Hybrid rendering
using Radeon Rays

Stijn Billiet

Graduation work 2018-19
Digital Arts And Entertainment
howest.be

Contents

Chapter 1

Abstract

This paper was written in an effort to elaborate on hybrid solutions between modern day rasterization engines and powerful ray-tracing ones and to explain rather thoroughly how one could implement this.
TODO methods TODO results

Chapter 2

Introduction

Ray-tracing is usually used in VFX productions to create the most accurate lighting effects, this doesn't come for free though seeing crisp ray-traced images require countless hours of rendering. With the rise of new ray-tracing capable hardware (e.g. Volta architecture NVIDIA) things are about to change, developers are already putting ray-tracing in some parts of their pipelines, they must however rely on optimization (e.g. De-noising) to keep up with the performance demands. So for the time being we will see a lot of hybrid solutions, but we should see full ray traced engines by the time that our hardware is capable enough.

That being said I would like to clarify that rasterization engines aren't bad, however they are very rough approximations and rely on specific techniques to achieve lighting effects sometimes at the cost of artifacts. We must keep in mind though rasterization engines are here to stay for a while longer due to their immense performance, and not to forget they have been around for a while, their maturity surely pays off seeing what we've already achieved with rasterization (e.g. The Witcher 3: Wild Hunt).

As the subtitle suggests we will use Radeon Rays for our ray-tracing algorithm, however there are other options out there like Microsoft's DirectX Raytracing. Nevertheless the topics outlined should still hold up when implementing an other API, only the Radeon Rays implementation details would be an exception to that rule. The same essentialy goes for our graphics card specification, OpenGL was used seeing that would be the logical combination, but the information should translate over nicely to other API's.

Chapter 3

Research

This extract should give you a rather thorough overview of the technologies used, however some knowledge regarding graphics programming and/or matrix mathematics is advised.

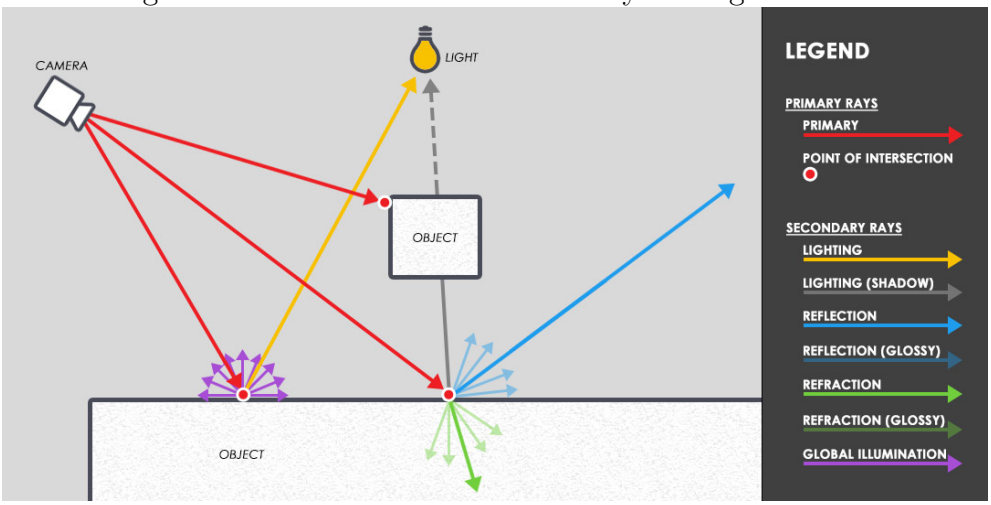
3.1 Rendering techniques

Over the past century many rendering algorithms have been researched/employed. They are all trying to find an answer to the problem of limited rendering capabilities. Seeing that our modern model of light would be completely impractical when it comes to (high speed) simulations, meaning it would take an obscene amount of time to calculate.

Therefore people have come up with ways to more efficiently model the phenomenon of light. The most prominent models being: ray tracing (ray casting) and rasterization. The aforementioned models will be covered in the following subsections.

3.1.1 Raytracing

Figure 3.1: Illustration of different ray-casting solutions



Definition

In short ray-tracing is a rendering technique that can produce lifelike lighting effects. A ray-tracing algorithm essentially achieves this by sending out rays for every pixel of our framebuffer (camera) towards the corresponding scene geometry. Based on the technique we can send out additional bounces (secondary rays). It then basically backtracks that chain of light-rays to accumulate material/light information to then calculate the final pixel color.

Ray casting

As addressed before ray-tracing essentially probes material and light information through the use of rays. There are 2 sets of rays, we've got primary rays and secondary rays. Primary rays are the first rays being sent out, the rays from camera towards the scene geometry. These primary rays are essentially doing the same as with rasterization in the sense that they only probe at material information. It becomes interesting however when we cast our secondary rays, these secondary rays have their origins at the interception point of the primary rays and the scene geometry. Their direction on the other hand are totally dependent on the technique that is employed, meaning that we will use different techniques to bounce/direct them based on the effect required (e.g. shadows, reflections, etc.).

3.1.2 Rasterization

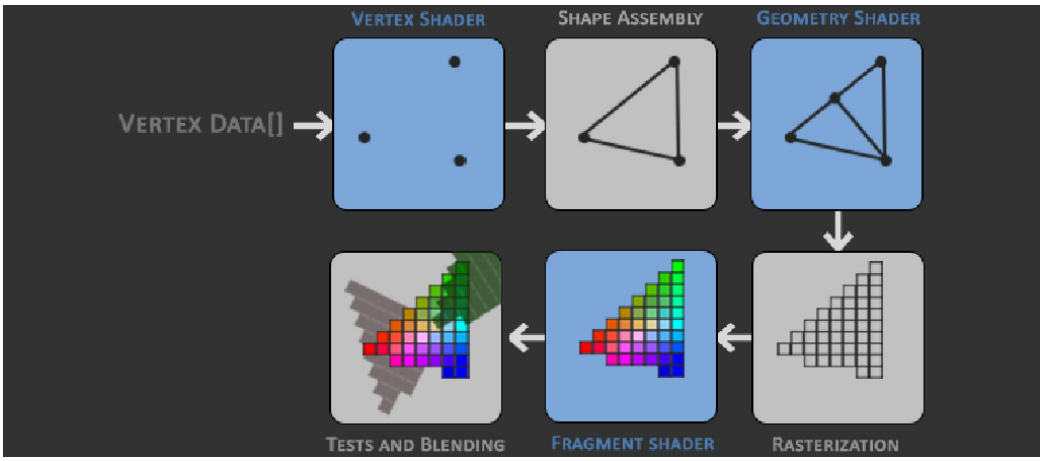


Figure 3.2: Rasterization graphics pipeline.

Definition

Rasterization is an algorithm designed to display three-dimensional objects on a two-dimensional screen. It's very fast seeing this algorithm is parallel in nature and can thus make full use of the GPU. The results of rasterization have gotten very good over the years, very convincing at relatively low compute performance. However there are certain limitations to this algorithm, seeing its main goal is to get graphics in screen space.

