_dir) * rp_uniforms->light_position.w;
   if(attenuation > 0.0f) {
        light_dir = normalize(light_dir);

        float3 diff_color = (float3)(0.0f, 0.0f, 0.0f);
        float3 spec_color = (float3)(0.0f, 0.0f, 0.0f);
        const float lambert_term = dot(output_attributes->normal.xyz, light_dir);
        if(lambert_term > 0.0f) {
            diff_color = rp_uniforms->light_color.xyz * lambert_term * attenuation;

            float3 view_dir = normalize(rp_uniforms->camera_position.xyz -
                                        output_attributes->vertex.xyz);
            float3 R = reflect(-light_dir, output_attributes->normal.xyz);
            float specular = pow(max(dot(R, view_dir), 0.0f), 16.0f);
            spec_color = rp_uniforms->light_color.xyz * attenuation * specular;
        }

        *fragment_color = (float4)(diff_color + spec_color, 1.0f);
   }
}
```

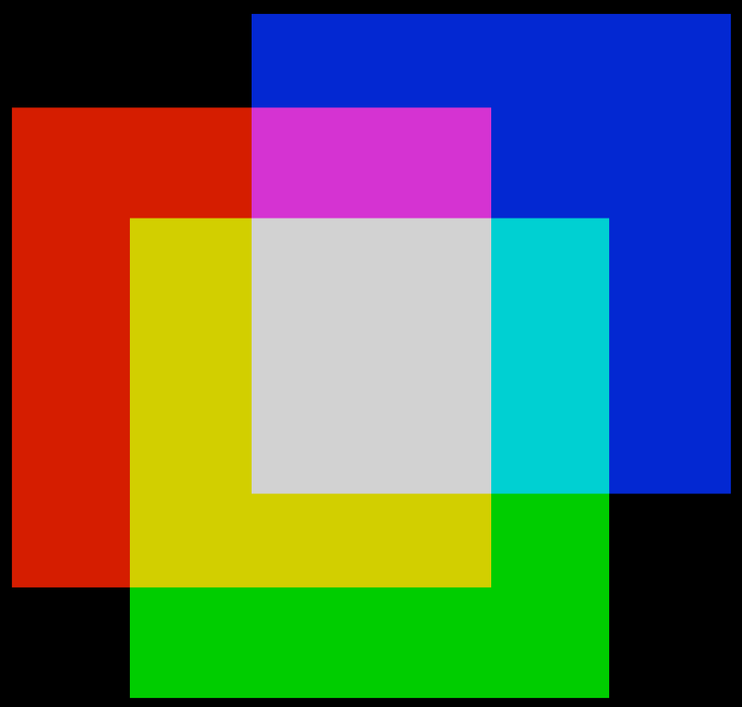
input: user data*, depth, fragment coord, barycentric coords

* user data is interpolated automatically (might need a specifier to not interpolate data)
* fragment_color (float4*) is a temporary solution, this will be replaced by an actual “write to framebuffer” function
* later: custom blending, depth testing, sampling, ...

