Flexible Rasterizer in OpenCL

Florian Ziesche

May 28, 2013

This is not the official print version!

(although the content is identical)

Contents

1. Introduction

Today's real-time computer graphics are mostly dominated by the two graphics APIs OpenGL and Direct3D, and their derivates, which provide a hardware abstraction on top of GPUs and a standardized way to program a predetermined graphics pipeline. While certain parts of these hardware graphics pipelines are programmable, other parts are not programmable at all or only allow a low degree of fixed-function parameter tweaking. At the same time, the graphics pipeline as a whole remains a largely rigid setup that doesn't allow any rearrangement or simplification of pipeline stages or the complete removal of specific stages that are deemed necessary. Further restrictions are a result of the software APIs and hardware vendors themselves, as they decide on how, which and where hardware features are exposed, which can lead to situations where only a subset of all possible GPU features are available to programmers.

This thesis will demonstrate the implementation of an essentially OpenGL 2.0-level software graphics pipeline, called *OCLRaster* (short for OpenCL Rasterizer), with the addition of some unique features and functionality of more recent OpenGL versions, but also the exclusion of some other features. The pipeline is written and accelerated by OpenCL C on the device side and C++ on the host side, and is capable of running on all OpenCL 1.1 desktop hardware. This includes most modern GPUs and CPUs.

Among the main goals are to provide a simple host API and an easy way to program the vertex and fragment stage, with the direct intention of being similar to a hardware graphics pipeline and API, and accordingly requiring no modification of the pipeline. Both of these should allow for a rather uncomplicated migration of OpenGL programs.

On the other hand, this project should also provide functionality that facilitates complete or partial customization of the graphics pipeline.

In regard to the implemented features, this software pipeline supports fully programmable depth testing and blending, which are both not possible on today's graphics hardware, instanced rendering, scissor testing, the previously mentioned vertex and fragment stage programmability, miscellaneous buffer objects in a simplified and unified way, 2D images, framebuffers and multiple render targets with less restrictions than hardware pipelines, and of course rendering with perspective and orthographic projection modes. Other OpenGL 2.0-level features are however not supported. These include stencil testing (which can however be partially simulated in software by simply using an additional framebuffer attachment), antialiasing, 1D and 3D images, occlusion querrying and all of the now obsolete legacy

draw functions and modes. The reasons for this are not of any technical nature that would prevent their implementation, but rather due to the time constraints of this thesis.

Additional benefits that a software pipeline might provide over a hardware pipeline is to perform functions that hardware simply can't do. This can be, for example, the direct rendering of user-defined non-triangle primitives (Bézier surfaces or actual quads), or the already mentioned programmable blending and depth testing, or custom image and framebuffer formats with the possibility of arbitrary types (if implemented in software), or stripping parts of the pipeline that aren't used or necessary or extend the pipeline by a new stage (image or fullscreen shader). Hybrid techniques are also possible, e.g. a more direct approach at combining rasterization and ray-tracing [DEG+12] or performing occlusion culling on the CPU [Int13] or generally relieving the GPU by offloading some work to any other free device.

In general, a software pipeline, and OpenCL in particular, allow for a more flexible and heterogeneous way of pipeline programming, as the same highly modifiable code runs on all devices and all of these devices are able to take part in the rendering of a frame. Furthermore, adjustment and creation of pipeline stages, direct access to rasterization variables and buffers and strong programming language support via OpenCL all lead to a kind of programming that is closer to a classical CPU-centric software pipeline than it is to today's GPU-centric hardware pipeline.

2. OpenCL

After the advent of modern GPU programming with CUDA in 2007 [NVI07] and earlier GPGPU adventures like BrookGPU [BFH+04b, BFH+04a] or Close to Metal [AMD06] that were closely hardware dependent, the need for an open, standardized, cross-platform framework for parallel programming of modern processors arose. Initiated by Apple and in close collaboration with the major CPU and GPU vendors AMD, Intel and NVIDIA (among others), the first draft specification was handed over to the Khronos Group in June 2008 where the Khronos Compute Working Group (now OpenCL Working Group) was formed [Khr08a, Khr08c]. Following a fast standardization process, the final OpenCL 1.0 specification was released and approved by all Khronos members in December 2008 [Khr08b]. Revisions to the standard were released in 2010 with OpenCL 1.1 [Khr10] and in 2011 with OpenCL 1.2 [Khr11], which was further updated with additional extensions in November 2012 [Khr12] and is, as of this writing, the most current version of OpenCL.

OpenCL has been widely adopted by many hardware vendors and today runs on a multitude of hardware ranging from NVIDIA and AMD GPUs, to Intel, IBM, ARM and AMD CPUs, to Texas Instruments DSPs, to Xilinx and Altera FPGAs.

This thesis and its implementation are based on OpenCL 1.1, as this is the most widely available OpenCL standard as of today that also provides a certain minimum set of features that are required for its operation. Furthermore, the implementation has been tested on and should run on all three major desktop operating systems (Windows, OS X and Linux) on top of the CPU and GPU OpenCL implementations of AMD, Apple, Intel and NVIDIA. This translates to OpenGL 4 capable GPUs, SSE2+ capable CPUs (AMD platform) and SSE4.2+ CPUs (Intel platform). At the beginning, the project has been shortly tested on iOS 6.1 as well, but further development has been dropped, since iOS only supports the ARM CPU device and OpenCL support as-is is still inofficial and experimental with many remaining implementation issues. In general, OpenCL and this project should also run on most modern mobile graphics hardware. Imagination and ARM both advertise OpenCL 1.1 compliance for their current mobile GPUs [Ima13, ARM13, Khr13b], but widespread support is still missing.

This chapter will mostly reference chapter 3, *The OpenCL Architecture*, of the OpenCL specification [OWG12a] and show how it maps to actual hardware, with the inclusion of a few examples.

2.1. OpenCL Platform Model

From top to bottom, the OpenCL platform model consists of a *host*, i.e. the operating system and its attached hardware (CPU, RAM, ...) that controls everything, one or more OpenCL implementations (the *OpenCL platforms*) provided by the OS or the respective hardware vendors, which in turn give access to one or more *OpenCL devices* (CPUs, GPUs, ...).

Devices are further divided up into multiple *compute units*, which are physically represented by individual *CPU cores* (usually logical CPU cores or H/W threads), *(streaming) multiprocessors* (NVIDIA GPUs [NVI12]), or combined *SIMD units* (AMD GPUs [AMD12]). Note that multiple platforms can give access to the same devices (e.g., this is the case when you have both the Intel and the AMD OpenCL implementations installed with each providing access to the CPU device).

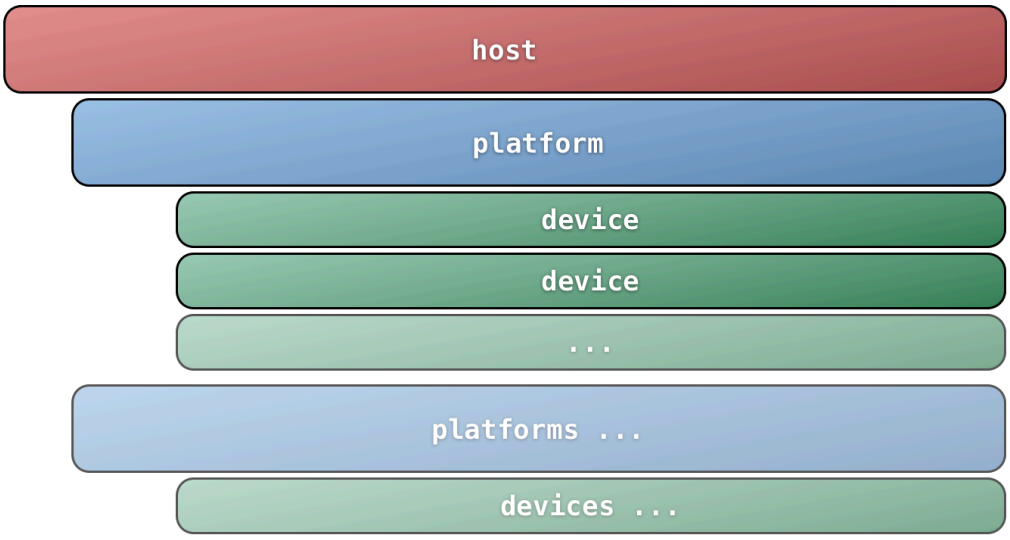

Figure 1: OpenCL Platform Model

Compute units are divided up into one or more *processing elements* that are responsible for the actual execution of OpenCL device code. These are represented either by single *compute cores* (NVIDIA), or again by *SIMD units* (AMD) or logical *CPU cores*.

In most cases, multiple processing elements are not physically distinguishable as their execution is bundled in SIMD units (which currently range from 4-wide with SSE, to 8-wide with AVX, to 16-wide on AMD GPUs and Intel MICs, to 32-wide so called *warps* on NVIDIA hardware). On hardware that purely consists out of SIMD units (GPUs), branched execution, or any kind of execution that requires less processing elements than the width of the SIMD unit, is generally achieved through an execution mask that simply *disables* the unnecessary parts of the unit

(i.e. these parts are unused during execution). Additionally, there are usually several SIMD units per compute unit, which makes program execution within a compute unit an odd mixture of SIMD and SPMD that one should be very aware of when programming such devices (know the execution order, where to put memory fences and barriers and how these are executed).

2.2. OpenCL Execution Model

OpenCL execution happens in so called *OpenCL kernels*, which are programs written in OpenCL C that are compiled for and run on each OpenCL device. Kernels are organized in *OpenCL programs*, which contain many different kernels for multiple devices, but the relation is usually 1:1 (at least in this project, for simplicity reasons).

OpenCL kernel execution occurs in 1D, 2D or 3D ranges that describe a specific problem size, i.e. they determine into how many parts the workload is split and in what kind of dimensionality the problem is processed in. This range is called the *global range*. The work is then further divided into smaller *local ranges* that are processed by so called *work-groups*. Work-groups consist of numerous *work-items*, which finally perform the actual kernel execution. Each work-item executes a kernel exactly once.

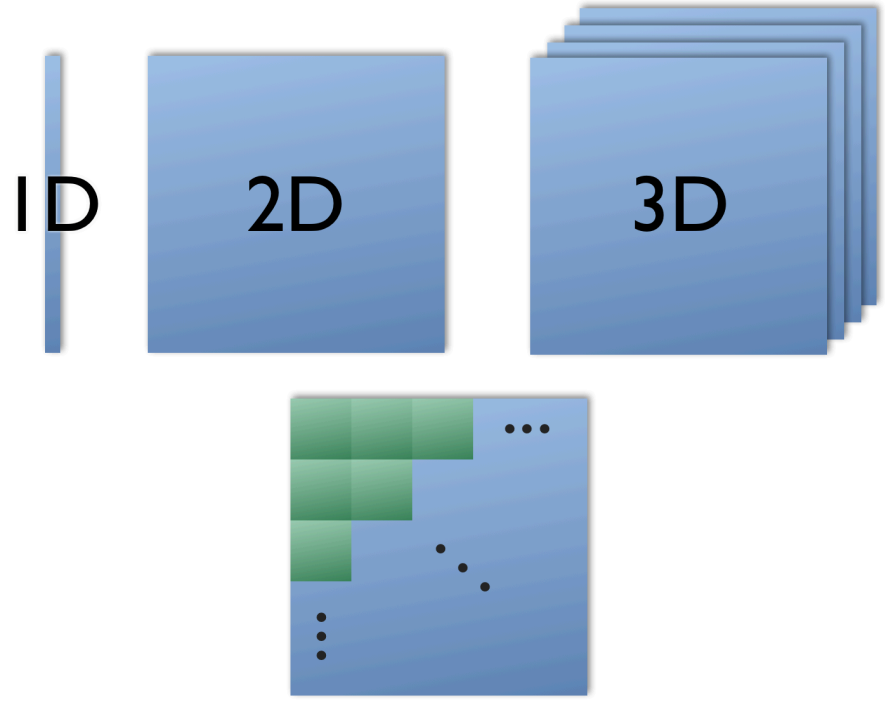

Figure 2: OpenCL Execution Model:
global ranges (blue), local ranges (green)

Furthermore, the user doesn't have to care about the work distribution, as this is entirely and automatically done by the OpenCL implementation and the OpenCL device according to the specified global and local range. There are however ways to circumvent this to some degree, mainly by using persistent kernels and distributing the workload manually with some form of global synchronization (as will be demonstated later on).