Graphics pipeline

With rasterization objects on screen are created from meshes. A mesh is a collection of three-dimensional points a.k.a. vertices. These vertices are the main resource needed for rendering, these go through what's called the rendering pipeline to then be converted into screen pixels. The rendering pipeline or more formerly known as the graphics pipeline acts as a conceptual model that describes what steps a graphics system needs to performs to be able to render aforementioned meshes to a 2D screen.

Vertex shader

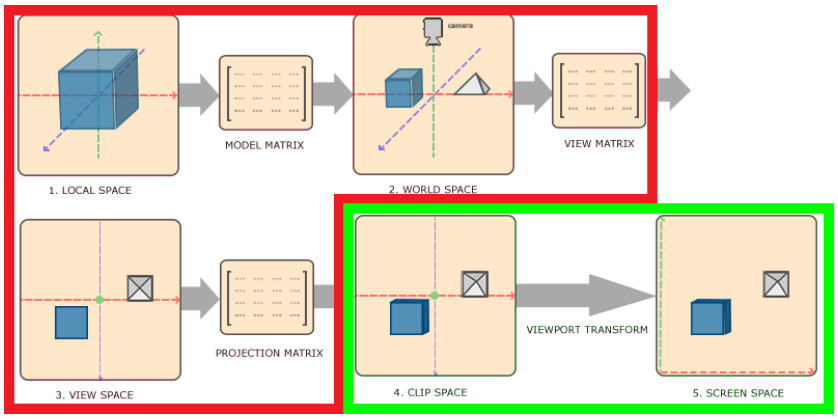


Figure 3.3: Model space to screen space.

The first stage is the vertex shader which essentially transforms the vertices to their correct positions, the vertices have to be transformed seeing that they are defined in their own local transform space(vertices positions are defined relative to their own center point). The vertices are defined this way as a means to make it easier for modelers, they quite simply have to define the vertices relative to the modeling software's gizmo.

Shape assembly

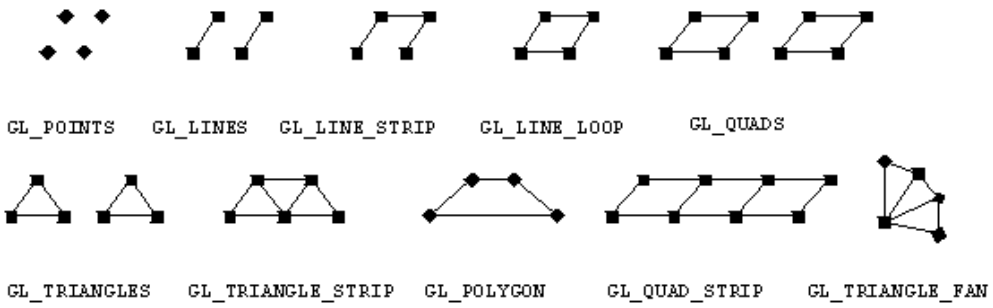


Figure 3.4: List of possible OpenGL primitives.

The second stage is the shape assembly which is in charge of creating primitives out of these verts. The developers have to specify the wanted primitive (e.g. triangle) the GPU then uses that information to interpret the a collection of vertices as said primitive.

Geometry shader

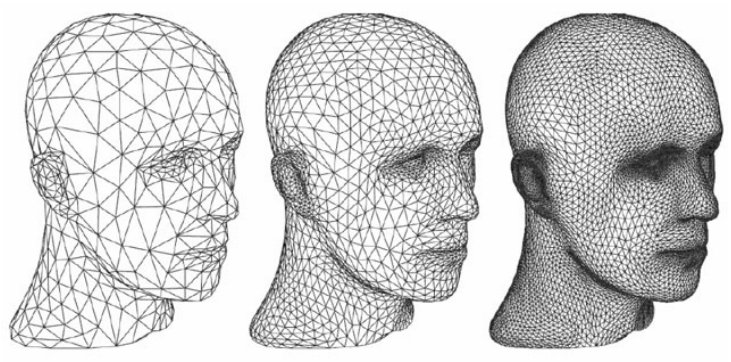
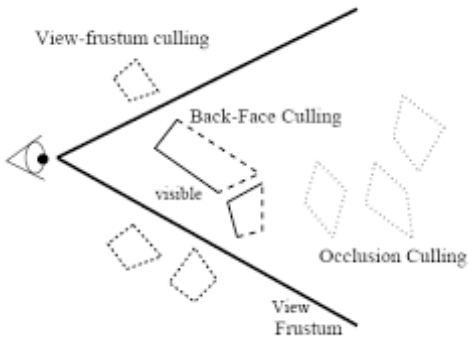


Figure 3.5: Purpose of geometry shader.

The third stage is an optional geometry shader, this stage won't be covered in depth, but it is essentially in charge of creating additional geometry (extra vertices) at runtime.

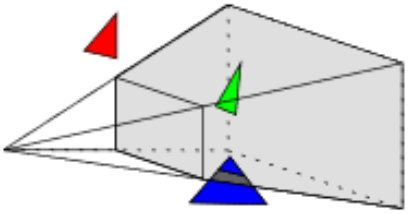
Rasterization stage

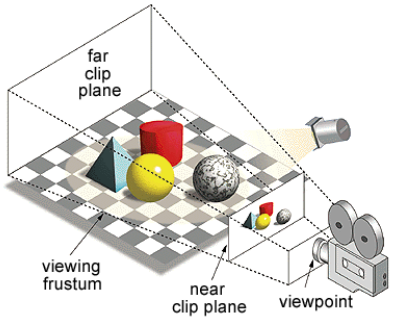
The fourth stage is rasterization, this is where the algorithm gets its name. The rasterizer is in charge of a not only rasterization, but also homogeneous divide, clipping, culling and viewport transform.



The first task of the rasterizer is primitive culling. The shapes that made it to the rasterizer are checked based on their normals. The normal of a primitive/vertex tells us if the face is facing away from us or not. This simple operation can be demonstrated using the dotproduct, however this does get handled by the specifications (e.g. DirectX/OpenGL) rasterizer. In its essence the calculation will translate to if the dotproduct is negative then the face is facing away, if it's positive then it's facing toward us.

The second task is the clipping stage, in this stage we clip triangles that lie (partly) outside of the view frustum. This effectively removes those triangles and prevents them from being projected/rendered.



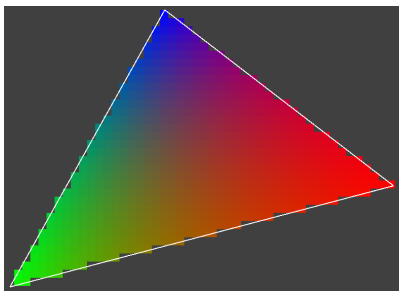


The third task of the rasterizer is the homogeneous divide. The previous stages were in charge of transforming the vertices from model space into clipping space, this is done by multiplying the world, view and projection matrix with the vertices this yields a valid projection but does not account for depth information (f.e. objects that are nearby will appear the same as objects from afar) this is where the z-component comes in, seeing the z stores our depth information. Before we apply the projectionmatrix we store the z-component as an additional w component.