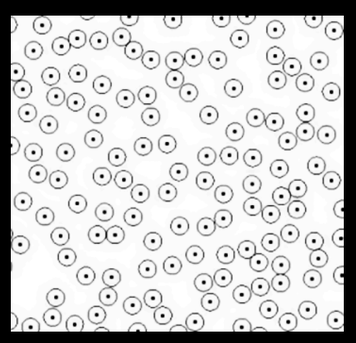
- combine, compile and use/bind as usual
- OpenCL C
- some limitations

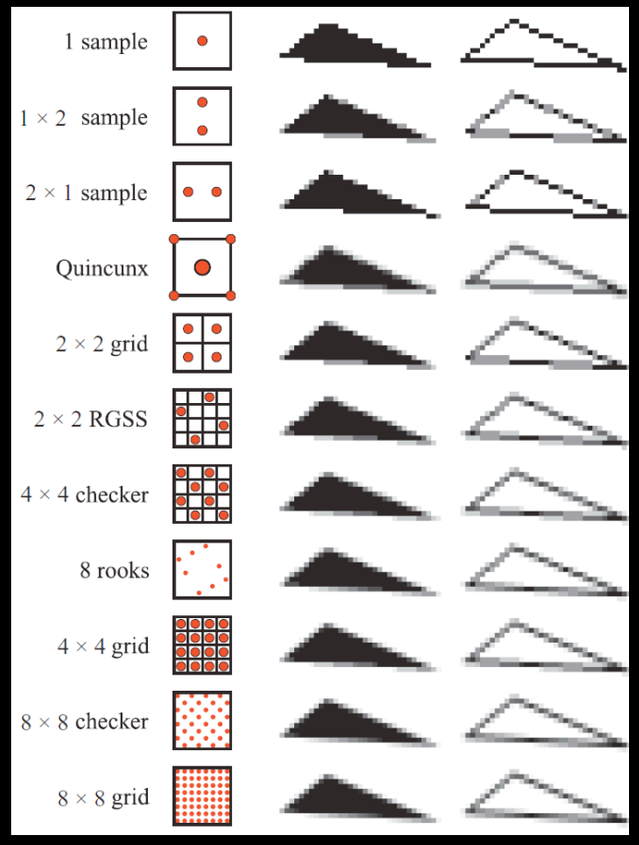
* sadly: no built-in matrix types and functions
* limitations for oclraster structs (in/out/uniforms):
* no preprocessor (this would require a compiler)
* no interior structs/unions (will possibly allow this)
* no multi-variable declarations (e.g. "float x, y, z;")
* no user defined types are allowed (again: requires a compiler)
* use of any oclraster struct specifier in other places is disallowed (no typedefs, comments, ...)
* otherwise standard C

Partial Pipeline Customization



```
f 153/184/106 151/180/106 121/241/106
f 153/184/106 121/241/106 123/238/106
f 118/242/107 148/179/107 150/178/107
f 118/242/107 150/178/107 120/239/107
```



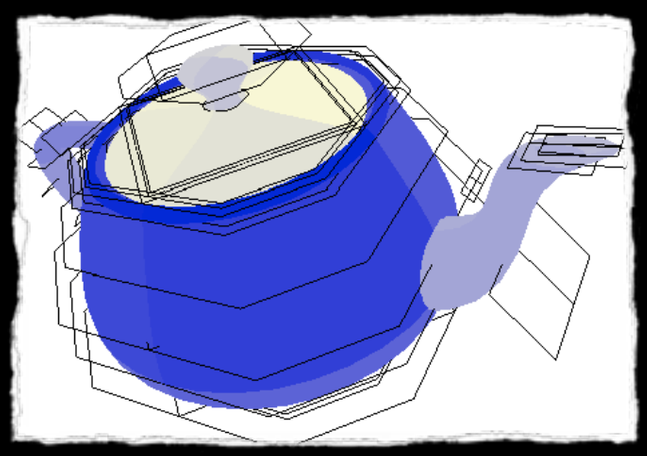


* replace transform and/or rasterization stage
* will directly support this

* possibilities in the transform stage:
* self-defined primitive types
* multi-index buffers (-> .obj and others)
* per triangle/primitive shader/program
* direct buffer access (access other triangles or vertex attributes)
* no actual input, generate input/output in program

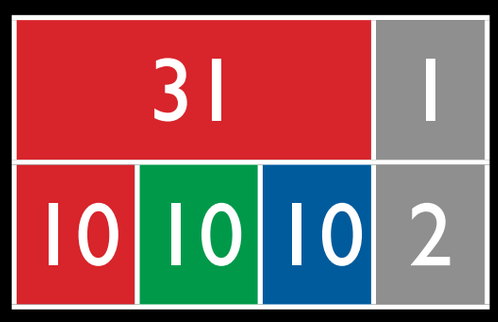
* possibilities in the fragment stage:
* multi-framebuffer support with self-defined types
* custom anti-aliasing and sampling methods (e.g. custom sample patterns; only do multi-sampling on triangle borders)
* transformed buffers and render queues could be stored and used multiple times (beneficial for certain multi-pass renderers, e.g. light pre-pass rendering, deferred/inferred rendering)
* custom blending

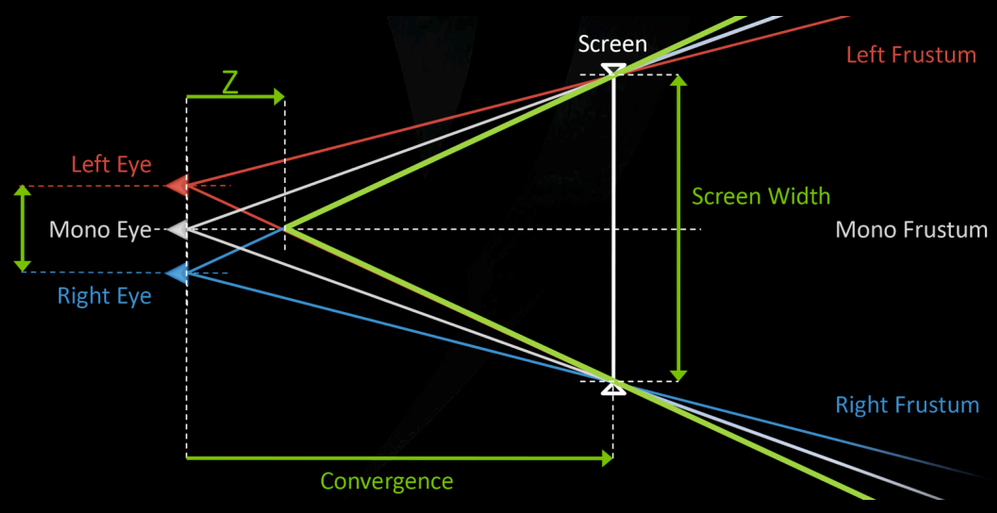
Complete Pipeline Customization



CMYK

YCrCb





* “replace anything”
* will also need custom pipeline code on the host side

* combinations of previous partial pipeline customizations
* direct framebuffer access (access pixels directly, or simplified geometry -> pixel mapping, e.g. orthographic projection)
* more compact stereoscopic rendering (render both images at the same time)
* rasterization / ray-tracing combination
* direct rendering of non-triangle primitives (e.g. Bézier surfaces, variably sized polygons)
* custom framebuffer/texture formats (although faster if hardware support)
