A tour around the world of Mesa and Linux graphics drivers

JULY 18, 2014

For some time now I have decided to focus my work at Igalia on the graphics stack. As a result of this I had the chance to participate in a couple of very interesting projects like implementing Wayland support in WebKitGtk+ (a topic I have visited in this blog a number of times) and, lately, work on graphics drivers for Linux in the Mesa framework.

The graphics stack in Linux is complex and it is not always easy to find information and technical documentation that can aid beginners in their firsts steps. This is usually a very demanding domain, the brave individuals who decide to put their energy into it usually have their hands full hacking on the code and they don't have that much room for documenting what they do in a way that is particularly accessible to newcomers.

As I mentioned above, I have been hacking on Mesa lately (particularly on the Intel i965 driver) and so far it as been a lot of fun, probably the most exciting work I have done at Igalia in all these years, but it is also certainly challenging, requiring me to learn a lot of new things and some times fairly complex stuff.

Getting involved in this is no easy endeavor, the learning curve is steep because the kind of work you do here is probably unlike anything you

have done before: for starters it requires a decent understanding of OpenGL and capacity to understand OpenGL specifications and what they mean in the context of the driver, you also need to have a general understanding of how modern 3D-capable GPUs work and finally, you have to dig deeper and understand how the specific GPU that your driver targets works and what is the role that the driver needs to play to make that hardware work as intended. And that's not all of it, a driver may need to support multiple generations of GPUs which sometimes can be significantly different from each other, requiring driver developers to write and merge multiple code paths that handle these differences. You can imagine the maintenance burden and extra complexity that comes from this.

Finally, we should also consider the fact that graphics drivers are among the most critical pieces of code you can probably have in a system, they need to be performant and stable for all supported hardware generations, which adds to the overall complexity.

All this stuff can be a bit overwhelming in the beginning for those who attempt to give their first steps in this world but I believe that this initial steep learning curve can be smoothed out by introducing some of the most important concepts in a way that is oriented specifically to new developers. The rest will still not be an easy task, it requires hard work, some passion, be willing to learn and a lot of attention to detail, but I think anyone passionate enough should be able to get into it with enough dedication.

I had to go through all this process myself lately, so I figured I am in a very good situation to try and address this problem myself, so that's why I decided to write a series of posts to introduce people to the world of Mesa and 3D graphics drivers, with a focus on OpenGL and Intel GPUs, which

is the area were I am currently developing my work. Although I'll focus on Intel hardware I believe that many of the concepts that I will be introducing here are general enough so that they are useful also to people interested in other GPUs. I'll try to be clear about when I am introducing general concepts and when I am discussing Intel specific stuff.

My next post, which will be the first in this series, will serve as an introduction to the Linux graphics stack and Linux graphics drivers. We will discuss what Mesa brings to the table exactly and what we mean when we talk about graphics drivers in Linux exactly. I think that should put us on the right track to start looking into the internals of Mesa.

So that's it, if you are interested in learning more about Linux graphics and specifically Mesa and 3D graphics drivers, stay tuned! I'll try my best to post regularly and often.

A brief introduction to the Linux graphics stack

JULY 29, 2014

This post attempts to be a brief and simple introduction to the Linux graphics stack, and as such, it has an introductory nature. I will focus on giving enough context to understand the role that Mesa and 3D drivers in general play in the stack and leave it to follow up posts to dive deeper into the guts of Mesa in general and the Intel DRI driver specifically.

A bit of history

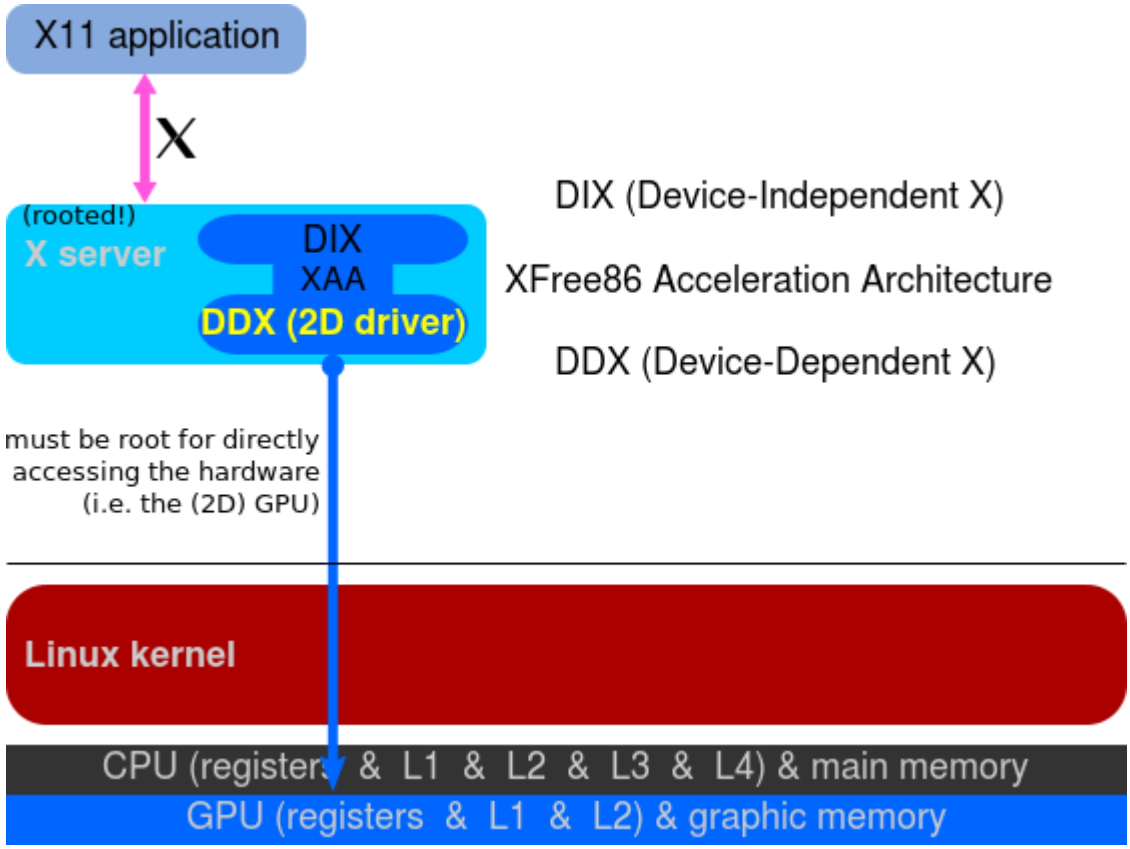
In order to understand some of the particularities of the current graphics stack it is important to understand how it had to adapt to new challenges throughout the years.

You see, nowadays things are significantly more complex than they used to be, but in the early times there was only a single piece of software that had direct access to the graphics hardware: the X server. This approach made the graphics stack simpler because it didn't need to synchronize access to the graphics hardware between multiple clients.

In these early days applications would do all their drawing indirectly, through the X server. By using Xlib they would send rendering commands over the X11 protocol that the X server would receive, process and translate to actual hardware commands on the other side of a socket.

Notice that this “translation” is the job of a driver: it takes a bunch of hardware agnostic rendering commands as its input and translates them into hardware commands as expected by the targeted GPU.

Since the X server was the only piece of software that could talk to the graphics hardware by design, these drivers were written specifically for it, became modules of the X server itself and an integral part of its architecture. These userspace drivers are called DDX drivers in X server argot and their role in the graphics stack is to support 2D operations as exported by Xlib and required by the X server implementation.



DDX drivers in the X server (image via wikipedia)

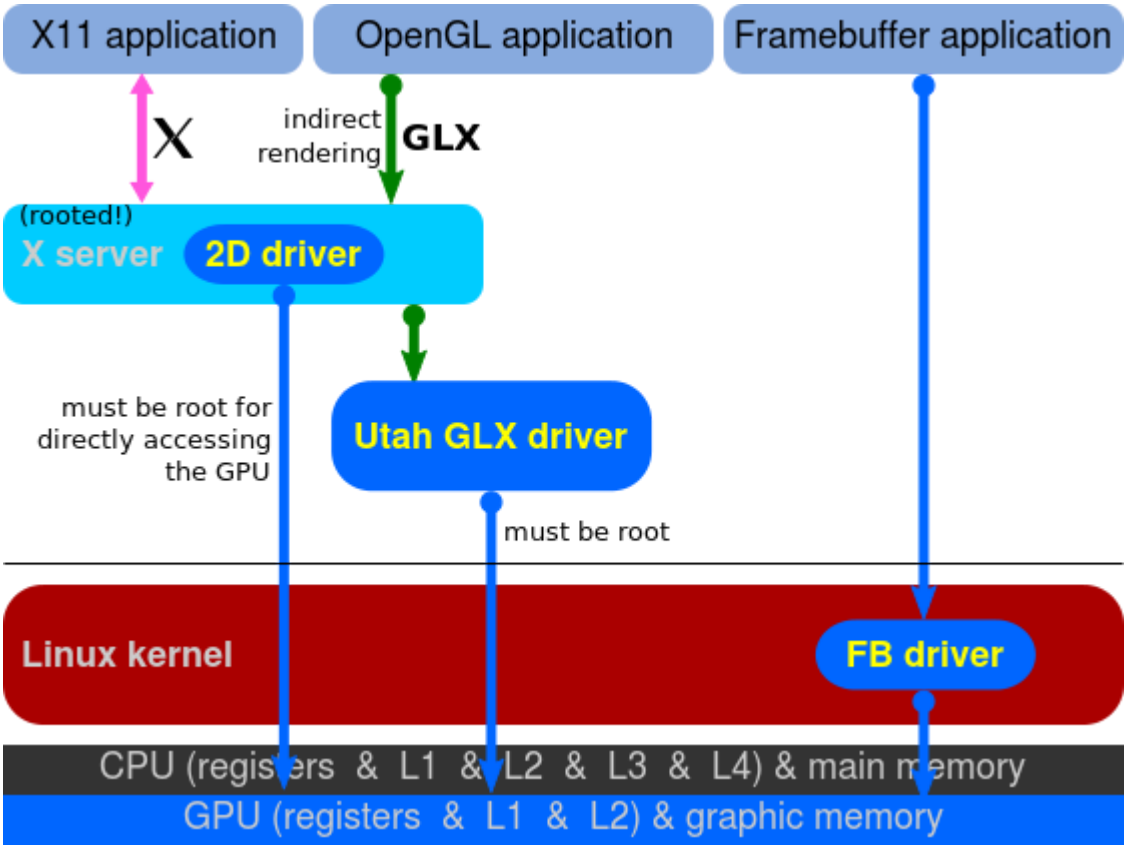
In my Ubuntu system, for example, the DDX driver for my Intel GPU comes via the xserver-xorg-video-intel package and there are similar packages for other GPU vendors.

3D graphics

The above covers 2D graphics as that is what the X server used to be all about. However, the arrival of 3D graphics hardware changed the scenario significantly, as we will see now.

In Linux, 3D graphics is implemented via OpenGL, so people expected an implementation of this standard that would take advantage of the fancy new 3D hardware, that is, a hardware accelerated *libGL.so*. However, in a system where only the X server was allowed to access the graphics hardware we could not have a *libGL.so* that talked directly to the 3D hardware. Instead, the solution was to provide an implementation of OpenGL that would send OpenGL commands to the X server through an extension of the X11 protocol and let the X server translate these into actual hardware commands as it had been doing for 2D commands before.

We call this *Indirect Rendering*, since applications do not send rendering commands directly to the graphics hardware, and instead, render indirectly through the X server.



OpenGL with Indirect Rendering (image via wikipedia)

Unfortunately, developers would soon realize that this solution was not sufficient for intensive 3D applications, such as games, that required to render large amounts of 3D primitives while maintaining high frame rates. The problem was clear: wrapping OpenGL calls in the X11 protocol was not a valid solution.

In order to achieve good performance in 3D applications we needed these to access the hardware directly and that would require to rethink a large chunk of the graphics stack.

Enter Direct Rendering Infrastructure (DRI)

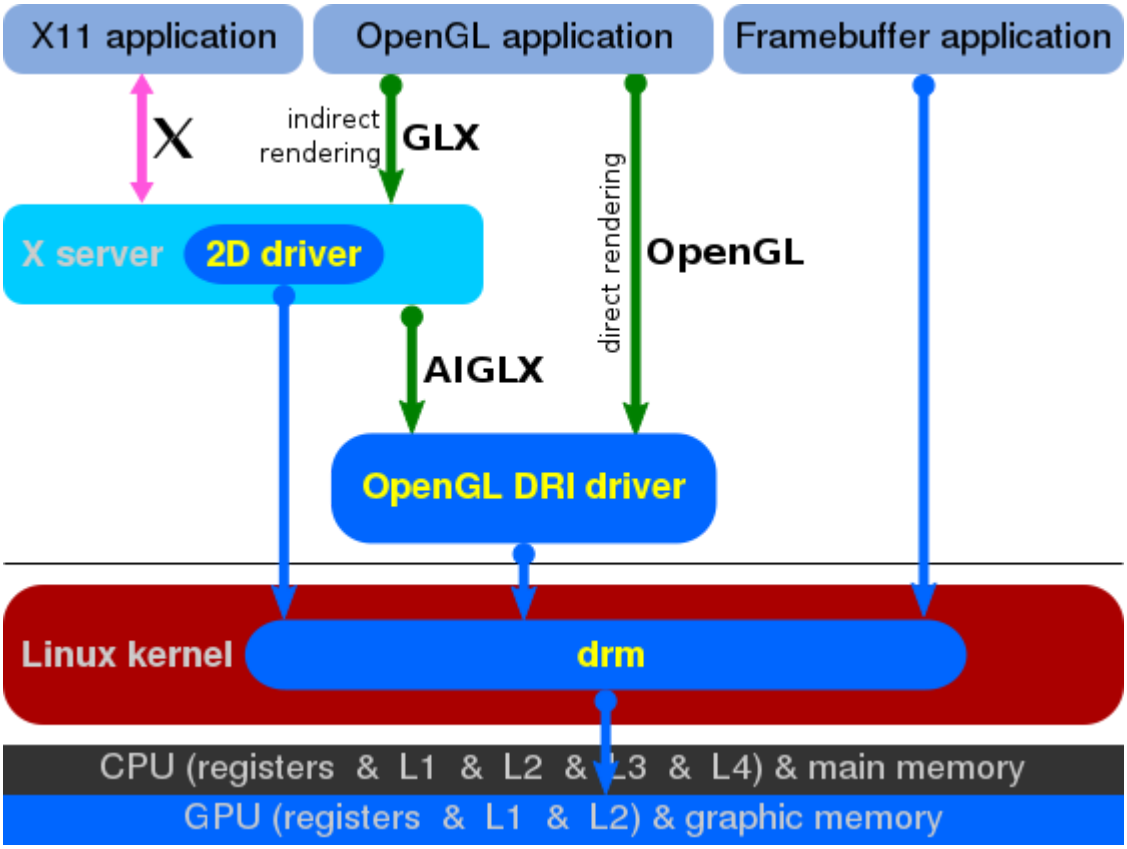
Direct Rendering Infrastructure is the new architecture that allows X clients to talk to the graphics hardware directly. Implementing DRI required

changes to various parts of the graphics stack including the X server, the kernel and various client libraries.

Although the term DRI usually refers to the complete architecture, it is often also used to refer only to the specific part of it that involves the interaction of applications with the X server, so be aware of this dual meaning when you read about this stuff on the Internet.

Another important part of DRI is the Direct Rendering Manager (DRM). This is the kernel side of the DRI architecture. Here, the kernel handles sensitive aspects like hardware locking, access synchronization, video memory and more. DRM also provides userspace with an API that it can use to submit commands and data in a format that is adequate for modern GPUs, which effectively allows userspace to communicate with the graphics hardware.

Notice that many of these things have to be done specifically for the target hardware so there are different DRM drivers for each GPU. In my Ubuntu system the DRM module for my Intel GPU is provided via the libdrm-intel1:amd64 package.



OpenGL with Direct Rendering (image via wikipedia)

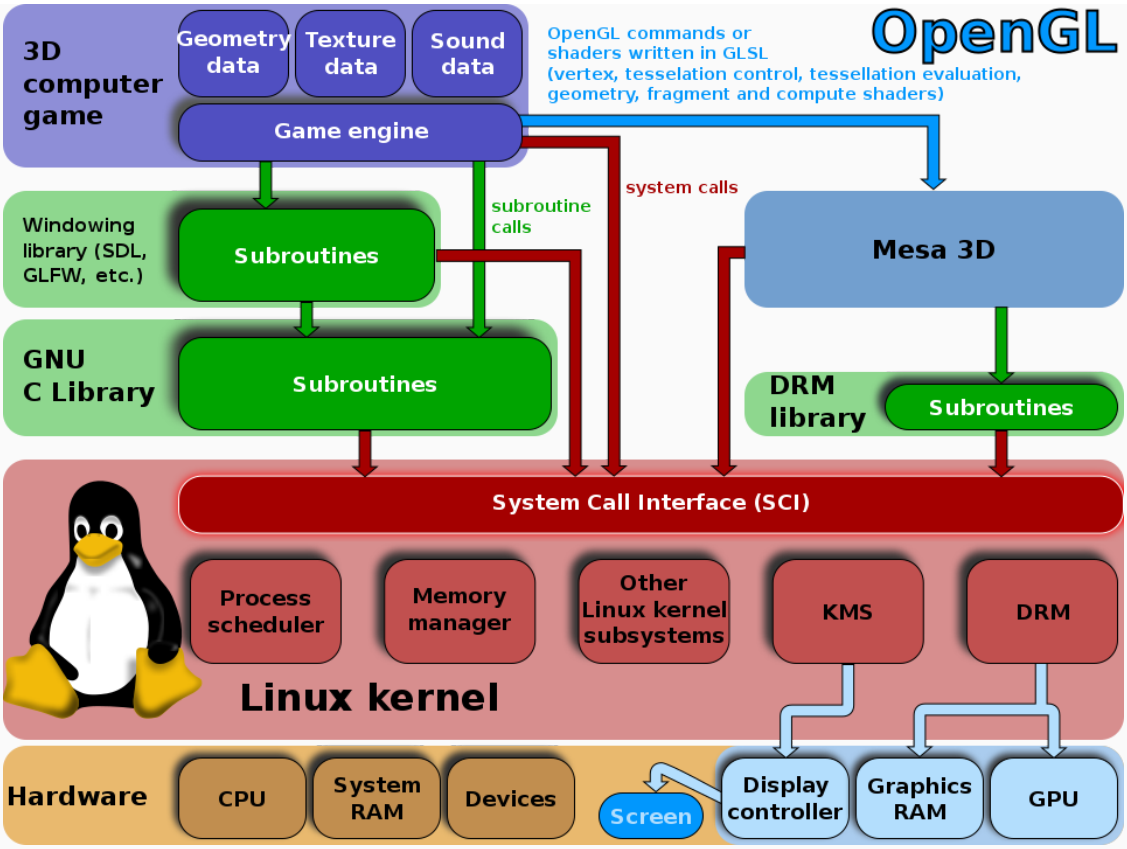
DRI/DRM provide the building blocks that enable userspace applications to access the graphics hardware directly in an efficient and safe manner, but in order to use OpenGL we need another piece of software that, using the infrastructure provided by DRI/DRM, implements the OpenGL API while respecting the X server requirements.

Enter Mesa

Mesa is a free software implementation of the OpenGL specification, and as such, it provides a *libGL.so*, which OpenGL based programs can use to output 3D graphics in Linux. Mesa can provide accelerated 3D graphics by taking advantage of the DRI architecture to gain direct access to the underlying graphics hardware in its implementation of the OpenGL API.

When our 3D application runs in an X11 environment it will output its graphics to a surface (window) allocated by the X server. Notice, however, that with DRI this will happen without intervention of the X server, so naturally there is some synchronization to do between the two, since the X server still owns the window Mesa is rendering to and is the one in charge of displaying its contents on the screen. This synchronization between the OpenGL application and the X server is part of DRI. Mesa's implementation of GLX (the extension of the OpenGL specification that addresses the X11 platform) uses DRI to talk to the X server and accomplish this.

Mesa also has to use DRM for many things. Communication with the graphics hardware happens by sending commands (for example "draw a triangle") and data (for example the vertex coordinates of the triangle, their color attributes, normals, etc). This process usually involves allocating a bunch of buffers in the graphics hardware where all these commands and data are copied so that the GPU can access them and do its work. This is enabled by the DRM driver, which is the one piece that takes care of managing video memory and which offers APIs to userspace (Mesa in this case) to do this for the specific target hardware. DRM is also required whenever we need to allocate and manage video memory in Mesa, so things like creating textures, uploading data to textures, allocating color, depth or stencil buffers, etc all require to use the DRM APIs for the target hardware.



OpenGL/Mesa in the context of 3D Linux games (image via wikipedia)

What's next?

Hopefully I have managed to explain what is the role of Mesa in the Linux graphics stack and how it works together with the Direct Rendering Infrastructure to enable efficient 3D graphics via OpenGL. In the next post we will cover Mesa in more detail, we will see that it is actually a framework where multiple OpenGL drivers live together, including both hardware and software variants, we will also have a look at its directory structure and identify its main modules, introduce the Gallium framework and more.

Diving into Mesa

AUGUST 8, 2014

Recap

In my last post I gave a quick introduction to the Linux graphics stack. There I explained how what we call a graphics driver in Linux is actually a combination of three different drivers:

- the user space X server DDX driver, which handles 2D graphics.
- the user space 3D OpenGL driver, that can be provided by Mesa.
- the kernel space DRM driver.

Now that we know where Mesa fits let's have a more detailed look into it.

DRI drivers and non-DRI drivers

As explained, Mesa handles 3D graphics by providing an implementation of the OpenGL API. Mesa OpenGL drivers are usually called DRI drivers too. Remember that, after all, the DRI architecture was brought to life precisely to enable efficient implementation of OpenGL drivers in Linux and, as I introduced in my previous post, DRI/DRM are the building blocks of the OpenGL drivers in Mesa.

There are other implementations of the OpenGL API available too. Hardware vendors that provide drivers for Linux will provide their own

implementation of the OpenGL API, usually in the form of a binary blob. For example, if you have an NVIDIA GPU and install NVIDIA's proprietary driver this will install its own libGL.so.

Notice that it is possible to create graphics drivers that do not follow the DRI architecture in Linux. For example, the NVIDIA proprietary driver installs a Kernel module that implements similar functionality to DRM but with a different API that has been designed by NVIDIA, and obviously, their corresponding user space drivers (DDX and OpenGL) will use this API instead of DRM to communicate with the NVIDIA kernel space driver.

Mesa, the framework

You have probably noticed that when I talk about Mesa I usually say 'drivers', in plural. That is because Mesa itself is not really a driver, but a project that hosts multiple drivers (that is, multiple implementations of the OpenGL API).

Indeed, Mesa is best seen as a framework for OpenGL implementators that provides abstractions and code that can be shared by multiple drivers. Obviously, there are many aspects of an OpenGL implementation that are independent of the underlying hardware, so these can be abstracted and reused.

For example, if you are familiar with OpenGL you know it provides a state based API. This means that many API calls do not have an immediate effect, they only modify the values of certain variables in the driver but do not require to push these new values to the hardware immediately. Indeed, usually that will happen later, when we actually render something by calling *glDrawArrays()* or a similar API: it is at that point that the driver will configure the 3D pipeline for rendering according to all the state that has

been set by the previous API calls. Since these APIs do not interact with the hardware their implementation can be shared by multiple drivers, and then, each driver, in their implementation of *glDrawArrays()*, can fetch the values stored in this state and translate them into something meaningful for the hardware at hand.

As such, Mesa provides abstractions for many things and even complete implementations for multiple OpenGL APIs that do not require interaction with the hardware, at least not immediate interaction.

Mesa also defines hooks for the parts where drivers may need to do hardware specific stuff, for example in the implementation of *glDrawArrays()*.

Looking into *glDrawArrays()*

Let's see an example of these hooks into a hardware driver by inspecting the stacktrace produced from a call to *glDrawArrays()* inside Mesa. In this case, I am using the Mesa Intel DRI driver and I am calling *glDrawArrays()* from a function named *render()* in my program. This is the relevant part of the stacktrace:

```
brw_upload_state () at brw_state_upload.c:651
brw_try_draw_prims () at brw_draw.c:483
brw_draw_prims () at brw_draw.c:578
vbo_draw_arrays () at vbo/vbo_exec_array.c:667
vbo_exec_DrawArrays () at
vbo/vbo_exec_array.c:819
render () at main.cpp:363
```

Notice that *glDrawArrays()* is actually *vbo_exec_DrawArrays()*. What is interesting about this stack is that *vbo_exec_DrawArrays()* and *vbo_draw_arrays()* are hardware independent and reused by many drivers inside Mesa. If you don't have an Intel GPU like me, but also use a Mesa,

your backtrace should be similar. These generic functions would usually do things like checks for API use errors, reformatting inputs in a way that is more appropriate for later processing or fetching additional information from the current state that will be needed to implement the actual operation in the hardware.

At some point, however, we need to do the actual rendering, which involves configuring the hardware pipeline according to the command we are issuing and the relevant state we have set in prior API calls. In the stacktrace above this starts with *brw_draw_prims()*. This function call is part of the Intel DRI driver, it is the hook where the Intel driver does the stuff required to configure the Intel GPU for drawing and, as you can see, it will later call something named *brw_upload_state()*, which will upload a bunch of state to the hardware to do exactly this, like configuring the various shader stages required by the current program, etc.

Registering driver hooks

In future posts we will discuss how the driver configures the pipeline in more detail, but for now let's just see how the Intel driver registers its hook for the *glDrawArrays()* call. If we look at the stacktrace, and knowing that *brw_draw_prims()* is the hook into the Intel driver, we can just inspect how it is called from *vbo_draw_arrays()*:

```
1  static void
2  vbo_draw_arrays(struct gl_context *ctx, GLenum
3  mode, GLint start,
4                  GLsizei count, GLuint
5  numInstances, GLuint baseInstance)
6  {
7     struct vbo_context *vbo = vbo_context(ctx);
8     (...)
9     vbo->draw_prims(ctx, prim, 1, NULL,
10 GL_TRUE, start, start + count - 1,
```

```
                                NULL, NULL);
    (...)
}
```

So the hook is *draw_prims()* inside vbo_context. Doing some trivial searches in the source code we can see that this hook is setup in *brw_draw_init()* like this:

```
void brw_draw_init( struct brw_context *brw )
{
   struct vbo_context *vbo = vbo_context(ctx);
   (...)
   /* Register our drawing function:
    */
   vbo->draw_prims = brw_draw_prims;
   (...)
}
```

Let's put a breakpoint there and see when Mesa calls into that:

```
brw_draw_init () at brw_draw.c:583
brwCreateContext () at brw_context.c:767
driCreateContextAttribs () at dri_util.c:435
dri2_create_context_attribs () at
dri2_glx.c:318
glXCreateContextAttribsARB () at
create_context.c:78
setupOpenGLContext () at main.cpp:411
init () at main.cpp:419
main () at main.cpp:477
```

So there it is, Mesa (unsurprisingly) calls into the Intel DRI driver when we setup the OpenGL context and it is there when the driver will register various hooks, including the one for drawing primitives.

We could do a similar thing to see how the driver registers its hook for the context creation. We will see that the Intel driver (as well as other drivers in Mesa) assign a global variable with the hooks they need like this:

```
static const struct __DriverAPIRec
brw_driver_api = {
   .InitScreen           = intelInitScreen2,
   .DestroyScreen        = intelDestroyScreen,
   .CreateContext        = brwCreateContext,
   .DestroyContext       =
intelDestroyContext,
   .CreateBuffer         = intelCreateBuffer,
   .DestroyBuffer        = intelDestroyBuffer,
   .MakeCurrent          = intelMakeCurrent,
   .UnbindContext        = intelUnbindContext,
   .AllocateBuffer       =
intelAllocateBuffer,
   .ReleaseBuffer        = intelReleaseBuffer
};

PUBLIC const __DRIextension
**__driDriverGetExtensions_i965(void)
{
   globalDriverAPI = &brw_driver_api;

   return brw_driver_extensions;
}
```

This global is then used throughout the DRI implementation in Mesa to call into the hardware driver as needed.

We can see that there are two types of hooks then, the ones that are needed to link the driver into the DRI implementation (which are the main entry points of the driver in Mesa) and then the hooks they add for tasks that are related to the hardware implementation of OpenGL bits, typically registered by the driver at context creation time.

In order to write a new DRI driver one would only have to write implementations for all these hooks, the rest is already implemented in Mesa and reused across multiple drivers.

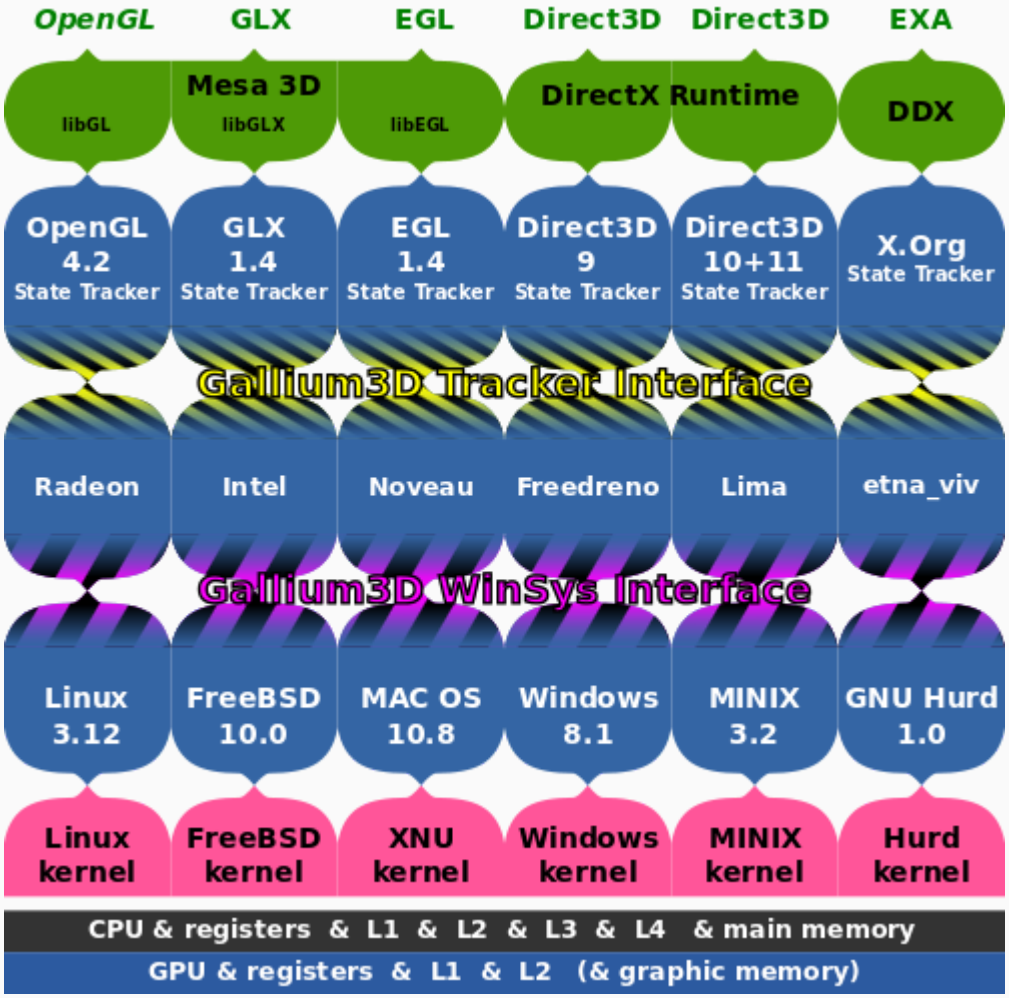
Gallium3D, a framework inside a framework

Currently, we can split Mesa DRI drivers in two kinds: the classic drivers (not based on the Gallium3D framework) and the new Gallium drivers.

Gallium3D is part of Mesa and attempts to make 3D driver development easier and more practical than it was before. For example, classic Mesa drivers are tightly coupled with OpenGL, which means that implementing support for other APIs (like Direct3D) would pretty much require to write a completely new implementation/driver. This is addressed by the Gallium3D framework by providing an API that exposes hardware functions as present in modern GPUs rather than focusing on a specific API like OpenGL.

Other benefits of Gallium include, for example, support for various Operating Systems by separating the part of the driver that relies on specific aspects of the underlying OS.

In the last years we have seen a lot of drivers moving to the Gallium infrastructure, including nouveau (the open source driver for NVIDIA GPUs), various radeon drivers, some software drivers (swrast, llvmpipe) and more.



Gallium3D driver model (image via wikipedia)

Although there were some efforts to port the Intel driver to Gallium in the past, development of the Intel Gallium drivers (i915g and i965g) is stalled now as far as I know. Intel is focusing in the classic version of the drivers instead. This is probably because it would take a large amount of time and effort to bring the current classic driver to Gallium with the same features and stability that it has in its current classic form for many generations of Intel GPUs. Also, there is a lot of work going on to add support for new OpenGL features to the driver at the moment, which seems to be the priority right now.

Gallium and LLVM

As we will see in more detail in future posts, writing a modern GPU driver involves a lot of native code generation and optimization. Also, OpenGL includes the OpenGL Shading Language (GLSL) which directly requires to have a GLSL compiler available in the driver too.

It is no wonder then that Mesa developers thought that it would make sense to reuse existing compiler infrastructure rather than building and using their own: enter LLVM.

By introducing LLVM into the mix, Mesa developers expect to bring new and better optimizations to shaders and produce better native code, which is critical to performance.

This would also allow to eliminate a lot of code from Mesa and/or the drivers. Indeed, Mesa has its own complete implementation of a GLSL compiler, which includes a GLSL parser, compiler and linker as well as a number of optimizations, both for abstract representations of the code, in Mesa, and for the actual native code for a specific GPU, in the actual hardware driver.

The way that Gallium plugs LLVM is simple: Mesa parses GLSL and produces LLVM intermediary representation of the shader code that it can then pass to LLVM, which will take care of the optimization. The role of hardware drivers in this scenario is limited to providing LLVM backends that describe their respective GPUs (instruction set, registers, constraints, etc) so that LLVM knows how it can do its work for the target GPU.

Hardware and Software drivers

Even today I see people who believe that Mesa is just a software implementation of OpenGL. If you have read my posts so far it should be

clear that this is not true: Mesa provides multiple implementations (drivers) of OpenGL, most of these are hardware accelerated drivers but Mesa also provides software drivers.

Software drivers are useful for various reasons:

- For developing and testing purposes, when you want to take the hardware out of the equation. From this point of view, a software representation can provide a reference for expected behavior that is not tied or constrained by any particular hardware. For example, if you have an OpenGL program that does not work correctly we can run it with the software driver: if it works fine then we know the problem is in the hardware driver, otherwise we can suspect that the problem is in the application itself.
- To allow execution of OpenGL in systems that lack 3D hardware drivers. It would obviously be slow, but in some scenarios it could be sufficient and it is definitely better than not having any 3D support at all.

I initially intended to cover more stuff in this post, but it is already getting long enough so let’s stop here for now. In the next post we will discuss how we can check and change the driver in use by Mesa, for example to switch between a software and hardware driver, and we will then start looking into Mesa’s source code and introduce its main modules.

Driver loading and querying in Mesa

SEPTEMBER 4, 2014

Recap

In my previous post I explained that Mesa is a framework for OpenGL driver development. As such, it provides code that can be reused by multiple driver implementations. This code is, of course, hardware agnostic, but frees driver developers from doing a significant part of the work. The framework also provides hooks for developers to add the bits of code that deal with the actual hardware. This design allows multiple drivers to co-exist and share a significant amount of code.

I also explained that among the various drivers that Mesa provides, we can find both hardware drivers that take advantage of a specific GPU and software drivers, that are implemented entirely in software (so they work on the CPU and do not depend on a specific GPU). The latter are obviously slower, but as I discussed, they may come in handy in some scenarios.