In hardware, kernel execution translates as follows: one work-group is processed by exactly one compute unit, work-items within the work-group are distributed to all processing elements within the compute unit (SIMD units). After a work-group execution has concluded, the compute unit is assigned the next work-group, and so on, until all work-groups have been completed.

As an example, let's imagine that we want to perform some kind of per-pixel operation on a framebuffer. This framebuffer is $1280px * 720px$ in size, which consequently leads to a global 2D range of $(1280, 720)$. Determining the local range is not as obvious, since it highly depends on the hardware and how many processing elements per compute unit it possesses. There are however hints, as well as device and kernel specific information that OpenCL provides that aid in computing the best local range. For now, let's assume a local 2D range of $(16, 16)$, i.e. 256 work-items per work-group. In total, this gives us $80 * 45 = 3600$ work-groups and $3600 * 256 = 921600$ work-items. In this setup, each work-item will process exactly one pixel.

Figure 3: OpenCL Execution Model example

2.3. OpenCL Memory Model

OpenCL knows four distinct device address spaces: *global* memory, *local* memory, *private* memoy and *constant* memory. *Global memory* is represented by big, but relatively slow device RAM that all compute units and the host have access to (read and write). *Local memory* is directly coupled with every compute unit and is only accessible to the processing elements of a compute unit (read and write), although the available amount of local memory can be controlled from the host, which has otherwise no access to it. Its speed is usually very high, while its size is very limited: the OpenCL 1.1 specification requries at least 32KiB of local memory (on current hardware this equates to 48KiB on Fermi and Kepler NVIDIA GPUs and 32KiB on AMD GPUs and Intel and AMD CPUs). *Private memory* is the memory that is only available to a processing element, usually represented by hardware registers and thus is the fastest, but also the smallest type of memory. Private memory is completely inaccessible from the host. *Constant memory* is a small amount of memory (at least 64KiB) that is read-only to all compute units, but can be written to from the host. It is generally located in the global memory and access to it is cached on most hardware, but this can vary from device to device.

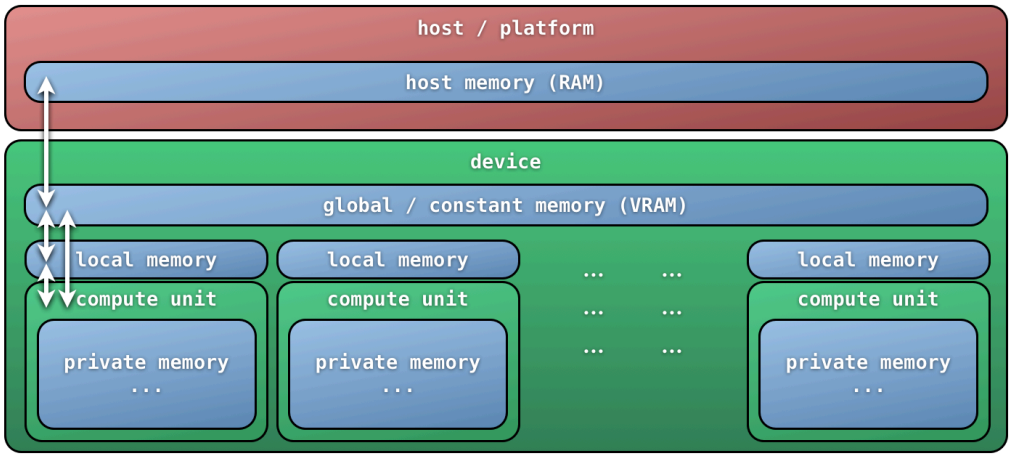

Figure 4: OpenCL Memory Model

For all types of memory except global memory, devices are only capable of static allocation, i.e. the size must be known at compile time. Allocation of global memory is not possible at all from the device. The host is capable of dynamically allocating global, constant and local memory.

OpenCL 1.1 also introduced *host unified memory*, which makes host memory (RAM) directly accessible from the OpenCL device. This is mostly beneficial for devices that share the same physical memory (e.g. integrated GPUs or CPU devices themselves) or in cases where explicitly copying data from the device to the

host or allocating huge amounts of device memory is unwanted (note however that these cases rarely give any, if any at all, performance boost, as the data still has to copied in some way).

2.4. OpenCL Programming

OpenCL devices are programmed in OpenCL C, which is basically a subset of ISO C99 with a few extensions. OpenCL C does not provide a C standard library, and in fact, does not supply any header files at all. OpenCL uses a different model, in which all built-in functions are available everywhere. This follows the model that other GPU-centric languages have previously used, including GLSL, HLSL and Cg, but it might slightly confuse more CPU-centric programmers.

Nevertheless, OpenCL offers a huge amount of built-in functions, as described in chapter 6.12 of the specification [OWG12a]. Math, integer, geometric and relational functions are mostly identical to what OpenGL provides with GLSL, with a few extra functions in OpenCL that are mostly known from the C standard library. In addition to those, OpenCL also provides atomic functions (both to global and local memory, as required since OpenCL 1.1), work-group internal synchronization functions (barrier and memory fence functions), direct image read and write functions (as an extension, even write support to 3D images), explicit vector load and store functions, asynchronous memory copy functions and optionally a printf function.

An additional extension that OpenCL C delivers is the built-in support for vector types of 2-, 3-, 4-, 8- and 16-width for all natively supported data types (float, int, uint, ...). This allows for easy writing of vector code that can be conveniently utilized by OpenCL compilers to directly generate SIMD code.

Among the C99 features that were removed from OpenCL C are dynamic memory allocation (on the device), function recursion (mostly due to earlier GPU hardware not supporting this) and variadic macros (although not really a hardware limitation, and supported by most compilers). Further restrictions apply to the address space qualifiers, determining how and where these can be used. More extensive information is available in chapter 6.9 of the OpenCL specification [OWG12a].

OpenCL kernel source code is usually compiled at runtime by the OpenCL C compiler that is provided with each OpenCL platform implementation. OpenCL also allows to retrieve compiled program binaries. The binary format is however specific to each OpenCL implementation and each device and might even differ across versions of the same OpenCL implementation. Thus, it is not recommended to ship any kind of kernel binaries, but always provide and compile kernel source code

at runtime. In the future, this might be partially mitigated by SPIR [OWG12b], which is a vendor neutral intermediate bytecode representation that is based on LLVM IR. It should be relatively easy to implement, as most OpenCL compilers are Clang/LLVM based.

To provide some insight into OpenCL C, I will show and shortly explain two simple OpenCL kernels. I will not show any OpenCL host code in here, as this is rather uninteresting and not really helpful at this point, as OCLRaster builds upon *cl.hpp* [OWG13], the C++ abstraction layer on top of the OpenCL host interfaces and provides additional functionality on top of that, so that users of this project won't have to write any direct OpenCL host code.

Onto the first example:

```
kernel void simple_kernel(global const float4* input,
                          global float4* output) {
    // work-item id for dimension #0
    const size_t id = get_global_id(0);

    // clamps input to [0, 1] and writes it to output
    output[id] = clamp(input[id], 0.0f, 1.0f);
}
```

Listing 1: a simple OpenCL kernel

Kernel functions are always specified using the `kernel` qualifier and always have to return `void`. Kernel function parameters are completely in the hands of the user, with some restrictions that apply to the allowed address space qualifiers (e.g. no private memory parameters). In this example, the kernel function is supplied with two global memory buffers, the `input` buffer being read-only (`const`), and the `output` buffer having read-write access. Inside the kernel function, we first fetch the global ID (index or unique ID in $[0, \textit{global range})$), which is then used to get the id-*th* element of the input buffer, clamp its value and write it to the id-*th* position of the output buffer. As one might also notice, both buffers use 4-width float vector types for their elements, which OpenCL is easily and directly able to handle.

The second example is slightly more complex and shows some more advanced OpenCL C features (image sampling, type conversion and atomic functions):

```
kernel void histogram(read_only image2d_t image,
                      global unsigned int* buckets) {
    // global range is in 2D
    const size_t idx = get_global_id(0);
    const size_t idy = get_global_id(1);

    // define an image sampler, necessary for reading from images
    const sampler_t point_sampler = (// coordinates are in [0, image size)
                                     CLK_NORMALIZED_COORDS_FALSE |
                                     // nearest/point sampling
                                     CLK_FILTER_NEAREST |
                                     // no address checking
                                     CLK_ADDRESS_NONE);

    // sample the image at coordinate (idx, idy) using the point sampler
    const float value = read_imagef(image, point_sampler, (int2)(idx, idy)).x;

    // explicit saturated convert (value will be clamped to [0, 255])
    const unsigned char uchar_value = convert_uchar_sat(value * 255.0f);

    // atomically increment the count for value
    // note that atomic_inc only takes a global or local address
    atomic_inc(&buckets[uchar_value]);
}
```

Listing 2: a simple histogram kernel in OpenCL

This example computes the histogram of an 8-bit grayscale 2D image. Consequently, the global range is in 2D and should match the image size. The image is provided as the first kernel parameter (also explicitly being read-only) and the bucket buffer is provided as the second parameter. The bucket buffer should contain 256 elements of type `unsigned int` and should be zero initialized on the host side.

Again, we first retrieve the global ID, but this time in 2D. We then create an image sampler that is always required when reading (sampling) images, which in this case happens with nearest (or point) filtering with coordinates that are not normalized (we directly address the image texel using absolute integer coordinates) and also don't require address checking (this tells OpenCL that coordinates are guaranteed to be valid - other modes allow for coordinate clamping, mirroring and repeating). We then continue to actually read the texel as a normalized float value. This is followed by a multiply by 255 to get the value into $[0, 255]$ range and a saturated convert to an 8-bit `unsigned char`. And finally, the bucket buffers `uchar_value`-*th* element is atomically increased.

Note that this kernel is purely for demonstration purposes and shouldn't be used in real-world code. For one thing, OpenCL also allows direct reading of (non-normalized) `int` and `unsigned int` image values. Additionally, performing atomic operations on such a low amount of values is not recommended and will result in a lot of waiting as multiple work-items try to atomically increase the same value at the same time. It would be better to synchronize the bucket buffer in local memory first and then write it to the global buffer, or write some kind of reduction kernel.

3. Previous and Related Work

This project mostly builds upon the *3D Rasterization* paper [DEG+12] for its rasterization part, with some simplifications when using orthographic rendering. On the pipeline side, it is heavily inspired by and expands on the *High-Performance Software Rasterization on GPUs* paper [LK11].

3.1. High-Performance Software Rasterization on GPUs

This paper already demonstrated that a GPU based software graphics pipeline is possible and that performance in the range of 2x - 8x slower than a hardware graphics pipeline can be achieved. The implementation is however written in CUDA and very GPU dependent, as all work-sizes and its design closely match the specific GPU hardware that was used in the paper. This project, and OpenCL, necessitate a more universal approach to that.

CUDA also provides work-group internal synchronization functions that OpenCL unfortunately lacks, thus making a direct OpenCL portation impossible. Some of its general work distribution and synchronization methods were however helpful and have been utilized in OCLRaster, as will be shown later on.