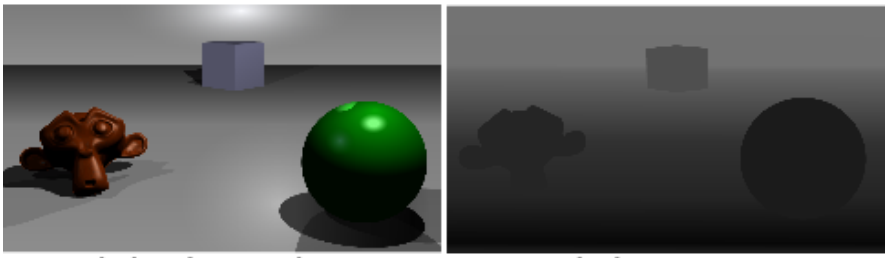
The homogeneous divide is just a specific term for converting the homogeneous coordinate back to w=1 meaning we put the vertices in their normalized device coordinates (in the -1 to 1 range) effectively scaling all objects to their rightful dimensions.

The fourth stage is the viewport transform. In the viewport transform we essentially translate all the vertex positions that are now in normalized device coordinates (NDC) and transform them to window space. These are then the coordinates that are being rasterized to the output image. This process is outlined in figure 3.3 (part with green outline).

The final task pixelisation (rasterisation) is the process of converting continuous primitives into discrete pixel fragments. The rasterizer does so by interpolating between vertex attributes (data that vertex holds) like color, normals, texture coordinates and positions. It then launches a fragment shader for every pixel fragment with the information just calculated.

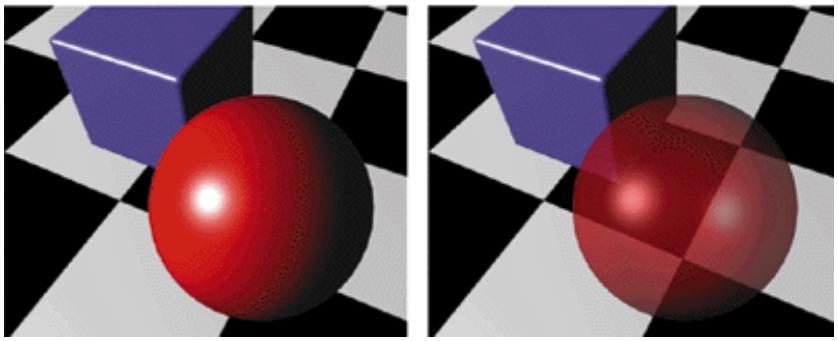


Fragment shader



The fifth stage, the fragment stage is a shader stage that will process all pixel fragments processed by the rasterizer. Depending on the technique we collapse all information into a set of colors (color buffer) and a single depth value (depth buffer), additionally we can augment this by writing away a stencil value (stencil buffer).

Test and blending stage



Finally we've got the testing and blending stage. Up until this point "pixels" where addressed as pixel fragments, this is in-fact a clear distinction to make seeing that pixel fragments are not guaranteed to show up on screen, they could be discarded in the shaders or they could be transparent. If there are transparent objects in the scene we have to keep both pixels, those from the object behind it and the transparent object, we can then simply blend those pixels together to receive the final pixel colour. This blending operation is in-fact quite costly seeing we need to make sure the objects appear in the right order and we've got the load of additional pixels on our hands. A more optimized technique can be used for objects that are fully transparent however, here we can employ alpha testing where we simply discard those pixel fragments in the fragment shader.

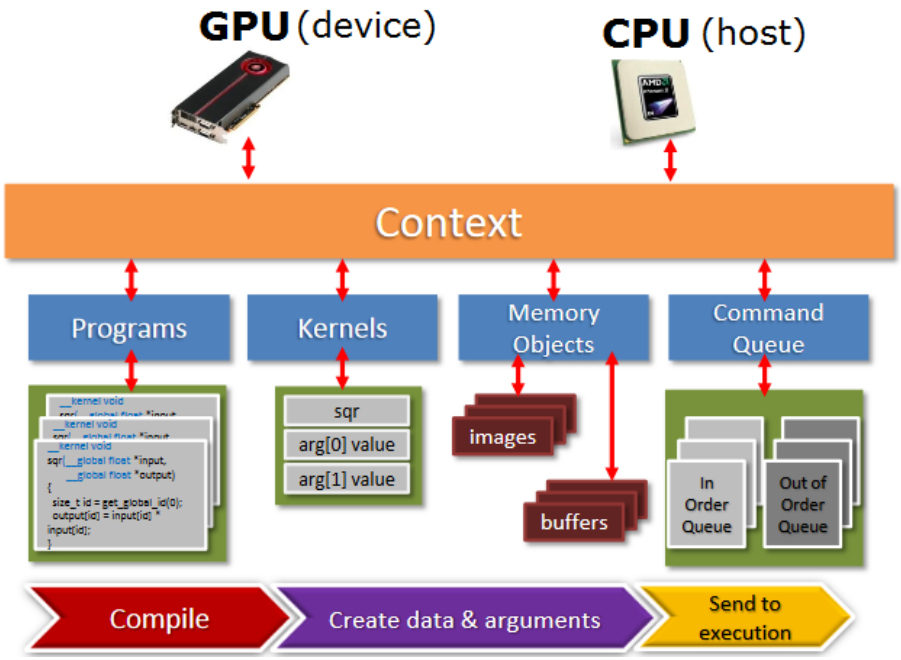
3.2 API's and tools

The main API's that have been researched were Radeon Rays, OpenCL and OpenGL. However to speed up the development process tools were employed. The main part of the tools were tailored towards the OpenGL pipeline (e.g. model loading, texture loading, etc.), but some small frameworks were also employed to speed up the raytracer development (e.g. CLW). Keep in mind however that these tools are not necessary seeing it could all be made on your own, but it sure helps to have these around.

3.2.1 Raytracing

OpenCL

Open Computing Language is a framework for writing programs that execute across compute devices. This includes among other hardware accelerators, the central processing unit (CPU) and the graphics processing unit (GPU). These aforementioned programs are called kernels and are also written on a pseudo c-language. OpenCL provides the programmer of a standard interface to perform parallel computing tasks. The programmer has to set up the OpenCL execution pipeline and write the kernels.



Under this section you'll find a quick rundown of aforementioned execution model.

- **Platform**

  The platform that we are running this code on, so which vendor (e.g. Nvidia).

- **Device**

  What device are we going to run this on could be anything from CPU, GPU to APU.

- **Context**

  This is the runtime interface between the host program (run on the CPU) and the device (CPU/GPU/etc.). The context is in charge of managing all the OpenCL resources so programs, kernels, command queues and buffers.