* 3DR: transform+render into a non-planar viewport (also needs custom clipping!)
* generally: many optimization and simplification possibilities if you know what you are doing and what you want

Hardware

- tested on Nvidia GPUs, Intel CPUs, AMD CPUs and ARM so far
- will hopefully be testing on a Xeon Phi
- possibly multi-device rendering
- runs on iOS/ARM (just for the fun)

* iOS demo? (buggy and slow, but essentially working)
* K20/Titan? ;)

Future

- always: need larger + faster memory and caches
- access to fixed rasterization hardware not necessarily beneficial
- better debugging tools + protected memory ...
- OpenCL C++ (need templates and classes)

* available cache and/or local memory per ALU on latest Nvidia consumer hardware compared to previous hardware is laughable (256 bytes max vs 1536 bytes max for L1, ~341 bytes max vs 1536 bytes max for L2)
* for a seamless transform stage: need huge pre-allocated transform buffers
* fixed function hardware: too limited to be of use / no generic solution to this
* need an actual opencl source code level debugger (might go the CUDA route)
* graphics hardware is easily crashable; TDR/similar doesn't always help -> trial and reboot
* C++ support would simplify a lot of things (actual matrix type, custom texturing/image support, specialization, ...)

Future

- better OpenCL support
- multi-device support
- User Frontend additions
- OpenCL ⇒ CUDA translator/wrapper

* better OpenCL support: specifically looking at Nvidia/AMD software side
* multi-device support: CPU+GPU+... rendering (frame interleaving or in-frame splitting)
* possible additions: easy tessellation support
* OpenCL -> CUDA translator
 * for performance comparison and GPU debugging support
 * mostly working (no easy way to support textures/images, CUDA requires more info)

The End

- clEnqueueQA()
- Code and *this available at: https://github.com/a2flo/oclraster

Image Credits

- https://developer.apple.com/softwarelicensing/agreements/opencl.html
- http://mynameismjp.wordpress.com/2012/10/24/msaa-overview/
- http://www.geeks3d.com/20100628/3d-programming-ready-to-use-64-sample-poisson-disc/
- http://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_07
- http://developer.download.nvidia.com/assets/gamedev/docs/Siggraph2011-Stereoscopy_From_XY_to_Z-SG.pdf
- http://cg.ibds.kit.edu/publications/p2012/3dr/gi2012.pdf