Additionally, OCLRaster draws upon its general pipeline design, however merging the *bin rasterizer* and *coarse rasterizer* to a single *binning stage* and splitting the *triangle setup stage* into two parts, the *transform stage* and the *primitive processing & assembly stage*. These will be discussed in chapter 5. In the same way as this paper, OCLRaster also preserves the primitive input order and guarantees hole-free rasterization.

Furthermore, OCLRaster gets rid of the viewport limitation of this paper and allows arbitrary viewport sizes that are only limited by the amount of framebuffer memory that can be allocated with OpenCL on a device.

3.2. 3D Rasterization

This paper introduces 3D triangle edge equations based on the Plücker ray-triangle intersection test. These can be used as an alternative to 2D edge function of a more traditional rasterization pipeline. 3D rasterization provides a simpler rasterization variable setup and requires less computation than when using 2D edge equations. In addition, it guarantees rasterization consistency and provides rules for it.

Triangle clipping isn't utilized by 3D rasterization, but OCLRaster still implements soft-clipping, since it requires certain triangle information in the binning stage and to some extent when deciding which triangles should be culled in the primitive processing & assembly stage.

For future reference, when rendering with orthographic projection using only 2D vertex coordinates and a fixed forward vector of $(0,0,1)^T$, the 3D rasterization variable setup can be simplified to:

$$
\begin{aligned}
V_{i,x} &= d_x \cdot (p_{((i+2) \bmod 3),y} - p_{((i+1) \bmod 3),y}) \\
V_{i,y} &= d_y \cdot (p_{((i+1) \bmod 3),x} - p_{((i+2) \bmod 3),x}) \\
V_{i,z} &= V_{i,y} \cdot p_{((i+2) \bmod 3),y} - V_{i,x} \cdot p_{((i+2) \bmod 3),x}
\end{aligned}
$$

V defines the volume, d_x and d_y are part of the image plane parameterization (right and up vector) and p specifies the triangles vertices.

3.3. Frustum Culling

OCLRaster also uses a highly optimized form of frustum culling [Gie10] which is implemented using the p/n-vertex approach. The code has been adapted for OpenCL and some minor modifications have been made. Since OCLRaster uses no far plane and the near plane is tested separately (see chapter 5.2 for an explanation), this is only used for the side planes of the view frustum. Furthermore, the frustum planes and the plane normals are transposed, so the loop can, in one step, test all respective x, y or z values of all planes at once.

```
const float3 aabb_min = fmin(fmin(vertices[0], vertices[1]), vertices[2]);
const float3 aabb_max = fmax(fmax(vertices[0], vertices[1]), vertices[2]);
// actual scale doesn't matter, but both must have the same scale (no * 0.5f req.):
const float3 aabb_center = (aabb_max + aabb_min);
const float3 aabb_extent = (aabb_max - aabb_min);
float4 fc_dot = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
for(unsigned int i = 0; i < 3; i++) {
    const float4 plane_normal = cdata->frustum_normals[i];
    const uint4 plane_sign = *(const uint4*)&plane_normal & (uint4)(0x80000000);
    const uint4 flipped_extent =
        (uint4)(((const uint*)&aabb_extent)[i]) ^ plane_sign;
    const float4 dot_param =
        (float4)(((const float*)&aabb_center)[i]) + *(const float4*)&flipped_extent;
    fc_dot += dot_param * plane_normal;
}
// if any dot product is less than 0 (aabb is completely outside any plane) -> cull
if(any(signbit(fc_dot))) {
    discard();
}
```

Listing 3: Frustum Culling as used in OCLRaster

4. Pipeline Overview

As any graphics pipeline, OCLRaster consists of three parts: the user programming interfaces and pipeline management code on the host side, as well as the actual graphics pipeline code on the device side. All of these will be explained in detail in the following chapters.

4.1. Device Stages

OCLRaster implements four device pipeline stages: The *Transform Stage*, where all vertex transformation happens and which is directly user-programmable. The *Primitive Processing & Primitive Assembly Stage*, which is responsible for assembling transformed vertices into primitives and processing these primitives by computing the required *3D Rasterization* variables and cull any invisible primitives. The *Binning Stage* computes which triangles are visible in which bins and creates triangle render queues for each bin. The *Rasterization Stage* finally rasterizes the triangles that have been determined to be visible by the *Binning Stage*. This stage, or the fragment shading to be precise, is also directly user-programmable.

4.2. User Frontend

The User Frontend section will show what kind of user programs are directly supported by the pipeline and how they can be programmed. Additionally, it will list all supported buffer types on the host and device side and also show how a user can program the depth-test and blending.

4.3. Host Interfaces

This chapter will provide an overview over all host interfaces, with a focus on the important ones that are required to run user programs and to modify the pipeline. This includes both programmable pipeline programs and the program base class, the pipeline main class and pipeline stage classes, the image and framebuffer classes, the opencl abstraction and a short overview over the core classes.

5. Pipeline Device Stages

The Rasterization Pipeline is divided up into four logical parts, with each part representing a separate pipeline stage and OpenCL kernel implementation. These are consecutively executed by the host when issuing a draw command.

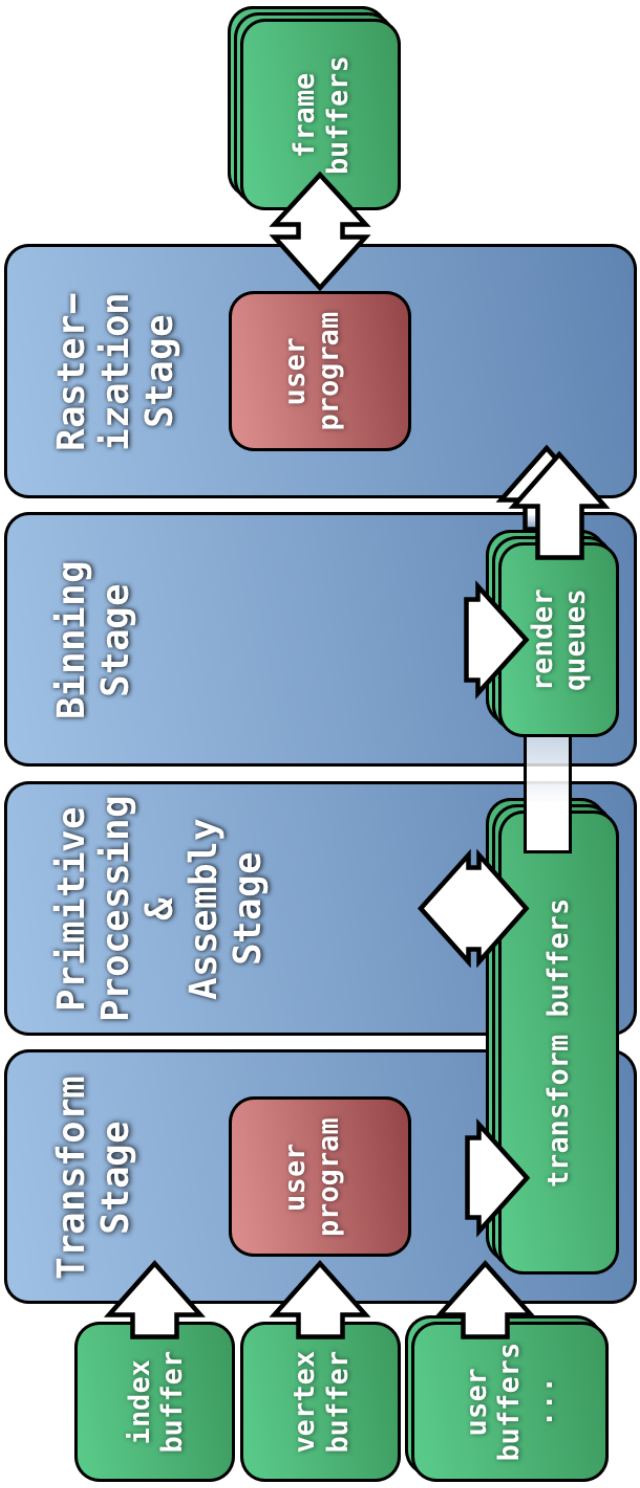

Figure 5: the OCLRaster pipeline

5.1. Transform Stage

The *transform stage* is responsible for transforming all vertices by calling a user-defined *transformation program* and storing all output data in a *transform buffer* (or multiple ones) that has been previously created by, and is automatically managed by the pipeline. This buffer is created according to the users output specification, meaning the pipeline has to know the size of each output element and how many there are (this will be further explained in chapter 6.2).

The actual implementation is relatively straightforward, each vertex is processed by exactly one work-item. This facilitates very good device utilization and parallelism, since every work-item can work independently and there are as many work-items as there are vertices. Vertex output is written in a 1:1 relationship, with each work-item knowing where it has to write its data and thus requiring no synchronization.

In case instancing is used, the workload is simply multiplied by the number of instances, e.g. N vertices and M instances will result in $(N * M)$ work-items. This implementation is rather simple and hardly gives any performance improvement in a software pipeline. The only benefit is that the user doesn't have to do M draw calls or duplicate the index buffer M times. As instancing doesn't matter for the rest of the pipeline (just more primitives to handle), the transformed data is simply written into the same transform buffer, although still segmented in instances. In any case, the rasterization stage is still able to compute the instance ID, both for the user program and for using the correct indices and transform data.

In the future this could be improved upon by handling the same vertex inside a work-group with the work-items processing the different instances, so to make use of any hardware caching (vertex data is the same for all instances). Of course this only makes sense if there are enough instances to ensure that all work-items have work to do.

Note that this stage is purely doing modelview transformations via the users program and any perspective or camera related transformation is done automatically by the pipeline, either already in this stage (substract the cameras position) or in the processing stage.

Discarded vertices are set to float infinity and will directly lead to discarded triangles in the processing stage as well.

5.2. Primitive Processing & Primitive Assembly Stage

This stage will assemble the previously transformed vertices into primitives (at the moment only triangles) and perform different kinds of processing on the primitive: frustum culling, culling in general, 3D rasterization setup and some soft-clipping.

Again, each work-item is able to work independently, as every work-item processes exactly one primitive and knows where to write its output without requiring any synchronization.

At first, the work-item reads the indices from the index buffer according to the employed primitive type (full triangles, triangles strips and triangles fans are currently possible), followed by reading the transformed vertex data from the transform buffer according to the read indices. If any of the primitives vertices has been discarded, the triangle is directly discarded at this point.

Next, the work-item will perform frustum culling on the primitive as shown in chapter 3.3. The near plane testing is conducted and stored separately, as this information is required later for use with the soft-clipping. As OCLRaster doesn't utilize a far plane, this test can consequently not be performed. A far plane in OCLRaster is not necessary, because all depth values are stored and handled as linear depth (in 32-bit floats) and no perspective transformation requiring a far plane definition is performed. All primitives that fail the frustum culling test are again directly discarded at this point.

Any primitive that has made it thus far has been established to be within the view frustum of the camera and is, to a high chance, probably visible. Subsequently, we can now do all of the complex computation without there being much waste.

In the next step, this stage will compute and set up the 3D rasterization variables, i.e. compute the volume variables V_0, V_1 and V_2 and the depth variable V_{depth} using the transformed primitive data and the camera setup that has been specified in the pipeline on the host side. Consider this the *2D edge equation setup* and *projective transformation* equivalent of a regular hardware graphics pipeline.

An additional step that is unfortunately also required in this stage, is the soft-clipping (the primitive doesn't actually get clipped, only the clip coordinates are computed) and 2D bounds computation of a primitive (screen space AABB). This information will be used in the binning stage to determine if a primitive covers a bin (is visible in a bin). 3D rasterization itself doesn't require this information per se, but there is no other way of directly knowing if a triangle is visible or not prior to the rasterization stage which will execute this test on a per-pixel basis, as it was intended. In this future, this could be improved by computing a per-bin frustum in the binning stage itself and perform the frustum culling on a per-bin basis.