- **Program**

  The entire OpenCL program, this could be one or more kernels/device functions bundled together into a single program file.

- **Kernel**

  The starting point/building blocks of a OpenCL program. Kernels are called from the CPU, they provide the basic units of executable code that run on a OpenCL device. The kernels are preceded with the keyword "__kernel"

- **Command queue**

  The command queue allows the kernels execution commands to be sent to the device. This effectively queues them for execution, this enqueuing can be in-order or out-of-order.

- **Memory objects**

  The memory objects are what is provided to the kernels. Kernels do accept basic types like floats, ints, bools etc. and self defined types like Ray, Intersection etc, but if there is a large buffer of those it's kept in a Memory buffer (e.g. images).

A large part of the parallelization control is in the hands of the programmer. Upon launching a kernel the programmer is in control of how many work items need to be handled per compute unit, this effectively bundles them in a workgroup. Each workgroup gets its own compute unit, a work group can exist out of one or multiple work items. This degree of freedom is further reinforced by the level of control over different memory levels that can be employed to speed up development.

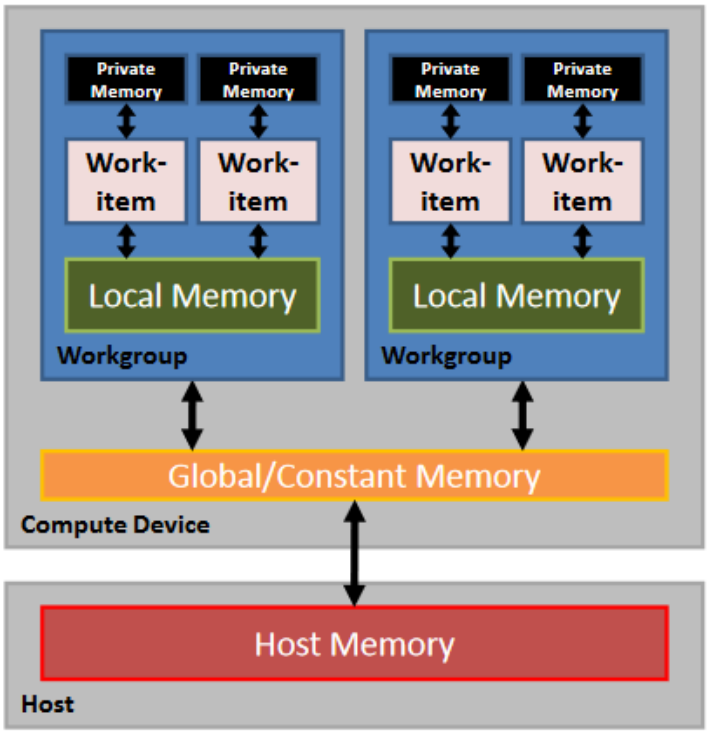


Figure 3.6: Illustration of the memory execution model.

Under this section is a roundup of the OpenCL memory model (GPU memory model).

- **Global memory**

  Global memory is very similar to RAM, it is the largest memory buffer but also the slowest. Global memory can be written to / read from by all work items (or threads).

- **Constant memory**

  A small chunk of global memory on the device, can be read from by all work items, but can not be written to. However the host application has full read/write access to this memory pool. Constant memory is a bit faster than

- **Local memory**

  The local memory is a shared memory pool among work items in the same work group, similar to cache on the CPU. Local memory allows work items to share their results with work items belonging in the same work group. Local memory is up to 100 times faster than global memory.

- **Private memory**

  The fastest type of memory, similar to registers on the cpu. Each work item (thread) has a tiny buffer of private memory to store intermediate results in.

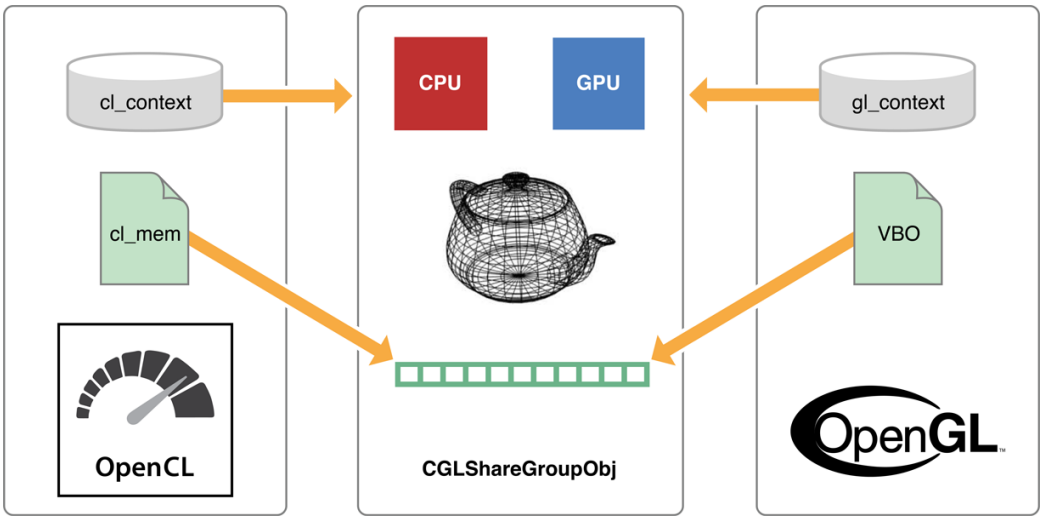


Figure 3.7: OpenCL/OpenGL interop context

A great thing about this control over memory is that it empowers the developer to make use of interoperation contexts. This can be used to achieve zero-copy architectures where memory objects can be shared by multiple compatible api's. A prime example of this is an OpenCL/OpenGL interop context. Such a context allows the graphics programmers to send screen buffers, textures and openGL objects to the kernels for further processing without the need to copy them, further enabling them to use a kernel in the pipeline everywhere they want.

While this is great, there are still some limitations that have to be set in place to enable this kind of behaviour. For one the target device needs to support this sharegroup behaviour and for seconds shared memory can cause issues seeing that both contexts don't know about each others existance. All of this can be alliviated by setting blocking calls (e.g. GPU fences) in place, with these acting as a sort of failsafe that makes sure the object is done reading from/ writing to by the target context. This can account for slowdowns seeing that the CPU is needed to help the 2 parties synchronize, luckily for the programmer there is hope that said GPU supports the ARB_cl_event extension.3 The event extension allows the two api's to communicate through event management (e.g. wait for event to begin) ensuring that they don't need to wait a few ticks of CPU-time to synchronize.

Radeon Rays

Radeon-Rays is a GPU intersection library(raytracing framework), was developed by AMD and it is featured in their AMD ProRenderer (raytraced renderer). Radeon Rays exposes a well defined C++ API for scene construction and performing asynchronous ray intersection queries.