Driver selection

So, Mesa provides multiple drivers, but how does it select the one that fits the requirements of a specific system?

You have probably noticed that Mesa is deployed in multiple packages. In my Ubuntu system, the one that deploys the DRI drivers is *libgl1-mesa-dri:amd64*. If you check its contents you will see that this package installs OpenGL drivers for various GPUs:

```
# dpkg -L libgl1-mesa-dri:amd64
(...)
/usr/lib/x86_64-linux-gnu/gallium-
pipe/pipe_radeonsi.so
/usr/lib/x86_64-linux-gnu/gallium-
pipe/pipe_r600.so
/usr/lib/x86_64-linux-gnu/gallium-
pipe/pipe_nouveau.so
/usr/lib/x86_64-linux-gnu/gallium-
pipe/pipe_vmwgfx.so
/usr/lib/x86_64-linux-gnu/gallium-
pipe/pipe_r300.so
/usr/lib/x86_64-linux-gnu/gallium-
pipe/pipe_swrast.so
/usr/lib/x86_64-linux-gnu/dri/i915_dri.so
/usr/lib/x86_64-linux-gnu/dri/i965_dri.so
/usr/lib/x86_64-linux-gnu/dri/r200_dri.so
/usr/lib/x86_64-linux-gnu/dri/r600_dri.so
/usr/lib/x86_64-linux-gnu/dri/radeon_dri.so
/usr/lib/x86_64-linux-gnu/dri/r300_dri.so
/usr/lib/x86_64-linux-gnu/dri/vmwgfx_dri.so
/usr/lib/x86_64-linux-gnu/dri/swrast_dri.so
/usr/lib/x86_64-linux-
gnu/dri/nouveau_vieux_dri.so
/usr/lib/x86_64-linux-gnu/dri/nouveau_dri.so
/usr/lib/x86_64-linux-gnu/dri/radeonsi_dri.so
(...)
```

Since I have a relatively recent Intel GPU, the driver I need is the one provided in i965_dri.so. So how do we tell Mesa that this is the one we need? Well, the answer is that we don’t, Mesa is smart enough to know which driver is the right one for our GPU, and selects it automatically when you load *libGL.so*. The part of Mesa that takes care of this is called the *‘loader’*.

You can, however, point Mesa to look for suitable drivers in a specific directory other than the default, or force it to use a software driver using various environment variables.

What driver is Mesa actually loading?

If you want to know exactly what driver Mesa is loading, you can instruct it to dump this (and other) information to *stderr* via the *LIBGL_DEBUG* environment variable:

```
# LIBGL_DEBUG=verbose glxgears
libGL: screen 0 does not appear to be DRI3
capable
libGL: pci id for fd 4: 8086:0126, driver i965
libGL: OpenDriver: trying /usr/lib/x86_64-
linux-gnu/dri/tls/i965_dri.so
libGL: OpenDriver: trying /usr/lib/x86_64-
linux-gnu/dri/i965_dri.so
```

So we see that Mesa checks the existing hardware and realizes that the *i965* driver is the one to use, so it first attempts to load the TLS version of that driver and, since I don't have the TLS version, falls back to the normal version, which I do have.

The code in *src/loader/loader.c (loader_get_driver_for_fd)* is the one responsible for detecting the right driver to use (*i965* in my case). This receives a device fd as input parameter that is acquired previously by calling *DRI2Connect()* as part of the DRI bring up process. Then the actual driver file is loaded in *glx/dri_common.c (driOpenDriver)*.

We can also obtain a more descriptive indication of the driver we are loading by using the *glxinfo* program that comes with the *mesa-utils* package:

```
# glxinfo | grep -i "opengl renderer"
OpenGL renderer string: Mesa DRI Intel(R)
Sandybridge Mobile
```

This tells me that I am using the Intel hardware driver, and it also shares information related with the specific Intel GPU I have (*SandyBridge*).

Forcing a software driver

I have mentioned that having software drivers available comes in handy at times, but how do we tell the loader to use them? Mesa provides an environment variable that we can set for this purpose, so switching between a hardware driver and a software one is very easy to do:

```
# LIBGL_DEBUG=verbose LIBGL_ALWAYS_SOFTWARE=1
glxgears
libGL: OpenDriver: trying /usr/lib/x86_64-
linux-gnu/dri/tls/swrast_dri.so
libGL: OpenDriver: trying /usr/lib/x86_64-
linux-gnu/dri/swrast_dri.so
```

As we can see, setting *LIBGL_ALWAYS_SOFTWARE* will make the loader select a software driver (*swrast*).

If I force a software driver and call *glxinfo* like I did before, this is what I get:

```
# LIBGL_ALWAYS_SOFTWARE=1 glxinfo | grep -i
"opengl renderer"
OpenGL renderer string: Software Rasterizer
```

So it is clear that I am using a software driver in this case.

Querying the driver for OpenGL features

The *glxinfo* program also comes in handy to obtain information about the specific OpenGL features implemented by the driver. If you want to check if the Mesa driver for your hardware implements a specific OpenGL extension you can inspect the output of glxinfo and look for that extension:

```
# glxinfo | grep GL_ARB_texture_multisample
```

You can also ask glxinfo to include hardware limits for certain OpenGL features including the *-l* switch. For example:

```
# glxinfo -l | grep GL_MAX_TEXTURE_SIZE
GL_MAX_TEXTURE_SIZE = 8192
```

Coming up next

In my next posts I will cover the directory structure of the Mesa repository, identifying its main modules, which should give Mesa newcomers some guidance as to where they should look for when they need to find the code that deals with something specific. We will then discuss how modern 3D hardware has changed the way GPU drivers are developed and explain how a modern 3D graphics pipeline works, which should pave the way to start looking into the real guts of Mesa: the implementation of shaders.

An eagle eye view into the Mesa source tree

SEPTEMBER 8, 2014

Recap

My last post introduced Mesa's loader as the module that takes care of auto-selecting the right driver for our hardware. If the loader fails to find a suitable hardware driver it will fall back to a software driver, but we can also force this situation ourselves, which may come in handy in some scenarios. We also took a quick look at the *glxinfo* tool that we can use to query the capabilities and features exposed by the selected driver.

The topic of today focuses on providing a quick overview of the Mesa source code tree, which will help us identify the parts of the code that are relevant to our interests depending on the driver and/or the feature we intend to work on.

Browsing the source code

First off, there is already some documentation on this topic available on the Mesa 3D website that is a good place to start. Since that already gives some insight on what goes into each part of the repository I'll focus on complementing that information with a little bit more of detail for some of the most important parts I have interacted with so far:

- In *src/egl/* we have the implementation of the EGL standard. If you are working on EGL-specific features, tracking down an EGL-specific problem or you are simply curious about how EGL links into the GL implementation, this is the place you want to visit. This includes the EGL implementations for the X11, DRM and Wayland platforms.

- In *src/glx/* we have the OpenGL bits relating specifically to X11 platforms, known as GLX. So if you are working on the GLX layer, this is the place to go. Here there is all the stuff that takes care of interacting with the XServer, the client-side DRI implementation, etc.

- *src/glsl/* contains a critical aspect of Mesa: the GLSL compiler used by all Mesa drivers. It includes a GLSL parser, the definition of the Mesa IR, also referred to as GLSL IR, used to represent shader programs internally, the shader linker and various optimization passes that operate on the Mesa IR. The resulting Mesa IR produced by the GLSL compiler is then consumed by the various drivers which transform it into native GPU code that can be loaded and run in the hardware.

- *src/mesa/main/* contains the core Mesa elements. This includes hardware-independent views of core objects like textures, buffers, vertex array objects, the OpenGL context, etc as well as basic infrastructure, like linked lists.

- *src/mesa/drivers/* contains the actual classic drivers (not Gallium). DRI drivers in particular go into *src/mesa/drivers/dri*. For example the Intel i965 driver goes into *src/mesa/drivers/dri/i965*. The code here is, for the most part, very specific to the underlying hardware platforms.

- *src/mesa/swrast*/* and *src/mesa/tnl*/* provide software implementations for things like rasterization or vertex transforms. Used by some software

drivers and also by some hardware drivers to implement certain features for which they don't have hardware support or for which hardware support is not yet available in the driver. For example, the *i965* driver implements operations on the accumulation and selection buffers in software via these modules.

- *src/mesa/vbo/* is another important module. Across its various versions, OpenGL has specified many ways in which a program can tell OpenGL about its vertex data, from using functions of the *glVertex*()* family inside *glBegin()/glEnd()* blocks, to things like vertex arrays, vertex array objects, display lists, etc… The drivers, however, do not need to deal with all this, Mesa makes it so that they always receive their vertex data as collection of vertex arrays, significantly reducing complexity on the side of the driver implementator. This is the module that takes care of managing all this, so no matter what type of drawing you GL program is doing or how it specifies its vertex data, it will always go through this module before it reaches the driver.

- *src/loader/*, as we have seen in my previous post, contains the Mesa driver loader, which provides the logic necessary to decide which Mesa driver is the right one to use for a specific hardware so that Mesa's libGL.so can auto-select the right driver when loaded.

- *src/gallium/* contains the *Gallium3D* framework implementation. If, like me, you only work on a classic driver, you don't need to care about the contents of this at all. If you are working on Gallium drivers however, this is the place where you will find the various Gallium drivers in development (inside *src/gallium/drivers/*), like the various Gallium ATI/AMD drivers, Nouveau or the LLVM based software driver (llvmpipe) and the Gallium state trackers.

So with this in mind, one should have enough information to know where to start looking for something specific:

- If are interested in how vertex data provided to OpenGL is manipulated and uploaded to the GPU, the *vbo* module is probably the right place to look.

- If we are looking to work on a specific aspect of a concrete hardware driver, we should go to the corresponding directory in *src/mesa/drivers/* if it is a classic driver, or *src/gallium/drivers* if it is a Gallium driver.

- If we want to know about how Mesa, the framework, abstracts various OpenGL concepts like textures, vertex array objects, shader programs, etc. we should look into *src/mesa/main/*.

- If we are interested in the platform specific support, be it EGL or GLX, we want to look into *src/egl* or *src/glx*.

- If we are interested in the GLSL implementation, which involves anything from the compiler to the intermediary IR and the various optimization passes, we need to look into *src/glsl/*.

Coming up next

So now that we have an eagle view of the contents of the Mesa repository let’s see how we can prepare a development environment so we can start hacking on
some stuff. I’ll cover this in my next post.

Setting up a development environment for Mesa

SEPTEMBER 15, 2014

Recap

In my previous post I provided an overview of the Mesa source tree and identified some of its main modules.

Since we are on that subject I thought it would make sense to give a few tips on how to setup the development environment for Mesa too, so here I go.

Development environment

Mesa is mostly written in a combination of *C* and *C++*, uses *autotools* for its build system and *Git* for version control, so it should be a fairly familiar environment for many people. I am not going to explain how to build *autotools* projects here, there is plenty of documentation available on that subject, so instead I will focus on the specifics of Mesa.

First we need to checkout the source code. If you do not have a developer account then do an anonymous checkout:

```
1 # git clone
  git://anongit.freedesktop.org/git/mesa/mesa
```

If you do have a developer account do this instead:

```
# git clone
git+ssh://username@git.freedesktop.org/git/mesa/me
```

Next, we will have to deal with dependencies. This should not be too hard though. Mesa is fairly low in the software stack so it does not have many and the ones it has seem to have a fairly stable API and don't change too often, so typically, you should be able to build Mesa if you have a recent distribution and you keep it up to date. For reference, as of now I can build Mesa on my Ubuntu 14.04 without any problems.

In any case, the actual dependencies you will need to get may vary depending on the drivers you want to build, the target platform and the features you want to enable. For example, the *R300 Gallium* driver requires *LLVM*, but the *Intel i965* driver doesn't.

Notice, however, that if you are hacking on features that require specific builds of the *XServer*, *Wayland/Weston* or similar stuff the required setup will be more complex, since you would probably need to include these other projects into the mix, together with their respective dependencies.

Configuring the source tree

Here I will mention some of the Mesa specific options that I found to be more useful in my time with Mesa:

–*enable-debug*: This is necessary, at least, to get assertions to work, and you want this while you are developing. Mesa and the drivers have assertions on many places to make sure that new code does not break certain assumptions or violate hardware constraints, so you really want to

make sure that you have these activated when you are developing. It also adds *“-g -O0”* to enable debug support.

–*with-dri-driver*s: This is the list of classic Mesa DRI drivers you want to build. If you know you will only hack on the *i965* driver, for example, then building other drivers will only slow down your builds.

–with-gallium-drivers: This is the list of Gallium drivers you want to build. Again, if you are hacking on the classic DRI *i965* driver you are probably not interested in building any Gallium drivers.

Notice that if you are working on the Mesa framework layer, that is, the bits shared by all drivers, instead of the internals of a specific driver, you will probably want to include more drivers in the build to make sure that they keep building after your changes.

–*with-egl-platforms*: This is a list of supported platforms. Same as with the options above, you probably only want to build Mesa for the platform or platforms you are working on.

Besides using a combination of these options, you probably also want to set your CFLAGS and CXXFLAGS (remember that Mesa uses both *C* and *C*++). I for one like to pass *“-g3”*, for example.

Using your built version of Mesa

Once you have built Mesa you can type ‘make install’ to install the libraries and drivers. Probably, you have configured autotools (via the *–-prefix* option) to do this to a safe location that does not conflict with your distribution installation of Mesa and now your problem is to tell your

OpenGL programs that they should use this version of Mesa instead of the one provided by your distro.

You will have to adjust a couple of environment variables for this:

LIBGL_DRIVERS_PATH: Set this to the path where your built drivers have been installed. This will tell Mesa's loader to look for the drivers here.

LD_LIBRARY_PATH: Set this to the path where your Mesa libraries have been installed. This will make it so that OpenGL programs load your recently built libGL.so rather than your system's.

For more tips I'd suggest to read this short thread in the Mesa mailing list, which has some Mesa developers discussing their development environment setup.

Coming up next

In the next post I will provide an introduction to modern 3D graphics hardware. After all, the job of the graphics driver is all about programming the hardware, so having a basic understanding of how it works is a requirement if want to do any meaningful driver development.

A brief overview of the 3D pipeline

NOVEMBER 11, 2014

Recap

In the previous post I discussed the Mesa development environment and gave a few tips for newcomers, but before we start hacking on the code we should have a look at how modern GPUs look like, since that has a definite impact on the design and implementation of driver code. Let's get to it.

Fixed Function vs Programmable hardware

Before the advent of shading languages like *GLSL* we did not have the option to program the 3D hardware at will. Instead, the hardware would have specific units dedicated to implement certain operations (like vertex transformations) that could only be used through specific APIs, like those exposed by OpenGL. These units are usually labeled as *Fixed Function*, to differentiate them from modern GPUs that also expose fully programmable units.

What we have now in modern GPUs is a *fully programmable pipeline*, where graphics developers can code graphics algorithms of various sorts in high level programming languages like *GLSL*. These programs are then compiled and loaded into the GPU to execute specific tasks. This gives graphics developers a huge amount of freedom and power, since they are

no longer limited to preset APIs exposing fixed functionality (like the old OpenGL lightning models for example).

Modern graphics drivers

But of course all this flexibility and power that graphics developers enjoy today come at the expense of significantly more complex hardware and drivers, since the drivers are responsible for exposing all that flexibility to the developers while ensuring that we still obtain the best performance out of the hardware in each scenario.

Rather than acting as a bridge between a fixed API like OpenGL and fixed function hardware, drivers also need to handle general purpose graphics programs written in high-level languages. This is a big change. In the case of OpenGL, this means that the driver needs to provide an implementation of the *GLSL* language, so suddenly, the driver is required to incorporate a full compiler and deal with all sort of problems that belong to the realm of compilers, like choosing an intermediary representation for the program code (IR), performing optimization passes and generating native code for the GPU.

Overview of a modern 3D pipeline

I have mentioned that modern GPUs expose fully programmable hardware units. These are called shading units, and the idea is that these units are connected in a pipeline so that the output of a shading unit becomes the input of the next. In this model, the application developer pushes vertices to one end of the pipeline and usually obtains rendered pixels on the other side. In between these two ends there are a number of units making this transition possible and a number of these will be programmable, which

means that the graphics developer can control how these vertices are transformed into pixels at different stages.

The image below shows a simplified example of a 3D graphics pipeline, in this case as exposed by the *OpenGL 4.3* specification. Let's have a quick look at some of its main parts:

The OpenGL 4.3 3D pipeline (image via www.brightsideofnews.com)

Vertex Shader (VS)

This programmable shading unit takes vertices as input and produces vertices as output. Its main job is to transform these vertices in any way the graphics developer sees fit. Typically, this is were we would do transforms like vertex projection,
rotation, translation and, generally, compute per-vertex attributes that we won't to provide to later stages in the pipeline.

The vertex shader processes vertex data as provided by APIs like *glDrawArrays* or *glDrawElements* and outputs shaded vertices that will be assembled into primitives as indicated by the OpenGL draw command (*GL_TRIANGLES, GL_LINES*, etc).

Geometry Shader

Geometry shaders are similar to vertex shaders, but instead of operating on individual vertices, they operate on a geometry level (that is, a line, a triangle, etc), so they can take the output of the vertex shader as their input.

The geometry shader unit is programmable and can be used to add or remove vertices from a primitive, clip primitives, spawn entirely new primitives or modify the geometry of a primitive (like transforming triangles into quads or points into triangles, etc). Geometry shaders can also be used to implement basic tessellation even if dedicated tessellation units present in modern hardware are a better fit for this job.

In *GLSL*, some operations like layered rendering (which allows rendering to multiple textures in the same program) are only accessible through geometry shaders, although this is now also possible in vertex shaders via a particular extension.

The output of a geometry shader are also primitives.

Rasterization

So far all the stages we discussed manipulated vertices and geometry. At some point, however, we need to render pixels. For this, primitives need to be rasterized, which is the process by which they are broken into individual fragments that would then be colored by a fragment shader and eventually turn into pixels in a frame buffer. Rasterization is handled by the rasterizer fixed function unit.

The rasterization process also assigns depth information to these fragments. This information is necessary when we have a 3D scene where multiple polygons overlap on the screen and we need to decide which polygon's fragments should be rendered and which should be discarded because they are hidden by other polygons.

Finally, the rasterization also interpolates per-vertex attributes in order to compute the corresponding fragment values. For example, let's say that

we have a line primitive where each vertex has a different color attribute, one red and one green. For each fragment in the line the rasterizer will compute interpolated color values by combining red and green depending on how close or far the fragments are to each vertex. With this, we will obtain red fragments on the side of the red vertex that will smoothly transition to green as we move closer to the green vertex.

In summary, the input of the rasterizer are the primitives coming from a vertex, tessellation or geometry shader and the output are the fragments that build the primitive's surface as projected on the screen including color, depth and other interpolated per-vertex attributes.

Fragment Shader (FS)

The programmable fragment shader unit takes the fragments produced by the rasterization process and executes an algorithm provided by a graphics developer to compute the final color, depth and stencil values for each fragment. This unit can be used to achieve numerous visual effects, including all kinds of post-processing filters, it is usually where we will sample textures to color polygon surfaces, etc.

This covers some of the most important elements in 3D the graphics pipeline and should be sufficient, for now, to understand some of the basics of a driver. Notice, however that have not covered things like transform feedback, tessellation or compute shaders. I hope I can get to cover some of these in future posts.

But before we are done with the overview of the 3D pipeline we should cover another topic that is fundamental to how the hardware works: parallelization.

Parallelization

Graphics processing is a very resource demanding task. We are continuously updating and redrawing our graphics 30/60 times per second. For a full HD resolution of 1920×1080 that means that we need to redraw over 2 million pixels in each go (124.416.000 pixels per second if we are doing 60 FPS). That's a lot.

To cope with this the architecture of GPUs is massively parallel, which means that the pipeline can process many vertices/pixels simultaneously. For example, in the case of the Intel Haswell GPUs, programmable units like the VS and GS have multiple *Execution Units (EU)*, each with their own set of *ALUs*, etc that can spawn up to 70 threads each (for GS and VS) while the fragment shader can spawn up to 102 threads. But that is not the only source of parallelism: each thread may handle multiple objects (vertices or pixels depending on the case) at the same time. For example, a VS thread in Intel hardware can shade two vertices simultaneously, while a FS thread can shade up to 8 (SIMD8) or 16 (SIMD16) pixels in one go.

Some of these means of parallelism are relatively transparent to the driver developer and some are not. For example, SIMD8 vs SIMD16 or single vertex shading vs double vertex shading requires specific configuration and writing driver code that is aligned with the selected configuration. Threads are more transparent, but in certain situations the driver developer may need to be careful when writing code that can require a sync between all running threads, which would obviously hurt performance, or at least be careful to do that kind of thing when it would hurt performance the least.

Coming up next

So that was a very brief introduction to how modern 3D pipelines look like. There is still plenty of stuff I have not covered but I think we can go through a lot of that in later posts as we dig deeper into the driver code. My next post will discuss how Mesa models various of the programmable pipeline stages I have introduced here, so stay tuned!

An introduction to Mesa’s GLSL compiler (I)

MARCH 3, 2015

Recap

In my last post I explained that modern 3D pipelines are programmable and how this has impacted graphics drivers. In the following posts we will go deeper into this aspect by looking at different parts of Mesa’s GLSL compiler. Specifically, this post will cover the GLSL parser, the Mesa IR and built-in variables and functions.

The GLSL parser

The job of the parser is to process the shader source code string provided via glShaderSource and transform it into a suitable binary representation that is stored in RAM and can be efficiently processed by other parts of the compiler in later stages.

The parser consists of a set of *Lex/Yacc* rules to process the incoming shader source. The lexer (*glsl_parser.ll*) takes care of tokenizing the source code and the parser (*glsl_parser.yy*) adds meaning to the stream of tokens identified in
the lexer stage.

Similarly, just like in *C* or *C++*, GLSL includes a pre-processor that goes through the shader source code before the main parser kicks in. Mesa's implementation of the GLSL pre-processor lives in *src/glsl/glcpp* and is also based on *Lex/Yacc* rules.

The output of the parser is an *Abstract Syntax Tree (AST)* that lives in RAM memory, which is a binary representation of the shader source code. The nodes that make this tree are defined in *src/glsl/ast.h*.

For someone familiar with all the *Lex/Yacc* stuff, the parser implementation in Mesa should feel familiar enough.

The next step takes care of converting from the *AST* to a different representation that is better suited for the kind of operations that drivers will have to do with it. This new representation, called the IR (Intermediate Representation), is usually referenced in Mesa as *Mesa IR*, *GLSL IR* or simply *HIR*.

The *AST* to *Mesa IR* conversion is driven by the code in *src/glsl/ast_to_hir.cpp*.

Mesa IR

The *Mesa IR* is the main data structure used in the compiler. Most of the work that the compiler does can be summarized as:

- Optimizations in the IR
- Modifications in the IR for better/easier integration with GPU hardware
- Linking multiple shaders (multiple IR instances) into a single program.
- Generating native assembly code for the target GPU from the IR

As we can see, the Mesa IR is at the core of all the work that the compiler has to do, so understanding how it is setup is necessary to work in this part of Mesa.

The nodes in the Mesa IR tree are defined in *src/glsl/ir.h*. Let's have a look at the most important ones:

At the top of the class hierarchy for the IR nodes we have *exec_node*, which is Mesa's way of linking independent instructions together in a list to make a program. This means that each instruction has previous and next pointers to the instructions that are before and after it respectively. So, we have *ir_instruction*, the base class for all nodes in the tree, inherit from *exec_node*.

Another important node is *ir_rvalu*e, which is the base class used to represent expressions. Generally, anything that can go on the right side of an assignment is an *ir_rvalue*. Subclasses of *ir_rvalue* include *ir_expression*, used to represent all kinds of unary, binary or ternary operations (supported operators are defined in the *ir_expression_operation* enumeration), *ir_texture*, which is used to represent texture operations like a texture lookup, *ir_swizzle*, which is used for swizzling values in vectors, all the *ir_dereferenc*e nodes, used to access the values stored in variables, arrays, structs, etc. and *ir_constant*, used to represent constants of all basic types (bool, float, integer, etc).

We also have *ir_variable*, which represents variables in the shader code. Notice that the definition of *ir_variable* is quite large… in fact, this is by large the node with the most impact in the memory footprint of the compiler when compiling shaders in large games/applications. Also notice that the IR differentiates between variables and variable dereferences (the

fact of looking into a variable's value), which are represented as an *ir_rvalue*.

Similarly, the IR also defines nodes for other language constructs like *ir_loop, ir_if, ir_assignment*, etc.

Debugging the IR is not easy, since the representation of a shader program in IR nodes can be quite complex to traverse and inspect with a debugger. To help with this Mesa provides means to print the IR to a human-readable text format. We can enable this by using the environment variable *MESA_GLSL=dump*. This will instruct Mesa to print both the original shader source code and its IR representation. For example:

```
$ MESA_GLSL=dump ./test_program

GLSL source for vertex shader 1:
#version 140
#extension GL_ARB_explicit_attrib_location :
enable

layout(location = 0) in vec3 inVertexPosition;
layout(location = 1) in vec3 inVertexColor;

uniform mat4 MVP;
smooth out vec3 out0;

void main()
{
  gl_Position = MVP * vec4(inVertexPosition,
1);
  out0 = inVertexColor;
}

GLSL IR for shader 1:
(
(declare (sys ) int gl_InstanceID)
(declare (sys ) int gl_VertexID)
(declare (shader_out ) (array float 0)
gl_ClipDistance)
```

```
(declare (shader_out ) float gl_PointSize)
(declare (shader_out ) vec4 gl_Position)
(declare (uniform ) (array vec4 56)
gl_CurrentAttribFragMESA)
(declare (uniform ) (array vec4 33)
gl_CurrentAttribVertMESA)
(declare (uniform ) gl_DepthRangeParameters
gl_DepthRange)
(declare (uniform ) int gl_NumSamples)
(declare () int gl_MaxVaryingComponents)
(declare () int gl_MaxClipDistances)
(declare () int
gl_MaxFragmentUniformComponents)
(declare () int gl_MaxVaryingFloats)
(declare () int gl_MaxVertexUniformComponents)
(declare () int gl_MaxDrawBuffers)
(declare () int gl_MaxTextureImageUnits)
(declare () int
gl_MaxCombinedTextureImageUnits)
(declare () int gl_MaxVertexTextureImageUnits)
(declare () int gl_MaxVertexAttribs)
(declare (shader_in ) vec3 inVertexPosition)
(declare (shader_in ) vec3 inVertexColor)
(declare (uniform ) mat4 MVP)
(declare (shader_out smooth) vec3 out0)
(function main
  (signature void
    (parameters
    )
    (
      (declare (temporary ) vec4 vec_ctor)
      (assign  (w) (var_ref vec_ctor)
(constant float (1.000000)) )
      (assign  (xyz) (var_ref vec_ctor)
(var_ref inVertexPosition) )
      (assign  (xyzw) (var_ref gl_Position)
            (expression vec4 * (var_ref MVP)
(var_ref vec_ctor) ) )
      (assign  (xyz) (var_ref out0)  (var_ref
inVertexColor) )
    ))
)
)
```

Notice, however, that the IR representation we get is not the one that is produced by the parser. As we will see later, that initial IR will be modified in multiple ways by Mesa, for example by adding different kinds of optimizations, so the IR that we see is the result after all these processing passes over the original IR. Mesa refers to this post-processed version of the IR as *LIR* (low-level IR) and to the initial version of the IR as produced by the parser as *HIR* (high-level IR). If we want to print the *HIR* (or any intermediary version of the IR as it transforms into the final *LIR*), we can edit the compiler and add calls to _mesa_print_ir as needed.

Traversing the Mesa IR

We mentioned before that some of the compiler's work (a big part, in fact) has to do with optimizations and modifications of the IR. This means that the compiler needs to traverse the IR tree and identify subtrees that are relevant to this kind of operations. To achieve this, Mesa uses the visitor design pattern.

Basically, the idea is that we have a visitor object that can traverse the IR tree and we can define the behavior we want to execute when it finds specific nodes.

For instance, there is a very simple example of this in *src/glsl/linker.cpp*: *find_deref_visitor*, which detects if a variable is ever read. This involves traversing the IR, identifying *ir_dereference_variable* nodes (the ones where a variable's value is accessed) and check if the name of that variable matches the one we are looking for. Here is the visitor class definition:

```
1 /**
2  * Visitor that determines whether or not a
3 variable is ever read.
```

```
 */
class find_deref_visitor : public
ir_hierarchical_visitor {
public:
   find_deref_visitor(const char *name)
      : name(name), found(false)
   {
      /* empty */
   }

   virtual ir_visitor_status
visit(ir_dereference_variable *ir)
   {
      if (strcmp(this->name, ir->var->name) ==
0) {
         this->found = true;
         return visit_stop;
      }

      return visit_continue;
   }

   bool variable_found() const
   {
      return this->found;
   }

private:
   const char *name;       /**< Find writes to
a variable with this name. */
   bool found;             /**< Was a write to
the variable found? */
};
```

And this is how we get to use this, for example to check if the shader code ever reads *gl_Vertex*:

```
find_deref_visitor find("gl_Vertex");
find.run(sh->ir);
if (find.variable_found()) {
  (...)
}
```

Most optimization and lowering passes in Mesa are implemented as visitors and follow a similar idea. We will look at examples of these in a later post.

Built-in variables and functions

GLSL defines a set of built-in variables (with ‘gl_’ prefix) for each shader stage which Mesa injects into the shader code automatically. If you look at the example where we used *MESA_GLSL=dump* to obtain the generated *Mesa IR* you can see some of these variables.

Mesa implements support for built-in variables in *_mesa_glsl_initialize_variables()*, defined in *src/glsl/builtin_variables.cpp*.

Notice that some of these variables are common to all shader stages, while some are specific to particular stages or available only in specific versions of GLSL.

Depending on the type of variable, Mesa or the hardware driver may be able to provide the value immediately (for example for variables holding constant values like *gl_MaxVertexAttribs* or *gl_MaxDrawBuffers*). Otherwise, the driver will probably have to fetch (or generate) the value for the variable from the hardware at program run-time by generating native code that is added to the user program. For example, a geometry shader that uses *gl_PrimitiveID* will need that variable updated for each primitive processed by the Geometry Shader unit in a draw call. To achieve this, a driver might have to generate native code that fetches the current primitive ID value from the hardware and puts stores it in the register that provides the storage for the *gl_PrimitveID* variable before the user code is executed.

The GLSL language also defines a number of available built-in functions that must be provided by implementators, like *texture(), mix(), or dot()*, to name a few examples. The entry point in Mesa's GLSL compiler for built-in functions
is *src/glsl/builtin_functions.cpp.*

The method *builtin_builder::create_builtins()* takes care of registering built-in functions, and just like with built-in variables, not all functions are always available: some functions may only be available in certain shading units, others may only be available in certain GLSL versions, etc. For that purpose, each built-in function is registered with a predicate that can be used to test if that function is at all available in a specific scenario.

Built-in functions are registered by calling the *add_function()* method, which registers all versions of a specific function. For example *mix()* for float, vec2, vec3, vec4, etc Each of these versions has its own availability predicate. For instance, *mix()* is always available for float arguments, but using it with integers requires *GLSL 1.30* and the *EXT_shader_integer_mix* extension.

Besides the availability predicate, *add_function()* also takes an *ir_function_signature*, which tells Mesa about the specific signature of the function being registered. Notice that when Mesa creates signatures for the functions it also defines the function body. For example, the following code snippet defines the signature for *modf()*:

```
1  ir_function_signature *
2  builtin_builder::_modf(builtin_available_predicat
3  avail,
4                         const glsl_type *type)
5  {
6     ir_variable *x = in_var(type, "x");
7     ir_variable *i = out_var(type, "i");
8     MAKE_SIG(type, avail, 2, x, i);
```

```
 9
10      ir_variable *t = body.make_temp(type, "t");
11      body.emit(assign(t, expr(ir_unop_trunc, x)));
12      body.emit(assign(i, t));
13      body.emit(ret(sub(x, t)));
14
15      return sig;
    }
```

GLSL's *modf()* splits a number in its integer and fractional parts. It assigns the integer part to an output parameter and the function return value is the fractional part.

This signature we see above defines input parameter 'x' of type 'type' (the number we want to split), an output parameter 'i' of the same type (which will hold the integer part of 'x') and a return type 'type'.

The function implementation is based on the existence of the unary operator *ir_unop_trunc*, which can take a number and extract its integer part. Then it computes the fractional part by subtracting that from the original number.

When the *modf()* built-in function is used, the call will be expanded to include this IR code, which will later be transformed into native code for the GPU by the corresponding hardware driver. In this case, it means that the hardware driver is expected to provide an implementation of the *ir_unop_trunc* operator, for example, which in the case of the Intel i965 driver is implemented as a single hardware instruction (see *brw_vec4_visitor.cpp* or *brw_fs_visitor.cpp* in *src/mesa/drivers/dri/i965*).

In some cases, the implementation of a built-in function can't be defined at the IR level. In this case the implementation simply emits an ad-hoc IR

node that drivers can identify and expand appropriately. An example of this is *EmitVertex()* in a geometry shader. This is not really a function call in the traditional sense, but a way to signal the driver that we have defined all the attributes of a vertex and it is time to “push” that vertex into the current primitive. The meaning of “pushing the vertex” is something that can’t be defined at the IR level because it will be different for each driver/hardware. Because of that, the built-in function simply injects an IR node *ir_emit_vertex* that drivers can identify and implement properly when the time comes. In the case of the Intel code, pushing a vertex involves a number of steps that are very intertwined with the hardware, but it basically amounts to generating native code that implements the behavior that the hardware expects for that to happen. If you are curious, the implementation of this in the i965 driver code can be found in *brw_vec4_gs_visitor.cpp*, in the *visit()* method that takes an *ir_emit_vertex* IR node as parameter.

Coming up next

In this post we discussed the parser, which is the entry point for the compiler, and introduced the *Mesa IR*, the main data structure. In following posts we will delve deeper into the GLSL compiler implementation. Specifically, we will look into the lowering and optimization passes as well as the linking process and the hooks for hardware drivers that deal with native code generation.

An introduction to Mesa's GLSL compiler (II)

MARCH 6, 2015

Recap

My previous post served as an initial look into Mesa's GLSL compiler, where we discussed the Mesa IR, which is a core aspect of the compiler. In this post I'll introduce another relevant aspect: IR lowering.

IR lowering

There are multiple lowering passes implemented in Mesa (check *src/glsl/lower_*.cpp* for a complete list) but they all share a common denominator: their purpose is to re-write certain constructs in the IR so they fit better the underlying GPU hardware.

In this post we will look into the *lower_instructions.cpp* lowering pass, which rewrites expression operations that may not be supported directly by GPU hardware with different implementations.

The lowering process involves traversing the IR, identifying the instructions we want to lower and modifying the IR accordingly, which fits well into the visitor pattern strategy discussed in my previous post. In this case, expression lowering is handled by the *lower_instructions_visitor*

class, which implements the lowering pass in the *visit_leave()* method for ir_expression nodes.

The hierarchical visitor class, which serves as the base class for most visitors in Mesa, defines *visit()* methods for leaf nodes in the IR tree, and *visit_leave()/visit_enter()* methods for non-leaf nodes. This way, when traversing intermediary nodes in the IR we can decide to take action as soon as we enter them or when we are about to leave them.

In the case of our *lower_instructions_visitor* class, the *visit_leave()* method implementation is a large *switch()* statement with all the operators that it can lower.

The code in this file lowers common scenarios that are expected to be useful for most GPU drivers, but individual drivers can still select which of these lowering passes they want to use. For this purpose, hardware drivers create instances of the *lower_instructions* class passing the list of lowering passes to enable. For example, the Intel i965 driver does:

```
const int bitfield_insert = brw->gen >= 7
                            ?
BITFIELD_INSERT_TO_BFM_BFI
                            : 0;
lower_instructions(shader->base.ir,
                   MOD_TO_FLOOR |
                   DIV_TO_MUL_RCP |
                   SUB_TO_ADD_NEG |
                   EXP_TO_EXP2 |
                   LOG_TO_LOG2 |
                   bitfield_insert |
                   LDEXP_TO_ARITH);
```

Notice how in the case of Intel GPUs, one of the lowering passes is conditionally selected depending on the hardware involved. In this case, *brw->gen >= 7* selects GPU generations since *IvyBridge*.