Due to floating point imprecision and possible invalid input (degenerate triangles), it is also necessary to cull triangles with a surface area that is smaller than a certain epsilon value. This epsilon is currently set to $\frac{1}{256}$ (of a pixel). In the future, in particular when using anti-aliasing, this should ideally be set to a value that relates to the desired sub-pixel precision and it should additionally be tested if the primitive is on top of a sample point. Additional cross-checks are also performed to rule out or confirm if clipping completely failed due to imprecision issues.

When using orthographic rendering, this pipeline stage will use a different and much simplified path. For one thing, this uses the specialized 3D rasterization setup (as shown in chapter 3.2) and for another, the 3D frustum culling and soft-clipping code is unnecessary, since the actual 2D vertex positions can already be computed in this stage, thus simplifying the viewport test and the scissor test which can also already be performed here, and the direct ability to test if the primitive area is large enough (cull if degenerate).

If everything has been successful, the 3D rasterization variables are written to the internal transform buffer (10 float values per primitive) and the primitive bounds are written to another buffer (4 float values). These are stored in distinct buffers for logical reasons, as there are used by different pipeline stages (the former by the rasterization stage, the latter by the binning stage) and for performance reasons, because sequential buffer reads are faster than strided reads (especially in this case where neither 10 nor 14 is a power-of-two value, leading to even worse performance).

5.3. Binning Stage

The *binning stage* sits at the center between the previous two stages and the rasterizer, interacting with all of them as it determines which triangles are finally being rendered. The result of this stage are per-bin *render queues* that contain the primitive indices of all primitives that are visible in a bin. Due to its relative complexity and close hardware dependence, the implementation provides two paths: a GPU friendly version and a CPU friendly version. The GPU version employs manual work distribution and a manual primitive "cache" in local memory which is highly beneficial for GPUs and their high number of concurrently executing work-items, while the CPU version uses none of these and makes use of the much more simple automatic work distribution. Bins are currently set to be $32px * 32px$ in size, as this has shown to provide the best performance on different kinds of hardware (x86 CPUs as well as NVIDIA and AMD GPUs). Note that this is a simple global define that can easily be changed.

The manual work distribution of the GPU path is implemented using a permanent kernel and a fixed amount of work-items, determined by the *amount of compute units* times the *maximum allowed amount of work-items per work-group* (e.g. 8 compute units and a max of 1024 work-items will result in 8 work-groups with 1024 work-items each and 8192 work-items in total). To synchronize and distribute the work across work-groups, it utilizes a global memory counter, atomic functions (for global synchronization) and local memory barriers (for local synchronization). Each work-group processes one batch of primitives at a time, while each work-item processes that batch for exactly one bin. If there are more bins than there are work-items, it will simply iterate of the remaining bins, assigning work-items a new bin when they have fully processed their current one. In case there are less or all work done, work-items will simply wait until all work-items of a work-group have finished. Batch assignment is realized through a global memory counter that holds the next batch index (it is initialized with 0). The first work-item of a work-group will then do an atomic increment of this counter to receive the next batch index. This value is then shared with the other work-items in the group through local memory and a barrier instruction. This approach is similar to the one in [LK11]. Additionally, the GPU version will manually "cache" the primitive bounds of all the primitives in a batch in local memory, using an asynchronous work-group memory copy call from global memory to local memory (a built-in OpenCL function). This gives a 2x speed increase over reading this memory manually later on.

In the CPU path, using automatic work distribution, there are as many work-groups as there are bins and as many work-items per group as possible (although only as many work-items as primitive batches). This way, one work-item handles exactly one triangle batch for one bin at a time.

The primitive batch size is currently limited to a maximum of 256 primitives in an attempt to decrease the required memory of render queues by storing the primitive indices within a batch as 8-bit values (relative offsets from the absolute batch primitive index). This is necessary, because the render queues will be stored ("cached") in local memory in the rasterization stage, making them the only limiting factor on how many primitives can be rendered at once (without reading the next batch of render queues) and how much time is required to read the memory of all of these queues (4x less memory reads are very much noticeable). In the future, this could be reduced even further by using bitsets and only storing the visibility of a primitive in a batch in 1 bit, leading to 8x less memory reads and size. This would however necessitate more work to "pack" an "unpack" these bitsets and will definitely require some performance testing. In addition, this would also allow arbitrary batch sizes.

Now to the actual logic part of this stage, which is the same for both versions. Each work-item has a private memory queue where it stores the indices of visible primitives (all invisible ones will simply be ignored). The most inner loop sequentially iterates over all 256 primitives of a batch and each work-item will perform the visibility test on each primitive. The test itself is very simple, as it only checks if the previously computed primitive bounds intersect the bin bounds. If this is the case, the primitive is visible to the bin and the work-item stores the 8-bit loop index in its private memory queue.

After a batch has been processed, there are three different ways how the private memory queue of a work-item is stored as the render queue of a bin. If the queue is completely empty (no primitives of a batch are visible), it will simply store the 16-bit value `0xFFFF` at the start of the render queue to directly signal that this is an empty queue. If the queue is completely full, it will simply store all 256 indices in the render queue and if it's only partially full, it will write a `0x00` after the index of the last passing primitive to signal the end of the queue. In both cases, the render queue write happens via two `vstore16` calls (storing `ulong16` values), each copying 128 bytes from the private memory queue to the global memory render queue. As a small optimization, in case the queue contains less than 128 primitives, it will only perform one `vstore16` call. These calls to store big chunks of data all at once are necessary for efficiency reasons (copying every single byte manually is slow).

This stage will also perform a coarse scissor test, in that bins that are completely outside the scissor rectangle are not used in the first place (no work is ”scheduled” for them).

In the future, it might be beneficial to already perform some kind of depth test in this stage to directly cull primitives that would fail the depth test for all fragments in a bin, i.e. it would be better to do this here per-bin than doing this per-fragment in the rasterization stage.

5.4. Rasterization Stage

The *rasterization stage* is the last part of the pipeline which will finally (and sometimes hopefully) produce visible output. It uses the render queues computed in the binning stage to rasterize the primitives of each queue for each fragment of each bin, and read and writes fragments from and to the user specified framebuffer(s).

Similar to the binning stage, this stage also provides a GPU friendly and a CPU friendly implementation, with similar manual work distribution in the GPU version and automatic work distribution in the CPU version.

The work distribution in the GPU path is identical to the one in the binning stage, with the only logical difference being a change in terminology: work-groups process bins (instead of batches) and work-items process fragments (instead of bins). Again, if there are more fragments than there are work-items, work-items will simply process the remaining fragments in sequence, but still only one at a time. This stage uses the available local memory as manual cache as well. This time, it will cache the render queue of a bin in local memory (as this queue is shared among all work-items in a work-group), once more using an asynchronous work-group memory copy call, copying data from global to local memory. This however makes it the limiting factor on how many primitives can be rendered in one run, but it is still a lot faster than doing single memory reads for each element of a render queue. With local memory sizes currently ranging from 32 KiB to 48 KiB, this allows for 32768 to 49512 primitives to be rendered in one run (minus a small amount, because of local memory used elsewhere).

In the CPU path, using automatic work distribution, one work-group contains exactly one work-item that processes exactly one bin, i.e. it handles all render queues and all fragments of a bin. There are as many work-groups as there are compute units (cores). The restriction to only one work-item per group has shown to provide the best performance on a CPU (on all OpenCL CPU implementations!), with additional work-items reducing the performance significantly (up to a 100x loss when using >100 work-items). At this point, I'm not entirely sure why this happens, but I'd attribute it to the sheer complexity of the rasterization kernel and due to that compiler limitations.

From here on, the actual work being done is the same for both versions. This covers the per-fragment computations.

At first, the viewport and scissor test is performed on the computed fragment coordinate and the fragment is directly discarded if either one fails. After that, all framebuffer values for the specific fragment (coordinate) are read, including possible conversions to 32-bit float for 8-bit and 16-bit integer, unsigned integer and half-float image formats. If a depth buffer is attached, the depth value will be read as well.

Next, the work-item will iterate over all batches in the render queue (quickly skipping over the empty ones, as described in the previous stage). The most inner loop then iterates over the primitives in a batch and there, will do the actual primitive rasterization for the fragment.

Using the fragment coordinate and the 3D rasterization variables computed in the primitive processing stage, compute the barycentric coordinate for the primitive at the fragments location. If any component of the barycentric coordinate is greater or equal to 0 (i.e. the fragment is located outside the primitive), directly discard the primitive (or less than 0 in case orthographic rendering is used - this is due to necessarily flipped normals in the 3D rasterization setup code). After that, apply the consistency rules (top-left filling convention, as described in the 3D rasterization paper) and discard the fragment if it fails any of these.

The work-item will then continue to compute the depth value of the fragment, discard the fragment if it has a negative depth (which can happen for triangles that clip the near plane) and perform the early depth test if it hasn't been disabled by the user. This must be disabled if the user wants to write a custom depth value in the rasterization program and perform a depth test on that value instead (late depth test after calling the users program).

If the early depth test succeeds or is disabled, read the primitive specific transform buffer data, interpolate the data according to the computed barycentric coordinate and call the users rasterization program with all gathered input parameters, including the fragments framebuffer data to which the user will have direct access.

After the render queue and batches for the bin have been processed, this will write the fragment framebuffer values back to the users framebuffers, of course only if at least one primitive has been rendered (passed the depth test). This will again also handle any possible conversion, here, into the opposite direction (32-bit float values to the used 8-bit or 16-bit image format values).

5.5. Implementation

As listing the implementations code would easily go beyond the scope of this document and wouldn't provide much additional value at this point, I'd rather list the specific files locations, so anyone who is interested in the actual code can find it without the trouble of looking through all the source code.

Transform stage code is located at *lib/program/transform_program.cpp*, primitive processing and assembly stage code at *data/kernels/processing.cl*, binning stage code at *data/kernels/bin_rasterize.cl* and rasterization stage code at *lib/program/rasterization_program.cpp*.
See Appendix A on where to find the source code.

6. Pipeline User Frontend

6.1. OpenCL Built-in Functions

As previously mentioned in the OpenCL chapter, OpenCL C complies to ISO C99 for the most part and does provide numerous built-in functions. The similarity to GLSL should facilitate easy porting of GLSL programs to OCLRaster.

6.2. User Programs

The current OCLRaster implementation provides the possibility to directly program two pipeline stages: the transformation stage and the rasterization stage. These two program types are known as vertex shader and fragment (or pixel) shader in a hardware graphics pipeline. For consistency reasons these will simply be refered to as transformation program and rasterization program as their names correspond to the pipeline stages they are used in.

Both of those programs must provide a *main* function that will be called from within the respective pipeline kernel. Additionally, the user can provide *input* and *output* parameters that are or will be stored in buffers, *image* objects that can be read from or written to, *uniform* buffer objects (constant parameters), generic global memory *buffers* that can be randomly accessed, and only as part of the rasterization stage: a *framebuffer* that can contain multiple image objects.

This section will describe the types of buffers that are common among both program types. Note that all program parameters must be stored inside buffers, as OpenCL doesn't allow global variables except for constant memory variables that are known at compile time. From a performance viewpoint this also makes sense, since it's faster to set and upload combined buffers than it is to set each variable individually. The user can however define and use several buffers of one type. That way, it is for example still possible to use a separate buffer for vertices, texture coordinates and normals instead of storing them inside one single buffer. This is largely identical to what a hardware graphics pipeline allows.

type	program typename	description
input	`oclraster_in`	defines the per-vertex input
output	`oclraster_out`	defines the per-vertex transform output
uniform	`oclraster_uniforms`	defines constant parameters
buffer	`oclraster_buffers`	list of global memory buffers
image	`oclraster_images`	list of images
framebuffer	`oclraster_framebuffer`	list of attached framebuffer images (rasterization program only)

Table 1: available oclraster struct types

The syntax should be familiar to anyone who has ever written a standard C struct, with the latter three struct types only listing types without creating an actual object:

```
oclraster_in [struct_typename] {
    (variable_type variable_name;)*
} object_name;

oclraster_out struct_typename {
    (variable_type variable_name;)*
} object_name;

oclraster_uniforms struct_typename {
    (variable_type variable_name;)*
} object_name;

oclraster_buffers {
    (variable_type variable_name;)*
};

oclraster_images {
    ([access_type] image2d[<DATA_TYPE_HINT, CHANNEL_HINT>] image_name;)*
};

oclraster_framebuffer {
    (image2d | depth_image)[<DATA_TYPE_HINT, CHANNEL_HINT>] image_name;)*
};
```

Listing 4: syntax of oclraster structs

For an example user program where most of these struct types are used, have a look at the second part of chapter 8.1.