Radeon-Rays employs OpenCL kernels to do all these calculations and to speed up the ray intersection they order their objects in bounding volume hierarchies. Seeing they use OpenCL to speed up their computations by sending it of to the GPU, this library will run on almost any GPU (meaning it will also work on Nvidia/Intel products).

3.2.2 Rasterization

OpenGL

Loading libraries

OpenGL is a specification that has many different versions, implementations and features. The hardware manufacturers have to make sure their hardware/drivers provides the graphics programmers of those features. However when working with immensely different target platforms, ranging from old hardware to new and from obscure cards to bleeding edge ones the graphics programmers need some kind of way to determine what's possible on that specific piece of hardware. That is where loading libraries come in, loading libraries essentially provide the programmer with function pointers to OpenGL agnostic functions at runtime, meaning they provide us the memory addresses of those functions seeing they depend from hardware to hardware.

An additional task of the loading libraries is extension loading where programmers can use this API to ask if a specific extension/feature is supported on the hardware. Loading libraries provide their own OpenGL implementations (e.g. 'gl.h'), but your own implementation can be provided by making sure it is set as the last include in your build order. A word of caution however is that some features might be declared in your specification that the loading library has no knowledge of, but this is an edge case seeing that most people won't thread that route.

There are numerous libraries out there that do just that, libraries like GLEW, GL3W, Glad, Glee and many more. All libraries that exist out there all deliver on the same promise, meaning that the choice of library really boils down to personal preference.

GLM

When it comes to math libraries that plug in nicely with an OpenGL renderer there's really only one good option namely GLM. OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL).

GLM exposes a well defined interface for everything from calculus to quaternion rotations (e.g. vector operations, matrix calculations).

Window creation

A window allows the rendering specification to offload its framebuffer to the viewport, it also acts as a layer between the operating system and the game when it comes to input etc. The creation of a window is entirely dependent on the operating system it's creating the window for. For example when creating a window on linux the X window system should be addressed, whereas on Windows you would need to hook on to the WinMain function and use the WndProc callback to fetch any input. Therefore it is much smarter to use a library that hides all this operating system agnostic stuff in an abstraction layer.

SDL, SFML and GLFW are all low level libraries that handle multimedia indirection. They all deliver the same features in some shape or form. The choice really boils down to maturity of the library, documentation, usefulness of its features and frequency of the updates (if any).

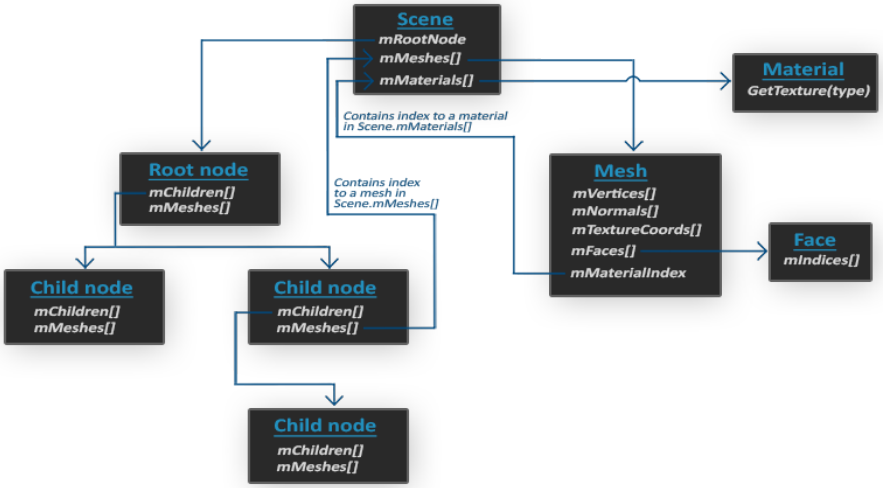
Texture loading

A texture is a 2D (could be 3D or 1D) image packed with color data that is used to feed into the graphics pipeline to texturize geometry. Textures can be stored in a dozen of file formats, each with their own structure to store the pixel data. Textures are stored in a binary format and can therefore be parsed by a binary reader. The problem however with doing textureloading is that every format is different so the programmer would be in charge of writing a dozen different loaders to support the most basic of texture types. Seeing that this is rather cumbersome most people would opt for a library that does this for them. Again there are quite a lot of libraries out there that do this.

SOIL, GLi, DevIL, FreeImage and stb_image are all texture loading libraries that promise on loading the most prevalent of types.

Model loading

A 3D file format is a specification to store information about 3D models in the form of plain text or as binary data. The main purpose of such format is to encode 3D model's geometry, material information and animation data. Seeing there are loads of formats out there, creating your own model loading library has become something of a novelty. Fortunately there are loads of libraries out there that load about any format you throw at it.



Asset import or AssImp for short is a model loading library designed to load as many formats as possible and to unify them into a searchable structure to be used by the host application. It structures the entire scene into a scene object, which consecutively holds a rootnode and a collection of meshes and materials. That rootnote then has leads to other nodes that all hold the same information namely a list of indices to the corresponding meshes and a next connection. This leads to a clear distinction as to what are seperate objects in the global model file, which can then later be used by the game-engine to ensure perfect relationships between aforementioned meshes.

3.3 Hybrid rendering approach

The different rendering techniques have been covered, now onto a hybrid solution where rasterization and raytracing are combined to achieve a new pipeline. Most hybrid approaches replace the first stage of ray-tracing, which is sending out our primary rays to retrieve surface information, by rasterization. This effectively reduces the amount of rays by half. Ray-tracing comes into play for sending out our reflected rays, which could be anything from sending out new primary rays to secondary (shadow) rays. This introduction of ray-tracing essentially brings us world information whereas normally you'd only have surface information. To be able to augment rasterization with raytracing we will need to postpone our lighting effects to a later stage.

3.3.1 Raytracing in rasterization

Forward vs deferred rendering

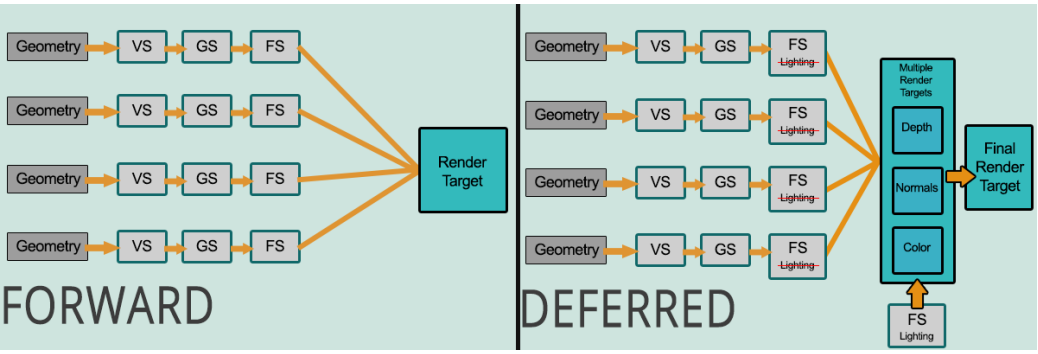


Figure 3.8: Difference between forward and deferred pipeline.