Let's have a look at the implementation of some of these lowering passes. For example, *SUB_TO_ADD_NEG* is a very simple one that transforms subtractions into negative additions:

```
void
lower_instructions_visitor::sub_to_add_neg(ir_expr
*ir)
{
   ir->operation = ir_binop_add;
   ir->operands[1] =
      new(ir) ir_expression(ir_unop_neg, ir-
>operands[1]->type,
                            ir->operands[1], NULL)
   this->progress = true;
}
```

As we can see, the lowering pass simply changes the operator used by the *ir_expression* node, and negates the second operand using the unary negate operator (ir_unop_neg), thus, converting the original *a = b – c* into *a = b + (-c)*.

Of course, if a driver does not have native support for the subtraction operation, it could still do this when it processes the IR to produce native code, but this way Mesa is saving driver developers that work. Also, some lowering passes may enable optimization passes after the lowering that drivers might miss otherwise.

Let's see a more complex example: *MOD_TO_FLOOR*. In this case the lowering pass provides an implementation of *ir_binop_mod* (modulo) for GPUs that don't have a native modulo operation.

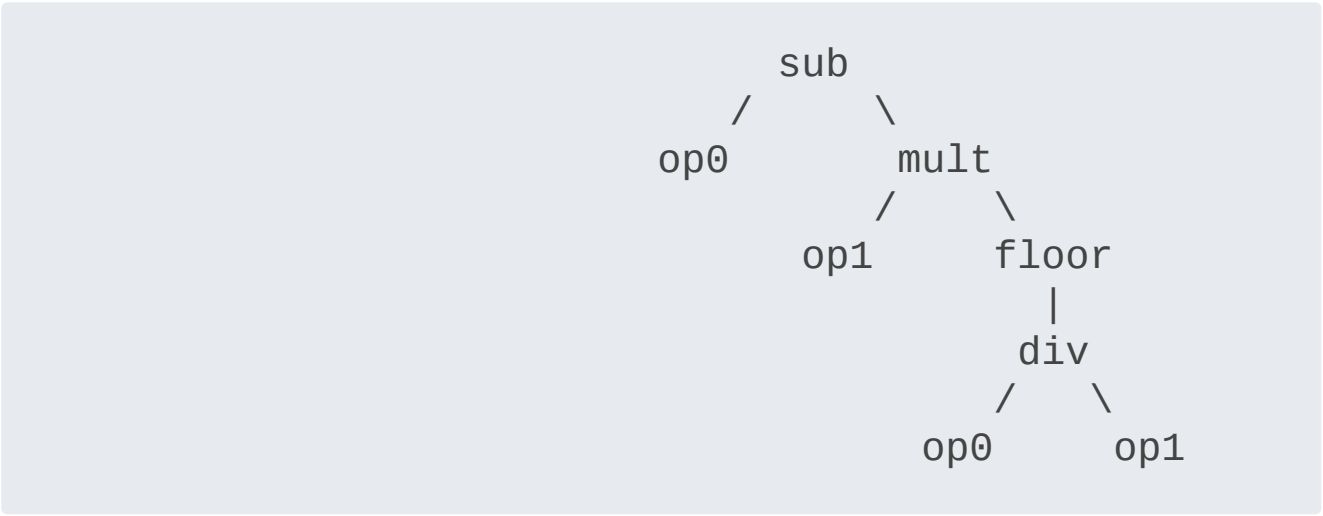
The modulo operation takes two operands (op0, op1) and implements the C equivalent of the '*op0 % op1*', that is, it computes the remainder of the division of op0 by op1. To achieve this the lowering pass breaks the

modulo operation as *mod(op0, op1) = op0 – op1 * floor(op0 / op1)*, which requires only multiplication, division and subtraction. This is the implementation:

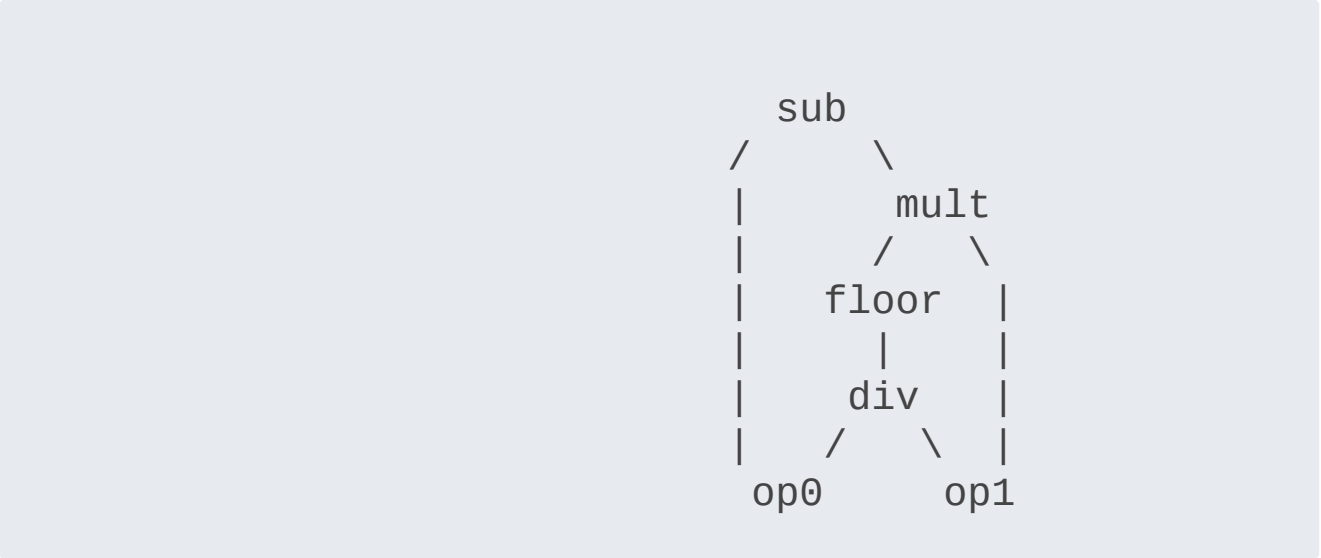
```
ir_variable *x = new(ir) ir_variable(ir->operands
>type, "mod_x",
                                     ir_var_tempo
ir_variable *y = new(ir) ir_variable(ir->operands
>type, "mod_y",
                                     ir_var_tempo
this->base_ir->insert_before(x);
this->base_ir->insert_before(y);

ir_assignment *const assign_x =
   new(ir) ir_assignment(new(ir)
ir_dereference_variable(x),
                         ir->operands[0], NULL);
ir_assignment *const assign_y =
   new(ir) ir_assignment(new(ir)
ir_dereference_variable(y),
                         ir->operands[1], NULL);

this->base_ir->insert_before(assign_x);
this->base_ir->insert_before(assign_y);

ir_expression *const div_expr =
   new(ir) ir_expression(ir_binop_div, x->type,
                         new(ir)
ir_dereference_variable(x),
                         new(ir)
ir_dereference_variable(y));

/* Don't generate new IR that would need to be lo
in an additional
 * pass.
 */
if (lowering(DIV_TO_MUL_RCP) && (ir->type->is_flo
||
    ir->type->is_double()))
   div_to_mul_rcp(div_expr);

ir_expression *const floor_expr =
```

```
39    new(ir) ir_expression(ir_unop_floor, x->type,
40 div_expr);
41
42 if (lowering(DOPS_TO_DFRAC) && ir->type->is_doubl
43    dfloor_to_dfrac(floor_expr);
44
   ir_expression *const mul_expr =
      new(ir) ir_expression(ir_binop_mul,
                            new(ir)
   ir_dereference_variable(y),
                            floor_expr);

   ir->operation = ir_binop_sub;
   ir->operands[0] = new(ir) ir_dereference_variable
   ir->operands[1] = mul_expr;
   this->progress = true;
```

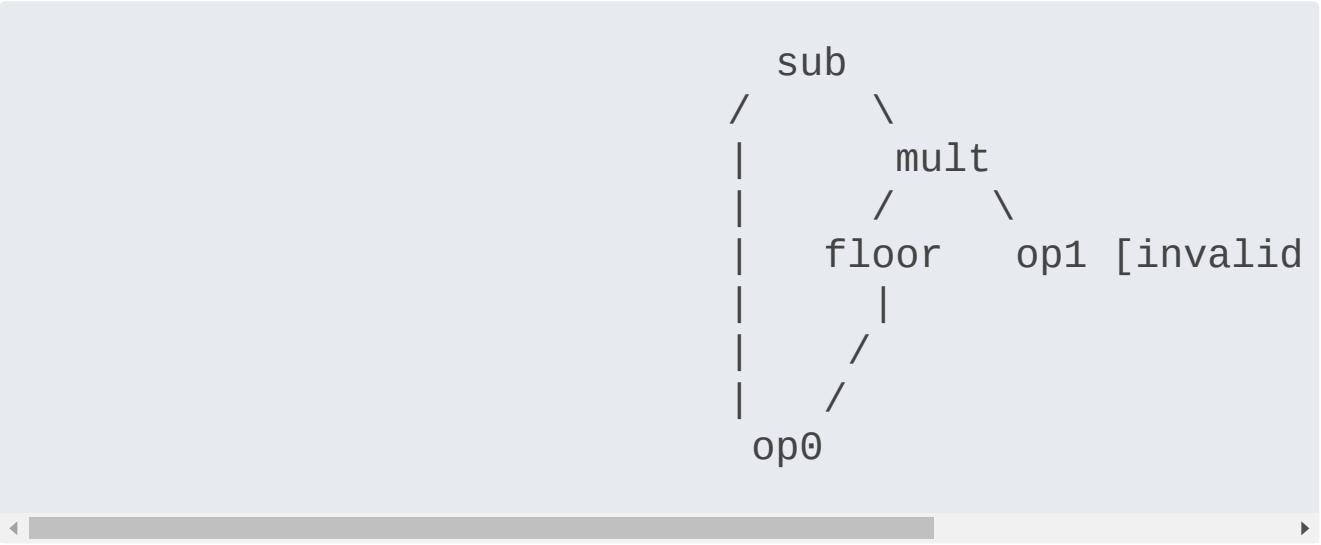
Notice how the first thing this does is to assign the operands to a variable. The reason for this is a bit tricky: since we are going to implement *ir_binop_mod* as *op0 – op1 * floor(op0 / op1)*, we will need to refer to the IR nodes op0 and op1 twice in the tree. However, we can't just do that directly, for that would mean that we have the same node (that is, the same pointer) linked from two different places in the IR expression tree. That is, we want to have this tree:



Instead of this other tree:

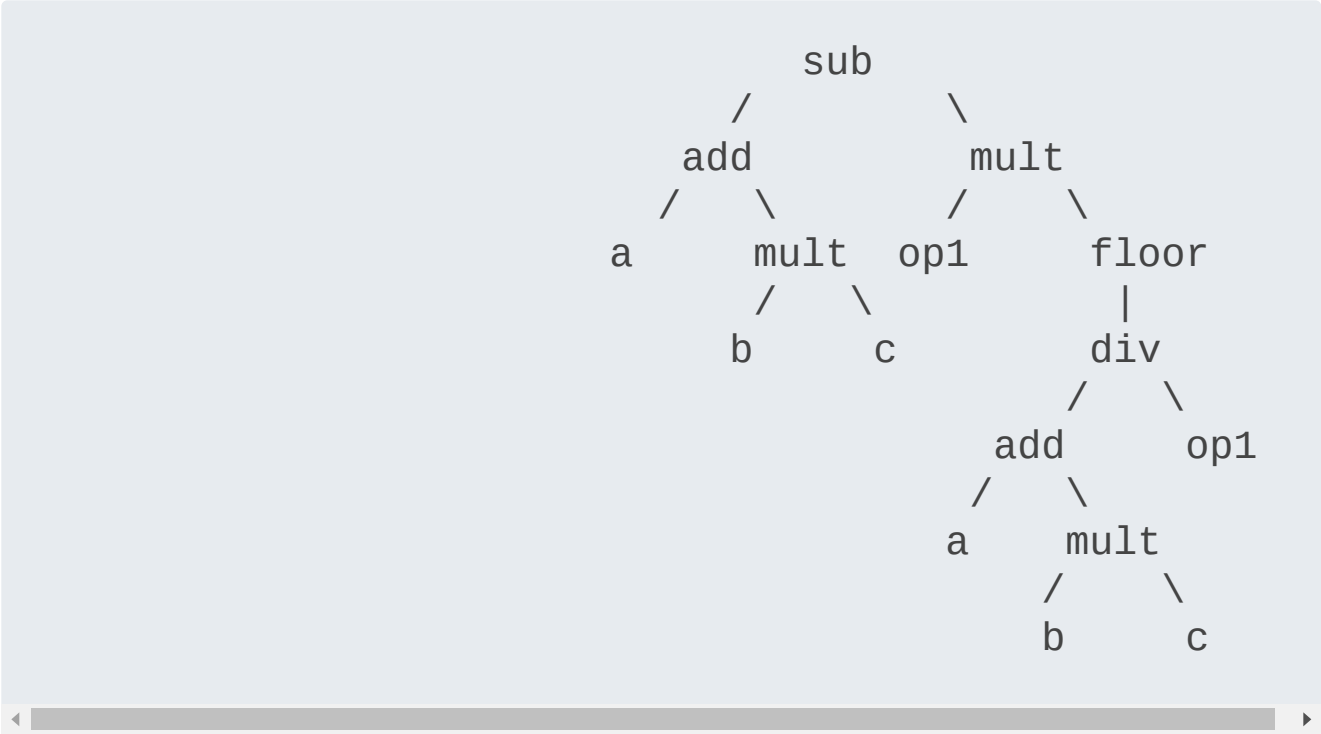


This second version of the tree is problematic. For example, let's say that a hypothetical optimization pass detects that *op1* is a constant integer with value *1*, and realizes that in this case *div(op0/op1)* == *op0*. When doing that optimization, our *div* subtree is removed, and with that, *op1* could be removed too (and possibily freed), leaving the other reference to that operand in the IR pointing to an invalid memory location… we have just corrupted our IR:

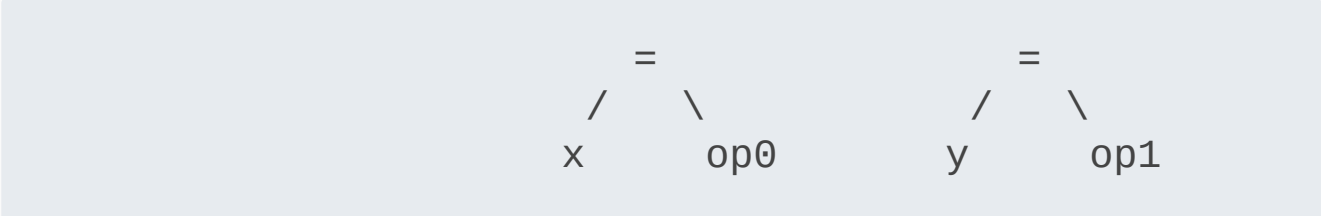


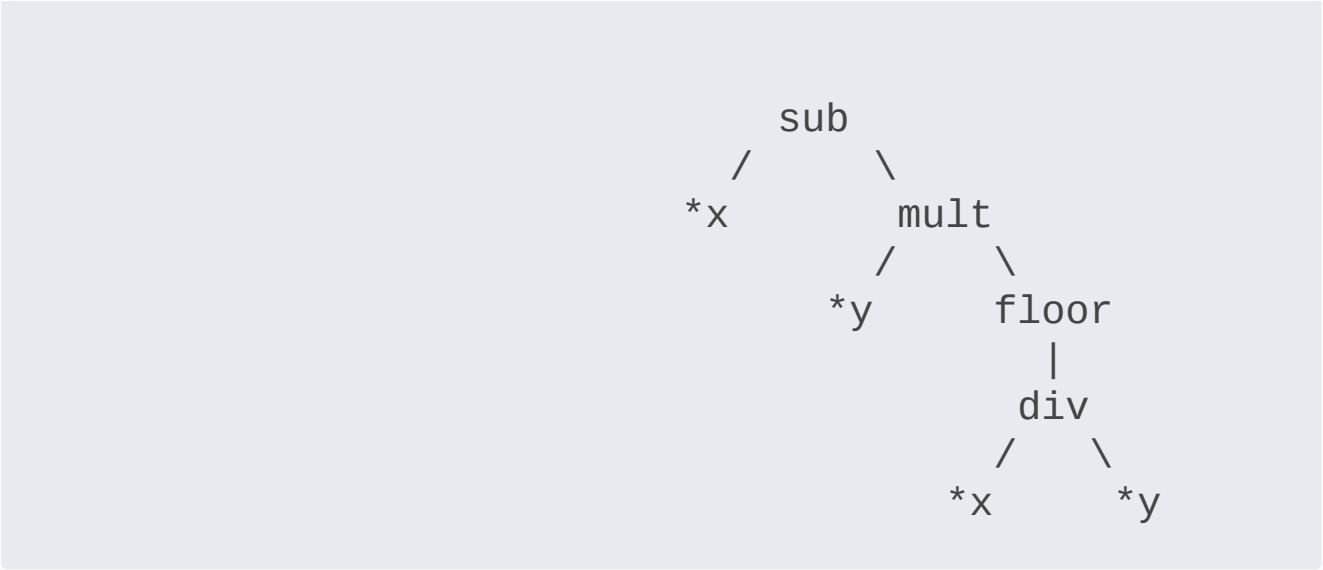
Instead, what we want to do here is to clone the nodes each time we need a new reference to them in the IR. All IR nodes have a *clone()* method for this purpose. However, in this particular case, cloning the nodes creates a

new problem: *op0* and *op1* are *ir_expression* nodes so, for example, *op0* could be the expression *a + b * c*, so cloning the expression would produce suboptimal code where the expression gets replicated. This, at best, will lead to slower
compilation times due to optimization passes needing to detect and fix that, and at worse, that would go undetected by the optimizer and lead to worse performance where we compute the value of the expression multiple times:



The solution to this problem is to assign the expression to a variable, then dereference that variable (i.e., read its value) wherever we need. Thus, the implementation defines two variables *(x, y)*, assigns *op0* and *op1* to them and creates new dereference nodes wherever we need to access the value of the *op0* and *op1* expressions:





In the diagram above, each variable dereference is marked with an *'*'*, and each one is a new IR node (so both appearances of *'*x'* refer to different IR nodes, both representing two different reads of the same variable). With this solution we only evaluate the *op0* and *op1* expressions once (when they get assigned to the corresponding variables) and we never refer to the same IR node twice from different places (since each variable dereference is a new IR node).

Now that we know why we assign these two variables, let's continue looking at the code of the lowering pass:

In the next step we implement *op0 / op1* using a ir_binop_div expression. To speed up compilation, if the driver has the *DIV_TO_MUL_RCP* lowering pass enabled, which transforms *a / b* into *a * 1 / b* (where *1 / b* could be a native instruction), we immediately execute the lowering pass for that expression. If we didn't do this here, the resulting IR would contain a division operation that might have to be lowered in a later pass, making the compilation process slower.

The next step uses a *ir_unop_floor* expression to compute *floor(op0/op1)*, and again, tests if this operation should be lowered too, which might be

the case if the type of the operands is a 64bit double instead of a regular 32bit float, since GPUs may only have a native floor instruction for 32bit floats.

Next, we multiply the result by *op1* to get *op1 * floor(op0 / op1)*.

Now we only need to subtract this from op0, which would be the root IR node for this expression. Since we want the new IR subtree spawning from this root node to replace the old implementation, we directly edit the IR node we are lowering to replace the *ir_binop_mod* operator with *ir_binop_sub*, make a dereference to *op1* in the first operand and link the expression holding *op1 * floor(op0 / op1)* in the second operand, effectively attaching our new implementation in place of the old version. This is how the original and lowered IRs look like:

Original IR:

```
  [prev inst] -> mod -> [next inst]
                 /   \
              op0     op1
```

Lowered IR:

```
[prev inst] -> var x -> var y ->   =   ->   =   ->   s
                                  / \      / \      /
                                 x  op0   y  op1  *x
```

Finally, we return true to let the compiler know that we have optimized the IR and that as a consequence we have introduced new nodes that may be subject to further lowering passes, so it can run a new pass. For example, the subtraction we just added may be lowered again to a negative addition as we have seen before.

Coming up next

Now that we learnt about lowering passes we can also discuss optimization passes, which are very similar since they are also based on the visitor implementation in Mesa and also transform the *Mesa IR* in a similar way.

Bringing ARB_shader_storage_buffer_object to Mesa and i965

MAY 20, 2015

In the last weeks I have been working together with my colleague Samuel on bringing support for **ARB_shader_storage_buffer_object**, an **OpenGL 4.3** feature, to *Mesa* and the *Intel i965* driver, so I figured I would write a bit on what this brings to *OpenGL/GLSL* users. If you are interested, read on.

Introducing Shader Storage Buffer Objects

This extension introduces the concept of **shader storage buffer objects (SSBOs)**, which is a new type of *OpenGL* buffer. *SSBOs* allow GL clients to create buffers that shaders can then map to variables (known as **buffer variables**) via *interface blocks*. If you are familiar with *Uniform Buffer Objects (UBOs)*, *SSBOs* are pretty similar but:

- They are read/write, unlike *UBOs*, which are read-only.
- They allow a number of atomic operations on them.
- They allow an optional unsized array at the bottom of their definitions.

Since *SSBOs* are read/write, they create a bidirectional channel of communication between the GPU and CPU spaces: the GL application

can set the value of shader variables by writing to a regular OpenGL buffer, but the shader can also update the values stored in that buffer by assigning values to them in the shader code, making the changes visible to the GL application. This is a major difference with *UBOs*.

In a parallel environment such as a GPU where we can have multiple shader instances running simultaneously (processing multiple vertices or fragments from a specific rendering call) we should be careful when we use *SSBOs*. Since all these instances will be simultaneously accessing the same buffer there are implications to consider relative to the order of reads and writes. The spec does not make many guarantees about the order in which these take place, other than ensuring that the order of reads and writes *within a specific execution of a shader* is preserved. Thus, it is up to the graphics developer to ensure, for example, that each execution of a fragment or vertex shader writes to a different offset into the underlying buffer, or that writes to the same offset always write the same value. Otherwise the results would be undefined, since they would depend on the order in which writes and reads from different instances happen in a particular execution.

The spec also allows to use *glMemoryBarrier()* from shader code and *glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT)* from a GL application to add sync points. These ensure that all memory accesses to buffer variables issued before the barrier are completely executed before moving on.

Another tool for developers to deal with concurrent accesses is *atomic operations*. The spec introduces a number of new atomic memory functions for use with buffer variables: **atomicAdd, atomicMin, atomicMax, atomicAnd, atomicOr, atomicXor, atomicExchange**

(atomic assignment to a buffer variable), **atomicCompSwap** (atomic conditional assignment to a buffer variable).

The optional unsized array at the bottom of an *SSBO* definition can be used to push a dynamic number of entries to the underlying buffer storage, up to the total size of the buffer allocated by the GL application.

Using shader storage buffer objects (GLSL)

Okay, so how do we use *SSBOs*? We will introduce this through an example: we will use a buffer to record information about the fragments processed by the fragment shader. Specifically, we will group fragments according to their X coordinate (by computing an index from the coordinate using a modulo operation). We will then record how many fragments are assigned to a particular index, the first fragment to be assigned to a given index, the last fragment assigned to a given index, the total number of fragments processed and the complete list of fragments processed.

To store all this information we will use the *SSBO* definition below:

```
layout(std140, binding=0) buffer SSBOBlock {
   vec4 first[8];     // first fragment
coordinates assigned to index
   vec4 last[8];      // last fragment
coordinates assigned to index
   int counter[8];    // number of fragments
assigned to index
   int total;         // number of fragments
processed
   vec4 fragments[];  // coordinates of all
fragments processed
};
```

Notice the use of the keyword **buffer** to tell the compiler that this is a *shader storage buffer object*. Also notice that we have included an unsized array called *fragments[]*, there can only be one of these in an *SSBO* definition, and in case there is one, it has to be the last field defined.

In this case we are using **std140** layout mode, which imposes certain alignment rules for the buffer variables within the *SSBO*, like in the case of *UBOs*. These alignment rules may help the driver implement read/write operations more efficiently since the underlying GPU hardware can usually read and write faster from and to aligned addresses. The downside of **std140** is that because of these alignment rules we also waste some memory and we need to know the alignment rules on the GL side if we want to read/write from/to the buffer. Specifically for *SSBOs*, the specification introduces a new layout mode: **std430**, which removes these alignment restrictions, allowing for a more efficient memory usage implementation, but possibly at the expense of some performance impact.

The *binding* keyword, just like in the case of *UBOs*, is used to select the buffer that we will be reading from and writing to when accessing these variables from the shader code. It is the application's responsibility to bound the right buffer to the binding point we specify in the shader code.

So with that done, the shader can read from and write to these variables as we see fit, but we should be aware of the fact that multiple instances of the shader could be reading from and writing to them simultaneously. Let's look at the fragment shader that stores the information we want into the *SSBO*:

```
1  void main() {
2     int index = int(mod(gl_FragCoord.x, 8));
3
4     int i = atomicAdd(counter[index], 1);
5     if (i == 0)
```

```
6           first[index] = gl_FragCoord;
7       else
8           last[index] = gl_FragCoord;
9
10      i = atomicAdd(total, 1);
11      fragments[i] = gl_FragCoord;
12  }
```

The first line computes an index into our integer array buffer variable by using *gl_FragCoord*. Notice that different fragments could get the same index. Next we increase in one unit *counter[index]*. Since we know that different fragments can get to do this at the same time we use an atomic operation to make sure that we don’t lose any increments.

Notice that if two fragments can write to the same index, reading the value of *counter[index]* after the *atomicAdd* can lead to different results. For example, if two fragments have already executed the *atomicAdd*, and assuming that *counter[index]* is initialized to 0, then both would read *counter[index] == 2*, however, if only one of the fragments has executed the atomic operation by the time it reads *counter[index]* it would read a value of 1, while the other fragment would read a value of 2 when it reaches that point in the shader execution. Since our shader intends to record the coordinates of the first fragment that writes to *counter[index]*, that won’t work for us. Instead, we use the return value of the atomic operation (which returns the value that the *buffer variable* had right before changing it) and we write to *first[index]* only when that value was 0. Because we use the atomic operation to read the previous value of *counter[index]*, only one fragment will read a value of 0, and that will be the fragment that first executed the atomic operation.

If this is not the first fragment assigned to that index, we write to *last[index]* instead. Again, multiple fragments assigned to the same index could do this simultaneously, but that is okay here, because we only care about the

the last write. Also notice that it is possible that different executions of the same rendering command produce different values of *first[]* and *last[]*.

The remaining instructions unconditionally push the fragment coordinates to the unsized array. We keep the last index into the unsized array *fragments[]* we have written to in the *buffer variable total*. Each fragment will atomically increase *total* before writing to the unsized array. Notice that, once again, we have to be careful when reading the value of *total* to make sure that each fragment reads a different value and we never have two fragments write to the same entry.

Using shader storage buffer objects (GL)

On the side of the GL application, we need to create the buffer, bind it to the appropriate binding point and initialize it. We do this as usual, only that we use the new *GL_SHADER_STORAGE_BUFFER* target:

```
typedef struct {
   float first[8*4];      // vec4[8]
   float last[8*4];       // vec4[8]
   int counter[8*4];      // int[8] padded as
per std140
   int total;             // int
   int pad[3];            // padding: as per
std140 rules
   char fragments[1024];  // up to 1024 bytes
of unsized array
} SSBO;

SSBO data;

(...)

memset(&data, 0, sizeof(SSBO));

GLuint buf;
glGenBuffers(1, &buf);
```

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0,
buf);
glBufferData(GL_SHADER_STORAGE_BUFFER,
sizeof(SSBO), &data, GL_DYNAMIC_DRAW);
```

The code creates a buffer, binds it to binding point 0 of *GL_SHADER_STORAGE_BUFFER* (the same we have bound our shader to) and initializes the buffer data to 0. Notice that because we are using **std140** we have to be aware of the alignment rules at work. We could have used **std430** instead to avoid this.

Since we have 1024 bytes for the *fragments[]* unsized array and we are pushing a vec4 (16 bytes) worth of data to it with every fragment we process then we have enough room for 64 fragments. It is the developer's responsibility to ensure that this limit is not surpassed, otherwise we would write beyond the allocated space for our buffer and the results would be undefined.

The next step is to do some rendering so we get our shaders to work. That would trigger the execution of our fragment shader for each fragment produced, which will generate writes into our buffer for each buffer variable the shader code writes to. After rendering, we can map the buffer and read its contents from the GL application as usual:

```
1  SSBO *ptr = (SSBO *) glMapNamedBuffer(buf,
2  GL_READ_ONLY);
3
4  /* List of fragments recorded in the unsized
5  array */
6  printf("%d fragments recorded:\n", ptr-
7  >total);
8  float *coords = (float *) ptr->fragments;
9  for (int i = 0; i < ptr->total; i++, coords
10 +=4) {
11    printf("Fragment %d: (%.1f, %.1f, %.1f,
12 %.1f)\n",
13
```

```
            i, coords[0], coords[1], coords[2],
coords[3]);
}

/* First fragment for each index used */
for (int i = 0; i < 8; i++) {
   if (ptr->counter[i*4] > 0)
      printf("First fragment for index %d:
(%.1f, %.1f, %.1f, %.1f)\n",
             i, ptr->first[i*4], ptr-
>first[i*4+1],
             ptr->first[i*4+2], ptr-
>first[i*4+3]);
}

/* Last fragment for each index used */
for (int i = 0; i < 8; i++) {
   if (ptr->counter[i*4] > 1)
      printf("Last fragment for index %d:
(%.1f, %.1f, %.1f, %.1f)\n",
             i, ptr->last[i*4], ptr-
>last[i*4+1],
             ptr->last[i*4+2], ptr-
>last[i*4+3]);
   else if (ptr->counter[i*4] == 1)
      printf("Last fragment for index %d:
(%.1f, %.1f, %.1f, %.1f)\n",
             i, ptr->first[i*4], ptr-
>first[i*4+1],
             ptr->first[i*4+2], ptr-
>first[i*4+3]);
}

/* Fragment counts for each index */
for (int i = 0; i < 8; i++) {
   if (ptr->counter[i*4] > 0)
      printf("Fragment count at index %d:
%d\n", i, ptr->counter[i*4]);
}
glUnmapNamedBuffer(buf);
```

I get this result for an execution where I am drawing a handful of points:

```
4 fragments recorded:
Fragment 0: (199.5, 150.5, 0.5, 1.0)
Fragment 1: (39.5, 150.5, 0.5, 1.0)
Fragment 2: (79.5, 150.5, 0.5, 1.0)
Fragment 3: (139.5, 150.5, 0.5, 1.0)

First fragment for index 3: (139.5, 150.5,
0.5, 1.0)
First fragment for index 7: (39.5, 150.5, 0.5,
1.0)

Last fragment for index 3: (139.5, 150.5, 0.5,
1.0)
Last fragment for index 7: (79.5, 150.5, 0.5,
1.0)

Fragment count at index 3: 1
Fragment count at index 7: 3
```

It recorded 4 fragments that the shader mapped to indices 3 and 7. Multiple fragments where assigned to index 7 but we could handle that gracefully by using the corresponding atomic functions. Different executions of the same program will produce the same 4 fragments and map them to the same indices, but the first and last fragments recorded for index 7 can change between executions.

Also notice that the first fragment we recorded in the unsized array (fragments[0]) is not the first fragment recorded for index 7 (fragments[1]). That means that the execution of fragments[0] got first to the unsized array addition code, but the execution of fragments[1] beat it in the race to execute the code that handled the assignment to the first/last arrays, making clear that we cannot make any assumptions regarding the execution order of reads and writes coming from different instances of the same shader execution.

So that's it, the patches are now in the *mesa-dev* mailing list undergoing review and will hopefully land soon, so look forward to it! Also, if you have

any interesting uses for this new feature, let me know in the comments.

Implementing ARB_shader_storage_buffer

JULY 8, 2015

In my previous post I introduced *ARB_shader_storage_buffer*, an *OpenGL 4.3* feature that is coming soon to *Mesa* and the *Intel i965 driver*. While that post focused on explaining the features introduced by the extension, in this post I'll dive into some of the implementation aspects, for those who are curious about this kind of stuff. Be warned that some parts of this post will be specific to Intel hardware.

Following the trail of UBOs

As I explained in my previous post, *SSBOs* are similar to *UBOs*, but they are read-write. Because there is a lot of code already in place in Mesa's GLSL compiler to deal with *UBOs*, it made sense to try and reuse all the data structures and code we had for *UBOs* and specialize the behavior for *SSBOs* where that was needed, that allows us to build on code paths that are already working well and reuse most of the code.

That path, however, had some issues that bit me a bit further down the road. When it comes to representing these operations in the IR, my first idea was to follow the trail of *UBO* loads as well, which are represented as *ir_expression* nodes. There is a fundamental difference between the two though: *UBO* loads are constant operations because uniform buffers are read-only. This means that a *UBO* load operation with the same

parameters will always return the same value. This has implications related to certain optimization passes that work based on the assumption that other *ir_expression* operations share this feature. *SSBO* loads are not like this: since the shader storage buffer is read-write, two identical *SSBO* load operations in the same shader may not return the same result if the underlying buffer storage has been altered in between by *SSBO* write operations within the same or other threads. This forced me to alter a number of optimization passes in Mesa to deal with this situation (mostly disabling them for the cases of *SSBO* loads and stores).

The situation was worse with *SSBO* stores. These just did not fit into *ir_expression* nodes: they did not return a value and had side-effects (memory writes) so we had to come up with a different way to represent them. My initial implementation created a new IR node for these, *ir_ssbo_store*. That worked well enough, but it left us with an implementation of loads and stores that was a bit inconsistent since both operations used very different IR constructs.

These issues were made clear during the review process, where it was suggested that we used *GLSL IR intrinsics* to represent load and store operations instead. This has the benefit that we can make the implementation more consistent, having both loads and stores represented with the same IR construct and follow a similar treatment in both the GLSL compiler and the i965 backend. It would also remove the need to disable or alter certain optimization passes to be *SSBO* friendly.

Read/Write coherence

One of the issues we detected early in development was that our reads and writes did not seem to work very well together: some times a read after a write would fail to see the last value written to a buffer variable. The

problem here also spawned from following the implementation trail of the *UBO* path. In the Intel hardware, there are various interfaces to access memory, like the Sampling Engine and the Data Port. The former is a read-only interface and is used, for example, for texture and *UBO* reads. The Data Port allows for read-write access. Although both interfaces give access to the same memory region, there is something to consider here: if you mix reads through the *Sampling Engine* and writes through the *Data Port* you can run into cache coherence issues, this is because the caches in use by the Sampling Engine and the Data Port functions are different. Initially, we implemented *SSBO* load operations like *UBO* loads, so we used the *Sampling Engine*, and ended up running into this problem. The solution, of course, was to rewrite *SSBO* loads to go though the *Data Port* as well.

Parallel reads and writes

GPUs are highly parallel hardware and this has some implications for driver developers. Take a sentence like this in a fragment shader program:

```
1 | float cx = 1.0;
```

This is a simple assignment of the value *1.0* to variable *cx* that is supposed to happen for each fragment produced. In Intel hardware running in *SIMD16* mode, we process 16 fragments simultaneously in the same GPU thread, this means that this instruction is actually 16 elements wide. That is, we are doing 16 assignments of the value *1.0* simultaneously, each one is stored at a different offset into the GPU register used to hold the value of *cx*.

If *cx* was a *buffer variable* in a *SSBO*, it would also mean that the assignment above should translate to 16 memory writes to the same offset into the buffer. That may seem a bit absurd: why would we want to write

16 times if we are always assigning the same value? Well, because things can get more complex, like this:

```
float cx = gl_FragCoord.x;
```

Now we are no longer assigning the same value for all fragments, each of the 16 values assigned with this instruction could be different. If cx was a buffer variable inside a *SSBO*, then we could be potentially writing 16 different values to it. It is still a bit silly, since only one of the values (the one we write last), would prevail.

Okay, but what if we do something like this?:

```
int index = int(mod(gl_FragCoord.x, 8));
cx[index] = 1;
```

Now, depending on the value we are reading for each fragment, we are writing to a separate offset into the *SSBO*. We still have a single assignment in the GLSL program, but that translates to 16 different writes, and in this case the order may not be relevant, but we want all of them to happen to achieve correct behavior.

The bottom line is that when we implement *SSBO* load and store operations, we need to understand the parallel environment in which we are running and work with test scenarios that allow us to verify correct behavior in these situations. For example, if we only test scenarios with assignments that give the same value to all the fragments/vertices involved in the parallel instructions (i.e. assignments of values that do not depend on properties of the current fragment or vertex), we could easily overlook fundamental defects in the implementation.

Dealing with helper invocations

From Section 7.1 of the GLSL spec version 4.5:

> *"Fragment shader helper invocations execute the same shader code as non-helper invocations, but will not have side effects that modify the framebuffer or other shader-accessible memory."*

To understand what this means I have to introduce the concept of helper invocations: certain operations in the fragment shader need to evaluate derivatives (explicitly or implicitly) and for that to work well we need to make sure that we compute values for adjacent fragments that may not be inside the primitive that we are rendering. The fragment shader executions for these added fragments are called *helper invocations*, meaning that they are only needed to help in computations for other fragments that are part of the primitive we are rendering.

How does this affect *SSBOs*? Because helper invocations are not part of the primitive, they cannot have side-effects, after they had served their purpose it should be as if they had never been produced, so in the case of *SSBOs* we have to be careful not to do memory writes for helper fragments. Notice also, that in a *SIMD16* execution, we can have both proper and helper fragments mixed in the group of 16 fragments we are handling in parallel.

Of course, the hardware knows if a fragment is part of a helper invocation or not and it tells us about this through a *pixel mask register* that is delivered with all executions of a fragment shader thread, this register has a bitmask stating which pixels are proper and which are helper. The Intel hardware also provides developers with various kinds of messages that we can use, via the Data Port interface, to write to memory, however, the tricky thing is that not all of them incorporate pixel mask information, so for

use cases where you need to disable writes from helper fragments you need to be careful with the write message you use and select one that accepts this sort of information.

Vector alignments

Another interesting thing we had to deal with are address alignments. *UBOs* work with layout *std140*. In this setup, elements in the *UBO* definition are aligned to 16-byte boundaries (the size of a vec4). It turns out that GPUs can usually optimize reads and writes to multiples of 16 bytes, so this makes sense, however, as I explained in my previous post, *SSBOs* also introduce a packed layout mode known as *std430*.

Intel hardware provides a number of messages that we can use through the *Data Port* interface to write to memory. Each message has different characteristics that makes it more suitable for certain scenarios, like the pixel mask I discussed before. For example, some of these messages have the capacity to write data in chunks of 16-bytes (that is, they write vec4 elements, or *OWORDS* in the language of the technical docs). One could think that these messages are great when you work with vector data types, however, they also introduce the problem of dealing with partial writes: what happens when you only write to an element of a vector? or to a buffer variable that is smaller than the size of a vector? what if you write columns in a row_major matrix? etc

In these scenarios, using these messages introduces the need to mask the writes because you need to disable the channels in the vec4 element that you don’t want to write. Of course, the hardware provides means to do this, we only need to set the writemask of the destination register of the message instruction to select the right channels. Consider this example:

```
1 struct TB {
```

```
    float a, b, c, d;
};

layout(std140, binding=0) buffer Fragments {
   TB s[3];
   int index;
};

void main()
{
   s[0].d = -1.0;
}
```

In this case, we could use a 16-byte write message that takes 0 as offset (i.e writes at the beginning of the buffer, where *s[0]* is stored) and then set the writemask on that instruction to *WRITEMASK_W* so that only the fourth data element is actually written, this way we only write one data element of 4 bytes (*-1*) at offset 12 bytes (*s[0].d*). Easy, right? However, how do we know, in general, the writemask that we need to use? In *std140* layout mode this is easy: since each element in the *SSBO* is aligned to a 16-byte boundary, we simply need to take the byte offset at which we are writing, divide it by 16 (to convert it to units of vec4) and the modulo of that operation is the byte offset into the chunk of 16-bytes that we are writing into, then we only have to divide that by 4 to get the component slot we need to write to (a number between 0 and 3).

However, there is a restriction: we can only set the writemask of a register at compile/link time, so what happens when we have something like this?:

```
s[i].d = -1.0;
```

The problem with this is that we cannot evaluate the value of *i* at compile/link time, which inevitably makes our solution invalid for this. In other words, if we cannot evaluate the actual value of the offset at which we are writing at compile/link time, we cannot use the writemask to select

the channels we want to use when we don’t want to write a vec4 worth of data and we have to use a different type of message.

That said, in the case of *std140* layout mode, since each data element in the *SSBO* is aligned to a 16-byte boundary you may realize that the actual value of *i* is irrelevant for the purpose of the modulo operation discussed above and we can still manage to make things work by completely ignoring it for the purpose of computing the writemask, but in *std430* that trick won’t work at all, and even in *std140* we would still have *row_major matrix* writes to deal with.

Also, we may need to tweak the message depending on whether we are running on the vertex shader or the fragment shader because not all message types have appropriate *SIMD* modes (*SIMD4x2, SIMD8, SIMD16,* etc) for both, or because different hardware generations may not have all the message types or support all the *SIMD* modes we need need, etc

The point of this is that selecting the right message to use can be tricky, there are multiple things and corner cases to consider and you do not want to end up with an implementation that requires using many different messages depending on various circumstances because of the increasing complexity that it would add to the implementation and maintenance of the code.

Closing notes

This post did not cover all the intricacies of the implementation of *ARB_shader_storage_buffer_object*, I did not discuss things like the optional unsized array or the compiler details of *std430* for example, but, hopefully, I managed to give an idea of the kind of problems one would

have to deal with when coding driver support for this or other similar features.

Story and status of ARB_gpu_shader_fp64 on Intel GPUs

JULY 13, 2016

In case you haven't heard yet, with the recently announced **Mesa 12.0** release, *Intel gen8+ GPUs* expose **OpenGL 4.3**, which is quite a big leap from the previous *OpenGL 3.3*!

The Mesa i965 Intel driver now exposes OpenGL 4.3 on Broadwell and later!

Although this might surprise some, the truth is that even if the *i965* driver only exposed *OpenGL 3.3* it had been exposing many of the *OpenGL 4.x* extensions for quite some time, however, there was one *OpenGL 4.0* extension in particular that was still missing and preventing the driver from exposing a higher version: **ARB_gpu_shader_fp64** (*fp64* for short). There was a good reason for this: it is a very large feature that has been in the works by *Intel* first and *Igalia* later for quite some time. We first started to work on this as far back as November 2015 and by that time *Intel* had already been working on it for months.

I won't cover here what made this such a large effort because there would be a lot of stuff to cover and I don't feel like spending weeks writing a series of posts on the subject :). Hopefully I will get a chance to talk about all that at *XDC* in September, so instead I'll focus on explaining why we only have this working in *gen8+* at the moment and the status of *gen7* hardware.

The plan for *ARB_gpu_shader_fp64* was always to focus on *gen8+* hardware (*Broadwell* and later) first because it has better support for the feature. I must add that it also has fewer hardware bugs too, although we only found out about that later ;). So the plan was to do *gen8+* and then extend the implementation to cover the quirks required by *gen7* hardware (*IvyBridge, Haswell, ValleyView*).

At this point I should explain that *Intel GPUs* have two code generation backends: scalar and vector. The main difference between both backends is that the vector backend (also known as *align16*) operates on vectors (surprise, right?) and has native support for things like swizzles and writemasks, while the scalar backend (known as *align1*) operates on scalars, which means that, for example, a *vec4 GLSL* operation running is broken up into 4 separate instructions, each one operating on a single

component. You might think that this makes the scalar backend slower, but that would not be accurate. In fact it is usually faster because it allows the *GPU* to exploit *SIMD* better than the vector backend.

The thing is that different hardware generations use one backend or the other for different shader stages. For example, *gen8+* used to run *Vertex*, *Fragment* and *Compute* shaders through the scalar backend and *Geometry* and *Tessellation* shaders via the vector backend, whereas *Haswell* and *IvyBridge* use the vector backend also for *Vertex* shaders.

Because you can use 64-bit floating point in any shader stage, the original plan was to implement *fp64* support on both backends. Implementing fp64 requires a lot of changes throughout the driver compiler backends, which makes the task anything but trivial, but the vector backend is particularly difficult to implement because the hardware only supports 32-bit swizzles. This restriction means that a hardware swizzle such as *XYZW* only selects components *XY* in a *dvecN* and therefore, there is no direct mechanism to access components *ZW*. As a consequence, dealing with anything bigger than a *dvec2* requires more creative solutions, which then need to face some other hardware limitations and bugs, etc, which eventually makes the vector backend require a significantly larger development effort than the scalar backend.

Thankfully, *gen8+* hardware supports scalar *Geometry* and *Tessellation* shaders and *Intel*'s Kenneth Graunke had been working on enabling that for a while. When we realized that the vector *fp64* backend was going to require much more effort than what we had initially thought, he gave a final push to the full scalar *gen8+* implementation, which in turn allowed us to have a full *fp64* implementation for this hardware and expose *OpenGL 4.0*, and soon after, *OpenGL 4.3*.

That does not mean that we don't care about *gen7* though. As I said above, the plan has always been to bring *fp64* and *OpenGL4* to *gen7* as well. In fact, we have been hard at work on that since even before we started sending the *gen8*+ implementation for review and we have made some good progress.

Besides addressing the quirks of *fp64* for *IvyBridge* and *Haswell* (yes, they have different implementation requirements) we also need to implement the full *fp64* vector backend support from scratch, which as I said, is not a trivial undertaking. Because *Haswell* seems to require fewer changes we have started with that and I am happy to report that we have a working version already. In fact, we have already sent a small set of patches for review that implement *Haswell*'s requirements for the scalar backend and as I write this I am cleaning-up an initial implementation of the vector backend in preparation for review (currently at about 100 patches, but I hope to trim it down a bit before we start the review process). *IvyBridge* and *ValleView* will come next.

The initial implementation for the vector backend has room for improvement since the focus was on getting it working first so we can expose *OpenGL4* in *gen7* as soon as possible. The good thing is that it is more or less clear how we can improve the implementation going forward (you can see an excellent post by Curro on that topic here).

You might also be wondering about *OpenGL 4.1's ARB_vertex_attrib_64bit*, after all, that kind of goes hand in hand with *ARB_gpu_shader_fp64* and we implemented the extension for *gen8*+ too. There is good news here too, as my colleague Juan Suárez has already implemented this for *Haswell* and I would expect it to mostly work on *IvyBridge* as is or with minor tweaks. With that we should be able to expose at least *OpenGL 4.2* on all *gen7* hardware once we are done.

So far, implementing *ARB_gpu_shader_fp64* has been quite the ride and I have learned a lot of interesting stuff about how the *i965* driver and *Intel GPUs* operate in the process. Hopefully, I'll get to talk about all this in more detail at *XDC* later this year. If you are planning to attend and you are interested in discussing this or other Mesa stuff with me, please find me there, I'll be looking forward to it.

Finally, I'd like to thank both Intel and Igalia for supporting my work on Mesa and i965 all this time, my igalian friends Samuel Iglesias, who has been hard at work with me on the *fp64* implementation all this time, Juan Suárez and Andrés Gómez, who have done a lot of work to improve the *fp64* test suite in *Piglit* and all the friends at *Intel* who have been helping us in the process, very especially Connor Abbot, Francisco Jerez, Jason Ekstrand and Kenneth Graunke.

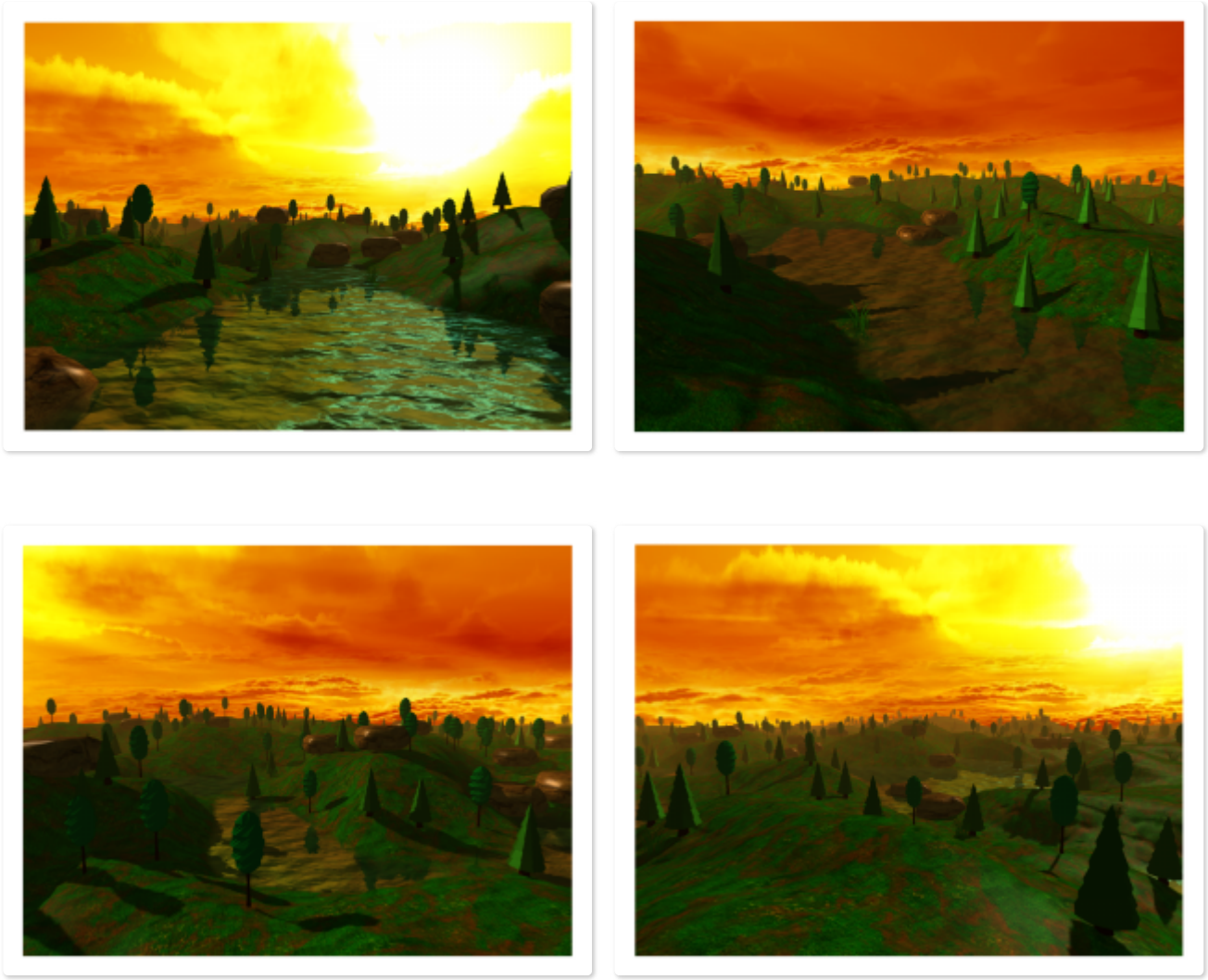
OpenGL terrain renderer

AUGUST 29, 2016

Lately I have been working on a simple terrain *OpenGL* renderer demo, mostly to have a playground where I could try some techniques like shadow mapping and water rendering in a scenario with a non trivial amount of geometry, and I thought it would be interesting to write a bit about it.

But first, here is a video of the demo running on my old *Intel IvyBridge GPU*:

And some screenshots too:



Note that I did not create any of the textures or 3D models featured in the video.

With that out of the way, let's dig into some of the technical aspects:

The terrain is built as a *251×251* grid of vertices elevated with a heightmap texture, so it contains *63,000 vertices* and *125,000 triangles*. It uses a single *512×512* texture to color the surface.

The water is rendered in 3 passes: refraction, reflection and the final rendering. Distortion is done via a *dudv* map and it also uses a *normal* map for lighting. From a geometry perspective it is also implemented as a grid of vertices with *750 triangles*.

I wrote a simple *OBJ file parser* so I could load some basic 3D models for the trees, the rock and the plant models. The parser is limited, but sufficient to load vertex data and simple materials. This demo features 4 models with these specs:

- Tree A: *280 triangles, 2 materials*.
- Tree B: *380 triangles, 2 materials*.
- Rock: *192 triangles, 1 material (textured)*
- Grass: 8*96 triangles (yes, really!), 1 material*.

The scene renders 2*00 instances of Tree A* another *200 instances of Tree B*, *50 instances of Rock* and *150 instances of Grass*, so *600 objects in total*.

Object locations in the terrain are randomized at start-up, but the demo prevents trees and grass to be under water (except for maybe their base section only) because it would very weird otherwise :), rocks can be fully submerged though.

Rendered objects fade in and out smoothly via *alpha blending* (so there is no *pop-in/pop-out* effect as they reach clipping planes). This cannot be observed in the video because it uses a static camera but the demo supports moving the camera around in real-time using the keyboard.

Lighting is implemented using the traditional *Phong reflection model* with a single *directional light*.

Shadows are implemented using a *4096×4096 shadow map* and *Percentage Closer Filter* with a 3*x3 kernel*, which, I read is (or was?) a very common technique for shadow rendering, at least in the times of the *PS3* and *Xbox 360*.

The demo features dynamic directional lighting (that is, the sun light changes position every frame), which is rather taxing. The demo also supports static lighting, which is significantly less demanding.

There is also a slight haze that builds up progressively with the distance from the camera. This can be seen slightly in the video, but it is more obvious in some of the screenshots above.

The demo in the video was also configured to use *4-sample multisampling*.

As for the rendering pipeline, it mostly has 4 stages:

- Shadow map.
- Water refraction.
- Water reflection.
- Final scene rendering.

A few notes on performance as well: the implementation supports a number of configurable parameters that affect the framerate: resolution, shadow rendering quality, clipping distances, multi-sampling, some aspects of the water rendering, N-buffering of dynamic VBO data, etc.

The video I show above runs at locked *60fps* at *800×600* but it uses relatively high quality shadows and dynamic lighting, which are very expensive. Lowering some of these settings (very specially turning off dynamic lighting, multisampling and shadow quality) yields framerates around *110fps-200fps*. With these settings it can also do *fullscreen 1600×900* with an unlocked framerate that varies in the range of *80fps-170fps*.

That’s all in the *IvyBridge GPU*. I also tested this on an *Intel Haswell GPU* for significantly better results: *160fps-400fps* with the “low” settings at *800×600* and roughly *80fps-200fps* with the same settings used in the video.

So that’s it for today, I had a lot of fun coding this and I hope the post was interesting to some of you. If time permits I intend to write follow-up posts that go deeper into how I implemented the various elements of the demo and I’ll probably also write some more posts about the optimization process I followed. If you are interested in any of that, stay tuned for more.

Source code for the OpenGL terrain renderer demo

OCTOBER 5, 2016

I have been quite busy with various things in the last few weeks, but I have finally found some time to clean up and upload the code of the OpenGL terrain render demo to Github.

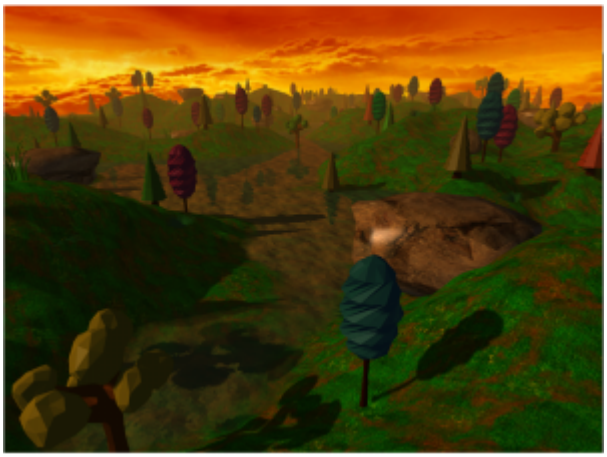
Since this was intended as a programming exercise I have not tried to be very elegant or correct during the implementation, so expect things like error handling to be a bit rough around the edges, but otherwise I think the code should be easy enough to follow.

Notice that I have only tested this on Intel GPUs. I know it works on NVIDIA too (thanks to Samuel and Chema for testing this) but there are a couple of rendering artifacts there, specifically at the edges of the skybox and some “pillars” showing up in the distance some times, probably because I am rendering one to many “rows” of the terrain and I end up rendering garbage. I may fix these some day.

The code I uploaded to the repository includes a few new features too:

- Model variants, which are basically color variations of the same model
- A couple of additional models (a new tree and plant) and a different rock type
- Collision detection, which makes navigating the terrain more pleasent

Here is a new screenshot:



In future posts I will talk a bit about some of the implementation details, so having the source code around will be useful. Enjoy!

OpenGL terrain renderer: rendering the terrain mesh

OCTOBER 13, 2016

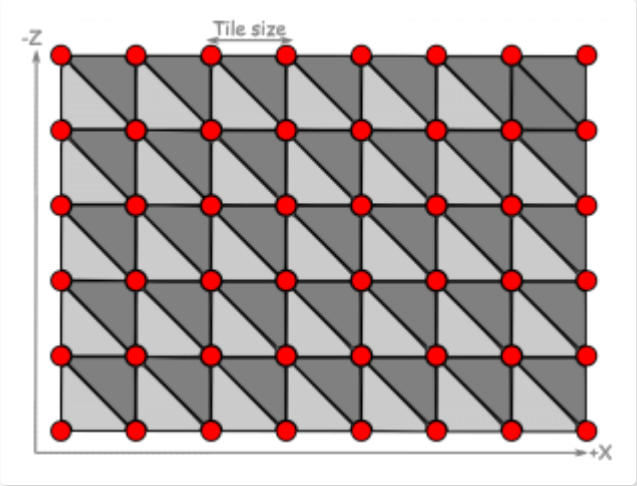
In this post I'll discuss how I setup and render terrain mesh in the OpenGL terrain rendering demo. Most of the relevant code for this is in the ter-terrain.cpp file.

Setting up a grid of vertices

Unless you know how to use a 3D modeling program properly, a reasonable way to create a decent mesh for a terrain consists in using a grid of vertices and elevate them according to a height map image. In order to create the grid we only need to decide how many rows and columns we want. This, in the end, determines the number of polygons and the resolution of the terrain.

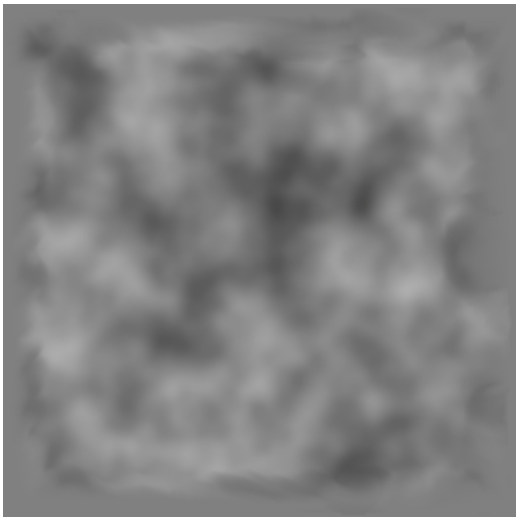
We need to map these vertices to world coordinates too. We do that by defining a tile size, which is the distance between consecutive vertices in world units. Larger tile sizes increase the size of the terrain but lower the resolution by creating larger polygons.

The image below shows an 8×6 grid that defines 35 tiles. Each tile is rendered using 2 triangles:



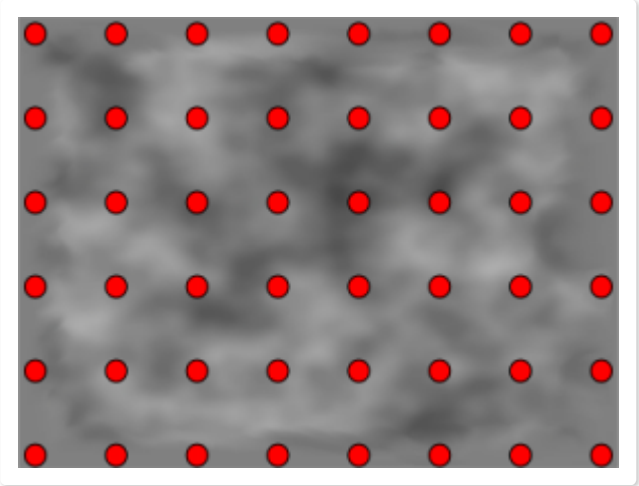
8×6 terrain grid

The next step is to elevate these vertices so we don’t end up with a boring flat surface. We do this by sampling the height map image for each vertex in the grid. A height map is a gray scale image where the values of the pixels represent altitudes at different positions. The closer the color is to white, the more elevated it is.



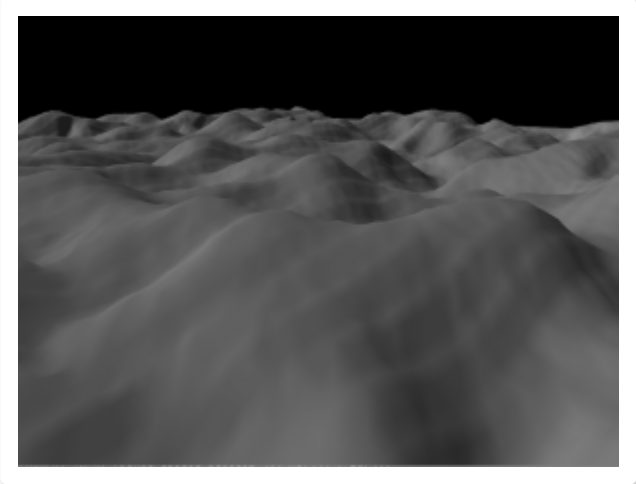
Heightmap image

Adding more vertices to the grid increases the number of sampling points from the height map and reduces the sampling distances, leading to a smoother and more precise representation of the height map in the resulting terrain.

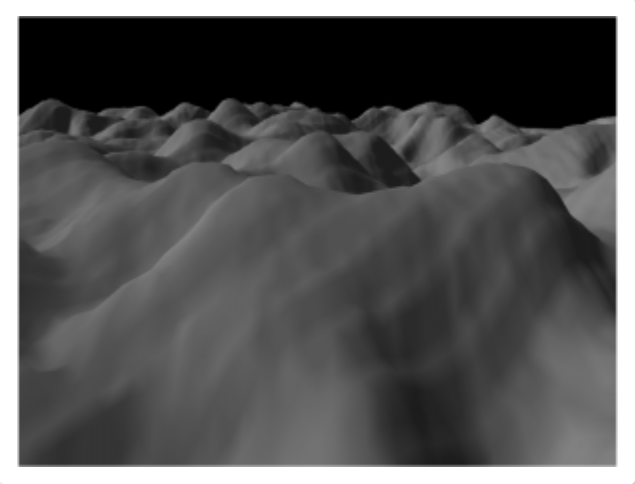


Sampling the heightmap to compute vertex heights

Of course, we still need to map the height map samples (gray scale colors) to altitudes in world units. In the demo I do this by normalizing the color values to [-1,+1] and then applying a scale factor to compute the altitude values in world space. By playing with the scaling factor we can make our terrain look more or less abrupt.



Altitude scale=6.0



Altitude scale=12.0

For reference, the height map sampling is implemented in ter_terrain_set_heights_from_texture().

Creating the mesh

At this point we know the full position (x, y, z) in world coordinates of all the vertices in our grid. The next step is to build the actual triangle mesh that we will use to render the terrain and the normal vectors for each triangle. This process is described below and is implemented in the ter_terrain_build_mesh() function.

Computing normals

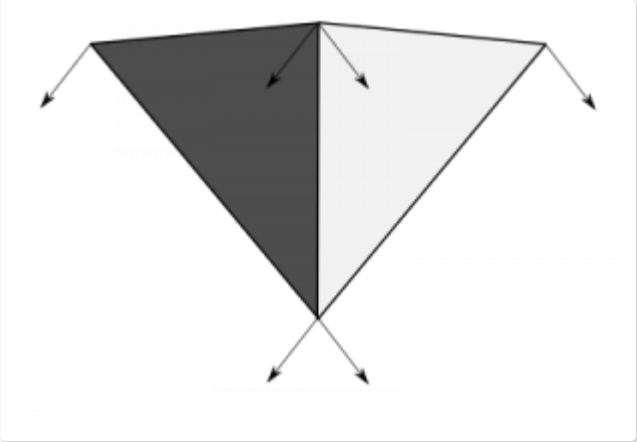
In order to get nice lighting on our terrain we need to compute normals for each vertex in the mesh. A simple way to achieve this is would be to compute the normal for each face (triangle) and use that normal for each vertex in the triangle. This works, but it has 3 problems:

1. Every vertex of each triangle has the same exact normal, which leads to a rather flat result.

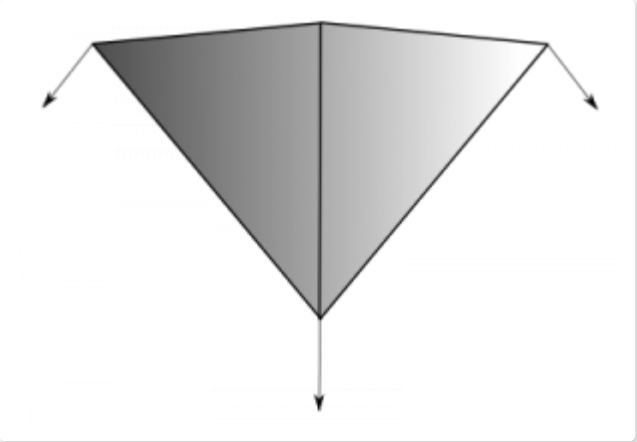
2. Adjacent triangles with different orientations showcase abrupt changes in the normal value, leading to significantly different lighting across the surfaces that highlight the individual triangles in the mesh.

3. Because each vertex in the mesh can have a different normal value for each triangle it participates in, we need to replicate the vertices when we render, which is not optimal.

Alternatively, we can compute the normals for each vertex considering the heights of its neighboring vertices. This solves all the problems mentioned above and leads to much better results thanks to the interpolation of the normal vectors across the triangles, which leads to smooth lighting reflection transitions:



Flat normals



Smooth normals

The implementation for this is in the function calculate_normal(), which takes the column and row indices of the vertex in the grid and calculates the Y coordinate by sampling the heights of the 4 nearby vertices in the grid.

Preparing the draw call

Now that we know the positions of all the vertices and their normal vectors we have all the information that we need to render the terrain. We still have to decide how exactly we want to render all the polygons.

The simplest way to render the terrain using a single draw call is to setup a vertex buffer with data for each triangle in the mesh (including position and normal information) and use *GL_TRIANGLES* for the primitive of the draw call. This, however, is not the best option from the point of view of performance.

Because the terrain will typically contain a large number of vertices and most of them participate in multiple triangles, we end up uploading a large amount of vertex data to the GPU and processing a lot of vertices in the

draw call. The result is large memory requirements and suboptimal performance.

For reference, the terrain I used in the demo from my original post used a 251×251 grid. This grid represents 250×250 tiles, each one rendered as two triangles (6 vertices/tile), so we end up with 250x250x6=375,000 vertices. For each of these vertices we need to upload 24 bytes of vertex data with the position and normal, so we end up with a GPU buffer that is almost 9MB large.

One obvious way to reduce this is to render the terrain using triangle strips. The problem with this is that in theory, we can't render the terrain with just one strip, we would need one strip (and so, one draw call) per tile column or one strip per tile row. Fortunately, we can use degenerate triangles to link separate strips for each column into a single draw call. With this we trim down the number of vertices to 126,000 and the size of the buffer to a bit below 3 MB. This alone produced a 15%-20% performance increase in the demo.

We can do better though. A lot of the vertices in the terrain mesh participate in various triangles across the large triangle strip in the draw call, so we can reduce memory requirements by using an index buffer to render the strip. If we do this, we trim things down to 63,000 vertices and ~1.5MB. This added another 4%-5% performance bonus over the original implementation.

Clipping

So far we have been rendering the full mesh of the terrain in each frame and we do this by uploading the vertex data to the GPU just once (for

example in the first frame). However, depending on where the camera is located and where it is looking at, just a fraction of the terrain may be visible.

Although the GPU will discard all the geometry and fragments that fall outside the viewport, it still has to process each vertex in the vertex shader stage before it can clip non-visible triangles. Because the number of triangles in the terrain is large, this is suboptimal and to address this we want to do CPU-side clipping before we render.

Doing CPU-side clippig comes with some additional complexities though: it requires that we compute the visible region of the terrain and upload new vertex data to the GPU in each frame preventin GPU stalls.

In the demo, we implement the clipping by computing a quad sub-region of the terrain that includes the visible area that we need to render. Once we know the sub-region that we want to render, we compute the new indices of the vertices that participate in the region so we can render it using a single triangle strip. Finally, we upload the new index data to the index buffer for use in the follow-up draw call.

Avoiding GPU stalls

Although all the above is correct, it actually leads, as described, to much worse performance in general. The reason for this is that our uploads of vertex data in each frame lead to frequent GPU stalls. This happens in two scenarios:

1. In the same frame, because we need to upload different vertex data for the rendering of the terrain for the shadow map and the scene (the shadow map renders the terrain from the point of view of the light, so the

visible region of the terrain is different). This creates stalls because the rendering of the terrain for the shadow map might not have completed before we attempt to upload new data to the index buffer in order to render the terrain for the scene.

2. Between different frames. Because the GPU might not be completely done rendering the previous frame (and thus, stills needs the index buffer data available) before we start preparing the next frame and attempt to upload new terrain index data for it.

In the case of the Intel Mesa driver, these GPU stalls can be easily identified by using the environment variable INTEL_DEBUG=perf. When using this, the driver will detect these situations and produce warnings informing about the stalls, the buffers affected and the regions of the buffers that generate the stall, such as:

```
Stalling on glBufferSubData(0, 503992) (492kb)
to a busy (0-1007984)
buffer object.  Use glMapBufferRange() to avoid
this.
```

The solution to this problem that I implemented (other than trying to put as much work as possible between read/write accesses to the index buffer) comes in two forms:

1. Circular buffers

In this case, we allocate a larger buffer than we need so that each subsequent upload of new index data happens in a separate sub-region of the allocated buffer. I set up the demo so that each circular buffer is large enough to hold the index data required for all updates of the index buffer happening in each frame (the shadow map and the scene).

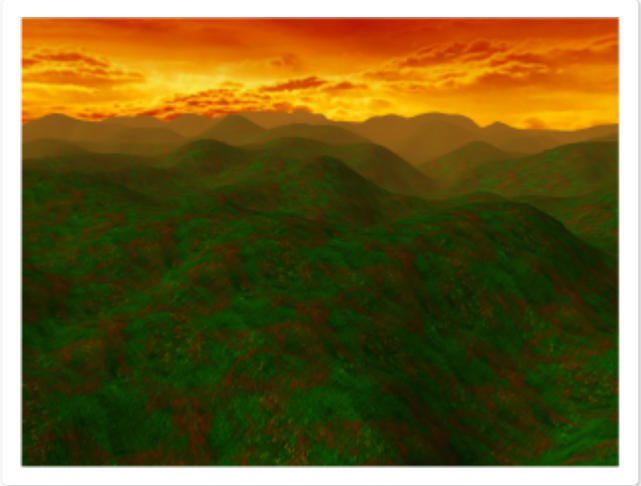
2. Multi-buffering

We allocate more than one circular buffer. When we don’t have enough free space at the end of the current buffer to upload the new index buffer data, we upload it to a different circular buffer instead. When we run out of buffers we circle back to the first one (which at this point will hopefully be free to be re-used again).

So why not just use a single, very large circular buffer? Mostly because there are limits to the size of the buffers that the GPU may be able to handle correctly (or efficiently). Also, why not having many smaller independent buffers instead of circular buffers? That would work just fine, but using fewer, larger buffers reduces the number of objects we need to bind/unbind and is better to prevent memory fragmentation, so that’s a plus.

Final touches

We are almost done, at this point we only need to add a texture to the terrain surface, add some slight fog effect for distant pixels to create a more realistic look, add a skybox (it is important to choose the color of the fog so it matches the color of the sky!) and tweak the lighting parameters to get a nice result:



Final rendering

I hope to cover some of these aspects in future posts, so stay tuned for more!

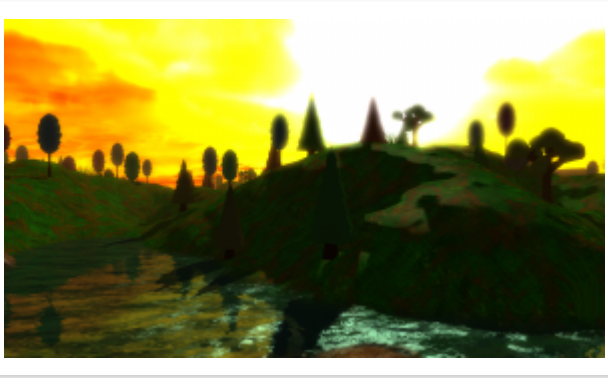
OpenGL terrain renderer update: Bloom and Cascaded Shadow Maps

OCTOBER 20, 2016

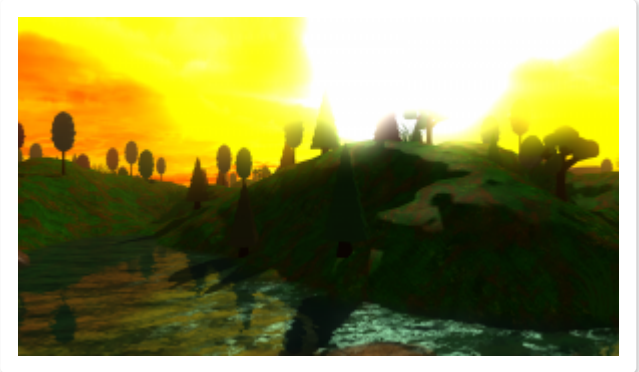
Bloom Filter

The bloom filter makes bright objects glow and bleed through other objects positioned in between them and the camera. It is a common post-processing effect used all the time in video games and animated movies. The demo supports a couple of configuration options that control the intensity and behavior of the filter, here are some screenshots with different settings:

Bloom filter Off



Bloom filter On, default settings



Bloom filter On, intensity increased

I particularly like the glow effect that this brings to the specular reflections on the water surface, although to really appreciate that you need to run the demo and see it in motion.

Cascaded Shadow Maps

I should really write a post about basic shadow mapping before going into the details of Cascaded Shadow Maps, so for now I'll just focus on the problem they try to solve.

One of the problems with shadow mapping is rendering high resolution shadows, specially for shadows that are rendered close to the camera. Generally, basic shadow mapping provides two ways in which we can improve the resolution of the shadows we render:

1. Increase the resolution of the shadow map textures. This one is obvious but comes at a high performance (and memory) hit.

2. Reduce the distance at which we can render shadows. But this is not ideal of course.

One compromise solution is to notice that, as usual with 3D computer graphics, it is far more important to render nearby objects in high quality than distant ones.

Cascaded Shadow Maps allow us to use different levels of detail for shadows that are rendered at different distances from the camera. Instead of having a single shadow map for all the shadows, we split the viewing frustum into slices and render shadows in each slice to a different shadow map.

There are two immediate benefits of this technique:

1. We have flexibility to define the resolution of the shadow maps for each level of the cascade, allowing us, for example, to increase the resolution of the levels closest to the camera and maybe reduce those that are further away.

2. Each level only records shadows in a slice of the viewing frustum, which increases shadow resolution even if we keep the same texture resolution we used with the original shadow map implementation for each shadow map level.

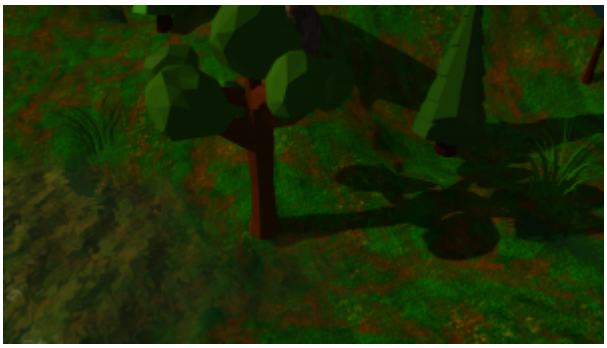
This approach also has some issues:

1. We need to render multiple shadow maps, which can be a serious performance hit depending on the resolutions of the shadow maps involved. This is why we usually lower the resolution of the shadow maps as distance from the camera increases.

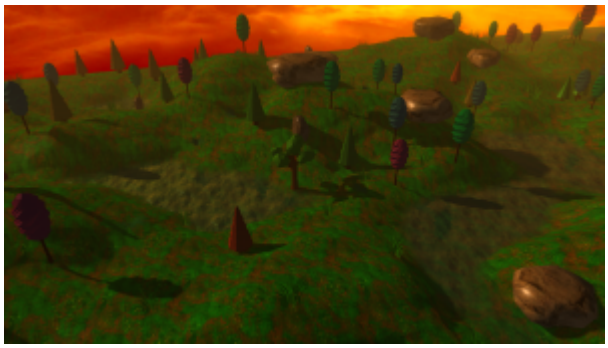
2. As we move closer to or further from shadowed objects we can see the changes in shadow quality pop-in. Of course we can control this by

avoiding drastic quality changes between consecutive levels in the cascade.

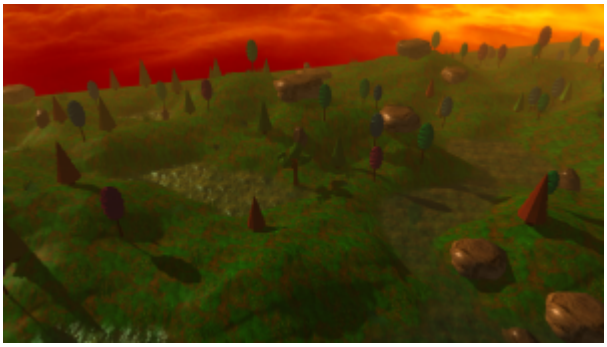
Here is an example that illustrates the second issue (in this case I have lowered the resolution of the 2nd and 3d cascade levels to 50% and 25% respectively so the effect was more obvious). The screenshots show the rendering of the shadows at different distances. We can see how the shadows in the close-up shot are very sharp and as the distance increases they become blurrier due to the use of a lower resolution shadow map:



CSM level 0 (4096×4096)



CSM level 1 (2048×2048)



CSM level 2 (1024×1024)

The demo supports up to 4 shadow map levels although the default configuration is to use 3. The resolution of each level can be configured separately too, in the default configuration I lowered the shadow resolution of the second and third levels to 75% and 50% respectively. If we

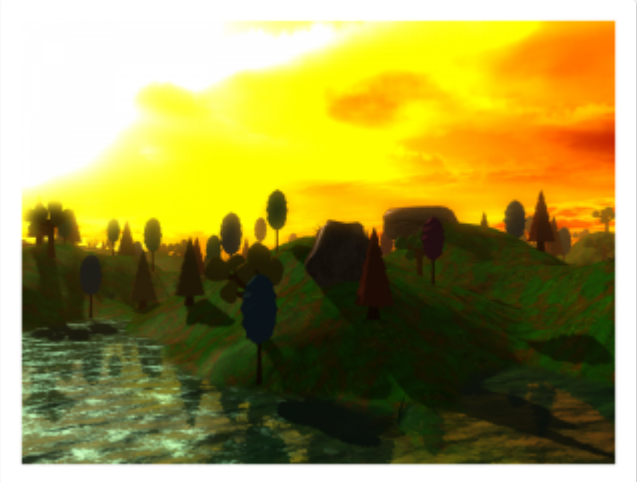
configure the demo to run on a single level (with 100% texture resolution), we are back to the original shadow map implementation, so it is easy to experiment with both techniques.

I intend to cover the details behind shadow mapping and the implementation of the bloom filter in more detail in a future post, so again, stay tuned for more!

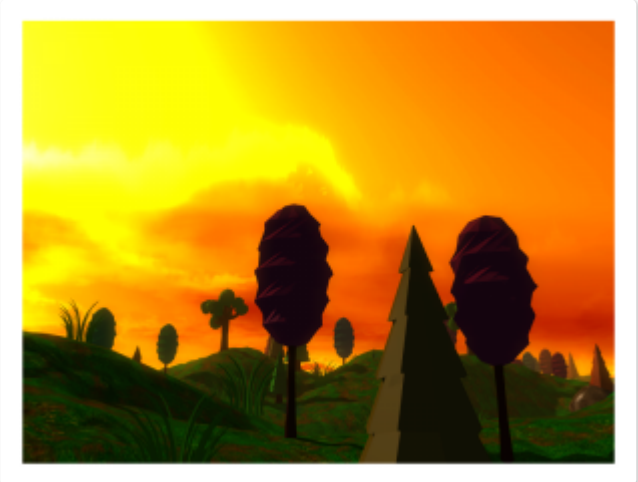
OpenGL terrain renderer update: Motion Blur

NOVEMBER 15, 2016

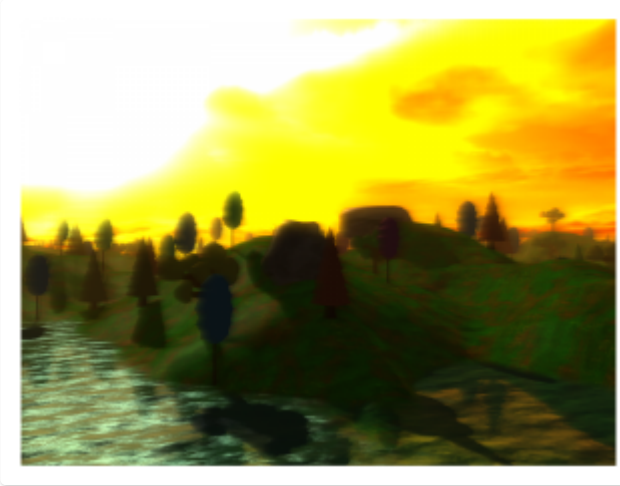
As usual, it can be turned on and off at build-time and there is some configuration available as well to control how the effect works. Here are some screen-shots:



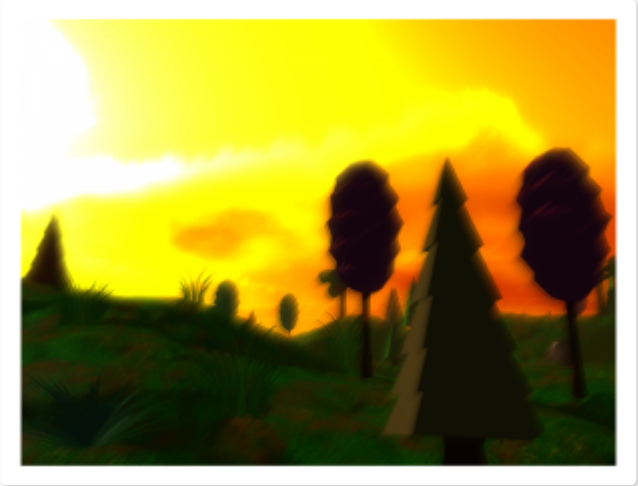
Motion Blur Off



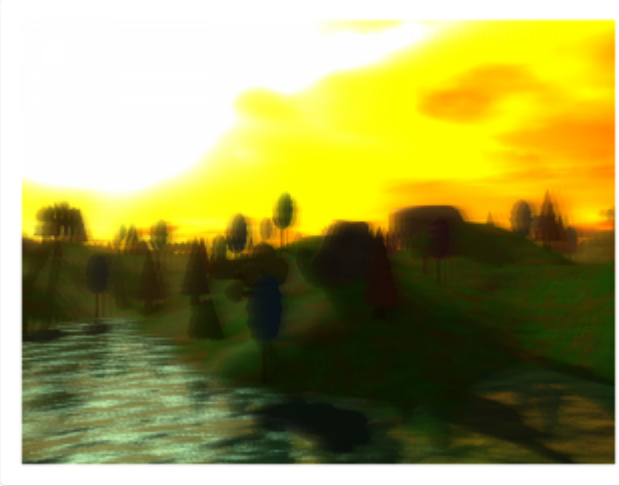
Motion Blur Off



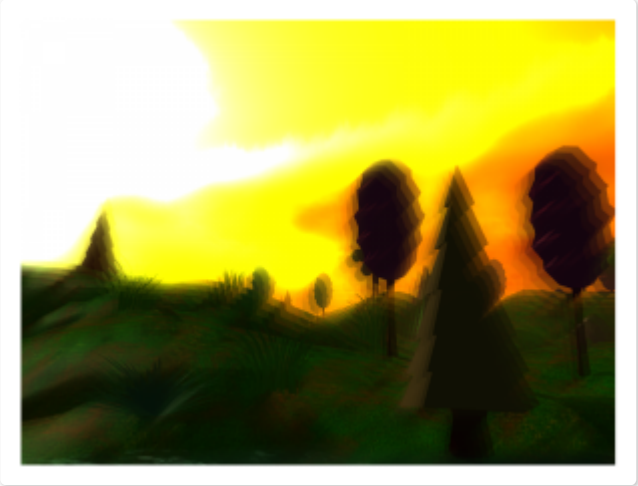
Motion Blur On, intensity 12.5%



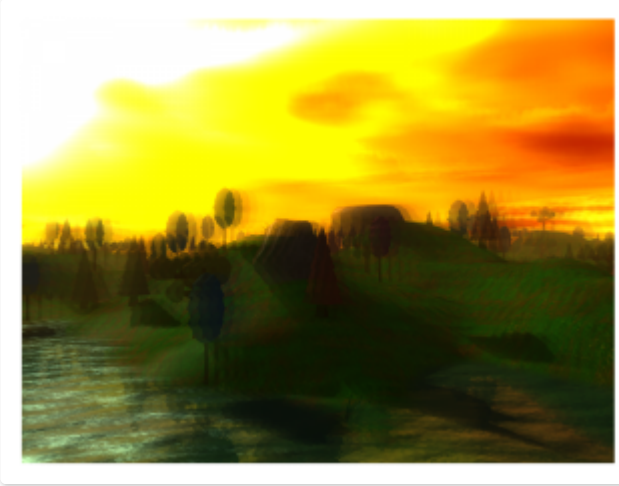
Motion Blur On, intensity 12.5%



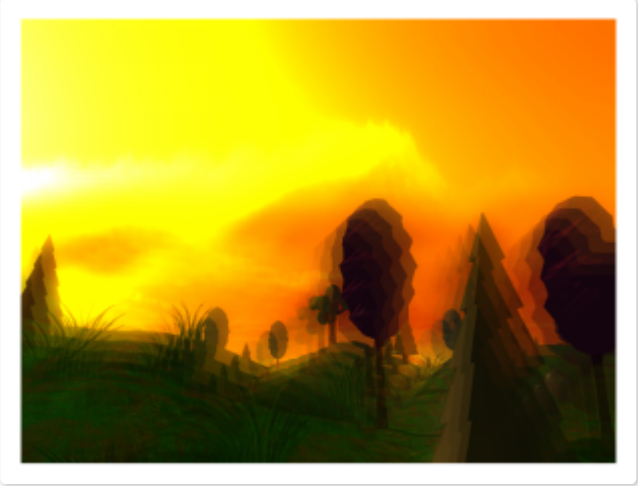
Motion Blur On, intensity 25%



Motion Blur On, intensity 25%



Motion Blur On, intensity 50%



Motion Blur On, intensity 50%

Motion blur is a technique used to make movement feel smoother than it actually is and is targeted at hiding the fact that things don’t move in continuous fashion, but rather, at fixed intervals dictated by the frame rate. For example, a fast moving object in a scene can “leap” many pixels between consecutive frames even if we intend for it to have a smooth animation at a fixed speed. Quick camera movement produces the same effect on pretty much every object on the scene. Our eyes can notice these gaps in the animation, which is not great. Motion blur applies a slight blur to objects across the direction in which they move, which aims at filling the animation gaps produced by our discrete frames, tricking the brain into perceiving a smoother animation as result.

In my demo there are no moving objects other than the sky box or the shadows, which are relatively slow objects anyway, however, camera movement can make objects change screen-space positions quite fast (specially when we rotate the view point) and the motion- blur effect helps us perceive a smoother animation in this case.

I will try to cover the actual implementation in some other post but for now I’ll keep it short and leave it to the images above to showcase what the filter actually does at different configuration settings. Notice that the smoothing effect is something that can only be perceived during motion, so still images are not the best way to showcase the result of the filter from the perspective of the viewer, however, still images are a good way to freeze the animation and see exactly how the filter modifies the original image to achieve the smoothing effect.

GL_ARB_gpu_shader_fp64 / OpenGL 4.0 lands for Intel/Haswell. More gen7 support coming soon!

JANUARY 6, 2017

2017 starts with good news for *Intel Haswell* users: it has been a long time coming, but we have finally landed *GL_ARB_gpu_shader_fp64* for this platform. Thanks to *Matt Turner* for reviewing the huge patch series!

Maybe you are not particularly excited about *GL_ARB_gpu_shader_fp64*, but that does not mean this is not an exciting milestone for you if you have a *Haswell* GPU (or even *IvyBridge*, read below): this extension was the last piece missing to finally bring *Haswell* to expose *OpenGL 4.0*!

If you want to give it a try but you don’t want to build the driver from the latest Mesa sources, don’t worry: the feature freeze for the *Mesa 13.1* release is planned to happen in just a few days and the current plan is to have the release in *early February*, so if things go according to plan you won’t have to wait too long for an official release.

But that is not all, now that we have landed *Fp64* we can also send for review the implementation of *GL_ARB_vertex_attrib_64bit*. This could be

a very exciting milestone, since I believe this is the only thing missing for *Haswell* to have all the extensions required for *OpenGL 4.5*!

You might be wondering about *IvyBridge* too, and 2017 also starts with good news for *IvyBridge* users. Landing *Fp64* for *Haswell* allowed us to send for review the *IvyBridge* patches we had queued up for *GL_ARB_gpu_shader_fp64* which will bring *IvyBridge* up to *OpenGL 4.0*. But again, that is not all, once we land *Fp64* we should also be able to send the patches for *GL_vertex_attrib_64bit* and get *IvyBridge* up to *OpenGL 4.2*, so look forward to this in the near future!

We have been working hard on *Fp64* and *Va64* during a good part of 2016, first for *Broadwell* and later platforms and later for *Haswell* and *IvyBridge*; it has been a lot of work so it is exciting to see all this getting to the last stages and on its way to the hands of the final users.

All this has only been possible thanks to Intel’s sponsoring and the great support and insight that our friends there have provided throughout the development and review processes, so big thanks to all of them and also to the team at Igalia that has been involved in the development with me.

Working with lights and shadows – Part I: Phong reflection model

JULY 6, 2017

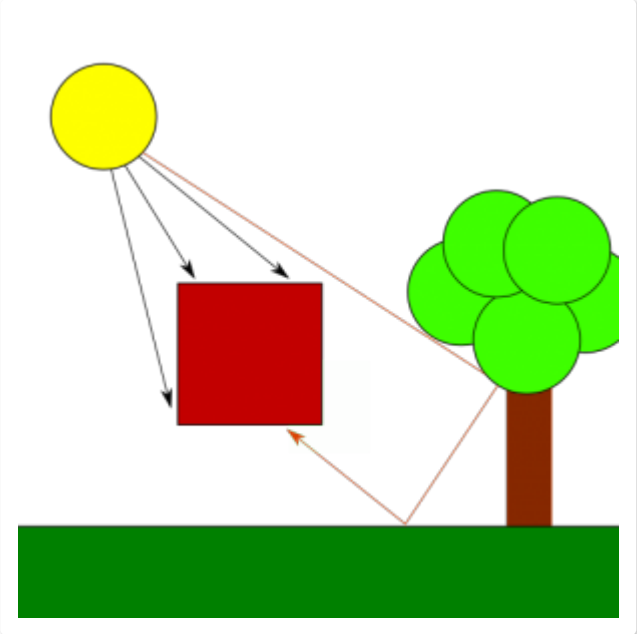
Some time ago I promised to write a bit more about how **shadow mapping** works. It has taken me a while to bring myself to actually deliver on that front, but I have finally decided to put together some posts on this topic, this being the first one. However, before we cover shadow mapping itself we need to cover some **lighting basics** first. After all, without light there can't be shadows, right?

This post will introdcuce the popular **Phong reflection model** as the basis for our lighting model. A lighting model provides a simplified representation of how light works in the natural world that allows us to simulate light in virtual scenes at reasonable computing costs. So let's dive into it:

Light in the natural world

In the real world, the light that reaches an object is a combination of both **direct and indirect light**. Direct light is that which comes straight from a light source, while indirect light is the result of light rays hitting other surfaces in the scene, bouncing off of them and eventually reaching the object as a result, maybe after multiple reflections from other objects.

Because each time a ray of light hits a surface it loses part of its energy, indirect light reflection is less bright than direct light reflection and its color might have been altered. The contrast between surfaces that are directly hit by the light source and surfaces that only receive indirect light is what creates shadows. A shadow isn't but the part of a scene that doesn't receive direct light but might still receive some amount of (less intense) indirect light.



Direct vs Indirect light

Light in the digital world

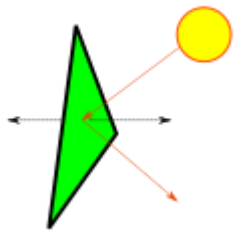
Unfortunately, implementing realistic light behavior like that is too expensive, specially for real-time applications, so instead we use simplifications that can produce similar results with much lower computing requirements. The *Phong reflection model* in particular describes the light reflected from surfaces or emitted by light sources as the combination of 3 components: **diffuse, ambient and specular**. The model also requires information about the direction in which a particular surface is facing,

provided via vectors called **surface normals**. Let's introduce each of these concepts:

Surface normals

When we study the behavior of light, we notice that the direction in which surfaces reflect incoming light affects our perception of the surface. For example, if we lit a shiny surface (such as a piece of metal) using a strong light shource so that incoming light is reflected off the surface in the exact opposite direction in which we are looking at it, we will see a strong reflection in the form of highlights. If we move around so that we look at the same surface from a different angle, then we will see the reflection get dimmer and the highlights will eventually disappear. In order to model this behavior we need to know the direction in which the surfaces we render reflect incoming light. The way to do this is by associating vectors called *normals* with the surfaces we render so that shaders can use that information to produce lighting calculations akin to what we see in the natural world.

Usually, modeling programs can compute normal vectors for us, even model loading libraries can do this work automatically, but some times, for example when we define vertex meshes programatically, we need to define them manually. I won't covere here how to do this in the general, you can see this article from Khronos if you're interested in specific algorithms, but I'll point out something relevant: given a plane, we can compute normal vectors in two opposite directions, one is correct for the front face of the plane/polygon and the other one is correct for the back face, so make sure that if you compute normals manually, you use the correct direction for each face, otherwise you won't be reflecting light in the correct direction and results won't be as you expect.



Light reflected using correct normal vector for the front face of the triangle

In most scenarios, we only render the front faces of the polygons (by enabling back face culling) and thus, we only care about one of the normal vectors (the one for the front face).

Another thing to notice about normal vectors is that they need to be transformed with the model to be correct for transformed models: if we rotate a model we need to rotate the normals too, since the faces they represent are now rotated and thus, their normal directions have rotated too. Scaling also affects normals, specifically if we don't use regular scaling, since in that case the orientation of the surfaces may change and affect the direction of the normal vector. Because normal vectors represent directions, their position in world space is irrelevant, so for the purpose of lighting calculations, a normal vector such as (1, 0, 0) defined for a surface placed at (0, 0, 0) is still valid to represent the same surface at any other position in the world; in other words, we do not need to apply translation transforms to our normal vectors.

In practice, the above means that we want to apply the rotation and scale transforms from our models to their normal vectors, but we can skip the translation transform. The matrix representing these transforms is usually called the *normal matrix*. We can compute the normal matrix from our model matrix by computing the transpose of the inverse of the 3×3 submatrix of the model matrix. Usually, we'd want to compute this matrix in the application and feed it to our vertex shader like we do with our model matrix, but for reference, here is how this can be achieved in the shader

code itself, plus how to use this matrix to transform the original normal vectors:

```
mat3 NormalMatrix =
transpose(inverse(mat3(ModelMatrix)));
vec3 out_normal = normalize(NormalMatrix *
in_normal);
```

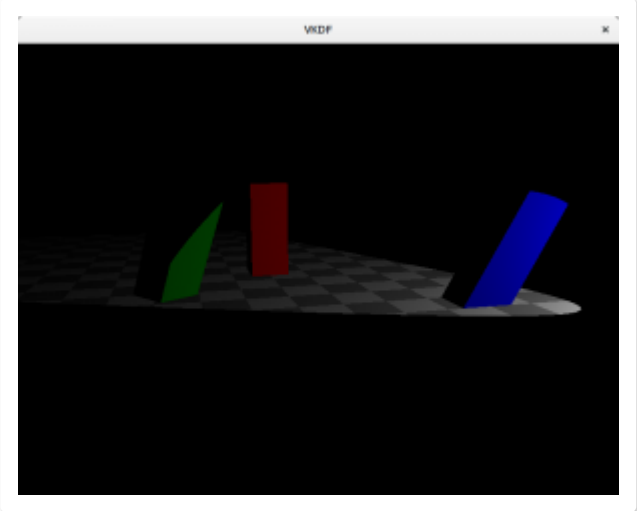
Notice that the code above normalizes the resulting normal before it is fed to the fragment shader. This is important because the rasterizer will compute normals for all fragments in the surface automatically, and for that it will interpolate between the normals for each vertex we emit. For the interpolated normals to be correct, all vertex normals we output in the vertex shader must have the same length, otherwise the larger normals will deviate the direction of the interpolated vectors towards them because their larger size will increase their *weight* in the interpolation computations.

Finally, even if we emit normalized vectors in the vertex shader stage, we should note that the interpolated vectors that arrive to the fragment shader are not guaranteed to be normalized. Think for example of the normal vectors (1, 0, 0) and (0, 1, 0) being assigned to the two vertices in a line primitive. At the half-way point in between these two vertices, the interpolator will compute a normal vector of (0.5, 0.5, 0), which is not unit-sized. This means that in the general case, input normals in the fragment shader will need to be normalized again even if have normalized vertex normals at the vertex shader stage.

Diffuse reflection

The diffuse component represents the reflection produced from direct light. It is important to notice that the intensity of the diffuse reflection is affected by the angle between the light coming from the source and the normal of

the surface that receives the light. This makes a surface looking straight at the light source be the brightest, with reflection intensity dropping as the angle increases:



Diffuse light (spotlight source)

In order to compute the diffuse component for a fragment we need its normal vector (the direction in which the surface is facing), the vector from the fragment's position to the light source, the diffuse component of the light and the diffuse reflection of the fragment's material:

```
vec3 normal = normalize(surface_normal);
vec3 pos_to_light_norm =
normalize(pos_to_light);
float dp_reflection = max(0.0, dot(normal,
pos_to_light_norm));
vec3 diffuse = material.diffuse * light.diffuse
* dp_reflection;
```

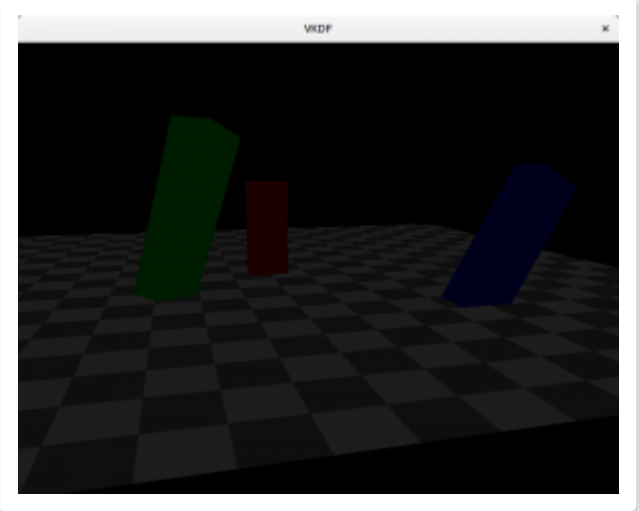
Basically, we multiply the diffuse component of the incoming light with the diffuse reflection of the fragment's material to produce the diffuse component of the light reflected by the fragment. The diffuse component of the surface tells how the object absorbs and reflects incoming light. For example, a pure yellow object (diffuse material vec3(1,1,0)) would absorb the blue component and reflect 100% of the red and green components of the incoming light. If the light is a pure white light (diffuse vec3(1,1,1)),

then the observer would see a yellow object. However, if we are using a red light instead (diffuse vec3(1,0,0)), then the light reflected from the surface of the object would only contain the red component (since the light isn’t emitting a green component at all) and we would see it red.

As we said before though, the intensity of the reflection depends on the angle between the incoming light and the direction of the reflection. We account for this with the dot product between the normal at the fragment (surface_normal) and the direction of the light (or rather, the vector pointing from the fragment to the light source). Notice that because the vectors that we use to compute the dot product are normalized, *dp_reflection* is exactly the cosine of the angle between these two vectors. At an angle of 0º the surface is facing straight at the light source, and the intensity of the diffuse reflection is at its peak, since cosine(0º)=1. At an angle of 90º (or larger) the cosine will be 0 or smaller and will be clamped to 0, meaning that no light is effectively being reflected by the surface (the computed diffuse component will be 0).

Ambient reflection

Computing all possible reflections and bounces of all rays of light from each light source in a scene is way too expensive. Instead, the Phong model approximates this by making indirect reflection from a light source constant across the scene. In other words: it assumes that the amount of indirect light received by any surface in the scene is the same. This eliminates all the complexity while still producing reasonable results in most scenarios. We call this constant factor ambient light.



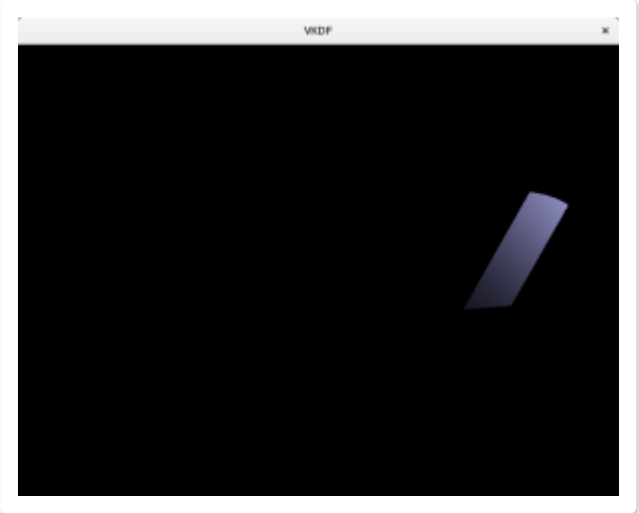
Ambient light

Adding ambient light to the fragment is then as simple as multiplying the light source's ambient light by the material's ambient reflection. The meaning of this product is exactly the same as in the case of the diffuse light, only that it affects the indirect light received by the fragment:

```
vec3 ambient = material.ambient *
light.ambient;
```

Specular reflection

Very sharp, smooth surfaces such as metal are known to produce specular highlights, which are those bright spots that we can see on shiny objects. Specular reflection depends on the angle between the observer's view direction and the direction in which the light is reflected off the surface. Specifically, the specular reflection is strongest when the observer is facing exactly in the opposite direction in which the light is reflected. Depending on the properties of the surface, the specular reflection can be more or less focused, affecting how the specular component scatters after being reflected. This property of the material is usually referred to as its shininess.



Specular light

Implementing specular reflection requires a bit more of work:

```
vec3 specular = vec3(0);
vec3 light_dir_norm =
normalize(vec3(light.direction));
if (dot(normal, -light_dir_norm) >= 0.0) {
   vec3 reflection_dir =
reflect(light_dir_norm, normal);
   float shine_factor = dot(reflection_dir,
normalize(in_view_dir));
   specular = light.specular.xyz *
material.specular.xyz *
         pow(max(0.0, shine_factor),
material.shininess.x);
}
```

Basically, the code above checks if there is any specular reflection at all by computing the cosine of the angle between the fragment's normal and the direction of the light (notice that, once again, both vectors are normalized prio to using them in the call to dot()). If there is specular reflection, then we compute how shiny the reflection is perceived by the viewer based on the angle between the vector from this fragment to the observer (in_view_dir) and the direction of the light reflected off the fragment's surface (reflection_dir). The smaller the angle, the more parallel the directions are, meaning that the camera is receiving more reflection and the specular component received is stronger. Finally, we modulate the

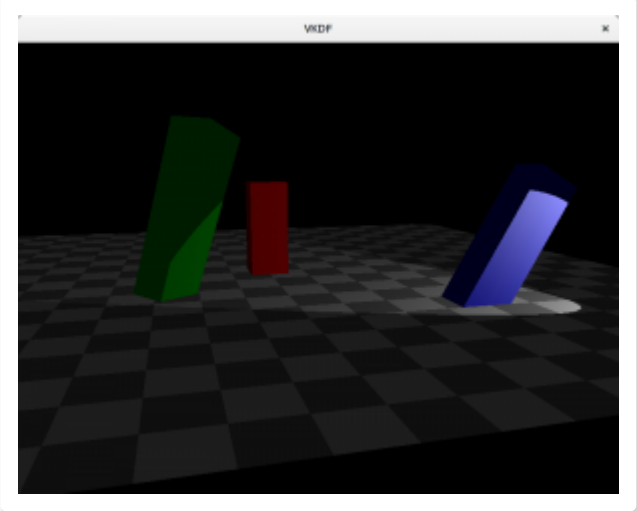
result based on the **shininess** of the fragment. We can compute *in_view_dir* in the vertex shader using the inverse of the View matrix like this:

```
mat4 ViewInv = inverse(View);
out_view_dir =
   normalize(vec3(ViewInv * vec4(0.0, 0.0, 0.0, 1.0) - world_pos));
```

The code above takes advantage of the fact that camera transformations are an illusion created by applying the transforms to everything else we render. For example, if we want to create the illusion that the camera is moving to the right, we just apply a translation to everything we render so they show up a bit to the left. This is what our *View* matrix achieves. From the point of view of GL or Vulkan, the camera is always fixed at (0,0,0). Taking advantage of this, we can compute the position of the virtual observer (the camera) in world space coordinates by applying the inverse of our camera transform to its fixed location (0,0,0). This is what the code above does, where *world_pos* is the position of this vertex in world space and *View* is the camera's view matrix.

In order to produce the final look of the scene according to the Phong reflection model, we need to compute these 3 components for each fragment and add them together:

```
out_color  = vec4(diffuse + ambient + specular, 1.0)
```



Diffuse + Ambient + Specular (spotlight source)

Attenuation

In most scenarios, light intensity isn't constant across the scene. Instead, it is brightest at its source and gets dimmer with distance. We can easily model this by adding an attenuation factor that is multiplied by the distance from the fragment to the light source. Typically, the intensity of the light decreases quite fast with distance, so a linear attenuation factor alone may not produce the best results and a quadratic function is preferred:

```
float attenuation = 1.0 /
    (light.attenuation.constant +
     light.attenuation.linear * dist +
     light.attenuation.quadratic * dist *
dist);

diffuse = diffuse * attenuation;
ambient = ambient * attenuation;
specular = specular * attenuation;
```

Of course, we may decide not to apply attenuation to the ambient component at all if we really want to make it look like it is constant across the scene, however, do notice that when multiple light sources are

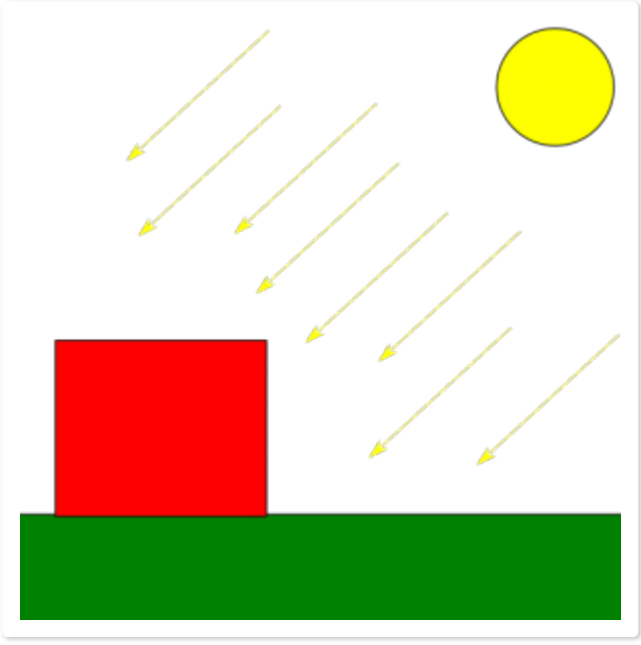
present, the ambient factors from each source will accumulate and may produce too much ambient light unless they are attenuated.

Types of lights

When we model a light source we also need to consider the kind of light we are manipulating:

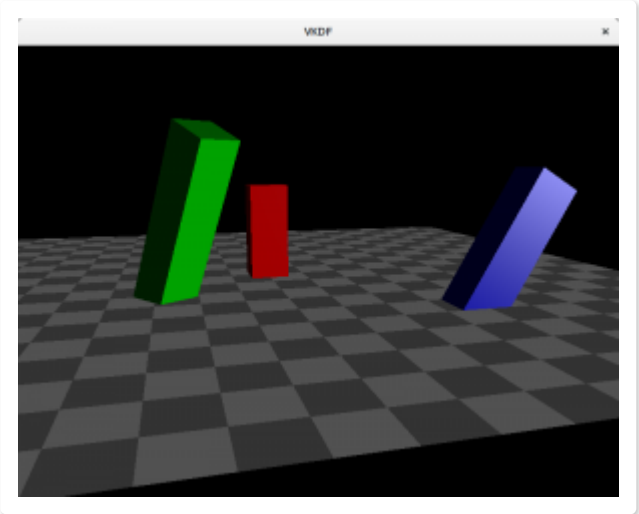
Directional lights

These are light sources that emit rays that travel along a specific direction so that all are parallel to each other. We typically use this model to represent bright, distant light sources that produce constant light across the scene. An example would be the sun light. Because the distance to the light source is so large compared to distances in the scene, the attenuation factor is irrelevant and can be discarded. Another particularity of directional light sources is that because the light rays are parallel, shadows casted from them are regular (we will talk more about this once we cover shadow mapping in future posts).



Directional light

If we had used a directional light in the scene, it would look like this:

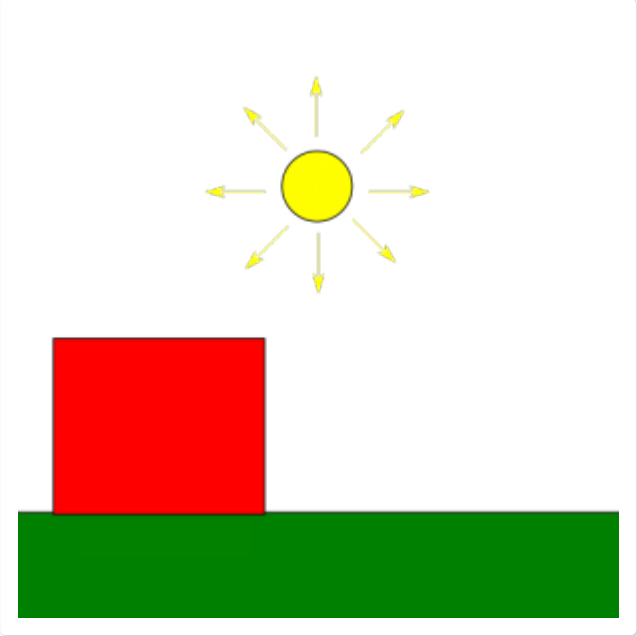


Scene with a directional light

Notice how the brightness of the scene doesn't lower with the distance to the light source.

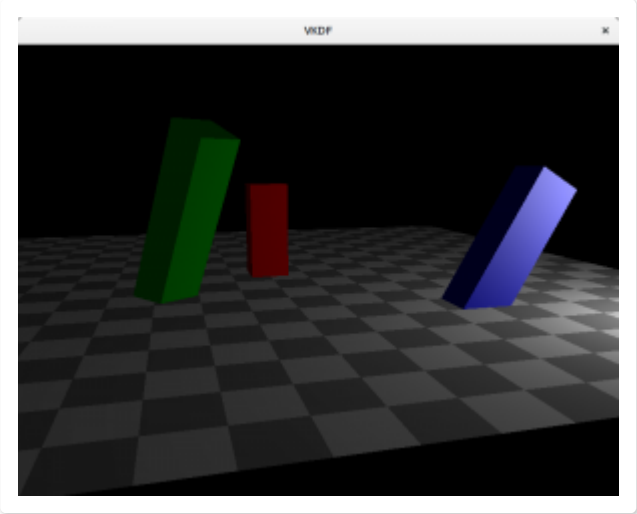
Point lights

These are light sources for which light originates at a specific position and spreads outwards in all directions. Shadows casted by point lights are not regular, instead they are projected. An example would be the light produced by a light bulb. The attenuation code I showed above would be appropriate to represent point lights.



Point light

Here is how the scene would look like with a point light:



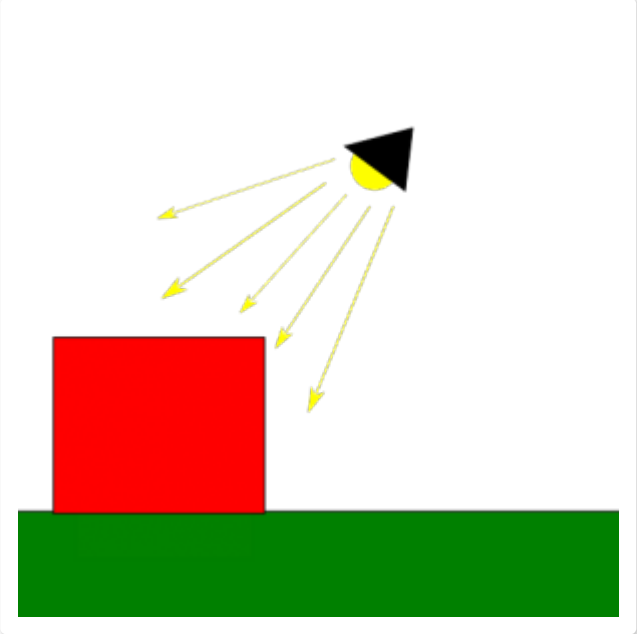
Scene with a point light

In this case, we can see how attenuation plays a factor and brightness lowers as we walk away from the light source (which is close to the blue cuboid).

Spotlights

This is the light source I used to illustrate the diffuse, ambient and specular components. They are similar to point lights, is the sense that

light originates from a specific point in space and spreads outwards, however, instead of scattering in all directions, rays scatter forming a cone with the tip at the origin of the light. The angle formed by the lights's direction and the sides of the cone is usually called the cutoff angle, because not light is casted outside its limits. Flashlights are a good example of this type of light.



Spotlight

In order to create spotlights we need to consider the cutoff angle of the light and make sure that no diffuse or specular component is reflected by a fragment which is beyond the cutoff threshold:

```
vec3 light_to_pos_norm = -pos_to_light_norm;
float dp = dot(light_to_pos_norm,
light_dir_norm);
if (dp <= light.cutoff) {
   diffuse = vec3(0);
   specular = vec3(0);
}
```

In the code above we compute the cosine of the angle between the light's direction and the vector from the light to the fragment (dp). Here, *light.cutoff* represents the cosine of the spotlight's cutoff angle too, so

when *dp* is smaller it means that the fragment is outside the light cone emitted by the spotlight and we remove its diffuse and specular reflections completely.

Multiple lights

Handling multiple lights is easy enough: we only need to compute the color contribution for each light separately and then add all of them together for each fragment (pseudocode):

```
vec3 fragColor = vec3(0);
foreach light in lights
    fragColor += compute_color_for_light(light,
...);
...
```

Of course, light attenuation plays a vital role here to limit the area of influence of each light so that scenes where we have multiple lights don’t get too bright.

An important thing to notice above the pseudocode above is that this process involves looping through costy per-fragment light computations for each light source, which can lead to important performance hits as the number of lights in the scene increases. This shading model, as described here, is called **forward rendering** and it has the benefit that it is very simple to implement but its downside is that we may incur in many costy lighting computations for fragments that, eventually, won’t be visible in the screen (due to them being occluded by other fragments). This is particularly important when the number of lights in the scene is quite large and its complexity makes it so that there are many occluded fragments. Another technique that may be more suitable for these situations is called **deferred rendering**, which postpones costy shader computations to a later stage (hence the word *deferred*) in which we only evaluate them for

fragments that are known to be visible, but that is a topic for another day, in this series we will focus on *forward rendering* only.

Lights and shadows

For the purpose of **shadow mapping** in particular we should note that objects that are directly lit by the light source reflect all 3 of the light components, while objects in the shadow only reflect the ambient component. Because objects that only reflect ambient light are less bright, they appear shadowed, in similar fashion as they would in the real world. We will see the details how this is done in the next post, but for the time being, keep this in mind.

Source code

The scene images in this post were obtained from a simple shadow mapping demo I wrote in **Vulkan**. The source code for that is available here, and it includes also the shadow mapping implementation that I'll cover in the next post. Specifically relevant to this post are the scene vertex and fragment shaders where lighting calculations take place.

Conclusions

In order to represent shadows we first need a means to represent light. In this post we discussed the Phong reflection model as a simple, yet effective way to model light reflection in a scene as the addition of three separate components: diffuse, ambient and specular. Once we have a representation of light we can start discussing shadows, which are parts of

the scene that only receive ambient light because other objects occlude the diffuse and specular components of the light source.

Working with lights and shadows – Part II: The shadow map

JULY 30, 2017

In the previous post we talked about the Phong lighting model as a means to represent light in a scene. Once we have light, we can think about implementing shadows, which are the parts of the scene that are not directly exposed to light sources. Shadow mapping is a well known technique used to render shadows in a scene from one or multiple light sources. In this post we will start discussing how to implement this, specifically, how to render the shadow map image, and the next post will cover how to use the shadow map to render shadows in the scene.

Note: although the code samples in this post are for Vulkan, it should be easy for the reader to replicate the implementation in OpenGL. Also, my OpenGL terrain renderer demo implements shadow mapping and can also be used as a source code reference for OpenGL.

Algorithm overview

Shadow mapping involves two passes, the first pass renders the scene from te point of view of the light with depth testing enabled and records depth information for each fragment. The resulting depth image (the shadow map) contains depth information for the fragments that are visible

from the light source, and therefore, are occluders for any other fragment behind them from the point of view of the light. In other words, these represent the only fragments in the scene that receive direct light, every other fragment is in the shade. In the second pass we render the scene normally to the render target from the point of view of the camera, then for each fragment we need to compute the distance to the light source and compare it against the depth information recorded in the previous pass to decice if the fragment is behind a light occluder or not. If it is, then we remove the diffuse and specular components for the fragment, making it look shadowed.

In this post I will cover the first pass: generation of the shadow map.

Producing the shadow map image

Note: those looking for OpenGL code can have a look at this file ter-shadow-renderer.cpp from my OpenGL terrain renderer demo, which contains the shadow map renderer that generates the shadow map for the sun light in that demo.

Creating a depth image suitable for shadow mapping

The shadow map is a regular depth image were we will record depth information for fragments in light space. This image will be rendered into and sampled from. In Vulkan we can create it like this:

```
...
VkImageCreateInfo image_info = {};
image_info.sType =
VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
image_info.pNext = NULL;
image_info.imageType = VK_IMAGE_TYPE_2D;
image_info.format = VK_FORMAT_D32_SFLOAT;
image_info.extent.width = SHADOW_MAP_WIDTH;
image_info.extent.height = SHADOW_MAP_HEIGHT;
image_info.extent.depth = 1;
image_info.mipLevels = 1;
image_info.arrayLayers = 1;
image_info.samples = VK_SAMPLE_COUNT_1_BIT;
image_info.initialLayout =
VK_IMAGE_LAYOUT_UNDEFINED;
image_info.usage =
VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT |
                    VK_IMAGE_USAGE_SAMPLED_BIT;
image_info.queueFamilyIndexCount = 0;
image_info.pQueueFamilyIndices = NULL;
image_info.sharingMode =
VK_SHARING_MODE_EXCLUSIVE;
image_info.flags = 0;

VkImage image;
vkCreateImage(device, &image_info, NULL,
&image);
...
```

The code above creates a 2D image with a 32-bit float depth format. The shadow map's width and height determine the resolution of the depth image: larger sizes produce higher quality shadows but of course this comes with an additional computing cost, so you will probably need to balance quality and performance for your particular target. In the first pass of the algorithm we need to render to this depth image, so we include the VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT usage flag, while in the second pass we will sample the shadow map from the fragment shader to decide if each fragment is in the shade or not, so we also include the VK_IMAGE_USAGE_SAMPLED_BIT.

One more tip: when we allocate and bind memory for the image, we probably want to request device local memory too (VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT) for optimal performance, since we won't need to map the shadow map memory in the host for anything.

Since we are going to render to this image in the first pass of the process we also need to create a suitable image view that we can use to create a framebuffer. There are no special requirements here, we just create a view with the same format as the image and with a depth aspect:

```
...
VkImageViewCreateInfo view_info = {};
view_info.sType =
VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
view_info.pNext = NULL;
view_info.image = image;
view_info.format = VK_FORMAT_D32_SFLOAT;
view_info.components.r =
VK_COMPONENT_SWIZZLE_R;
view_info.components.g =
VK_COMPONENT_SWIZZLE_G;
view_info.components.b =
VK_COMPONENT_SWIZZLE_B;
view_info.components.a =
VK_COMPONENT_SWIZZLE_A;
view_info.subresourceRange.aspectMask =
VK_IMAGE_ASPECT_DEPTH_BIT;
view_info.subresourceRange.baseMipLevel = 0;
view_info.subresourceRange.levelCount = 1;
view_info.subresourceRange.baseArrayLayer = 0;
view_info.subresourceRange.layerCount = 1;
view_info.viewType = VK_IMAGE_VIEW_TYPE_2D;
view_info.flags = 0;

VkImageView shadow_map_view;
vkCreateImageView(device, &view_info, NULL,
&view);
...
```

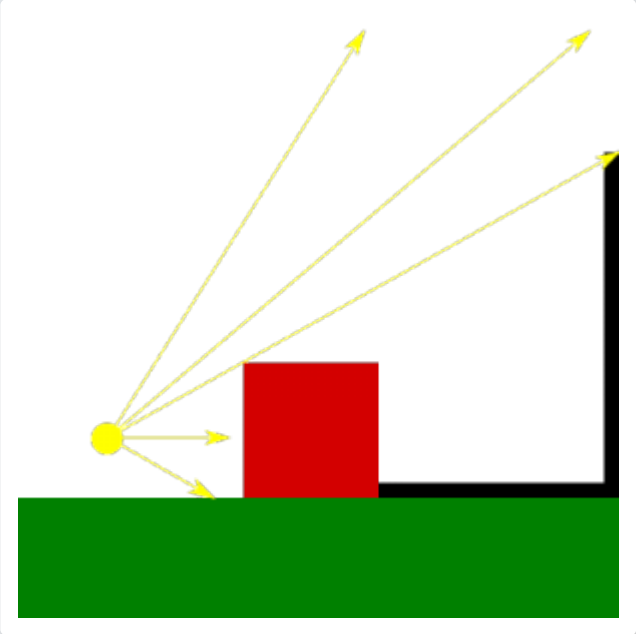
Rendering the shadow map

In order to generate the shadow map image we need to render the scene from the point of view of the light, so first, we need to compute the corresponding View and Projection matrices. How we calculate these matrices depends on the type of light we are using. As described in the previous post, we can consider 3 types of lights: spotlights, positional lights and directional lights.

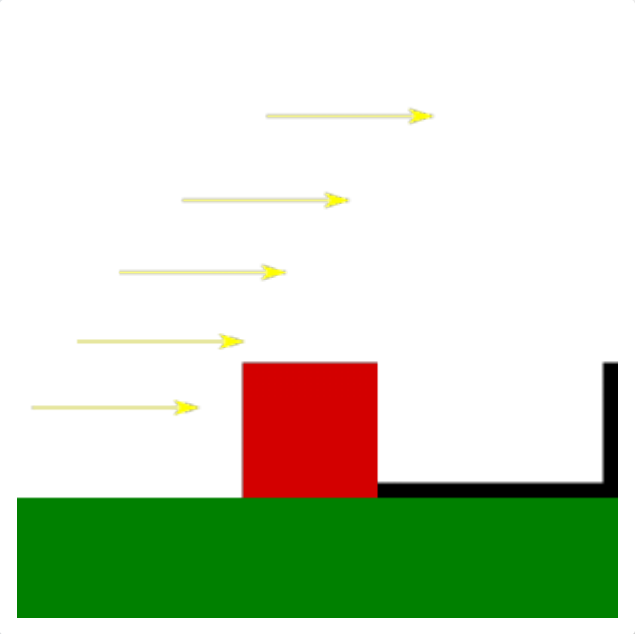
Spotlights are the easiest for shadow mapping, since with these we use regular perspective projection.

Positional lights work similar to spotlights in the sense that they also use perspective projection, however, because these are omnidirectional, they see the entire scene around them. This means that we need to render a shadow map that contains scene objects in all directions around the light. We can do this by using a cube texture for the shadow map instead of a regular 2D texture and render the scene 6 times adjusting the View matrix to capture scene objects in front of the light, behind it, to its left, to its right, above and below. In this case we want to use a field of view of 45º with the projection matrix so that the set of 6 images captures the full scene around the light source with no image overlaps.

Finally, we have directional lights. In the previous post I mentioned that these lights model light sources which rays are parallel and because of this feature they cast regular shadows (that is, shadows that are not perspective projected). Thus, to render shadow maps for directional lights we want to use orthographic projection instead of perspective projection.



Projected shadow from a point light source



Regular shadow from a directional light source

In this post I will focus on creating a shadow map for a spotlight source only. I might write follow up posts in the future covering other light sources, but for the time being, you can have a look at my OpenGL terrain renderer demo if you are interested in directional lights.

So, for a spotlight source, we just define a regular perspective projection, like this:

```
glm::mat4 clip = glm::mat4(1.0f, 0.0f, 0.0f,
0.0f,
                           0.0f,-1.0f, 0.0f,
0.0f,
                           0.0f, 0.0f, 0.5f,
0.0f,
                           0.0f, 0.0f, 0.5f,
1.0f);

glm::mat4 light_projection = clip *
      glm::perspective(glm::radians(45.0f),
                       (float) SHADOW_MAP_WIDTH
/ SHADOW_MAP_HEIGHT,
```

```
                                    LIGHT_NEAR, LIGHT_FAR);
```

The code above generates a regular perspective projection with a field of view of 45º. We should adjust the light's near and far planes to make them as tight as possible to reduce artifacts when we use the shadow map to render the shadows in the scene (I will go deeper into this in a later post). In order to do this we should consider that the near plane can be increased to reflect the closest that an object can be to the light (that might depend on the scene, of course) and the far plane can be decreased to match the light's area of influence (determined by its attenuation factors, as explained in the previous post).

The clip matrix is not specific to shadow mapping, it just makes it so that the resulting projection considers the particularities of how the Vulkan coordinate system is defined (Y axis is inversed, Z range is halved).

As usual, the projection matrix provides us with a projection frustrum, but we still need to point that frustum in the direction in which our spotlight is facing, so we also need to compute the view matrix transform of our spotlight. One way to define the direction in which our spotlight is facing is by having the rotation angles of spotlight on each axis, similarly to what we would do to compute the view matrix of our camera:

```
1  glm::mat4
2  compute_view_matrix_for_rotation(glm::vec3
3  origin, glm::vec3 rot)
4  {
5     glm::mat4 mat(1.0);
6     float rx = DEG_TO_RAD(rot.x);
7     float ry = DEG_TO_RAD(rot.y);
8     float rz = DEG_TO_RAD(rot.z);
9     mat = glm::rotate(mat, -rx, glm::vec3(1, 0,
10 0));
11    mat = glm::rotate(mat, -ry, glm::vec3(0, 1,
12 0));
13
```

```
    mat = glm::rotate(mat, -rz, glm::vec3(0, 0,
1));
    mat = glm::translate(mat, -origin);
    return mat;
}
```

Here, origin is the position of the light source in world space, and rot represents the rotation angles of the light source on each axis, representing the direction in which the spotlight is facing.

Now that we have the View and Projection matrices that define our light space we can go on and render the shadow map. For this we need to render scene as we normally would but instead of using our camera's View and Projection matrices, we use the light's. Let's have a look at the shadow map rendering code:

Render pass

```
static VkRenderPass
create_shadow_map_render_pass(VkDevice device)
{
   VkAttachmentDescription attachments[2];

   // Depth attachment (shadow map)
   attachments[0].format = VK_FORMAT_D32_SFLOAT;
   attachments[0].samples =
VK_SAMPLE_COUNT_1_BIT;
   attachments[0].loadOp =
VK_ATTACHMENT_LOAD_OP_CLEAR;
   attachments[0].storeOp =
VK_ATTACHMENT_STORE_OP_STORE;
   attachments[0].stencilLoadOp =
VK_ATTACHMENT_LOAD_OP_DONT_CARE;
   attachments[0].stencilStoreOp =
VK_ATTACHMENT_STORE_OP_DONT_CARE;
   attachments[0].initialLayout =
VK_IMAGE_LAYOUT_UNDEFINED;
   attachments[0].finalLayout =
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
```

```
    attachments[0].flags = 0;

    // Attachment references from subpasses
    VkAttachmentReference depth_ref;
    depth_ref.attachment = 0;
    depth_ref.layout =
VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

    // Subpass 0: shadow map rendering
    VkSubpassDescription subpass[1];
    subpass[0].pipelineBindPoint =
VK_PIPELINE_BIND_POINT_GRAPHICS;
    subpass[0].flags = 0;
    subpass[0].inputAttachmentCount = 0;
    subpass[0].pInputAttachments = NULL;
    subpass[0].colorAttachmentCount = 0;
    subpass[0].pColorAttachments = NULL;
    subpass[0].pResolveAttachments = NULL;
    subpass[0].pDepthStencilAttachment =
&depth_ref;
    subpass[0].preserveAttachmentCount = 0;
    subpass[0].pPreserveAttachments = NULL;

    // Create render pass
    VkRenderPassCreateInfo rp_info;
    rp_info.sType =
VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
    rp_info.pNext = NULL;
    rp_info.attachmentCount = 1;
    rp_info.pAttachments = attachments;
    rp_info.subpassCount = 1;
    rp_info.pSubpasses = subpass;
    rp_info.dependencyCount = 0;
    rp_info.pDependencies = NULL;
    rp_info.flags = 0;

    VkRenderPass render_pass;
    VK_CHECK(vkCreateRenderPass(device, &rp_info,
NULL, &render_pass));

    return render_pass;
}
```

The render pass is simple enough: we only have one attachment with the depth image and one subpass that renders to the shadow map target. We will start the render pass by clearing the shadow map and by the time we are done we want to store it and transition it to layout VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL so we can sample from it later when we render the scene with shadows. Notice that because we only care about depth information, the render pass doesn't include any color attachments.

Framebuffer

Every rendering job needs a target framebuffer, so we need to create one for our shadow map. For this we will use the image view we created from the shadow map image. We link this framebuffer target to the shadow map render pass description we have just defined:

```
VkFramebufferCreateInfo fb_info;
fb_info.sType =
VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
fb_info.pNext = NULL;
fb_info.renderPass = shadow_map_render_pass;
fb_info.attachmentCount = 1;
fb_info.pAttachments = &shadow_map_view;
fb_info.width = SHADOW_MAP_WIDTH;
fb_info.height = SHADOW_MAP_HEIGHT;
fb_info.layers = 1;
fb_info.flags = 0;

VkFramebuffer shadow_map_fb;
vkCreateFramebuffer(device, &fb_info, NULL,
&shadow_map_fb);
```

Pipeline description

The pipeline we use to render the shadow map also has some particularities:

Because we only care about recording depth information, we can typically skip any vertex attributes other than the positions of the vertices in the scene:

```
...
VkVertexInputBindingDescription vi_binding[1];
VkVertexInputAttributeDescription vi_attribs[1];

// Vertex attribute binding 0, location 0: positi
vi_binding[0].binding = 0;
vi_binding[0].inputRate = VK_VERTEX_INPUT_RATE_VE
vi_binding[0].stride = 2 * sizeof(glm::vec3);

vi_attribs[0].binding = 0;
vi_attribs[0].location = 0;
vi_attribs[0].format = VK_FORMAT_R32G32B32_SFLOAT
vi_attribs[0].offset = 0;

VkPipelineVertexInputStateCreateInfo vi;
vi.sType =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CRE
vi.pNext = NULL;
vi.flags = 0;
vi.vertexBindingDescriptionCount = 1;
vi.pVertexBindingDescriptions = vi_binding;
vi.vertexAttributeDescriptionCount = 1;
vi.pVertexAttributeDescriptions = vi_attribs;
...
pipeline_info.pVertexInputState = &vi;
...
```

The code above defines a single vertex attribute for the position, but assumes that we read this from a vertex buffer that packs interleaved positions and normals for each vertex (each being a vec3) so we use the binding’s stride to jump over the normal values in the buffer. This is because in this particular example, we have a single vertex buffer that we reuse for both shadow map rendering and normal scene rendering (which requires vertex normals for lighting computations).

Again, because we do not produce color data, we can skip the fragment shader and our vertex shader is a simple passthough instead of the normal vertex shader we use with the scene:

```
....
VkPipelineShaderStageCreateInfo shader_stages[1];
shader_stages[0].sType =
   VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE
shader_stages[0].pNext = NULL;
shader_stages[0].pSpecializationInfo = NULL;
shader_stages[0].flags = 0;
shader_stages[0].stage = VK_SHADER_STAGE_VERTEX_B
shader_stages[0].pName = "main";
shader_stages[0].module =
create_shader_module("shadowmap.vert.spv", ...);
...
pipeline_info.pStages = shader_stages;
pipeline_info.stageCount = 1;
...
```

This is how the shadow map vertex shader (shadowmap.vert) looks like in GLSL:

```
#version 400

#extension GL_ARB_separate_shader_objects :
enable
#extension GL_ARB_shading_language_420pack :
enable

layout(std140, set = 0, binding = 0) uniform
vp_ubo {
    mat4 ViewProjection;
} VP;

layout(std140, set = 0, binding = 1) uniform
m_ubo {
     mat4 Model[16];
} M;

layout(location = 0) in vec3 in_position;
```

```
19
20  void main()
21  {
       vec4 pos = vec4(in_position.x,
    in_position.y, in_position.z, 1.0);
       vec4 world_pos = M.Model[gl_InstanceIndex]
    * pos;
       gl_Position = VP.ViewProjection *
    world_pos;
    }
```

The shader takes the ViewProjection matrix of the light (we have already multiplied both together in the host) and a UBO with the Model matrices of each object in the scene as external resources (we use instanced rendering in this particular example) as well as a single vec3 input attribute with the vertex position. The only job of the vertex shader is to compute the position of the vertex in the transformed space (the light space, since we are passing the ViewProjection matrix of the light), nothing else is done here.

Command buffer

The command buffer is pretty similar to the one we use with the scene, only that we render to the shadow map image instead of the usual render target. In the shadow map render pass description we have indicated that we will clear it, so we need to include a depth clear value. We also need to make sure that we set the viewport and sccissor to match the shadow map dimensions:

```
1  ...
2  VkClearValue clear_values[1];
3  clear_values[0].depthStencil.depth = 1.0f;
4  clear_values[0].depthStencil.stencil = 0;
5
6  VkRenderPassBeginInfo rp_begin;
7  rp_begin.sType =
8  VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
```

```
rp_begin.pNext = NULL;
rp_begin.renderPass = shadow_map_render_pass;
rp_begin.framebuffer = shadow_map_framebuffer;
rp_begin.renderArea.offset.x = 0;
rp_begin.renderArea.offset.y = 0;
rp_begin.renderArea.extent.width =
SHADOW_MAP_WIDTH;
rp_begin.renderArea.extent.height =
SHADOW_MAP_HEIGHT;
rp_begin.clearValueCount = 1;
rp_begin.pClearValues = clear_values;

vkCmdBeginRenderPass(shadow_map_cmd_buf,
                     &rp_begin,
                     VK_SUBPASS_CONTENTS_INLINE);

VkViewport viewport;
viewport.height = SHADOW_MAP_HEIGHT;
viewport.width = SHADOW_MAP_WIDTH;
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
viewport.x = 0;
viewport.y = 0;
vkCmdSetViewport(shadow_map_cmd_buf, 0, 1,
&viewport);

VkRect2D scissor;
scissor.extent.width = SHADOW_MAP_WIDTH;
scissor.extent.height = SHADOW_MAP_HEIGHT;
scissor.offset.x = 0;
scissor.offset.y = 0;
vkCmdSetScissor(shadow_map_cmd_buf, 0, 1,
&scissor);
...
```

Next, we bind the shadow map pipeline we created above, bind the vertex buffer and descriptor sets as usual and draw the scene geometry.

```
...
vkCmdBindPipeline(shadow_map_cmd_buf,
                  VK_PIPELINE_BIND_POINT_GRAPHICS
                  shadow_map_pipeline);
```

```
5
6   const VkDeviceSize offsets[1] = { 0 };
7   vkCmdBindVertexBuffers(shadow_cmd_buf, 0, 1, vert
8   offsets);
9
10  vkCmdBindDescriptorSets(shadow_map_cmd_buf,
11                          VK_PIPELINE_BIND_POINT_GR
12                          shadow_map_pipeline_layou
13                          0, 1,
14                          shadow_map_descriptor_set
15                          0, NULL);
16
17  vkCmdDraw(shadow_map_cmd_buf, ...);
18
19  vkCmdEndRenderPass(shadow_map_cmd_buf);
    ...
```

Notice that the shadow map pipeline layout will be different from the one used with the scene too. Specifically, during scene rendering we will at least need to bind the shadow map for sampling and we will probably also bind additional resources to access light information, surface materials, etc that we don’t need to render the shadow map, where we only need the View and Projection matrices of the light plus the UBO with the model matrices of the objects in the scene.

We are almost there, now we only need to submit the command buffer for execution to render the shadow map:

```
1   ...
2   VkPipelineStageFlags shadow_map_wait_stages =
3   0;
4   VkSubmitInfo submit_info = { };
5   submit_info.pNext = NULL;
6   submit_info.sType =
7   VK_STRUCTURE_TYPE_SUBMIT_INFO;
8   submit_info.waitSemaphoreCount = 0;
9   submit_info.pWaitSemaphores = NULL;
10  submit_info.signalSemaphoreCount = 1;
11  submit_info.pSignalSemaphores = &signal_sem;
```

```
12  submit_info.pWaitDstStageMask = 0;
13  submit_info.commandBufferCount = 1;
14  submit_info.pCommandBuffers =
15  &shadow_map_cmd_buf;

    vkQueueSubmit(queue, 1, &submit_info, NULL);
    ...
```

Because the next pass of the algorithm will need to sample the shadow map during the final scene rendering,we use a semaphore to ensure that we complete this work before we start using it in the next pass of the algorithm.

In most scenarios, we will want to render the shadow map on every frame to account for dynamic objects that move in the area of effect of the light or even moving lights, however, if we can ensure that no objects have altered their positions inside the area of effect of the light and that the light's description (position/direction) hasn't changed, we may not need need to regenerate the shadow map and save some precious rendering time.

Visualizing the shadow map

After executing the shadow map rendering job our shadow map image contains the depth information of the scene from the point of view of the light. Before we go on and start using this as input to produce shadows in our scene, we should probably try to visualize the shadow map to verify that it is correct. For this we just need to submit a follow-up job that takes the shadow map image as a texture input and renders it to a quad on the screen. There is one caveat though: when we use perspective projection, Z values in the depth buffer are not linear, instead precission is larger at distances closer to the near plane and drops as we get closer to the far place in order to improve accuracy in areas closer to the observer and

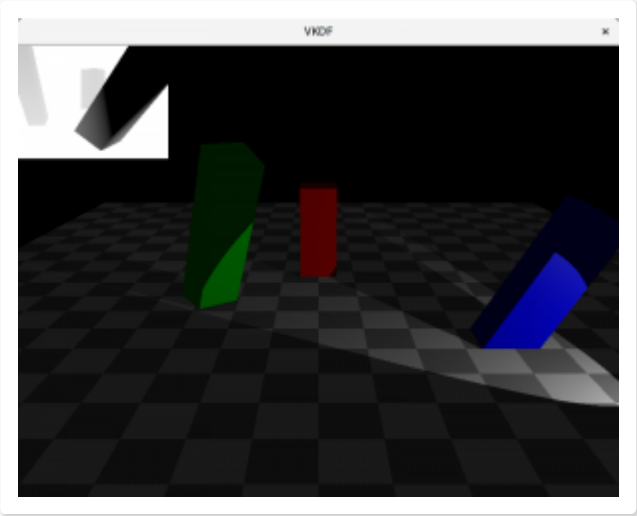
avoid Z-fighting artifacts. This means that we probably want to linearize our shadow map values when we sample from the texture so that we can actually see things, otherwise most things that are not close enough to the light source will be barely visible:

```
#version 400

#extension GL_ARB_separate_shader_objects :
enable
#extension GL_ARB_shading_language_420pack :
enable

layout(std140, set = 0, binding = 0) uniform
mvp_ubo {
    mat4 mvp;
} MVP;

layout(location = 0) in vec2 in_pos;
layout(location = 1) in vec2 in_uv;

layout(location = 0) out vec2 out_uv;

void main()
{
   gl_Position = MVP.mvp * vec4(in_pos.x,
in_pos.y, 0.0, 1.0);
   out_uv = in_uv;
}
```

```
#version 400

#extension GL_ARB_separate_shader_objects :
enable
#extension GL_ARB_shading_language_420pack :
enable

layout (set = 1, binding = 0) uniform
sampler2D image;

layout(location = 0) in vec2 in_uv;

layout(location = 0) out vec4 out_color;
```

```
14
15  void main()
16  {
       float depth = texture(image, in_uv).r;
       out_color = vec4(1.0 - (1.0 - depth) *
    100.0);
    }
```

We can use the vertex and fragment shaders above to render the contents of the shadow map image on to a quad. The vertex shader takes the quad's vertex positions and texture coordinates as attributes and passes them to the fragment shader, while the fragment shader samples the shadow map at the provided texture coordinates and then "linearizes" the depth value so that we can see better. The code in the shader doesn't properly linearize the depth values we read from the shadow map (that requires to pass the Z-near and Z-far values used in the projection), but for debugging purposes this works well enough for me, if you use different Z clipping planes you may need to alter the '100.0' value to get good results (or you might as well do a proper conversion considering your actual Z-near and Z-far values).



Visualizing the shadow map

The image shows the shadow map on top of the scene. Darker colors represent smaller depth values, so these are fragments closer to the light

source. Notice that we are not rendering the floor geometry to the shadow map since it can’t cast shadows on any other objects in the scene.

Conclusions

In this post we have described the shadow mapping technique as a combination of two passes: the first pass renders a depth image (the shadow map) with the scene geometry from the point of view of the light source. To achieve this, we need a passthrough vertex shader that only transforms the scene vertex positions (using the view and projection transforms from the light) and we can skip the fragment shader completely since we do not care for color output. The second pass, which we will cover in the next post, takes the shadow map as input and uses it to render shadows in the final scene.

Working with lights and shadows – Part III: rendering the shadows

OCTOBER 2, 2017

In the previous post in this series I introduced how to render the shadow map image, which is simply the depth information for the scene from the view point of the light. In this post I will cover how to use the shadow map to render shadows.

The general idea is that for each fragment we produce we compute the light space position of the fragment. In this space, the Z component tells us the depth of the fragment from the perspective of the light source. The next step requires to compare this value with the shadow map value for that same X,Y position. If the fragment's light space Z is larger than the value we read from the shadow map, then it means that this fragment is behind an object that is closer to the light and therefore we can say that it is in the shadows, otherwise we know it receives direct light.

Changes in the shader code

Let's have a look at the vertex shader changes required for this:

```
1  void main()
2  {
3
```

```
    vec4 pos = vec4(in_position.x,
in_position.y, in_position.z, 1.0);
    out_world_pos = Model * pos;
    gl_Position = Projection * View *
out_world_pos;

    [...]

    out_light_space_pos = LightViewProjection *
out_world_pos;
}
```

The vertex shader code above only shows the code relevant to the shadow mapping technique. *Model* is the model matrix with the spatial transforms for the vertex we are rendering, *View* and *Projection* represent the camera's view and projection matrices and the *LightViewProjection* represents the product of the light's view and projection matrices. The variables prefixed with *'out'* represent vertex shader outputs to the fragment shader.

The code generates the world space position of the vertex (world_pos) and clip space position (*gl_Position*) as usual, but then also computes the light space position for the vertex (*out_light_space_pos*) by applying the *View* and *Projection* transforms of the light to the world position of the vertex, which gives us the position of the vertex in light space. This will be used in the fragment shader to sample the shadow map.

The fragment shader will need to:

- Apply perspective division to compute *NDC* coordinates from the interpolated light space position of the fragment. Notice that this process is slightly different between *OpenGL* and *Vulkan*, since *Vulkan's NDC Z* is expected to be in the range [0, 1] instead of *OpenGL's* [-1, 1].

- Transform the X,Y coordinates from *NDC* space [-1, 1] to texture space [0, 1].

- Sample the shadow map and compare the result with the light space Z position we computed for this fragment to decide if the fragment is shadowed.

The implementation would look something like this:

```
float
compute_shadow_factor(vec4 light_space_pos,
sampler2D shadow_map)
{
   // Convert light space position to NDC
   vec3 light_space_ndc = light_space_pos.xyz
/= light_space_pos.w;

   // If the fragment is outside the light's
projection then it is outside
   // the light's influence, which means it is
in the shadow (notice that
   // such sample would be outside the shadow
map image)
   if (abs(light_space_ndc.x) > 1.0 ||
       abs(light_space_ndc.y) > 1.0 ||
       abs(light_space_ndc.z) > 1.0)
      return 0.0;

   // Translate from NDC to shadow map space
(Vulkan's Z is already in [0..1])
   vec2 shadow_map_coord = light_space_ndc.xy
* 0.5 + 0.5;

   // Check if the sample is in the light or
in the shadow
   if (light_space_ndc.z > texture(shadow_map,
shadow_map_coord.xy).x)
      return 0.0; // In the shadow

   // In the light
   return 1.0;
```

```
}
```

The function returns 0.0 if the fragment is in the shadows and 1.0 otherwise. Note that the function also avoids sampling the shadow map for fragments that are outside the light's frustum (and therefore are not recorded in the shadow map texture): we know that any fragment in this situation is shadowed because it is obviously not visible from the light. This assumption is valid for spotlights and point lights because in these cases the shadow map captures the entire influence area of the light source, for directional lights that affect the entire scene however, we usually need to limit the light's frustum to the surroundings of the camera, and in that case we probably want want to consider fragments outside the frustum as lighted instead.

Now all that remains in the shader code is to use this factor to eliminate the diffuse and specular components for fragments that are in the shadows. To achieve this we can simply multiply these components by the factor computed by this function.

Changes in the program

The list of changes in the main program are straight forward: we only need to update the pipeline layout and descriptors to attach the new resources required by the shaders, specifically, the light's view projection matrix in the vertex shader (which could be bound as a push constant buffer or a uniform buffer for example) and the shadow map sampler in the fragment shader.

Binding the light's *ViewProjection* matrix is no different from binding the other matrices we need in the shaders so I won't cover it here. The

shadow map sampler doesn’t really have any mysteries either, but since that is new let’s have a look at the code:

```
...
VkSampler sampler;
VkSamplerCreateInfo sampler_info = {};
sampler_info.sType =
VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
sampler_info.addressModeU =
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
sampler_info.addressModeV =
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
sampler_info.addressModeW =
VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
sampler_info.anisotropyEnable = false;
sampler_info.maxAnisotropy = 1.0f;
sampler_info.borderColor =
VK_BORDER_COLOR_INT_OPAQUE_BLACK;
sampler_info.unnormalizedCoordinates = false;
sampler_info.compareEnable = false;
sampler_info.compareOp = VK_COMPARE_OP_ALWAYS;
sampler_info.magFilter = VK_FILTER_LINEAR;
sampler_info.minFilter = VK_FILTER_LINEAR;
sampler_info.mipmapMode =
VK_SAMPLER_MIPMAP_MODE_NEAREST;
sampler_info.mipLodBias = 0.0f;
sampler_info.minLod = 0.0f;
sampler_info.maxLod = 100.0f;

VkResult result =
   vkCreateSampler(device, &sampler_info,
NULL, &sampler);
...
```

This creates the sampler object that we will use to sample the shadow map image. The address mode fields are not very relevant since our shader ensures that we do not attempt to sample outside the shadow map, we use linear filtering, but that is not mandatory of course, and we select nearest for the mipmap filter because we don’t have more than one miplevel in the shadow map.

Next we have to bind this sampler to the actual shadow map image. As usual in *Vulkan*, we do this with a descriptor update. For that we need to create a descriptor of type *VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER*, and then do the update like this:

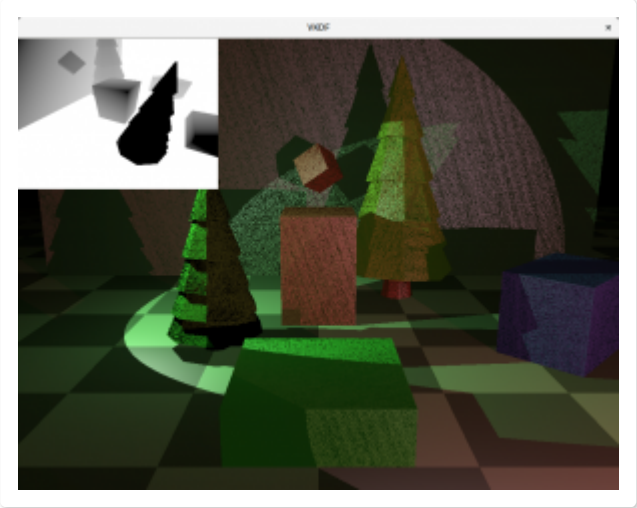
```
VkDescriptorImageInfo image_info;
image_info.sampler = sampler;
image_info.imageView = shadow_map_view;
image_info.imageLayout =
VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

VkWriteDescriptorSet writes;
writes.sType =
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
writes.pNext = NULL;
writes.dstSet = image_descriptor_set;
writes.dstBinding = 0;
writes.dstArrayElement = 0;
writes.descriptorCount = 1;
writes.descriptorType =
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
writes.pBufferInfo = NULL;
writes.pImageInfo = &image_info;
writes.pTexelBufferView = NULL;

vkUpdateDescriptorSets(ctx->device, 1,
&writes, 0, NULL);
```

A combined image sampler brings together the texture image to sample from (a *VkImageView* of the image actually) and the description of the filtering we want to use to sample that image (a *VkSampler*). As with all descriptor sets, we need to indicate its binding point in the set (in our case it is 0 because we have a separate descriptor set layout for this that only contains one binding for the combined image sampler).

Notice that we need to specify the layout of the image when it will be sampled from the shaders, which needs to be

VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL.
If you revisit the definition of our render pass for the shadow map image, you'll see that we had it automatically transition the shadow map to this layout at the end of the render pass, so we know the shadow map image will be in this layout immediately after it has been rendered, so we don't need to add barriers to execute the layout transition manually.

So that's it, with this we have all the pieces and our scene should be rendering shadows now. Unfortunately, we are not quite done yet, if you look at the results, you will notice a lot of dark noise in surfaces that are directly lit. This is an artifact of shadow mapping called self-shadowing or shadow acne. The next section explains how to get rid of it.



Self-shadowing artifacts

Eliminating self-shadowing

Self-shadowing can happen for fragments on surfaces that are directly lit by a source light for which we are producing a shadow map. The reason for this is that these are the fragments's Z coordinate in light space should exactly match the value we read from the shadow map for the same X,Y coordinates. In other words, for these fragments we expect:

```
light_space_ndc.z == texture(shadow_map, shadow_map_coord.xy).x.
```

However, due to different precession errors that can be generated on both sides of that equation, we may end up with slightly different values for each side and when the value we produce for *light_space_ndc.z* end ups being larger than what we read from the shadow map, even if it is a very small amount, it will mark the pixel as shadowed, leading to the result we see in that image.

The usual way to fix this problem involves adding a small depth offset or bias to the depth values we store in the shadow map so we ensure that we always read a larger value from the shadow map for the fragment. Another way to think about this is to think that when we record the shadow map, we push every object in the scene slightly away from the light source. Unfortunately, this depth offset bias should not be a constant value, since the angle between the surface normals and the vectors from the light source to the fragments also affects the bias value that we should use to correct the self-shadowing.

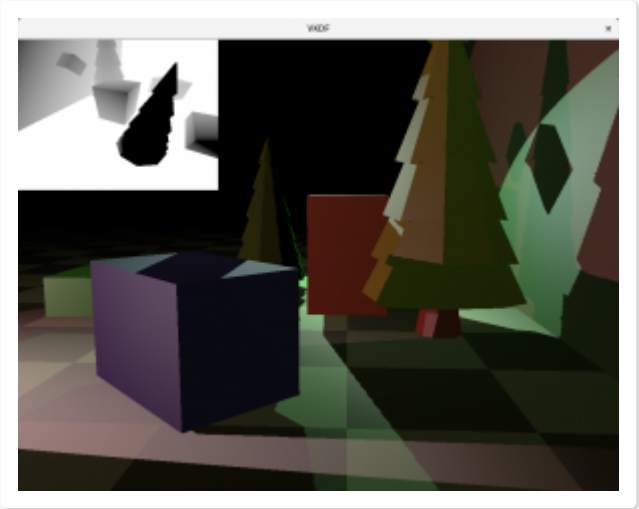
Thankfully, GPU hardware provides means to account for this. In *Vulkan*, when we define the rasterization state of the pipeline we use to create the shadow map, we can add the following:

```
VkPipelineRasterizationStateCreateInfo rs;
...
rs.depthBiasEnable = VK_TRUE;
rs.depthBiasConstantFactor = 4.0f;
rs.depthBiasSlopeFactor = 1.5f;
```

Where *depthBiasConstantFactor* is a constant factor that is automatically added to all depth values produced and *depthBiasSlopeFactor* is a factor that is used to compute depth offsets also based on the angle. This provides us with the means we need without having to do any extra work

in the shaders ourselves to offset the depth values correctly. In *OpenGL* the same functionality is available via *glPolygonOffset()*.

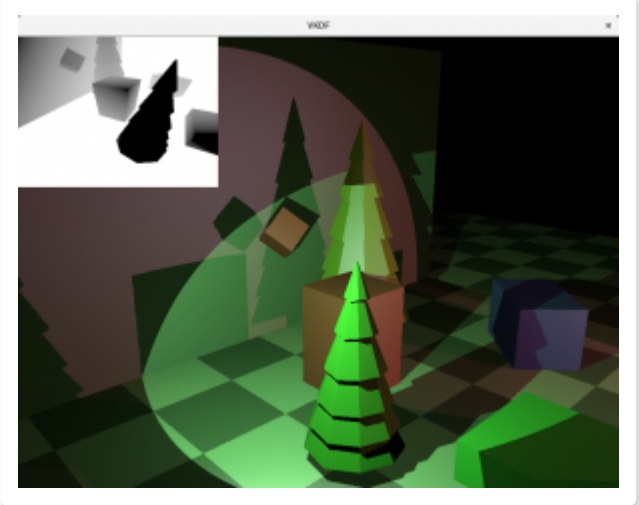
Notice that the bias values that need to be used to obtain the best results can change for each scene. Also, notice that too big values can lead to shadows that are “detached” from the objects that cast them leading to very unrealistic results. This effect is also known as *Peter Panning*, and can be observed in this image:



Peter Panning artifacts

As we can see in the image, we no longer have self-shadowing, but now we have the opposite problem: the shadows casted by the red and blue blocks are visibly incorrect, as if they were being rendered further away from the light source than they should be.

If the bias values are chosen carefully, then we should be able to get a good result, although some times we might need to accept some level of visible *self-shadowing* or visible *Peter Panning*:

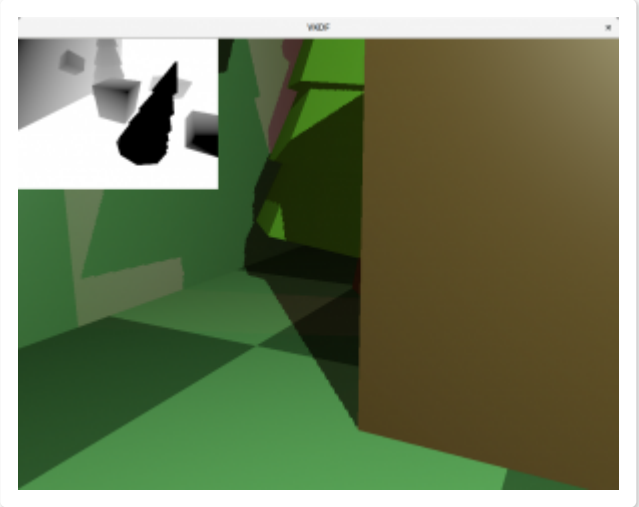


Correct shadowing

The image above shows correct shadowing without any *self-shadowing* or visible *Peter Panning*. You may wonder why we can't see some of the shadows from the red light in the floor where the green light is more intense. The reason is that even though it is not clear because I don't actually render the objects projecting the lights, the green light is mostly looking down, so its reflection on the floor (that has normals pointing upwards) is strong enough that the contribution from the red light to the floor pixels in this area is insignificant in comparison making the shadows casted from the red light barely visible. You can still see some shadowing if you get close enough with the camera though, I promise 😉

Shadow antialiasing

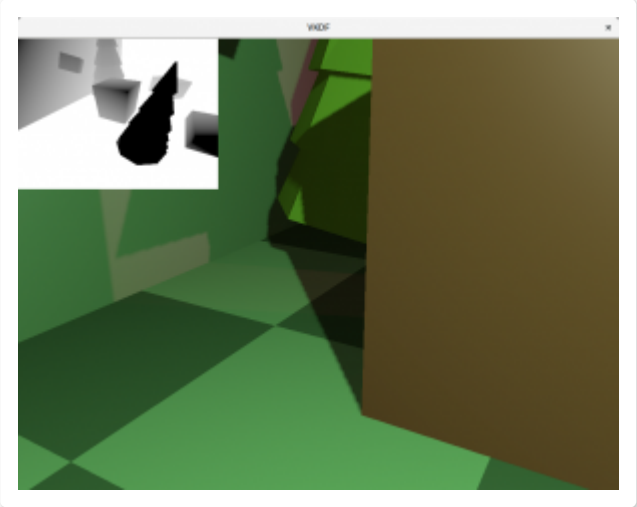
The images above show aliasing around at the edges of the shadows. This happens because for each fragment we decide if it is shadowed or not as a boolean decision, and we use that result to fully shadow or fully light the pixel, leading to aliasing:



Shadow aliasing

Another thing contributing to the aliasing effect is that a single pixel in the shadow map image can possibly expand to multiple pixels in camera space. That can happen if the camera is looking at an area of the scene that is close to the camera, but far away from the light source for example. In that case, the resolution of that area of the scene in the shadow map is small, but it is large for the camera, meaning that we end up sampling the same pixel from the shadow map to shadow larger areas in the scene as seen by the camera.

Increasing the resolution of the shadow map image will help with this, but it is not a very scalable solution and can quickly become prohibitive. Alternatively, we can implement something called **Percentage-Closer Filtering** to produce antialiased shadows. The technique is simple: instead of sampling just one texel from the shadow map, we take multiple samples in its neighborhood and average the results to produce shadow factors that do not need to be exactly 1 o 0, but can be somewhere in between, producing smoother transitions for shadowed pixels on the shadow edges. The more samples we take, the smoother the shadows edges get but do note that extra samples per pixel also come with a performance cost.



Smooth shadows with PCF

This is how we can update our *compute_shadow_factor()* function to add *PCF*:

```
float
compute_shadow_factor(vec4 light_space_pos,
                      sampler2D shadow_map,
                      uint shadow_map_size,
                      uint pcf_size)
{
   vec3 light_space_ndc = light_space_pos.xyz
/= light_space_pos.w;

   if (abs(light_space_ndc.x) > 1.0 ||
       abs(light_space_ndc.y) > 1.0 ||
       abs(light_space_ndc.z) > 1.0)
      return 0.0;

   vec2 shadow_map_coord = light_space_ndc.xy
* 0.5 + 0.5;

   // compute total number of samples to take
from the shadow map
   int pcf_size_minus_1 = int(pcf_size - 1);
   float kernel_size = 2.0 * pcf_size_minus_1
+ 1.0;
   float num_samples = kernel_size *
kernel_size;

   // Counter for the shadow map samples not
in the shadow
```

```
    float lighted_count = 0.0;

    // Take samples from the shadow map
    float shadow_map_texel_size = 1.0 /
shadow_map_size;
    for (int x = -pcf_size_minus_1; x <=
pcf_size_minus_1; x++)
    for (int y = -pcf_size_minus_1; y <=
pcf_size_minus_1; y++) {
        // Compute coordinate for this PFC
sample
        vec2 pcf_coord = shadow_map_coord +
vec2(x, y) * shadow_map_texel_size;

        // Check if the sample is in light or in
the shadow
        if (light_space_ndc.z <=
texture(shadow_map, pcf_coord.xy).x)
            lighted_count += 1.0;
    }

    return lighted_count / num_samples;
}
```

We now have a loop where we go through the samples in the neighborhood of the texel and average their respective shadow factors. Notice that because we sample the shadow map in texture space [0, 1], we need to consider the size of the shadow map image to properly compute the coordinates for the texels in the neighborhood so the application needs to provide this for every shadow map.

Conclusion

In this post we discussed how to use the shadow map image to produce shadows in the scene as well as typical issues that can show up with the shadow mapping technique, such as self-shadowing and aliasing, and how to correct them. This will be the last post in this series, there is a lot

more stuff to cover about lighting and shadowing, such as Cascaded Shadow Maps (which I introduced briefly in this other post), but I think (or I hope) that this series provides enough material to get anyone interested in the technique a reference for how to implement it.

Intel Mesa driver for Linux is now OpenGL 4.6 conformant

FEBRUARY 1, 2018

Khronos has recently announced the **conformance program for OpenGL 4.6** and I am very happy to say that **Intel has submitted successful conformance applications for various of its GPU models for the Mesa Linux driver**. For specifics on the conformant hardware you can check the list of conformant OpenGL products at the Khronos webstite.

Being **conformant on day one**, which the Intel Mesa Vulkan driver also obtained back in the day, is a significant achievement. Besides Intel Mesa, only NVIDIA managed to do this, which I think speaks of the amount of work and effort that one needs to put to achieve it. **The days where Linux implementations lagged behind are long gone**, we should all celebrate this and acknowledge the important efforts that companies like Intel have put into making this a reality.

Over the last 8-9 months or so, I have been working together with some of my Igalian friends to keep the Intel drivers (for both OpenGL and Vulkan) conformant, so I am very proud that we have reached this milestone. Kudos to all my work mates who have worked with me on this, to our friends at Intel, who have been providing reviews for our patches, feedback and additional driver fixes, and to many other members in the Mesa community who have contributed to make this possible in one way or another.

Of course, OpenGL 4.6 conformance requires that we have an implementation of **GL_ARB_gl_spirv,** which allows OpenGL applications to consume SPIR-V shaders. If you have been following Igalia's work, you have already seen some of my colleagues sending patches for this over the last months, but the feature is not completely upstreamed yet. We are working hard on this, but the scope of the implementation that we want for upstream is rather ambitious, since it involves to (finally) have a **full shader linker in NIR**. Getting that to be as complete as the current GLSL linker and in a shape that is good enough for review and upstreaming is going to take some time, but it is surely a worthwhile effort that will pay off in the future, so please look forward to it and be patient with us as we upstream more of it in the coming months.

It is also important to remark that OpenGL 4.6 conformance doesn't just validate new features in OpenGL 4.6, it is a full conformance program for OpenGL drivers that includes OpenGL 4.6 functionality, and as such, it is a super set of the OpenGL 4.5 conformance. The OpenGL 4.6 CTS does, in fact, incorporate a whole lot of bugfixes and expanded coverage for OpenGL features that were already present in OpenGL 4.5 and prior.

What is the conformance process and why is it important?

It is a well known issue with standards that different implementations are not always consistent. This can happen for a number of reasons. For example, implementations have bugs which can make something work on one platform but not on another (which will then require applications to implement work arounds). Another reason for this is that some times implementators just have different interpretations of the standard.

The Khronos conformance program is intended to ensure that products that implement Khronos standards (such as OpenGL or Vulkan drivers) do what they are supposed to do and they do it consistently across implementations from the same or different vendors. This is achieved by producing an extensive test suite, the Conformance Test Suite (or CTS for short), which aims to verify that the semantics of the standard are properly implemented by as many vendors as possible.

Why is CTS different to other test suites available?

One reason is that CTS development includes Khronos members who are involved in the definition of the API specifications. This means there are people well versed in the spec language who can review and provide feedback to test developers to ensure that the tests are good.

Another reason is that before new tests go in, it is required that there are at least a number of implementation (from different vendors) that pass them. This means that various vendors have implemented the related

specifications and these different implementations agree on the expected result, which is usually a good sign that the tests and the implementations are good (although this is not always enough!).

How does CTS and the Khronos conformance process help API implementators and users?

First, it makes it so that existing and new functionality covered in the API specifications is tested before granting the conformance status. This means that implementations have to run all these tests and pass them, producing the same results as other implementations, so as far as the test coverage goes, the implementations are correct and consistent, which is the whole point of this process: it wont' matter if you're running your application on Intel, NVIDIA, AMD or a different GPU vendor, if your application is correct, it should run the
same no matter the driver you are running on.

Now, this doesn't mean that your application will run smoothly on all conformant platforms out of the box. Application developers still need to be aware that certain aspects or features in the specifications are optional, or that different hardware implementations may have different limits for certain things. Writing software that can run on multiple platforms is always a challenge and some of that will always need to be addressed on the application side, but at least the conformance process attempts to ensure that for applications that do their part of the work, things will work as intended.

There is another interesting implication of conformance that has to do with correct API specification. Designing APIs that can work across hardware from different vendors is a challenging process. With the CTS, Khronos has an opportunity to validate the specifications against actual implementations. In other words, the CTS allows Khronos to verify that vendors can implement the specifications as intended and revisit the specification if they can't before releasing them. This ensures that API specifications are reasonable and a good match for existing hardware implementations.

Another benefit of CTS is that vendors working on any API implementation will always need some level of testing to verify their code during development. Without CTS, they would have to write their own tests (which would be biased towards their own interpretations of the spec anyway), but with CTS, they can leave that to Khronos and focus on the implementation instead, cutting down development times and sharing testing code with other vendors.

What about Piglit or other testing frameworks?

CTS doesn't make Piglit obsolete or redundant at all. While CTS coverage will improve over the years it is nearly impossible to have 100% coverage, so having other testing frameworks around that can provide extra coverage is always good.

My experience working on the Mesa drivers is that it is surprisingly easy to break stuff, specially on OpenGL which has a lot of legacy stuff in it. I have seen way too many instances of patches that seemed correct and in fact fixed actual problems only to later see Piglit, CTS and/or dEQP report

regressions on existing tests. The (not so surprising) thing is that many times, the regressions were reported on just one of these testing frameworks, which means they all provide some level of coverage that is not included in the others.

It is for this reason that the continuous integration system for Mesa provided by Intel runs all of these testing frameworks (and then some others). You just never get enough testing. And even then, some regressions slip into releases despite all the testing!

Also, for the case of Piglit in particular, I have to say that it is very easy to write new tests, specially shader runner tests, which is always a bonus. Writing tests for CTS or dEQP, for example, requires more work in general.

So what have I been doing exactly?

For the last 9 months or so, I have been working on ensuring that the Intel Mesa drivers for both Vulkan and OpenGL are conformant. If you have followed any of my work in Mesa over the last year or so, you have probably already guessed this, since most of the patches I have been sending to Mesa reference the conformance tests they fix.

To be more thorough, my work included:

- Reviewing and testing patches submitted for inclusion in CTS that either fixed test bugs, extended coverage for existing features or added new tests for new API specifications. CTS is a fairly active project with numerous changes submitted for review pretty much every day, for OpenGL, OpenGL ES and Vulkan, so staying on top of things requires a significant dedication.

- Ensuring that the Intel Mesa drivers passed all the CTS tests for both Vulkan and OpenGL. This requires to run the conformance tests, identifying test failures, identifying the cause for the failures and providing proper fixes. The fixes would go to CTS when the cause for the issue was a bogus test, to the driver, when it was a bug in our implementation or the fact that the driver was simply missing some functionality, or they could even go to the OpenGL or Vulkan specs, when the source of the problem was incomplete, ambiguous or incorrect spec language that was used to drive the test development. I have found instances of all these situations.

Where can I get the CTS code?

Good news, **it is open source** and available at GitHub.

This is a very important and welcomed change by Khronos. When I started helping Intel with OpenGL conformance, specifically for OpenGL 4.5, the CTS code was only available to specific Khronos members. Since then, Khronos has done a significant effort in working towards having an open source testing framework where anyone can contribute, so kudos to Khronos for doing this!

Going open source not only leverages larger collaboration and further development of the CTS. It also puts in the hands of API users a handful of small test samples that people can use to learn how some of the new Vulkan and OpenGL APIs released to the public are to be used, which is always nice.

What is next?

As I said above, CTS development is always ongoing, there is always testing coverage to expand for existing features, bugfixes to provide for existing tests, more tests that need to be adapted or changed to match corrections in in the spec language, new extensions and APIs that need testing coverage, etc.

And on the driver side, there are always new features to implement that come with their potential bugs that need to be fixed, occasional regressions that need to be addressed promptly, new bugs uncovered by new tests that need fixes, etc

So the fun never really stops 🙂

Final words

In this post, besides bringing the good news to everyone, I hope that I have made a good case for why the Khronos CTS is important for the industry and why we should care about it. I hope that I also managed to give a sense for the enormous amount of work that goes into making all of this possible, both on the side of Khronos and the side of the driver developer teams. I think all this effort means **better drivers for everyone** and I hope that we all, as users, come to appreciate it for that.

Finally, big thanks to Intel for sponsoring our work on Mesa and CTS, and also to Igalia, for having me work on this wonderful project.

Intel Mesa driver for Linux is now Vulkan 1.1 conformant

MARCH 12, 2018

It was only a few weeks ago when I posted that the Intel Mesa driver had successfully passed the Khronos OpenGL 4.6 conformance on day one, and now I am very proud that we can announce the same for the **Intel Mesa Vulkan 1.1 driver**, the new Vulkan API version announced by the Khronos Group last week. Big thanks to Intel for making Linux a first-class citizen for graphics APIs, and specially to Jason Ekstrand, who did most of the Vulkan 1.1 enablement in the driver.

At Igalia we are very proud of being a part of this: on the driver side, we have contributed the implementation of VK_KHR_16bit_storage, numerous bugfixes for issues raised by the Khronos Conformance Test Suite (CTS) and code reviews for some of the new Vulkan 1.1 features developed by Intel. On the CTS side, we have worked with other Khronos members in reviewing and testing additions to the test suite, identifying and providing fixes for issues in the tests as well as developing new tests.

Finally, I’d like to highlight the **strong industry adoption of Vulkan**: as stated in the Khronos press release, various other hardware vendors have already implemented conformant Vulkan 1.1 drivers, we are also seeing major 3D engines adopting and supporting Vulkan and AAA games that have already shipped with Vulkan-powered graphics. There is no doubt that this is only the beginning and that we will be seeing a lot more of Vulkan in the coming years, so look forward to it!

Improving shader performance with Vulkan’s specialization constants

MARCH 20, 2018

For some time now I have been working on and off on a personal project with no other purpose than toying a bit with Vulkan and some rendering and shading techniques. Although I’ll probably write about that at some point, in this post I want to focus on Vulkan’s specialization constants and how they can provide a very visible performance boost when they are used properly, as I had the chance to verify while working on this project.

The concept behind specialization constants is very simple: they allow applications to set the value of a shader constant at run-time. At first sight, this might not look like much, but it can have very important implications for certain shaders. To showcase this, let’s take the following snippet from a fragment shader as a case study:

```
1  layout(push_constant) uniform pcb {
2     int num_samples;
3  } PCB;
4
5  const int MAX_SAMPLES = 64;
6  layout(set = 0, binding = 0) uniform
7  SamplesUBO {
8     vec3 samples[MAX_SAMPLES];
9  } S;
10
```

```
void main()
{
   ...
   for(int i = 0; i < PCB.num_samples; ++i) {
      vec3 sample_i = S.samples[i];
      ...
   }
   ...
}
```

That is a snippet taken from a Screen Space Ambient Occlusion shader that I implemented in my project, a popular techinique used in a lot of games, so it represents a real case scenario. As we can see, the process involves a set of vector samples passed to the shader as a UBO that are processed for each fragment in a loop. We have made the maximum number of samples that the shader can use large enough to accomodate a high-quality scenario, but the actual number of samples used in a particular execution will be taken from a push constant uniform, so the application has the option to choose the quality / performance balance it wants to use.

While the code snippet may look trivial enough, let’s see how it interacts with the shader compiler:

The first obvious issue we find with this implementation is that it is preventing loop unrolling to happen because the actual number of samples to use is unknown at shader compile time. At most, the compiler could guess that it can’t be more than 64, but that number of iterations would still be too large for Mesa to unroll the loop in any case. If the application is configured to only use 24 or 32 samples (the value of our push constant uniform at run-time) then that number of iterations would be small enough that Mesa would unroll the loop if that number was known at shader compile time, so in that scenario we would be losing the

optimization just because we are using a push constant uniform instead of a constant for the sake of flexibility.

The second issue, which might be less immediately obvious and yet is the most significant one, is the fact that if the shader compiler can tell that the size of the samples array is small enough, then it can promote the UBO array to a push constant. This means that each access to S.samples[i] turns from an expensive memory fetch to a direct register access for each sample. To put this in perspective, if we are rendering to a full HD target using 24 samples per fragment, it means that we would be saving ourselves from doing 1920x1080x24 memory reads per frame for a very visible performance gain. But again, we would be loosing this optimization because we decided to use a push constant uniform.

Vulkan's specialization constants allow us to get back these performance optmizations without sacrificing the flexibility we implemented in the shader. To do this, the API provides mechanisms to specify the values of the constants at run-time, but before the shader is compiled.

Continuing with the shader snippet we showed above, here is how it can be rewritten to take advantage of specialization constants:

```
1  layout (constant_id = 0) const int NUM_SAMPLES
2  = 64;
3  layout(std140, set = 0, binding = 0) uniform
4  SamplesUBO {
5     vec3 samples[NUM_SAMPLES];
6  } S;
7
8  void main()
9  {
10    ...
11    for(int i = 0; i < NUM_SAMPLES; ++i) {
12       vec3 sample_i = S.samples[i];
13       ...
```

```
14      }
        ...
    }
```

We are now informing the shader that we have a specialization constant NUM_SAMPLES, which represents the actual number of samples to use. By default (if the application doesn't say otherwise), the specialization constant's value is 64. However, now that we have a specialization constant in place, we can have the application set its value at run-time, like this:

```
1  VkSpecializationMapEntry entry = { 0, 0,
2  sizeof(int32_t) };
3     VkSpecializationInfo spec_info = {
4        1,
5        &entry,
6        sizeof(uint32_t),
7        &config.ssao.num_samples
8     };
      ...
```

The application code above sets up specialization constant information for shader consumption at run-time. This is done via an array of *VkSpecializationMapEntry* entries, each one determining where to fetch the constant value to use for each specialization constant declared in the shader for which we want to override its default value. In our case, we have a single specialization constant (with id 0), and we are taking its value (of integer type) from offset 0 of a buffer. In our case we only have one specialization constant, so our buffer is just the address of the variable holding the constant's value (config.ssao.num_samples). When we create the Vulkan pipeline, we pass this specialization information using the *pSpecializationInfo* field of *VkPipelineShaderStageCreateInfo*. At that point, the driver will override the default value of the specialization constant with the value provided here before the shader code is optimized

and native GPU code is generated, which allows the driver compiler backend to generate optimal code.

It is important to remark that specialization takes place when we create the pipeline, since that is the only moment at which Vulkan drivers compile shaders. This makes specialization constants particularly useful when we know the value we want to use ahead of starting the rendering loop, for example when we are applying quality settings to shaders. However, If the value of the constant changes frequently, specialization constants are not useful, since they require expensive shader re-compiles every time we want to change their value, and we want to avoid that as much as possible in our rendering loop. Nevertheless, it it is possible to compile the same shader with different constant values in different pipelines, so even if a value changes often, so long as we have a finite number of combinations, we can generate optimized pipelines for each one ahead of the start of the redendering loop and just swap pipelines as needed while rendering.

Conclusions

Specialization constants are a straight forward yet powerful way to gain control over how shader compilers optimize your code. In my particular pet project, applying specialization constants in a small number of shaders allowed me to benefit from loop unrolling and, most importantly, UBO promotion to push constants in the SSAO pass, obtaining performance improvements that ranged from 10% up to 20% depending on the configuration.

Finally, although the above covered specialization constants from the point of view of Vulkan, this is really a feature of the SPIR-V language, so it is also available in OpenGL with the GL_ARB_gl_spirv extension, which is core since OpenGL 4.6.

Frame analysis of a rendering of the Sponza model

APRIL 17, 2018

For some time now I have been working on a personal project to render the well known Sponza model provided by Crytek using Vulkan. Here is a picture of the current (still a work-in-progress) result:

Sponza rendering

This screenshot was captured on my Intel Kabylake laptop, running on the Intel Mesa Vulkan driver (Anvil).

The following list includes the main features implemented in the demo:

- Depth pre-pass

- Forward and deferred rendering paths
- Anisotropic filtering
- Shadow mapping with Percentage-Closer Filtering
- Bump mapping
- Screen Space Ambient Occlusion (only on the deferred path)
- Screen Space Reflections (only on the deferred path)
- Tone mapping
- Anti-aliasing (FXAA)

I have been thinking about writing post about this for some time, but given that there are multiple features involved I wasn't sure how to scope it. Eventually I decided to write a "frame analysis" post where I describe, step by step, all the render passes involved in the production of the single frame capture showed at the top of the post. I always enjoyed reading this kind of articles so I figured it would be fun to write one myself and I hope others find it informative, if not entertaining.

To avoid making the post too dense I won't go into too much detail while describing each render pass, so don't expect me to go into the nitty-gritty of how I implemented Screen Space Ambient Occlussion for example. Instead I intend to give a high-level overview of how the various features implemented in the demo work together to create the final result. I will provide screenshots so that readers can appreciate the outputs of each step and verify how detail and quality build up over time as we include more features in the pipeline. Those who are more interested in the programming details of particular features can always have a look at the Vulkan source code (link available at the bottom of the article), look for specific tutorials available on the Internet or wait for me to write feature-specifc posts (I don't make any promises though!).

If you’re interested in going through with this then grab a cup of coffe and get ready, it is going to be a long ride!

Step 0: Culling

This is the only step in this discussion that runs on the CPU, and while optional from the point of view of the result (it doesn’t affect the actual result of the rendering), it is relevant from a performance point of view. Prior to rendering anything, in every frame, we usually want to cull meshes that are not visible to the camera. This can greatly help performance, even on a relatively simple scene such as this. This is of course more noticeable when the camera is looking in a direction in which a significant amount of geometry is not visible to it, but in general, there are always parts of the scene that are not visible to the camera, so culling is usually going to give you a performance bonus.

In large, complex scenes with tons of objects we probably want to use more sophisticated culling methods such as Quadtrees, but in this case, since the number of meshes is not too high (the Sponza model is slightly shy of 400 meshes), we just go though all of them and cull them individually against the camera’s frustum, which determines the area of the 3D space that is visible to the camera.

The way culling works is simple: for each mesh we compute an axis-aligned bounding box and we test that box for intersection with the camera’s frustum. If we can determine that the box never intersects, then the mesh enclosed within it is not visible and we flag it as such. Later on, at rendering time (or rather, at command recording time, since the demo has been written in Vulkan) we just skip the meshes that have been flagged.

The algorithm is not perfect, since it is possible that an axis-aligned bounding box for a particular mesh is visible to the camera and yet no part of the mesh itself is visible, but it should not affect a lot of meshes and trying to improve this would incur in additional checks that could undermine the efficiency of the process anyway.

Since in this particular demo we only have static geometry we only need to run the culling pass when the camera moves around, since otherwise the list of visible meshes doesn't change. If dynamic geometry were present, we would need to at least cull dynamic geometry on every frame even if the camera stayed static, since dynamic elements may step in (or out of) the viewing frustum at any moment.

Step 1: Depth pre-pass

This is an optional stage, but it can help performance significantly in many cases. The idea is the following: our GPU performance is usually going to be limited by the fragment shader, and very specially so as we target higher resolutions. In this context, without a depth pre-pass, we are very likely going to execute the fragment shader for fragments that will not end up in the screen because they are occluded by fragments produced by other geometry in the scene that will be rasterized to the same XY screen-space coordinates but with a smaller Z coordinate (closer to the camera). This wastes precious GPU resources.

One way to improve the situation is to sort our geometry by distance from the camera and render front to back. With this we can get fragments that are rasterized from background geometry quickly discarded by early depth tests before the fragment shader runs for them. Unfortunately, although this will certainly help (assuming we can spare the extra CPU work to

keep our geometry sorted for every frame), it won’t eliminate all the instances of the problem in the general case.

Also, some times things are more complicated, as the shading cost of different pieces of geometry can be very different and we should also take this into account. For example, we can have a very large piece of geometry for which some pixels are very close to the camera while some others are very far away and that has a very expensive shader. If our renderer is doing front-to-back rendering without any other considerations it will likely render this geometry early (since parts of it are very close to the camera), which means that it will shade all or most of its very expensive fragments. However, if the renderer accounts for the relative cost of the shader execution it would probably postpone rendering it as much as possible, so by the time it actually renders it, it takes advantage of early fragment depth tests to avoid as many of its expensive fragment shader executions as possible.

Using a depth-prepass ensures that we only run our fragment shader for visible fragments, and only those, no matter the situation. The downside is that we have to execute a separate rendering pass where we render our geometry to the depth buffer so that we can identify the visible fragments. This pass is usually very fast though, since we don’t even need a fragment shader and we are only writing to a depth texture. The exception to this rule is geometry that has opacity information, such as opacity textures, in which case we need to run a cheap fragment shader to identify transparent pixels and discard them so they don’t hit the depth buffer. In the Sponza model we need to do that for the flowers or the vines on the columns for example.

Depth pre-pass output

The picture shows the output of the depth pre-pass. Darker colors mean smaller distance from the camera. That’s why the picture gets brighter as we move further away.

Now, the remaining passes will be able to use this information to limit their shading to fragments that, for a given XY screen-space position, match exactly the Z value stored in the depth buffer, effectively selecting only the fragments that will be visible in the screen. We do this by configuring the depth test to do an EQUAL test instead of the usual LESS test, which is what we use in the depth-prepass.

In this particular demo, running on my Intel GPU, the depth pre-pass is by far the cheapest of all the GPU passes and it definitely pays off in terms of overall performance output.

Step 2: Shadow map

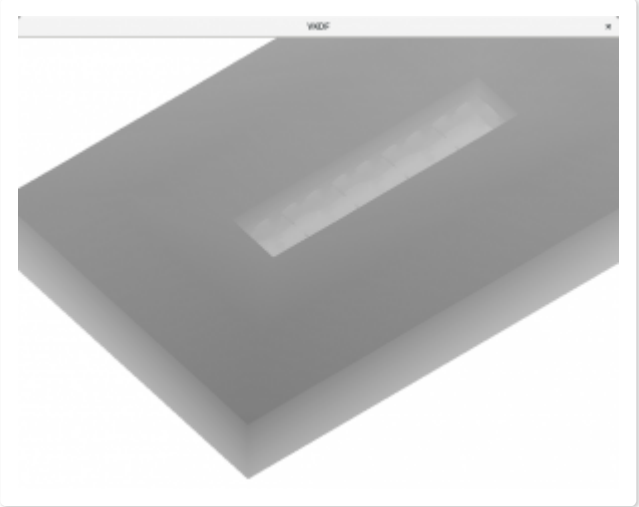
In this demo we have single source of light produced by a directional light that simulates the sun. You can probably guess the direction of the light by

checking out the picture at the top of this post and looking at the direction projected shadows.

I already covered how shadow mapping works in previous series of posts, so if you're interested in the programming details I encourage you to read that. Anyway, the basic idea is that we want to capture the scene from the point of view of the light source (to be more precise, we want to capture the objects in the scene that can potentially produce shadows that are visible to our camera).

With that information, we will be able to inform out lighting pass so it can tell if a particular fragment is in the shadows (not visible from our light's perspective) or in the light (visible from our light's perspective) and shade it accordingly.

From a technical point of view, recording a shadow map is exactly the same as the depth-prepass: we basically do a depth-only rendering and capture the result in a depth texture. The main differences here are that we need to render from the point of view of the light instead of our camera's and that this being a directional light, we need to use an orthographic projection and adjust it properly so we capture all relevant shadow casters around the camera.



In the image above we can see the shadow map generated for this frame. Again, the brighter the color, the further away the fragment is from the light source. The bright white area outside the atrium building represents the part of the scene that is empty and thus ends with the maximum depth, which is what we use to clear the shadow map before rendering to it.

In this case, we are using a 4096×4096 texture to store the shadow map image, much larger than our rendering target. This is because shadow mapping from directional lights needs a lot of precision to produce good results, otherwise we end up with very pixelated / blocky shadows, more artifacts and even missing shadows for small geometry. To illustrate this better here is the same rendering of the Sponza model from the top of this post, but using a 1024×1024 shadow map (floor reflections are disabled, but that is irrelevant to shadow mapping):

Sponza rendering with 1024×1024 shadow map

You can see how in the 1024×1024 version there are some missing shadows for the vines on the columns and generally blurrier shadows (when not also slightly distorted) everywhere else.

Step 3: GBuffer

In deferred rendering we capture various attributes of the fragments produced by rasterizing our geometry and write them to separate textures that we will use to inform the lighting pass later on (and possibly other passes).

What we do here is to render our geometry normally, like we did in our depth-prepass, but this time, as we explained before, we configure the depth test to only pass fragments that match the contents of the depth-buffer that we produced in the depth-prepass, so we only process fragments that we now will be visible on the screen.

Deferred rendering uses multiple render targets to capture each of these attributes to a different texture for each rasterized fragment that passes the depth test. In this particular demo our GBuffer captures:

- Normal vector
- Diffuse color
- Specular color
- Position of the fragment from the point of view of the light (for shadow mapping)

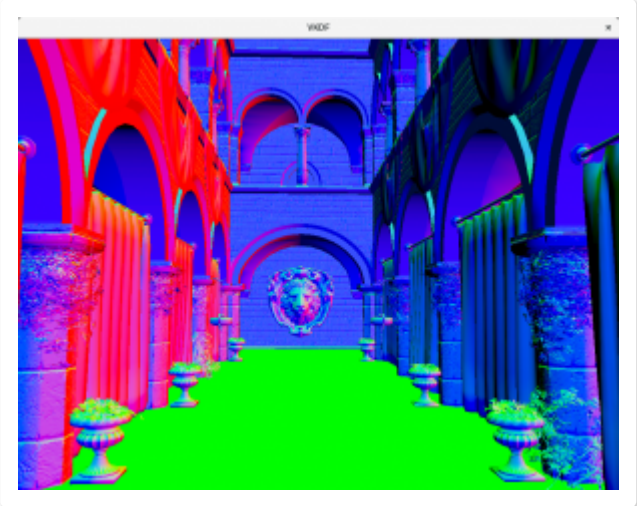
It is important to be very careful when defining what we store in the GBuffer: since we are rendering to multiple screen-sized textures, this pass has serious bandwidth requirements and therefore, we should use texture formats that give us the range and precision we need with the smallest pixel size requirements and avoid storing information that we can get or compute efficiently through other means. This is particularly relevant

for integrated GPUs that don’t have dedicated video memory (such as my Intel GPU).

In the demo, I do lighting in view-space (that is the coordinate space used takes the camera as its origin), so I need to work with positions and vectors in this coordinate space. One of the parameters we need for lighting is surface normals, which are conveniently stored in the GBuffer, but we will also need to know the view-space position of the fragments in the screen. To avoid storing the latter in the GBuffer we take advantage of the fact that we can reconstruct the view-space position of any fragment on the screen from its depth (which is stored in the depth buffer we rendered during the depth-prepass) and the camera’s projection matrix. I might cover the process in more detail in another post, for now, what is important to remember is that we don’t need to worry about storing fragment positions in the GBuffer and that saves us some bandwidth, helping performance.

Let’s have a look at the various GBuffer textures we produce in this stage:

Normal vectors

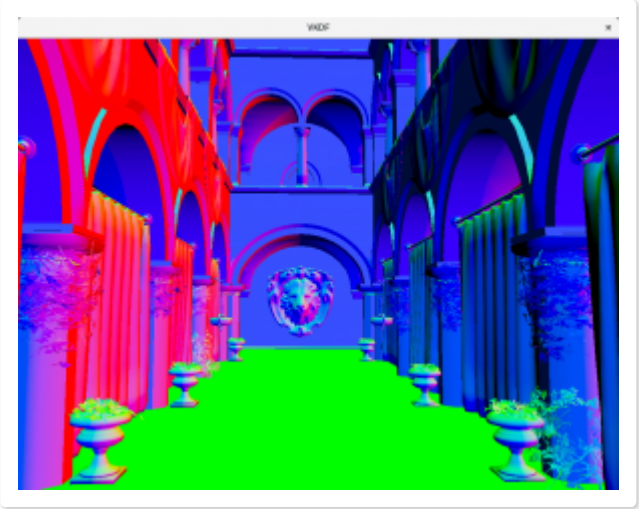


GBuffer normal texture

Here we see the normalized normal vectors for each fragment in view-space. This means they are expressed in a coordinate space in which our camera is at the origin and the positive Z direction is opposite to the camera's view vector. Therefore, we see that surfaces pointing to the right of our camera are red (positive X), those pointing up are green (positive Y) and those pointing opposite to the camera's view direction are blue (positive Z).

It should be mentioned that some of these surfaces use normal maps for bump mapping. These normal maps are textures that provide per-fragment normal information instead of the usual vertex normals that come with the polygon meshes. This means that instead of computing per-fragment normals as a simple interpolation of the per-vertex normals across the polygon faces, which gives us a rather flat result, we use a texture to adjust the normal for each fragment in the surface, which enables the lighting pass to render more nuanced surfaces that seem to have a lot more volume and detail than they would have otherwise.

For comparison, here is the GBuffer normal texture without bump mapping enabled. The difference in surface detail should be obvious. Just look at the lion figure at the far end or the columns and and you will immediately notice the addditional detail added with bump mapping to the surface descriptions:



GBuffer normal texture (bump mapping disabled)

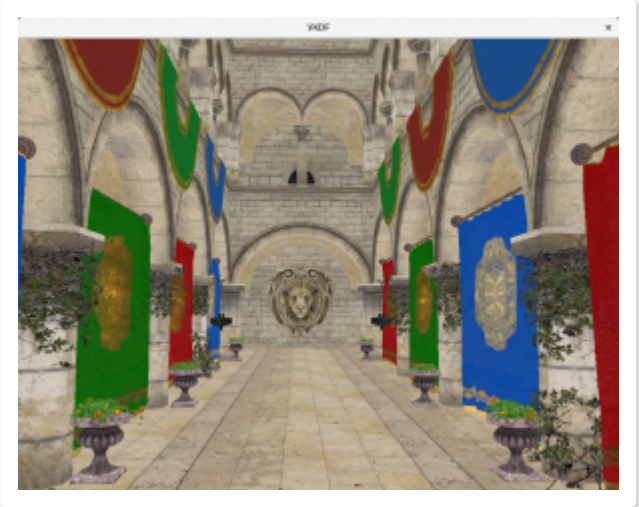
To make the impact of the bump mapping more obvious, here is a different shot of the final rendering focusing on the columns of the upper floor of the atrium, with and without bump mapping:

Bump mapping enabled

Bump mapping disabled

All the extra detail in the columns is the sole result of the bump mapping technique.

Diffuse color

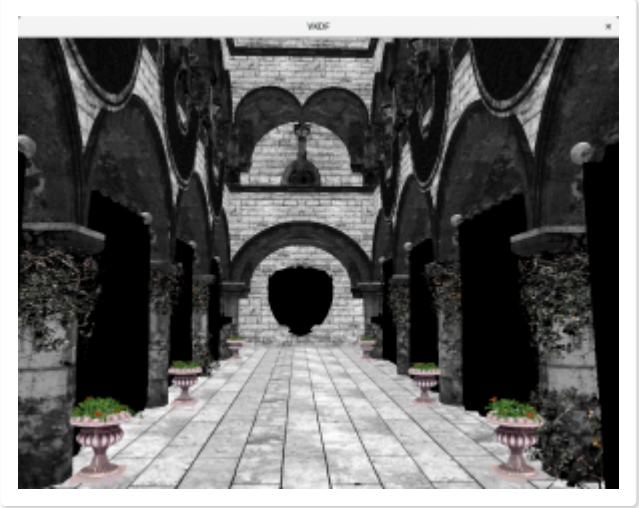


GBuffer diffuse texture

Here we have the diffuse color of each fragment in the scene. This is basically how our scene would look like if we didn't implement a lighting pass that considers how the light source interacts with the scene.

Naturally, we will use this information in the lighting pass to modulate the color output based on the light interaction with each fragment.

Specular color



GBuffer specular texture

This is similar to the diffuse texture, but here we are storing the color (and strength) used to compute specular reflections.

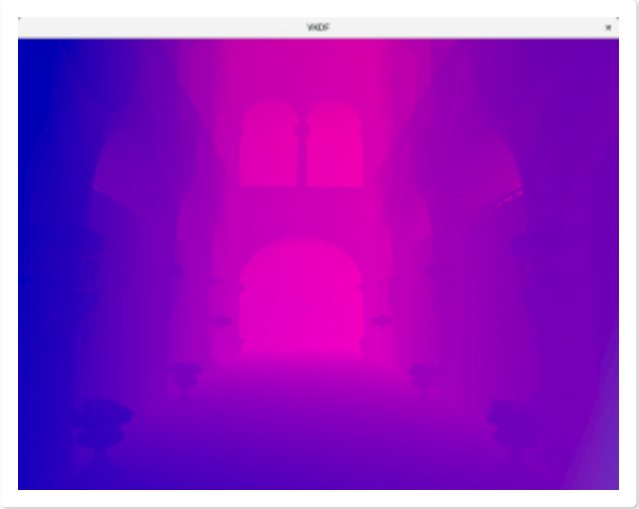
Similarly to normal textures, we use specular maps to obtain per-fragment specular colors and intensities. This allows us to simulate combinations of more complex materials in the same mesh by specifying different specular properties for each fragment.

For example, if we look at the cloths that hang from the upper floor of the atrium, we see that they are mostly black, meaning that they barely produce any specular reflection, as it is to be expected from textile materials. However, we also see that these same cloths have an embroidery that has specular reflection (showing up as a light gray color), which means these details in the texture have stronger specular reflections than its surrounding textile material:

Specular reflection on cloth embroidery

The image shows visible specular reflections in the yellow embroidery decorations of the cloth (on the bottom-left) that are not present in the textile segment (the blue region of the cloth).

Fragment positions from Light



GBuffer light-space position texture

Finally, we store fragment positions in the coordinate space of the light source so we can implement shadows in the lighting pass. This image may be less intuitive to interpret, since it is encoding space positions from the point of view of the sun rather than physical properties of the fragments. We will need to retrieve this information for each fragment during the lighting pass so that we can tell, together with the shadow map, which fragments are visible from the light source (and therefore are directly lit by the sun) and which are not (and therefore are in the shadows). Again, more detail on how that process works, step by step and including Vulkan source code in my series of posts on that topic.

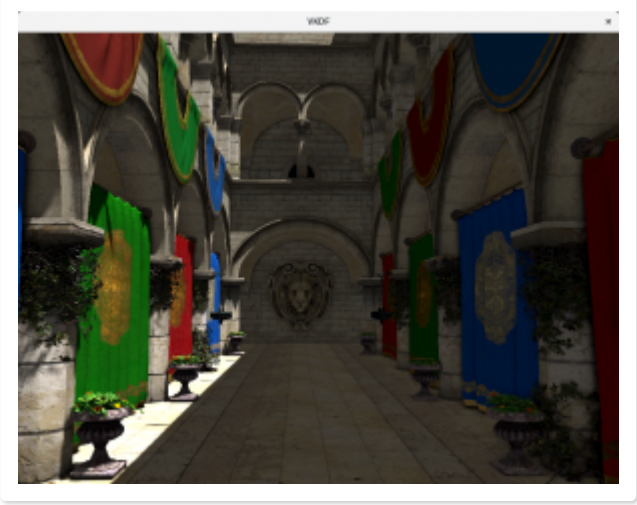
Step 4: Screen Space Ambient Occlusion

With the information stored in the GBuffer we can now also run a screen-space ambient occlusion pass that we will use to improve our lighting pass later on.

The idea here, as I discussed in my lighting and shadows series, the Phong lighting model simplifies ambient lighting by making it constant across the scene. As a consequence of this, lighting in areas that are not directly lit by a light source look rather flat, as we can see in this image:

SSAO disabled

Screen-space Ambient Occlusion is a technique that gathers information about the amount of ambient light occlusion produced by nearby geometry as a way to better estimate the ambient light term of the lighting equations. We can then use that information in our lighting pass to modulate ambient light accordingly, which can greatly improve the sense of depth and volume in the scene, specially in areas that are not directly lit:



SSAO enabled

Comparing the images above should illustrate the benefits of the SSAO technique. For example, look at the folds in the blue curtains on the right side of the images, without SSAO, we barely see them because the lighting is too flat across all the pixels in the curtain. Similarly, thanks to SSAO we can create shadowed areas from ambient light alone, as we can see behind the cloths that hang from the upper floor of the atrium or behind the vines on the columns.

To produce this result, the output of the SSAO pass is a texture with ambient light intensity information that looks like this (after some blur post-processing to eliminate noise artifacts):

SSAO output texture

In that image, white tones represent strong light intensity and black tones represent low light intensity produced by occlusion from nearby geometry. In our lighting pass we will source from this texture to obtain per-fragment ambient occlusion information and modulate the ambient term accordingly, bringing the additional volume showcased in the image above to the final rendering.

Step 6: Lighting pass

Finally, we get to the lighting pass. Most of what we showcased above was preparation work for this.

The lighting pass mostly goes as I described in my lighting and shadows series, only that since we are doing deferred rendering we get our per-fragment lighting inputs by reading from the GBuffer textures instead of getting them from the vertex shader.

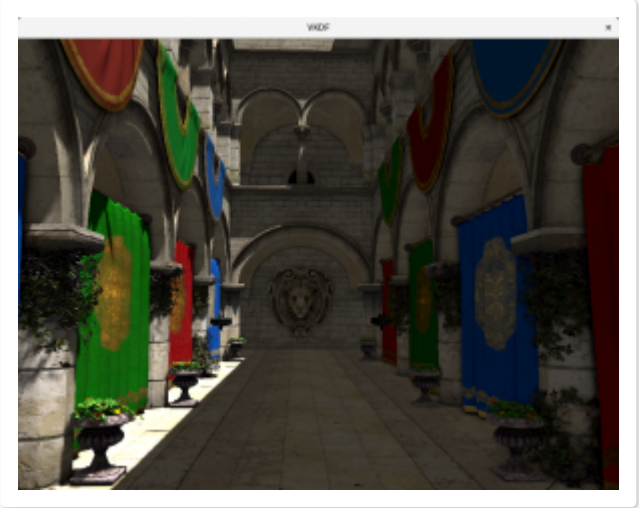
Basically, the process involves retrieving diffuse, ambient and specular color information from the GBuffer and use it as input for the lighting equations to produce the final color for each fragment. We also sample from the shadow map to decide which pixels are in the shadows, in which case we remove their diffuse and specular components, making them darker and producing shadows in the image as a result.

We also use the SSAO output to improve the ambient light term as described before, multipliying the ambient term of each fragment by the SSAO value we computed for it, reducing the strength of the ambient light for pixels that are surrounded by nearby geometry.

The lighting pass is also where we put bump mapping to use. Bump mapping provides more detailed information about surface normals, which the lighting pass uses to simulate more complex lighting interactions with mesh surfaces, producing significantly enhanced results, as I showcased earlier in this post.

After combining all this information, the lighting pass produces an output like this. Compare it with the GBuffer diffuse texture to see all the stuff that

this pass is putting together:



Lighting pass output

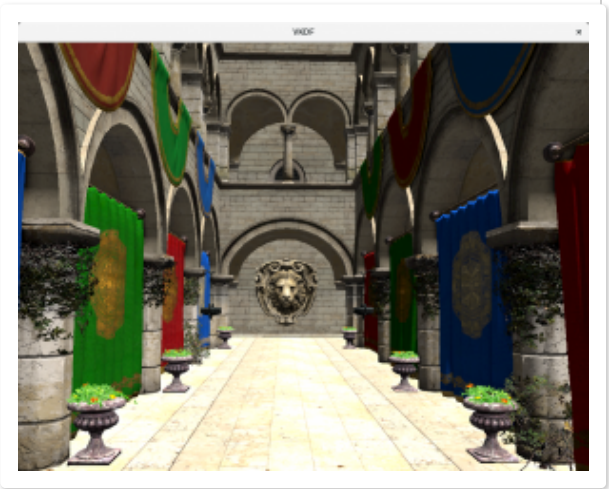
Step 7: Tone mapping

After the lighting pass we run a number of post-processing passes, of which tone mapping is the first one. The idea behind tone mapping is this: normally, shader color outputs are limited to the range [0, 1], which puts a hard cap on our lighting calculations. Specifically, it means that when our light contributions to a particular pixel go beyond 1.0 in any color component, they get clamped, which can distort the resulting color in unrealistic ways, specially when this happens during intermediate lighting calculations (since the deviation from the physically correct color is then used as input to more computations, which then build on that error).

To work around this we do our lighting calculations in High Dynamic Range (HDR) which allows us to produce color values with components larger than 1.0, and then we run a tone mapping pass to re-map the result to the [0, 1] range when we are done with the lighting calculations and we are ready for display.

The nice thing about tone mapping is that it gives the developer control over how that mapping happens, allowing us to decide if we are interested in preserving more detail in the darker or brighter areas of the scene.

In this particular demo, I used HDR rendering to ramp up the intensity of the sun light beyond what I could have represented otherwise. Without tone mapping this would lead to unrealistic lighting in areas with strong light reflections, since would exceed the 1.0 per-color-component cap and lead to pure white colors as result, losing the color detail from the original textures. This effect can be observed in the following pictures if you look at the lit area of the floor. Notice how the tone-mapped picture better retains the detail of the floor texture while in the non tone-mapped version the floor seems to be over-exposed to light and large parts of it just become white as a result (shadow mapping has been disabled to better showcase the effects of tone-mapping on the floor):



Tone mapping disabled

Tone mapping enabled

Step 8: Screen Space Reflections (SSR)

The material used to render the floor is reflective, which means that we can see the reflections of the surrounding environment on it.

There are various ways to capture reflections, each with their own set of pros and cons. When I implemented my OpenGL terrain rendering demo I implemented water reflections using “Planar Reflections”, which produce very accurate results at the expense of requiring to re-render the scene with the camera facing in the same direction as the reflection. Although this can be done at a lower resolution, it is still quite expensive and cumbersome to setup (for example, you would need to run an additional culling pass), and you also need to consider that we need to do this for each planar surface you want to apply reflections on, so it doesn’t scale very well. In this demo, although it is not visible in the reference screenshot, I am capturing reflections from the floor sections of both stories of the atrium, so the Planar Reflections approach might have required me to render twice when fragments of both sections are visible (admittedly, not very often, but not impossible with the free camera).

So in this particular case I decided to experiment with a different technique that has become quite popular, despite its many shortcomings, because it is a lot faster: Screen Space Reflections.

As all screen-space techniques, the technique uses information already present in the screen to capture the reflection information, so we don’t have to render again from a different perspective. This leads to a number

of limitations that can produce fairly visible artifacts, specially when there is dynamic geometry involved. Nevertheless, in my particular case I don't have any dynamic geometry, at least not yet, so while the artifacts are there they are not quite as distracting. I won't go into the details of the artifacts introduced with SSR here, but for those interested, here is a good discussion.

I should mention that my take on this is fairly basic and doesn't implement relevant features such as the Hierarchical Z Buffer optimization (HZB) discussed here.

The technique has 3 steps: capturing reflections, applying roughness material properties and alpha blending:

Capturing reflections

I only implemented support for SSR in the deferred path, since like in the case of SSAO (and more generally all screen-space algorithms), deferred rendering is the best match since we are already capturing screen-space information in the GBuffer.

The first stage for this requires to have means to identify fragments that need reflection information. In our case, the floor fragments. What I did for this is to capture the reflectiveness of the material of each fragment in the screen during the GBuffer pass. This is a single floating-point component (in the 0-1 range). A value of 0 means that the material is not reflective and the SSR pass will just ignore it. A value of 1 means that the fragment is 100% reflective, so its color value will be solely the reflection color. Values in between allow us to control the strength of the reflection for each fragment with a reflective material in the scene.

One small note on the GBuffer storage: because this is a single floating-point value, we don’t necessarily need an extra attachment in the GBuffer (which would have some performance penalty), instead we can just put this in the alpha component of the diffuse color, since we were not using it (the Intel Mesa driver doesn’t support rendering to RGB textures yet, so since we are limited to RGBA we might as well put it to good use).

Besides capturing which fragments are reflective, we can also store another piece of information relevant to the reflection computations: the material’s roughness. This is another scalar value indicating how much blurring we want to apply to the resulting reflection: smooth metal-like surfaces can have very sharp reflections but with rougher materials that have not smooth surfaces we may want the reflections to look a bit blurry, to better represent these imperfections.

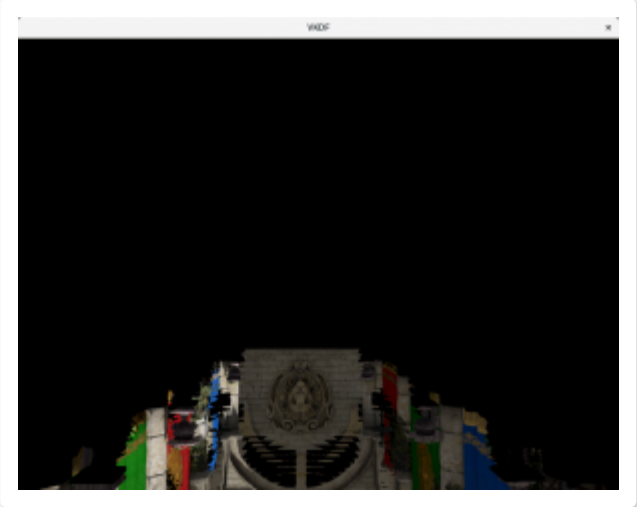
Besides the reflection and roughness information, to capture screen-space reflections we will need access to the output of the previous pass (tone mapping) from which we will retrieve the color information of our reflection points, the normals that we stored in the GBuffer (to compute reflection directions for each fragment in the floor sections) and the depth buffer (from the depth-prepass), so we can check for reflection collisions.

The technique goes like this: for each fragment that is reflective, we compute the direction of the reflection using its normal (from the GBuffer) and the view vector (from the camera and the fragment position). Once we have this direction, we execute a ray marching from the fragment position, in the direction of the reflection. For each point we generate, we take the screen-space X and Y coordinates and use them to retrieve the Z-buffer depth for that pixel in the scene. If the depth buffer value is smaller than our sample’s it means that we have moved past foreground geometry and we stop the process. If we got to this point, then we can do a binary

search to pin-point the exact location where the collision with the foreground geometry happens, which will give us the screen-space X and Y coordinates of the reflection point. Once we have that we only need to sample the original scene (the output from the tone mapping pass) at that location to retrieve the reflection color.

As discussed earlier, the technique has numerous caveats, which we need to address in one way or another and maybe adapt to the characteristics of different scenes so we can obtain the best results in each case.

The output of this pass is a color texture where we store the reflection colors for each fragment that has a reflective material:



Reflection texture

Naturally, the image above only shows reflection data for the pixels in the floor, since those are the only ones with a reflective material attached. It is immediately obvious that some pixels lack reflection color though, this is due to the various limitations of the screen-space technique that are discussed in the blog post I linked above.

Because the reflections will be alpha-blended with the original image, we use the reflectiveness that we stored in the GBuffer as the base for the

alpha component of the reflection color as well (there are other aspects that can contribute to the alpha component too, but I won't go into that here), so the image above, although not visible in the screenshot, has a valid alpha channel.

Considering material roughness

Once we have captured the reflection image, the next step is to apply the material roughness settings. We can accomplish this with a simple box filter based on the roughness of each fragment: the larger the roughness, the larger the box filter we apply and the blurrier the reflection we get as a result. Because we store roughness for each fragment in the GBuffer, we can have multiple reflective materials with different roughness settings if we want. In this case, we just have one material for the floor though.

Alpha blending

Finally, we use alpha blending to incorporate the reflection onto the original image (the output from the tone mapping) ot incorporate the reflections to the final rendering:

SSR output

Step 9: Anti-aliasing (FXAA)

So far we have been neglecting anti-aliasing. Because we are doing deferred rendering Multi-Sample Anti-Aliasing (MSAA) is not an option: MSAA happens at rasterization time, which in a deferred renderer occurs before our lighting pass (specifically, when we generate the GBuffer), so it cannot account for the important effects that the lighting pass has on the resulting image, and therefore, on the eventual aliasing that we need to correct. This is why deferred renderers usually do anti-aliasing via post-processing.

In this demo I have implemented a well-known anti-aliasing post-processing pass known as Fast Approximate Anti Aliasing (FXAA). The technique attempts to identify strong contrast across neighboring pixels in the image to identify edges and then smooth them out using linear filtering. Here is the final result which matches the one I included as reference at the top of this post:

Anti-aliased output

The image above shows the results of the anti-aliasing pass. Compare that with the output of the SSR pass. You can see how this pass has effectively removed the jaggies observed in the cloths hanging from the upper floor for example.

Unlike MSAA, which acts on geometry edges only, FXAA works on all pixels, so it can also smooth out edges produced by shaders or textures. Whether that is something we want to do or not may depend on the scene. Here we can see this happening on the foreground column on the left, where some of the imperfections of the stone are slightly smoothed out by the FXAA pass.

Conclusions and source code

So that's all, congratulations if you managed to read this far! In the past I have found articles that did frame analysis like this quite interesting so it's been fun writing one myself and I only hope that this was interesting to someone else.

This demo has been implemented in Vulkan and includes a number of configurable parameters that can be used to tweak performance and quality. The work-in-progress source code is available here, but beware that I have only tested this on Intel, since that is the only hardware I have available, so you may find issues if you run this on other GPUs. If that happens, let me know in the comments and I might be able to provide fixes at some point.

Intel Mesa Vulkan driver now supports shaderInt16

MAY 7, 2018

The Vulkan specification includes a number of *optional* features that drivers may or may not support, as described in chapter 30.1 Features. Application developers can query the driver for supported features via *vkGetPhysicalDeviceFeatures()* and then activate the subset they need in the *pEnabledFeatures* field of the *VkDeviceCreateInfo* structure passed at device creation time.

In the last few weeks I have been spending some time, together with my colleague Chema, adding support for one of these features in *Anvil*, the Intel Vulkan driver in Mesa, called **shaderInt16**, which we landed in Mesa master last week. This is an **optional feature available since Vulkan 1.0**. From the spec:

> *shaderInt16 specifies whether 16-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 16-bit integer types must not be used in shader code. This also specifies whether shader modules can declare the Int16 capability.*

It is probably relevant to highlight that this Vulkan capability also requires the **SPIR-V Int16 capability**, which basically means that the driver's

SPIR-V compiler backend can actually digest SPIR-V shaders that declare and use 16-bit integers, and which is really the core of the functionality exposed by the Vulkan feature.

Ideally, *shaderInt16* would **increase the overall throughput of integer operations** in shaders, leading to better performance when you don't need a full 32-bit range. It may also provide **better overall register usage** since you need less register space to store your integer data during shader execution. It is important to remark, however, that not all hardware platforms (Intel or otherwise) may have native support for all possible types of 16-bit operations, and thus, some of them might still need to run in 32-bit (which requires injecting type conversion instructions in the shader code). For Intel platforms, this is the case for operations associated with integer division.

From the point of view of the driver, this is the first time that we generally exercise lower bit-size data types in the driver compiler backend, so if you find any bugs in the implementation, please file bug reports in bugzilla!

Speaking of *shaderInt16*, I think it is worth mentioning its interactions with other Vulkan functionality that we implemented in the past: the Vulkan 1.0 **VK_KHR_16bit_storage** extension (which has been promoted to core in Vulkan 1.1). From the spec:

> *The VK_KHR_16bit_storage extension allows use of 16-bit types in shader input and output interfaces, and push constant blocks. This extension introduces several new optional features which map to SPIR-V capabilities and allow access to 16-bit data in Block-decorated objects in the Uniform and the StorageBuffer storage classes, and objects in the PushConstant storage class. This extension allows 16-bit variables to be declared and used as user-*

> *defined shader inputs and outputs but does not change location assignment and component assignment rules.*

While the *shaderInt16* capability provides the means to operate with 16-bit integers inside a shader, the *VK_KHR_16bit_storage* extension provides developers with the means to also feed shaders with 16-bit integer (and also floating point) input data, such as Uniform/Storage Buffer Objects or Push Constants, from the applications side, plus, it also gives the opportunity for linked shader stages in a graphics pipeline to consume 16-bit shader inputs and produce 16-bit shader outputs.

VK_KHR_16bit_storage and *shaderInt16* should be seen as two parts of a whole, each one addressing one part of a larger problem: *VK_KHR_16bit_storage* can help reduce memory bandwith for Uniform and Storage Buffer data accessed from the shaders and / or optimize Push Constant space, of which there are only a few bytes available, making it a precious shader resource, but without *shaderInt16*, shaders that are fed 16-bit input data are still required to convert this data to 32-bit internally for operation (and then back again to 16-bit for output if needed). Likewise, shaders that use *shaderInt16* without *VK_KHR_16bit_storage* can only operate with 16-bit data that is generated inside the shader, which largely limits its usage. Both together, however, give you the complete functionality.

Conclusions

We are very happy to continue expanding the feature set supported in Anvil and we look forward to seeing application developers making good

use of **shaderInt16** in Vulkan to improve shader performance. As noted above, this is the first time that we fully enable the shader compiler backend to do general purpose operations on lower bit-size data types and there might be things that we can still improve or optimize. If you hit any issues with the implementation, please contact us and / or file bug reports so we can continue to improve the implementation.