Some restrictions however apply:

- no interior/nested structs or unions
- no multi-variable declarations (e.g. `float x, y, z;`)
- no `__attribute__` (oclraster structs already use an `__attribute__` qualifier)
- use of any oclraster struct specifier in other places is disallowed (no typedefs, declarations, comments, ...)
- otherwise standard OpenCL C

These restrictions are necessary, since OCLRaster has to parse these structs without using or implementing a complex C parser itself (this would have exceeded the scope of this project). Previous limitations ([Zie13]) have however been lifted in the course of implementing OCLRaster by using an external C preprocessor project (Appendix B, [Bel13]). User-defined types, general preprocessor statements and include files are possible this way.

OCLRaster needs this information, because it has to automatically create correctly sized output buffers, correctly copy transform output variables and finally interpolate variables in the rasterization stage (all in case of *oclraster_out*), provide correct image or framebuffer objects/pointers (in case of *oclraster_images* and *oclraster_framebuffer*) and correctly provide buffers (in case of *oclraster_buffers*). Additionally, and because OCLRaster has to know this itself, the device specific memory layout of a struct must be retrieved and must also be made available to the user. The memory layout can be different for each device, but OpenCL unfortunately provides no method of accessing this information. To work around this limitation, OCLRaster generates and runs a simple OpenCL kernel from the parsed struct code in which it dumps the struct information (struct data member offsets and sizes, and the size of the struct itself), using standard C functionality, to a small integer buffer which is then read back on the host. As the memory layout can easily differ between the host and the device, especially when using small and odd-sized types, OCLRaster specifies an alignment of 16 bytes for each struct defintion which seems to work nicely in most uses cases.

The following sub-sections contain additional information that is specific to each program type.

6.2.1. Transformation Program

name	type	description
vertex_index	`unsigned int`	the index of the vertex inside the vertex buffer
instance_index	`unsigned int`	the index of the current instance
camera_position	`float3`	the absolute position of the camera

Table 2: available *built-in* main function parameters

6.2.2. Rasterization Program

name	type	description
primitive_index	`unsigned int`	the index of the primitive
instance_index	`unsigned int`	the index of the current instance
fragment_coord	`float2`	the fragments unnormalized screen space coordinate
fragment_depth	`float`	the fragments linear depth, computed by the pipeline
barycentric	`float3`	the barycentric coordinate at the fragments location
framebuffer	`oclraster_framebuffer*`	the user defined framebuffer (the entries for the specific fragment)

Table 3: available *built-in* main function parameters

6.3. Image Functions

Since OCLRaster provides an image abstraction on top of OpenCL to support non-native image formats and framebuffer functionality in general, this also requires functions to read and write image data. For natively supported image formats these functions directly forward to the corresponding OpenCL image functions. For software based images (buffer based images), OCLRaster implements all image read and write functions in software, including filtering and normalization support. The function prototypes match the built-in OpenCL image function prototypes, also allowing the same sampler specifications (see chapter 6.12.14 of the OpenCL specification [OWG12a] for both definitions).

```
/* return type */ image_read*(/* image type */ image,
                              const sampler_t sampler,
                              const float2 coordinate)
/* return type */ image_read*(/* image type */ image,
                              const sampler_t sampler,
                              const uint2 coordinate)

void image_write(/* image type */ image,
                 const uint2 coordinate,
                 const float4 color)
void image_write(/* image type */ image,
                 const uint2 coordinate,
                 const uint4 color)
void image_write(/* image type */ image,
                 const uint2 coordinate,
                 const int4 color)
```

Listing 5: available image functions

There are multiple image_read functions, depending on the read image data type. These are as follows: `image_read` (implicitly returns `float4` values), `image_read_int` (returns `int4` values), `image_read_uint` (returns `uint4` values), `image_read_long` (returns `long4` values) and `image_read_ulong` (returns `ulong4` values).

The following table contains the supported image data types and the corresponding enum type that can be used in image hints when writing `oclraster_images` and `oclraster_framebuffer` defintions.

image data type	**supported read & write types**	**enum name**
`uchar`	`float`, `uint`	`UINT_8`
`ushort`	`float`, `uint`	`UINT_16`
`uint`	`uint`	`UINT_32`
`ulong`	`ulong`	`UINT_64`
`char`	`float`, `int`	`INT_8`
`short`	`float`, `int`	`INT_16`
`int`	`int`	`INT_32`
`long`	`long`	`INT_64`
`half`	`float` (`half` is not directly supported)	`FLOAT_16`
`float`	`float`	`FLOAT_32`
`double`	`double`	`FLOAT_64`

Table 4: available image types

Note that the `double` type is only supported on OpenCL devices that have `double` support (the cl_khr_fp64 extension). Furthermore, OpenCL only has weak requirements on the `half` type, specifying that it only has to be supported as a data

storage type, not as a general variable data type. For this reason, OCLRaster will return read `half` values as `float` values and when `half` values should be written, `float` values must be specified. Also be aware that no OpenCL device (nor the OpenCL specification) support native 64-bit image formats, making `double`, `long` and `ulong` formats only available to software based images.

6.4. Framebuffer Functions

Framebuffers can contain almost any amount of normal image objects, but only one or no depth buffer for obvious logical reasons. As mentioned in chapter 6.2.2, each rasterization program has access to the specific fragment of the framebuffer that it is currently being computed. The framebuffer values are automatically read and possibly converted to the correct format at the beginning of the rasterization stage and again written to the framebuffer at the end of the rasterization stage, after all primitives in a bin queue have been processed.

As OpenCL doesn't allow read and writes to the same image object in a kernel (even though there is a `read_write` keyword, it is not supported), framebuffers in OCLRaster must use software based image objects (which use simple OpenCL buffers and can easily be synchronized).

Note that the amount of attached framebuffer image objects is only limited by the amount of kernel parameters that OpenCL allows. That amount has to be at least 1024 bytes (which translates to 256 4-byte parameters or pointers on a 32-bit device), while most implementations allow approximately 4KiB (about 1024 4-byte parameters). This should be more than enough for any sane use case.

The user also has full access to (and is actually responsible for) any fragment blending. The previous fragment values are fully accessible, and in fact, the user has to write to the same framebuffer variable that he is reading from when blending is to be employed. A built-in function called `linear_blend` that interpolates linearly between two values using a blend factor is also provided by OCLRaster.

6.5. Miscellaneous

Another advantage that OCLRaster provides over a hardware graphics pipeline is the ability to fully program the fragment depth test. The user can simply provide a custom `depth_test` function that takes the incomming fragment depth and the current fragment depth and returns true if the fragment should be discarded or not (i.e. the depth test fails or succeeds). See chapter 8.1 for an example of a custom

depth test function. All standard depth-test functions known from OpenGL are of course also available and can be set in the pipeline object.

The user should also be aware that, by default, the depth-test is executed prior to calling the rasterization program inside the rasterization stage (and in the future possibly also in the binning stage) to avoid unnecessary computation if a fragment is not visible anways (early Z-culling). This functionality can however be disabled, in which case the fragment depth *has* to be written by the rasterization program (the built-in `fragment_depth` value computed by the pipeline is still accessible regardless of which mode is used).

OCLRaster also includes scissor-testing functionality that is identical to the one in OpenGL and shouldn't require any more explanation. On the implementation side, the scissor test is performed in the binning stage (bins outside the scissor rectangle are completely ignored) and inside the rasterization stage (the fine, per-pixel scissor test).

7. Pipeline Host Interfaces

The main goals while creating the host interfaces of OCLRaster were to provide modern C++11 interfaces that are both easy to use and more object-oriented than what is provided with OpenGL. OCLRaster runs on all desktop operating systems that currently support OpenCL implementations, namely Linux, OS X and Windows. Due to the advanced use of C++11, OCLRaster currently only compiles with clang and libc++ [LLV13a, LLV13b] as all other compilers and STL implementations still lack important features (easy to set up on OS X and Linux, must use MinGW on Windows [Min13]). This should however be mitigated in the near future.

Cross-platform support is achieved through the usage of SDL [SDL13] and by writing platform independent code where possible, or writing platform specific code for all platforms in situations where it was necessary. This will also be noticable in all user programs, as these won't have to provide platform specific code.

As this is not the place nor the format to insert a Doxygen-like code documentation, this chapter will cover the host interfaces in a more general way, explaining the important classes, drawing similarites with OpenGL functionality, covering some design decisions and how one might extend and build upon the OCLRaster project. I would also highly encourage to read the respective header files, as these are more logically structured than any generated documentation could provide and possibly contain additional implementation details.

7.1. OpenCL

OCLRasters OpenCL abstraction is built on top of cl.hpp [OWG13], the C++ abstraction layer of OpenCL. OCLRaster provides an immense amount of additional functionality that make OpenCL easy to use, both in the OCLRaster implementation itself as well as in any user code that uses OpenCL directly. Among those features are cross-platform OpenCL context creation (including the ability to create shared OpenGL/OpenCL contexts and specifically restricting the context to certain platforms and device types), general OpenCL platform and device specific handling, simplified OpenCL kernel compilation (just specify a kernel file or code string and it'll handle the code compilation for all devices), simplified OpenCL buffer and image handling (note that these will have to be used when supplying the pipeline with user buffers), easy kernel execution, extended error handling, automatic command queue handling and many more.

Two features deserve particular mentioning and will be very helpful to developers. One being the ability to reload (and recompile) all kernels at runtime, of course requiring a small amount of handling code on the users side (OCLRaster emits an event for this that can be handled by the user program). The other one being automatic kernel binary dumping and consecutive binary disassembling (only on OS X), provided that the Xcode Tools [App13] (for CPU devices) and CUDA are installed [NVI07] (for NVIDIA GPU devices).

Note that OCLRaster also provides a CUDA implementation of its OpenCL base interface, wrapping many OpenCL functions in CUDA and supporting (simple) source transformation from OpenCL C to CUDA C. Currently, this is however defunct (not as advanced as required by OCLRasters pipeline), but with further development it might be of use in the future.

7.2. Pipeline

7.2.1. Pipeline & Stage Classes

The pipeline class is the main interface of OCLRaster. It is responsible for any rendering (draw calls) and stores and handles most of the rendering pipeline state (buffers, depth test function, camera setup, projection mode, ...). It also creates and handles objects of the four pipeline stages (transform_stage, processing_stage, binning_stage and rasterization_stage). These pipeline stage objects only handle their respective kernel parameter setup and execution, and possibly manage stage specific memory (e.g. render queues in the binning stage). The pipeline will call each of these stages in sequence when issueing a draw call.

Buffer and image objects are bound using unique identification names and are globally bound in regard to the pipeline object. This way, buffer and image objects don't have to be rebound across user program changes (if their name is unique). Internally this uses an `unordered_map` that provides constant lookup times, so its use should both be comfortable and fast.

Any kind of pipeline customization will have to happen in this pipeline class and any stage class that might need changing.

7.2.2. Image

The image class provides all basic 2D image functions that are also part of a hardware graphics pipeline implementation like OpenGL. Its usage is however more object-oriented than one might be used to with OpenGL. Functionality includes

read and write functions that can copy image data from and to the host as well as a copy function that can copy image data from another image object. Mapping of an image object to a host address is also possible.