In rasterization there are three main lighting pipelines. The default pipeline is forward rendering this is the pipeline that's discussed in the previous chapters. The main difference between both pipelines however is in the way they handle the lighting effects. In forward rendering the lighting is applied per object in the fragment shader, this means that even the transparent objects, whose pixel fragments are not sure to end up in the final image, are also being handled. This pipeline is fine for default drawing, but starts to fall behind when it comes to lighting performance.

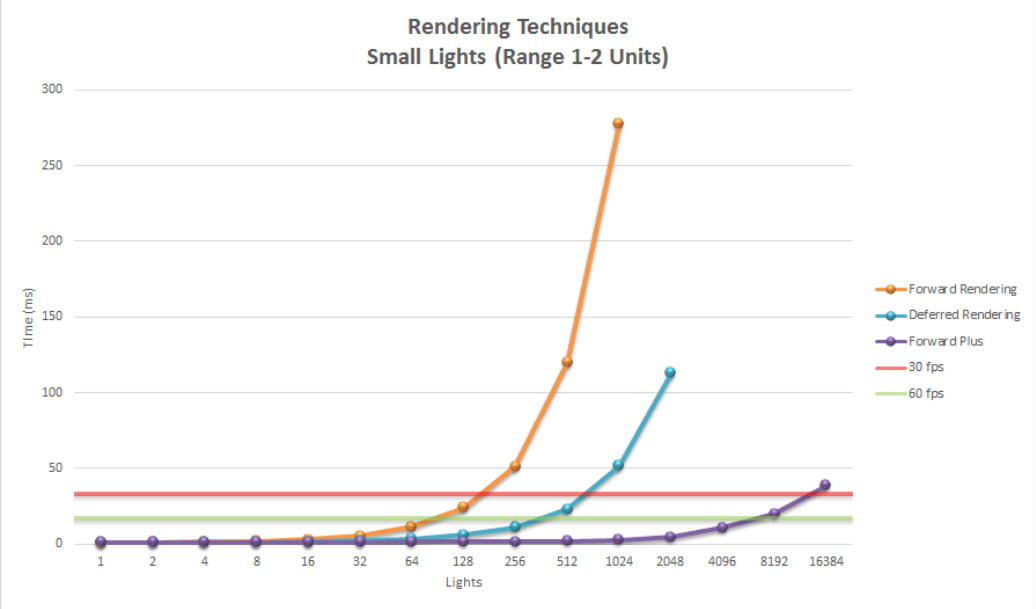


Figure 3.9: Lighting performance comparison between forward and deferred.

The second pipeline however as the name suggests defers from doing any lighting effects until all the objects are rendered in the off-screen g-buffers. This effectively divides the pipeline into two major parts, the fill pass and the lighting pass. In the fill pass we fill up all the off-screen buffers (or G-Buffers) with our pixel in formation. The light pass as the name suggests is then in charge of applying lighting to those pixels. This ensures that we can postpone our lighting to the lightpass enabling us to do additional effects that encompass the whole scene (e.g. raytracing.)

The third rendering pipeline, although the most performant will not be covered with explicit detail. The reason that this technique is not that applicable has primarily to do with the fact that we have no way of doing world space lighting effects. Forward+ is a rendering technique that improves upon regular forward rendering. Forward+ first determines which lights are overlapping which areas of the screen (screen space). During the shading phase that list is then consulted to only account for the lights that are potentially overlapping the current pixel fragment. The forward+ technique consists primarily of three simple passes. In the light culling pass, each light in the scene is sorted into screen space tiles. In the opaque pass, the light list generated from the light culling pass is used to compute the lighting for opaque geometry. The transparent pass is similar to the opaque pass in the sense that the light list is again consulted when shading the transparent pixel fragments.

Raytracing using g-buffers

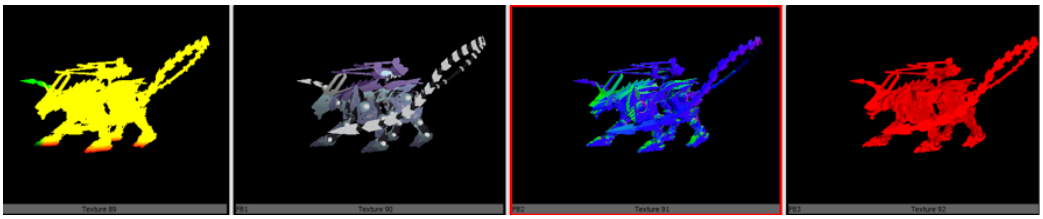


Figure 3.10: Illustration of g-buffers

As touched earlier deferred rendering fills the off screen render buffers (g-buffers) with scene information in its first pass. This effectively enables us to do lighting after the fact by then using that information to construct everything we need to apply our lighting.

The information that is stored in the g-buffers is entirely up to the programmer (e.g. position data, normal data, color data etc.). However to be able to use this in conjunction with ray construction the programmer needs atleast the position data and the normal data. Some people would however argue that the position data isn't necessary and that the position can be calculated by the z-buffer and that would be an entirely correct answer, but for simplicity sake people opt for the position data instead.

With the geometry data now in the g-buffers, that data can now be used to construct the rays. The ray construction can be sped up by doing it on the GPU, seeing that this operation is highly parallel in nature. Essentially what it boils down to is to make sure that meshes have been synchronized with the raytracer, meaning that the raytracer has all the right meshes in its scene to be able to fetch all occlusion/intersections. Then quite simply sample all the position entries and normal entries (for the entire screen) and send out a ray for every pixel, this will in turn correlate to the raytracer api seeing those position entries should correspond with the meshes in the scene.

Once the rays have been constructed the raytracer can now simply query the occlusion/intersections with the scene geometry. That data can then be used to shade the object respectively.

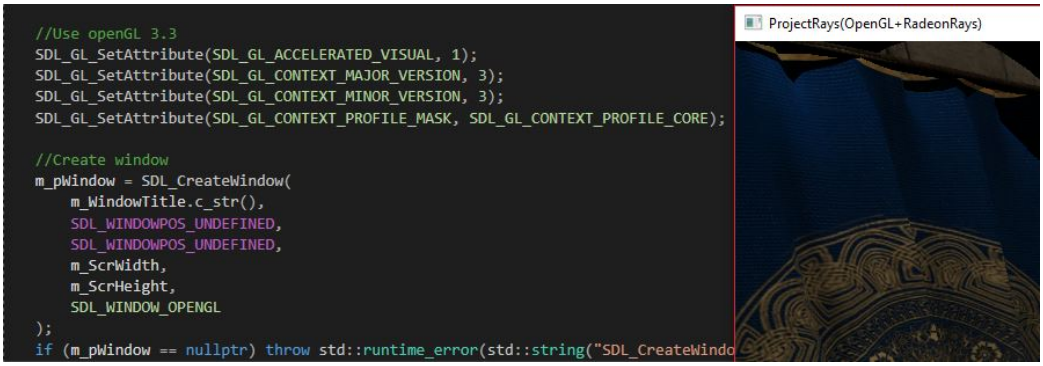
Chapter 4

Case study

In the following chapter an example rendering pipeline will be covered. This pipeline will make use of deferred shading (rasterization) for its main pipeline and raytracing for real-time shadows. The implementation of the shadows acts as an example effect to display that there is gain to be had when replacing that with a raytraced counterpart. The case study will act as a logical guide on how to implement this pipeline yourself, but some situation agnostic details will be left open for interpretation.