As mentioned in earlier chapters, OCLRaster supports native OpenCL image objects and also implements software based image objects (referred to as buffer backing, as the image data is stored in a simple OpenCL buffer) for use in framebuffers and when image formats that are not natively supported should be employed. OCLRaster also supports the modification of the image backing at runtime. The user should still be aware of the natively supported image formats when using this functionality (these can be queried from the OpenCL class).

7.2.3. Framebuffer

Framebuffers are basically just containers for image objects that can be attached to them. As mentioned in chapter 6.4, there are no limits to the amount of attached images but the kernel parameter count restrictions imposed by the OpenCL implementation. For logical reasons, OCLRaster only allows no or one depth buffer attachment, which must furthermore have a single-channel 32-bit float image format (as a 32-bit float data type is used for all depth computations).

Additionally, this class provides framebuffer clear functions with clear colors or values that are specific to the framebuffer object (not a global state!).

7.3. Program

The oclraster_program class provides the base and common functionality of both the transform_program and the rasterization_program. This class is responsible for parsing the user program code, processing it and finally handing it off to the OpenCL class for compilation. Any inheriting program type, as demonstrated in transform_program and the rasterization_program, should only perform functions that are specific to them (e.g. program specific kernel template code and user code injection).

In addition, oclraster_program handles the different kernels necessitated by the different kernel specifications that the pipeline requires. Since this is a software pipeline, any change in used image formats, depth test function or projection mode requires different kernel code.

7.4. Core Interfaces

OCLRasters core interfaces include many different basic math primitives, among them 2-, 3- and 4-wide vector types and a 4x4 matrix type with typedefs for all native C++ data types. These should specifically be used when specifying oclraster_structs for use in user programs, but they might also be of use in general user code.

Additional classes are as follows: a simple logger class (supports logging to a log file and the console, similar to printf, but with parameter checking using variadic templates), a basic camera class implementing a first-person camera (controlled via mouse and keyboard), an event handling system for system events as well as OCLRaster internal events, basic file I/O and XML read and write capabilities, a basic model loader (for .a2m, pretty much a binary version of the .obj format) and threading facilities (thread implementation on top of std::thread and a simple threaded task implementation).

8. Examples

8.1. Simple Example Program

For the sake of simplicity, clarity and less page bloat, I removed the event handling, the iOS specific code and some other less significant code from this sample, added a few more meaningful comments in some places and inlined everything in one file and function. The full source code of this sample can be found in the ”samples/simple” folder (Appendix A).

Since this is still a rather long code example, most of the program functionality is explained in the comments or should be obvious judging by the function and variable names.

```
// include all necessary OCLRaster headers
#include <oclraster/oclraster.h>
#include <oclraster/pipeline/pipeline.h>
#include <oclraster/pipeline/transform_stage.h>
#include <oclraster/pipeline/image.h>
#include <oclraster/core/a2m.h>
#include <oclraster/core/camera.h>
#include <oclraster/program/oclraster_program.h>
#include <oclraster/program/transform_program.h>
#include <oclraster/program/rasterization_program.h>

#define APPLICATION_NAME "oclraster simple sample"

// global vars (these are acessed by multiple functions in the actual sample code)
static bool done { false };
static camera* cam { nullptr };
static constexpr float3 cam_speeds { 0.01f, 0.1f, 0.001f };
static atomic<unsigned int> update_model { false };
static atomic<unsigned int> update_light { true };
static atomic<unsigned int> update_light_color { true };
static transform_program* transform_prog { nullptr };
static rasterization_program* rasterization_prog { nullptr };
static pipeline* p { nullptr };
static atomic<unsigned int> selected_material { 0 };
static constexpr size_t material_count { 5 };

int main(int argc oclr_unused, char* argv[]) {
    // initialize oclraster (binary location + data folder)
    oclraster::init(argv[0], (const char*)"../data/");
    oclraster::set_caption(APPLICATION_NAME);
    oclraster::acquire_context();
```

```
// try to use the "fastest" GPU as the primary opencl device (this is simply
// determined by multiplying the device clock speed by the amount of compute units)
// also note: ocl is the opencl abstraction (object) that is provided by oclraster
ocl->set_active_device(opencl_base::DEVICE_TYPE::FASTEST_GPU);

// create and init a standard first-person camera
cam = new camera();
cam->set_position(0.8f, 0.28f, 3.2f);
cam->set_rotation(-5.2f, 196.0f, 0.0f);
cam->set_speed(cam_speeds.x);
cam->set_rotation_speed(cam->get_rotation_speed() * 1.5f);
cam->set_wasd_input(true);

// create the pipeline, set the active camera and
// notify oclraster that this is the active pipeline
p = new pipeline();
p->set_camera(cam);
oclraster::set_active_pipeline(p);

// load the model (blender monkey with uv coordinates)
a2m* model = new a2m(oclraster::data_path("monkey_uv.a2m"));

// add event handlers (... omitted ...)

// load, compile and bind user shaders (inlined for clarity)
// this actually resides in the "load_programs" function, so programs can
// be reloaded at runtime (when a kernel reload event is triggered)
{
    if(transform_prog != nullptr) {
        delete transform_prog;
        transform_prog = nullptr;
    }
    if(rasterization_prog != nullptr) {
        delete rasterization_prog;
        rasterization_prog = nullptr;
    }

    string code_str;
    if(!file_io::file_to_string(
            oclraster::kernel_path("user/simple_texturing.cl"), code_str)) {
        oclr_error("couldn't open program file!");
        return -1;
    }

    // create the program objects (with their resp. main function name)
    transform_prog = new transform_program(code_str, "transform_main");
```

```
    rasterization_prog = new rasterization_program(code_str, "rasterize_main");

    // there is no full program "linking" required; also, programs are
    // automatically bound to the correct stage according to their type
    p->bind_program(*transform_prog);
    p->bind_program(*rasterization_prog);
}

// create / reference buffers
const opencl::buffer_object& index_buffer = model->get_index_buffer(0);
const opencl::buffer_object& input_attributes = model->get_vertex_buffer();

oclraster_struct tp_uniforms {
    matrix4f modelview;
    matrix4f rotation_matrix;
} transform_uniforms {
    matrix4f(),
    matrix4f()
};
opencl::buffer_object* tp_uniforms_buffer =
    ocl->create_buffer(// read-only buffer from the kernels perspective
                       opencl::BUFFER_FLAG::READ |
                       // initialize the buffer from the given data pointer
                       opencl::BUFFER_FLAG::INITIAL_COPY |
                       // always block on buffer writes
                       opencl::BUFFER_FLAG::BLOCK_ON_WRITE,
                       // the size of this buffer
                       sizeof(tp_uniforms),
                       // the data pointer (for initialization)
                       (void*)&transform_uniforms);

// simple light setup, with a light rotating around the origin at
// a distance of 10 and an "intensity" (light influence radius) of 32
float light_pos = PI, light_dist = 10.0f, light_intensity = 32.0f;
oclraster_struct rp_uniforms {
    float4 camera_position;
    float4 light_position; // .w = light radius ^ 2
    float4 light_color;
} rasterize_uniforms {
    float4(cam->get_position(), 1.0f),
    float4(sinf(light_pos) * light_dist,
           0.0f,
           cosf(light_pos) * light_dist,
           light_intensity * light_intensity),
    float4(0.0f, 0.3f, 0.7f, 1.0f)
};
opencl::buffer_object* rp_uniforms_buffer =
```

```
    ocl->create_buffer(opencl::BUFFER_FLAG::READ |
                       opencl::BUFFER_FLAG::INITIAL_COPY |
                       opencl::BUFFER_FLAG::BLOCK_ON_WRITE,
                       sizeof(rp_uniforms),
                       (void*)&rasterize_uniforms);

// materials/textures (5 sets of 3: diffuse + normal + height)
static constexpr size_t textures_per_material = 3;
static const array<string, material_count*textures_per_material>
    texture_names {{
        "light_512", "light_normal_512", "light_height_512",
        "planks_512", "planks_normal_512", "planks_height_512",
        "rockwall_512", "rockwall_normal_512", "rockwall_height_512",
        "acid_512", "acid_normal_512", "acid_height_512",
        "blend_test_512", "light_normal_512", "scale_gray",
}};

array<array<shared_ptr<image>, textures_per_material>, material_count>
    materials;
for(size_t i = 0; i < material_count; i++) {
    for(size_t j = 0; j < textures_per_material; j++) {
        const string filename = texture_names[(i*textures_per_material) + j]
                                + ".png";
        materials[i][j] = make_shared<image>(
            // provided static image class function for loading .png files
            // via SDL2_image and libpng
            image::from_file(oclraster::data_path(filename),
                             // try to use native image backing
                             image::BACKING::IMAGE,
                             // 8-bit per channel (unsigned char/byte)
                             IMAGE_TYPE::UINT_8,
                             // RGBA / four channels
                             IMAGE_CHANNEL::RGBA));
    }
}

// create a simple 512*512px single-channel float noise (random) texture
float* fp_noise_data = new float[512*512];
for(size_t i = 0; i < (512*512); i++) {
    fp_noise_data[i] = core::rand(0.0f, 1.0f);
}
image* fp_noise = new image(512, 512,
                            image::BACKING::IMAGE,
                            IMAGE_TYPE::FLOAT_32,
                            IMAGE_CHANNEL::R,
                            fp_noise_data);
delete [] fp_noise_data;
```

```
// init done
oclraster::release_context();

// main loop
float model_rotation = 0.0f;
while(!done) {
    // set caption (app name, fps info, cam info)
    if(oclraster::is_new_fps_count()) {
        // fps and average frame time values are automatically provided by oclraster
        const unsigned int fps = oclraster::get_fps();
        stringstream caption;
        caption << APPLICATION_NAME;
        caption << " | " << fps << " FPS";
        caption << " | ~" << oclraster::get_frame_time() << "ms ";
        caption << " | Cam: " << cam->get_position();
        caption << " " << cam->get_rotation();
        oclraster::set_caption(caption.str());
    }

    // signal oclraster that we're starting a new frame
    oclraster::start_draw();

    // run the camera and update the pipeline camera setup
    cam->run();
    p->set_camera(cam);

    // to also demonstrate how a custom depth-test function can be specified:
    // this is a combination of the "LESS" depth-test function and a function that
    // will pass and fail depth values in alternating steps of 0.05
    p->set_depth_function(DEPTH_FUNCTION::CUSTOM,
                          "bool depth_test(float incoming, float current) {" \
                          "    const float depth = fmod(incoming, 0.1f);" \
                          "    return (depth > 0.05f && incoming < current);" \
                          "}");

    // update uniforms
    if(update_model) {
        transform_uniforms.modelview = matrix4f().rotate_y(model_rotation);
        transform_uniforms.rotation_matrix = transform_uniforms.modelview;
        // (... remaining transform updates omitted ...)
        // update the transform uniform buffer
        ocl->write_buffer(tp_uniforms_buffer, &transform_uniforms);
    }

    if(update_light) {
        light_pos -= 0.125f;
```

```
            rasterize_uniforms.light_position.set(sinf(light_pos) * light_dist,
                                                  0.0f,
                                                  cosf(light_pos) * light_dist,
                                                  light_intensity * light_intensity);
        }

        // update the rasterization uniform buffer
        const auto& camera_position = cam->get_position();
        rasterize_uniforms.camera_position.set(camera_position.x, camera_position.y
                                               camera_position.z, 1.0f);
        ocl->write_buffer(rp_uniforms_buffer, &rasterize_uniforms);

        // finally: bind all buffers and images and draw the model
        p->bind_buffer("index_buffer", index_buffer); // "index_buffer" is a built-in name
        p->bind_buffer("input_attributes", input_attributes);
        p->bind_buffer("tp_uniforms", *tp_uniforms_buffer);
        p->bind_buffer("rp_uniforms", *rp_uniforms_buffer);
        p->bind_image("diffuse_texture", materials[selected_material][0]);
        p->bind_image("normal_texture", materials[selected_material][1]);
        p->bind_image("height_texture", materials[selected_material][2]);
        p->bind_image("fp_noise", *fp_noise);
        p->draw(PRIMITIVE_TYPE::TRIANGLE,
                model->get_vertex_count(),
                { 0, model->get_index_count(0) });

        // done with this frame -> stop/swap
        oclraster::stop_draw();
    }

    // cleanup (... partially omitted ...)
    delete p;
    oclraster::destroy();

    return 0;
}
```

Listing 6: simple sample host code

This code example has hopefully shown that writing code that uses OCLRaster is relatively straightforward and should in most cases be easier than writing OpenGL code.