4.1 Implementation rasterization

4.1.1 Window creation



The first main part that needs to be implemented to render is a window. As already outlined in the research part there are a lot of libraries out there. For this implementation SDL was chosen, simply for the reason that creating an SDL windowing context with OpenGL is a breeze. SDL expects the programmer to supply what version of OpenGL will be coupled to this window, what rendering mode will be used (e.g. immediate vs core) and the settings of the window. In our case we'll use OpenGL 3.3 seeing it was the first modern implementation of SDL and consecutively the easiest to comprehend, we'll also be using core mode seeing we need vertex buffer abilities. Once the window is in place the programmer is responsible for querying the accompanying OpenGL context.

4.1.2 OpenGL function bindings

Once the window and accompanying context is in place the programmer can move on to the OpenGL loading. For this implementation Glad was used. To get glad to work in this solution the respective package has to be created from their website `https://glad.dav1d.de/` For our purpose I selected the GL 3.3 package in core functionality mode. After the package selection has been done you should receive the source files. These can then be compiled into a solution by making use of CMake, to then consecutively build those into lib files. The final step was to link both the includes and libs into my solution. A simple call to gladLoadGLLoader(SDL_GL_GetProcAddress) later and it's done.

4.1.3 Shader loading

One of the major things that make the graphics pipeline modular is programmable shaders. Shaders are mostly written in a pseudo c-language like GLSL/HLSL. This language is relative to the underlying rendering api that is being used. The rendering api is therefore in charge for the shader compilation. The only obstacle that the host program has to overcome is loading the source from the shader file. In this implementation the default cpp iostream was used, but this can be tailored to the needs of your engine/implementation. Once the shader source has been loaded it needs to be compiled and hooked to an OpenGL object. In this implementation a clear distinction between a shader and a shader program was made, where a shader is seen as the smallest building block and a shader program as a collection of those blocks.

4.1.4 Hello Triangle

With OpenGL and SDL established it is time to test the renderer by doing to famous hello triangle test. The first step for the programmer is to supply the vertices of the triangle, seeing there are still no coordinate systems in place it is ok to define those vertices in normalized device coordinates.

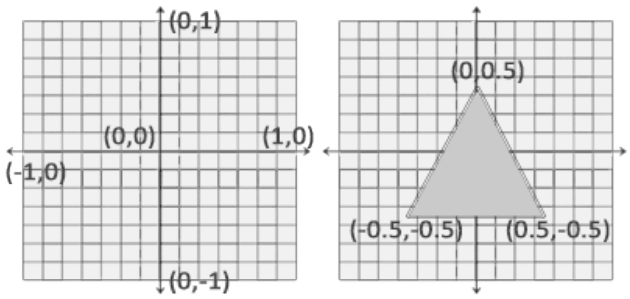
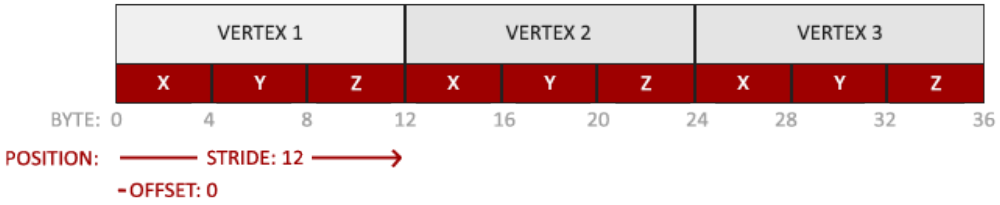


Figure 4.1: Normalized device coordinates for our triangle

Once the vertex data is defined the programmer needs to send it of to the first stage of the rendering pipeline (vertex shader). This means that the programmer should first get those vertices on the GPU, in modern OpenGL this is done by requesting a piece of memory called a vertex buffer object. After binding this buffer and providing it from its buffer data the object should be set in place.

Modern OpenGL requires the graphics programmer to atleast supply his own vertex and fragment shader, this is in contrary to DirectX where a default one is used if none supplied. Our vertex shader will consist of the vertex layout and the passing of our vertices position to the OpenGL system value gl_Position. Under vertex layout the programmer provides how his vertices store the data needed in the shader, this also needs to be done in the host program through the use of glVertexAttributePointer. The aforementioned gl_Position is there to provide a means of communication to specify where the vertex should appear on screen.



Then the final piece of the puzzle is to supply our pipeline of a fragment shader. The fragment shader in itself is quite simple only requiring us to specify the final fragments color, this can of course be any arbitrary value. After shader compilation, binding our shader, binding our vertex buffer and a final call to drawtriangles later a triangle appeared on screen.

4.1.5 Coordinate systems

Although you can render without a proper coordinate system in place, the implementation needed world space coordinates rather than screen space.

So for this implementation a transform component was set in place. This transform component stands in relationship with the GameObject. This relationship allows us to add childs to the GameObject while maintaining the correct local transforms in the component. If the position of a said object was changed we go ahead and rebuild our worldmatrix while keeping in check if the object had a parent.

The next part is the camera component, the camera component is in charge of holding the viewprojectionmatrix. Meaning that it is also in charge of holding the camera position and its settings.

The final part to the puzzle is the mesh draw component. To create a meshdrawcomponent the programmer has to provide it with a mesh filter which is loaded by the model loading utilities mentioned prior. Then the final assembly is to build the worldviewprojectionmatrix (by multiplying the world, view and projection matrices with each other) and send it to the shader.

4.1.6 Texture loading

For texture loading stb_image was used to accelerate this task. stb_image is a very simple library and is include file only, so there is not need to create a library file. stb_image provides the programmer with a very simple api for loading textures and it is up to the programmer to link those to openGL objects.

4.1.7 Model loading

As mentioned prior for this implementation AssImp was used. AssImp provides a very simple interface only requiring us programmers to call the importers read file and to store it into a aiScene object. This object is essentially our top level node that allows us to use this tree as a searchable structure. To keep things easy our implementation does only require us to make 3 functions being: ProcessNode, ProcessMesh and LoadMaterialTextures. Under this section you'll find a quick rundown of what each function does and why.