The following code listing contains the transformation & rasterization program that is used in this example (data/kernels/user/simple_texturing.cl):

```
// this is the same for both programs
// note that these are automatically interpolated in the rasterization program
oclraster_out simple_output {
    float4 vertex;
    float4 normal;
    float2 tex_coord;
} output_attributes;

#if defined(OCLRASTER_TRANSFORM_PROGRAM)
///////////////////////////////////////////////////////////////////
// transform program
oclraster_in simple_input {
    float4 vertex;
    float4 normal;
    float4 binormal; // unused
    float4 tangent; // unused
    float2 tex_coord;
} input_attributes;

oclraster_uniforms transform_uniforms {
    mat4 modelview_matrix;
    mat4 rotation_matrix;
} tp_uniforms;

float4 transform_main() {
    // transform the input vertex - note that this is only the modelview transformation,
    // a projection transformation is not needed
    const float4 mv_vertex = mat4_mul_vec4(tp_uniforms->modelview_matrix,
                                           input_attributes->vertex);
    output_attributes->vertex = mv_vertex;

    // the normal only needs to be rotated
    output_attributes->normal = mat4_mul_vec4(tp_uniforms->rotation_matrix,
                                              input_attributes->normal);

    // simply pass-through the texture coordinate
    output_attributes->tex_coord = input_attributes->tex_coord;

    // return the transformed vertex
    return mv_vertex;
}

#elif defined(OCLRASTER_RASTERIZATION_PROGRAM)
///////////////////////////////////////////////////////////////////
// rasterization program
```

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

oclraster_images {
    // hints are optional (will default to <UINT_8, RGBA>), but useful for specifying the
    // the image format of the first compiled kernel and possibly the intention of an image
    read_only image2d<UINT_8, RGBA> diffuse_texture;
    read_only image2d<FLOAT_32, R> fp_noise;
};
oclraster_framebuffer {
    image2d color;
    depth_image depth;
};

bool rasterize_main() {
    // we need two samplers for this:
    const sampler_t linear_sampler = (// normalized texture coordinates
                                      CLK_NORMALIZED_COORDS_TRUE |
                                      // repeat/wrap/modulo mode
                                      // in this case: wrap around 1.0
                                      CLK_ADDRESS_REPEAT |
                                      // linear sampling
                                      CLK_FILTER_LINEAR);
    const sampler_t point_sampler = (CLK_NORMALIZED_COORDS_TRUE |
                                     CLK_ADDRESS_REPEAT |
                                     // nearest/point sampling
                                     CLK_FILTER_NEAREST);

    // simple noise (depending on the texture coordinate and fragment_coord)
    // note that image_read always returns a float4 (akin to the built-in OpenCL functions)
    const float noise_offset = image_read(fp_noise,
                                          point_sampler,
                                          output_attributes->tex_coord).x;
    const float noise = image_read(fp_noise,
                                   linear_sampler,
                                   (fragment_coord / 100.0f) + noise_offset).x;

    // check if lit by light (compute attenuation)
    float3 light_dir = rp_uniforms->light_position.xyz - output_attributes->vertex.xyz;
    light_dir /= rp_uniforms->light_position.w; // / (light radius)^2
    float attenuation = 1.0f - dot(light_dir, light_dir) * rp_uniforms->light_position.w;
    if(attenuation > 0.0f) {
        light_dir = normalize(light_dir);
```

```
        // simple phong lighting
        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;
        }

        // combined light color
        float4 color = (float4)(diff_color + spec_color, 1.0f);

        // multiply material diffuse color
        color.xyz *= image_read(diffuse_texture,
                                linear_sampler,
                                output_attributes->tex_coord).xyz;

        // multiply noise
        color.xyz *= 0.5f + (noise * 0.5f);

        // update (overwrite) the framebuffer color
        framebuffer->color = color;
        // depth is written automatically, unless specified otherwise in the pipeline
    }

    // "the fragment passed"
    return true;
}
#endif
```

Listing 7: sample transformation & rasterization program

When compiling an OCLRaster program, a *transform program* will predefine `OCLRASTER_TRANSFORM_PROGRAM` while a *rasterization program* predefines `OCLRASTER_RASTERIZATION_PROGRAM`. This way, both programs can reside in the same file (or code text string) and can share common code, data structures or similar.

8.2. Other

The OCLRaster code repository (Appendix A) contains additional sample code for *render-to-texture functionality*, *rendering of other primitive types*, *volume rendering* and *GUI/2D rendering*. I will not go into these here, because they are not much different from the previous sample and I think they are rather easy to understand. However, for completeness sake, I had to mention that these samples do exist and might be interesting for anyone who wants to delve deeper into OCLRaster.

Nevertheless, actual screenshots are always interesting:

Figure 6: the result of the example program listed in 8.1, showing the custom depth test function (note the striped rendering) and simple lighting

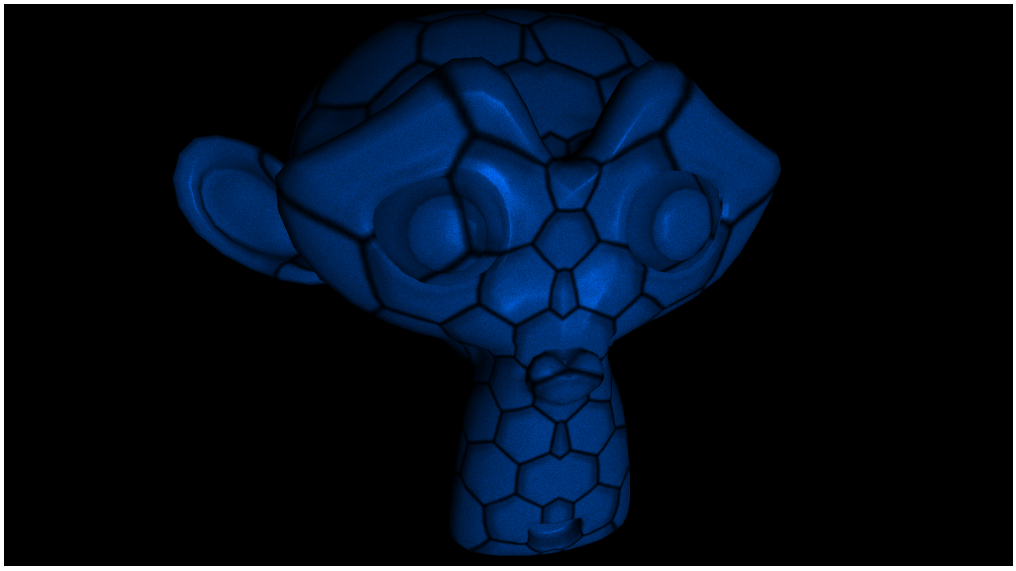

Figure 7: the same example program using the built-in ”LESS” depth test function

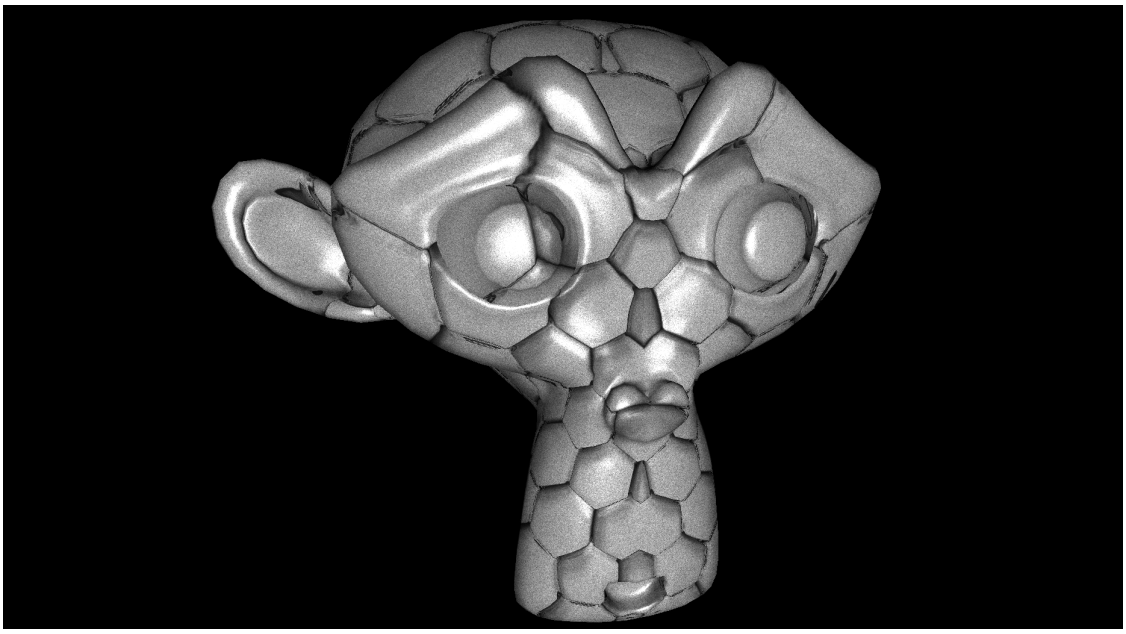

Figure 8: a more complex shading program showing parallax mapping

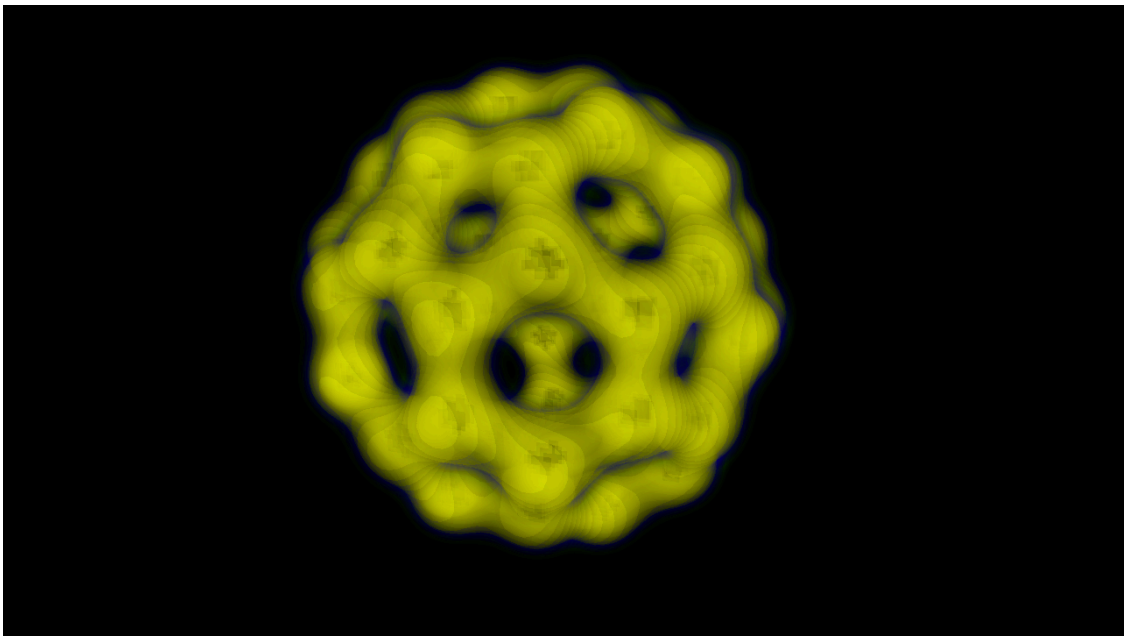

Figure 9: the volume example program, rendering the C60 volume using sliced volume rendering and also showing custom blending

Monkey model ”Suzanne” courtesy of Blender [Ble13], Buckminster Fuller carbon 60 volume data set courtesy of ImageVis3D [Ima11].

9. Conclusion & Future Work

This thesis has hopefully demonstrated that a software graphics pipeline is quite feasible on today's hardware. Of course some performance sacrifices have to be made, but with additional pipeline optimizations mentioned in earlier chapters and this chapter, this project might actually provide a genuine alternative to a hardware graphics pipeline.

There are many possible future extensions to the pipeline that could be included here, but I will concentrate on the most significant and compelling ones.

An important one is to increase the asynchronism by providing non-blocking draw calls. This would pretty much necessitate a client-server model instead of the current direct, but blocking, approach. In addition, this would allow the pipeline to combine multiple draw calls to a larger one, thus eliminating even more overhead. This would also require more asynchronism on the user buffer handling side by making all buffer read and write operations non-blocking, too.

Another possibly easy to fix performance problem is the displaying of the final framebuffer (or "framebuffer swap"). It is currently necessary to completely copy the framebuffer from an OpenCL buffer to an OpenGL texture in order to draw or blit it into the window frame, basically requiring two copies instead of one. One solution to this might be to move this OpenCL to OpenGL buffer copy to a separate thread and OpenGL context altogether. This would however need some kind of software double buffering and cause other issues like increased memory requirements and, depending on the OpenGL implementation, a greater frame latency. A better solution would have to be provided by a future OpenCL revision or extension that would allow OpenCL kernels to read and write to the same image object in the same kernel, thus eliminating the need for software image buffers and yielding the ability to use native images instead. This might increase general framebuffer performance as well, at least on devices with native image support.

On the feature side, OCLRaster is currently lacking two important ones, namely anti-aliasing and mip-mapping support, including general LOD texture lookup support and anisotropic filtering. Anti-aliasing should be rather trivial to implement, as it's just more fragments that need to be computed and the fragment coordinate (or sample location) can easily be changed. This should also enable interesting new anti-aliasing methods, since hardware limitations do not apply and all sample locations are purely software based, as is the number of samples and the framebuffer format. Mip-mapping is not as trivial, because OpenCL doesn't support mip-mapping itself or any other kind of texture chain with LOD lookup support, meaning this would have to be completely implemented in software. This is even

further impeded by how OpenCL handles native image objects, requiring that every texture level would have to be supplied as a separate kernel parameter.

Reading back the transformed vertex and primitive data is already known from OpenGL (transform feedback [Khr13a]), but with OCLRaster one could go a step further and additionally read back and cache the per-bin render queues from the binning stage, thus giving the ability to dump the complete pipeline state prior to the rasterization stage. This can be of benefit for certain multi-pass rendering methods that have to render the same geometry multiple times [KL09] or as an optimization for shading and overdraw heavy scenes, by simply filling the depth buffer in a first pass and then performing the actual fragment shading in a second pass.

Facilitated through the usage of *3D Rasterization* [DEG+12] and the fact that OCLRaster is a full software graphics pipeline, the combination of rasterization and ray tracing in a graphics pipeline is another interesting possibility.

As a result of OpenCLs design and due to its wide acceptance, as demonstrated by the number of available OpenCL implementations that exist today and which will probably soon be followed by additional ones on mobile devices, OCLRaster is capable of running on a lot of different hardware. This gives rise to two interesting applications. For one thing, this allows for a heterogenous graphics pipeline where computation is performed by the device that is most suited for a task (this is especially appealing on CPUs with integrated GPUs, with both sharing the same memory space). For another, multi-device rendering with very fine grained control is also possible. This can go from simply dividing the screen into multiple parts, each part being computed by a different device, deep down to splitting the vertex and primitive processing workload into several parts.

OpenCL also directly provides profiling functions that could be used in the future to measure the performance of new rendering methods, or just simply shader code in general, in a more direct manner than OpenGL allows, as every pipeline stage can be measured and dissected separately.

The prospect of complete pipeline programmability, both on the device and the host side, and the ability to run the same software on a multitude of different devices is very enticing and leads to many new possibilities.

A. OCLRaster on GitHub

For the full source code and future development, please have a look at the GitHub project page at `https://github.com/a2flo/oclraster`. All source code is released under the GPLv2 licence.

B. TCCPP

As a small side project to the OCLRaster project, mostly out of necessity to allow more complex user programs (chapter 6.2), I forked the TCC project (the Tiny C Compiler [Bel13]) and stripped it down to only its preprocessing parts. Additionally, some modifications were made that allow TCC to completely work in-memory (so no temporary files have to be created on disk or have to necessarily be read from disk - include files still work) and also made all TCC functions depend on a state object rather than depending on and using static global variables. This makes it possible to have multiple instances of TCC (state objects) within a process and thus "multi-threaded" program preprocessing. Preprocessing in itself is of course still single-threaded, but several programs can be preprocessed in different threads at the same time. Sadly, at the time of this writing, no OpenCL implementation is actually capable of building OpenCL programs from multiple threads at the same time, even though it's only a context related function and not command queue or device related. This makes OpenCL program compilation still a blocking operation.

Note that TCC is directly built along OCLRaster and resides in the same library folder. Building a standalone executable binary is still possible, although rather pointless.

The projects source code can be found at `https://github.com/a2flo/tccpp`.

C. OCLRaster Support Library

Also part of this project and meant to showcase and test more advanced rendering code, is an additional support library. This library implements a simple GUI system and provides 2D rendering capabilites that include the rendering of relatively complex primitives like rounded rectangles and ellipsoids with different kinds of shading (gradients, textures) as well as font rendering via FreeType [Fre13].

This is included in the OCLRaster GitHub repository at `https://github.com/a2flo/oclraster` and located in the *support* folder, with an UI example program in the *samples/ui* folder.

References

[AMD06] AMD. AMD Close to Metal. http://www.amd.com/us/press-releases/Pages/Press_Release_114147.aspx, November 2006.

[AMD12] AMD. AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE. http://www.amd.com/la/Documents/GCN_Architecture_whitepaper.pdf, June 2012.

[App13] Apple. Xcode Developer Tools. https://developer.apple.com/technologies/tools, 2013.

[ARM13] ARM. MaliTM-T600 Series GPU OpenCL. http://infocenter.arm.com/help/topic/com.arm.doc.dui0538e/DUI0538E_mali_t600_opencl_dg.pdf, 2013.

[Bel13] Fabrice Bellard. Tiny C Compiler. http://bellard.org/tcc, February 2013.

[BFH+04a] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[BFH+04b] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. BrookGPU. http://graphics.stanford.edu/projects/brookgpu, 2004.

[Ble13] Blender. Blender project - Free and Open 3D creation software. http://www.blender.org, 2013.

[DEG+12] Tomáš Davidovič, Thomas Engelhardt, Iliyan Georgiev, Philipp Slusallek, and Carsten Dachsbacher. 3D Rasterization: A Bridge between Rasterization and Ray Casting. In *Proceedings of Graphics Interface 2012*, GI '12, pages 201–208, Toronto, Ont., Canada, Canada, 2012. Canadian Information Processing Society. http://cg.ibds.kit.edu/publications/p2012/3dr/gi2012.pdf.

[Fre13] FreeType. The FreeType project. http://www.freetype.org, 2013.

[Gie10] Fabian Giesen. View frustum culling. http://fgiesen.wordpress.com/2010/10/17/view-frustum-culling/, October 2010.

[Ima11] ImageVis3D. ImageVis3D data sets. http://www.sci.utah.edu/download/IV3DData.html, 2011.

[Ima13] Imagination Technologies. PowerVR Series6 IP Core. http://www.imgtec.com/powervr/sgx_series6.asp, 2013.

[Int13] Intel. Software Occlusion Culling. http://download-software.intel.com/sites/default/files/softwareocclusionculling.pdf & http://software.intel.com/en-us/articles/software-occlusion-culling, January 2013.

[Khr08a] Khronos Group. Khronos Launches Heterogeneous Computing Initiative. https://www.khronos.org/news/press/khronos_launches_heterogeneous_computing_initiative, June 2008.

[Khr08b] Khronos Group. The Khronos Group Releases OpenCL 1.0 Specification. https://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification, December 2008.

[Khr08c] Khronos Group, Neil Trevett. OpenCL The Open Standard for Heterogeneous Parallel Programming. https://www.khronos.org/assets/uploads/developers/library/2008_siggraph_asia/OpenCL%20Overview%20SIGGRAPH%20Asia%20Dec08.pdf, December 2008.

[Khr10] Khronos Group. Khronos Drives Momentum of Parallel Computing Standard with Release of OpenCL 1.1 Specification. https://www.khronos.org/news/press/khronos-group-releases-opencl-1-1-parallel-computing-standard, June 2010.

[Khr11] Khronos Group. Khronos Releases OpenCL 1.2 Specification. https://www.khronos.org/news/press/khronos-releases-opencl-1.2-specification, November 2011.

[Khr12] Khronos Group. Khronos Releases Significant OpenCL 1.2 Specification Update. https://www.khronos.org/news/press/khronos-releases-significant-opencl-1.2-specification-update, November 2012.

[Khr13a] Khronos. The OpenGL Graphics System: A Specification, Version 4.3 (Core Profile). http://www.opengl.org/registry/doc/glspec43.core.20130214.pdf, February 2013.

[Khr13b] Khronos Group. Conformant Products. http://www.khronos.org/conformance/adopters/conformant-products, 2013.

[KL09] Scott Kircher and Alan Lawrance. Inferred lighting: Fast dynamic lighting and shadows for opaque and translucent objects. In *Proceedings of*

the 2009 ACM SIGGRAPH Symposium on Video Games, Sandbox '09, pages 39–45, New York, NY, USA, 2009. ACM.

[LK11] Samuli Laine and Tero Karras. High-Performance Software Rasterization on GPUs. In *Proceedings of High-Performance Graphics 2011*, 2011. https://mediatech.aalto.fi/~samuli/publications/laine2011hpg_paper.pdf.

[LLV13a] LLVM. clang: a C language family frontend for LLVM. http://clang.llvm.org, 2013.

[LLV13b] LLVM. "libc++" C++ Standard Library. http://libcxx.llvm.org, 2013.

[Min13] MinGW. "MinGW-w64. http://mingw-w64.sourceforge.net, 2013.

[NVI07] NVIDIA. CUDA. https://developer.nvidia.com/cuda-toolkit-archive & https://developer.nvidia.com/cuda-faq, 2007.

[NVI12] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[OWG12a] Khronos OpenCL Working Group. The OpenCL Specification, Version: 1.2, Document Revision: 19. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, November 2012.

[OWG12b] Khronos OpenCL Working Group, SPIR subgroup. SPIR 1.0 Specification for OpenCL. http://www.khronos.org/registry/cl/specs/spir_spec-1.0-provisional.pdf, August 2012.

[OWG13] Khronos OpenCL Working Group. The OpenCL C++ Wrapper API, Version: 1.2, Document Revision: 09. http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.2.pdf&http://www.khronos.org/registry/cl/api/1.2/cl.hpp, May 2013.

[SDL13] SDL. Simple DirectMedia Layer, version 2.0. http://www.libsdl.org, 2013.

[Zie13] Florian Ziesche. Flexible Rasterizer in OpenCL, Bachelor Seminar presentation. https://github.com/a2flo/oclraster/blob/master/etc/oclraster_presentation.pdf, February 2013.