- ProcessNode as the name suggest goes over the current node and recursively calls the ProcessMesh on its meshes. Once all the meshes within this node have been handled it goes and calls ProcessNode on its children.
- ProcessMesh has three major tasks the first one being fetching all the vertices of the mesh things like: storing position, normal data and texture coordinates. Secondly it makes sure all the indices are being loaded in before finally going over to the materials where it calls the LoadMaterialTextures function on each of its materials.
- LoadMaterialTextures then takes in the material and the type of texture that we are trying to fetch, it then checks the 3D object format for that type of format and stores it respectively. As an added optimization it keeps track of double textures, so it won't load the same texture twice.

4.1.8 Deferred shading

With a standard rendering pipeline in place it was time to go ahead and turn that pipeline into a deferred one. The steps beforehand did also not cover lighting effects for the simple reason that they would need to be re-written once going deferred.

The major implementation difference now is that the programmer needs to define his/her off screen render buffers (e.g. position buffer, normal buffer, etc.). First the programmer has to define what buffers are stored in the gbuffer object (frame buffer object). With the gbuffer object defined it's time to start moving on to the fill shader.

The fill shader is tasked with filling the gbuffers with all the geometry information respectively, so if the implementation uses four buffers the fill shader will be tasked with filling those four buffers. The fill shader is a shader program consisting of two shaders first a simple vertex shader that passes the information to the next stages and then a fragment shader who does all the heavy lifting and fills all the buffers. Then all the geometry gets rendered using this shader. After the bulk of the rendering is done we unbind the gbuffers (that are now filled with geometry information) and we move on to the lightpass.

The light pass shader is in charge of lighting and rendering the final full screen quad. It speaks for itself that before going into this lightpass we need to make sure the g-buffer is no longer bound as the active rendertarget and that the g-buffer textures can now be sent off to the lightpass to do its lighting calculations. Here again the implementation makes use of two shaders a very simple vertex shader that effectively passes texture coordinates and vertex position to the fragment stage. The fragment shader in this case will now resemble a traditional shader apart from the fact that it needs to sample all of its information from the geometry render buffers (g-buffer).

The final piece of code is in charge of rendering a full screen quad with the combined g-buffer information (through the use of the light pass) as its texture.

4.2 Implementation ray-tracing

4.2.1 OpenCL implementation

Setting up OpenCL can be a tedious process seeing there are multiple levels that need to be set up. For this implementation a small toolkit CLW was used. All this toolkit does is hide some of the abstractions behind very simple api calls. Nontheless it does take some freedom away from the programmer, but it does allow the programmer to jerryrig its own implementation to theirs.

As outlined in the research part there is a lot of performance to be had by just simply creating the right context through OpenCL. Seeing that our pipeline has three major parts that need communicating with each other (e.g. OpenCL/OpenGL/RadeonRays). To benefit from almost free communication (e.g. zero copy) the programmer first has to make an interop context.

The first steps for setting up OpenCL are getting the devices and platforms. Once the devices have been fetched and after verifying that a valid device was found the next logical step would be to create the context. An additional step to the context creation call is to specify context properties, now here is where it gets exponentially more interesting seeing that those properties allow the programmer to set up his interop context.

4.2.2 Radeon rays implementation

With OpenCL now set in place it is time to move on to Radeon Rays. To build RadeonRays the programmer can simply download it from `https://github.com/GPUOpen-LibrariesAndSDKs/RadeonRays_SDK` and build the binaries using CMake. Like addressed in the research portion of this paper Radeon Rays uses a OpenCL context to run its intersection kernels, so when initializing Radeon Rays it automatically creates a new OpenCL context. Luckily though there is an overloaded constructor taking an already existing OpenCL context, further enabling the programmer to share resources between OpenCL and Radeon Rays.

4.2.3 Shadow implementation

The shadow implementation will heavily benefit from GPU acceleration seeing that the bulk of these operations are very parallel in nature.

Ray creation

First the GPU will be used to create a shadow ray buffer, so it will be tasked with creating the 2073600 rays (ray for every pixel in 1920x1080p resolution). As you can probably tell this is an enormous amount of data, but each ray has no knowledge of the other one existing making it perfect for parallel execution.

Here the power of OpenCL and deferred rendering are combined to run a kernel for every entry in the position buffer and normal buffer. Effectively creating a ray that travels from the objects world position towards the lights (in this case directional light).

```
//Sample worldposition and normal information from g-Buffertargets
float4 worldPos = read_imagef(worldPosBuffer, imgCoord);
float4 Normal = normalize(read_imagef(normalBuffer, imgCoord));
float4 dir = lightPos - worldPos;

//Get 1D global index
int k = imgCoord.y * dimensions.x + imgCoord.x;

//Create rays
__global Ray* my_ray = rays + k;
my_ray->o = worldPos + Normal * 0.005f;
my_ray->d = normalize(dir);
my_ray->o.w = length(dir);
```

Figure 4.2: Code snippet from generate shadow rays kernel

Light mask

Before moving on to the light mask the programmer needs to send the ray buffer of to RadeonRays where its kernels will do the ray intersection. Once the intersection query has taken place the library provides us from an intersection buffer. This buffer is stored in GPU memory and can thus be shared with our own OpenCL kernels, alleviating the need to copy from CPU to GPU.

Again for every pixel in the light buffer, which is now white, we run a light mask kernel. The light mask kernel is in charge of fetching the intersection results and checking if they reached the light or not. For hard shadows this is easy, when the pixel gets occluded the light mask gets filled with a zero, if the ray reached the light the value stays one. For lighting calculations this means that pixels that have a one will get lit and pixel that have a zero will not get lit. This isn't totally accurate seeing we'd need to rely on global illumination to account for light bleeding.

It gets a little more interesting when looking at soft shadows however. Traditional ray tracing relies on multiple rays per light source to create soft shadow gradients, like covered in the research part this wouldn't be a viable option seeing it would be extremely performance heavy. So for this implementation of the light mask occlusion delta value was written instead of the occlusion value. The occlusion delta value gets calculated by dividing the rays maximum length (distance to lightsource) by the distance to the occluder. This give a bit more of a smooth transition instead of the hard one and zero values. This will suffice for hard shadows, but does still need a lot of extra work for a soft shadow implementation.

Penumbra sampling

Chapter 5

Conclusion

Bibliography

[1] Alexandru Voica, 2014, Practical techniques for ray tracing in games, http://www.gamasutra.com/blogs/AlexandruVoica/20140318/213148/Practical_techniques_for_ray_tracing_in_games.php

[2] Alexandru Voica, 2015, Ray tracing made easy, http://www.alexvoica.com/ray-tracing-made-easy/#sthash.q1sHqqiT.dpbs

[3] Mathieu Einig, 2017, Hybrid rendering for real-time lighting: ray tracing vs rasterization, https://www.imgtec.com/blog/hybrid-rendering-for-real-time-lighting/

[4] Brian Caulfield, 2018, What's the difference between ray tracing and rasterization?, https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/

[5] Wikipedia, 2018, Graphics pipeline, https://en.wikipedia.org/wiki/Graphics_pipeline

[6] Scratchapixel, 2018, Ray-Tracing: Generating Camera Rays, https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays

Appendices