
**UNIVERSITÀ DEGLI STUDI DI MILANO**

Department of Science and Technology
Master's Degree in Computer Science

# Distributed Rendering in Vulkan

Giacomo Parolini

Supervisors

**Prof. Dario Maggiorini**
**Dott. Davide Gadia**

Academic year 2017–2018

# Contents

# Introduction

The ever-increasing demand of high-end graphics, sophisticate AI, realistic animations and content richness in modern video games today requires consumers to buy more and more performant (and expensive) hardware in order to unlock the full potential games have to offer. Moreover, the increasing graphics fidelity and sheer amount of content of modern games means they are getting bigger and bigger in size very rapidly. Since nowadays the vast majority of videogames are delivered through the Internet rather than on physical media, this translates in longer and longer waiting times between the acquisition of a game and its fruition. This problem is made worse by the fact that most videogames, both multiplayer and singleplayer, are patched and updated on a regular basis. Updates for high-end videogames can easily reach hundreds, if not thousands, of megabytes of size. This trend of never-ending growth exists since the birth of videogames and shows no signs of decreasing.

The expansion of broadband availability worldwide in recent years brought to a brand new approach to those problems. At the dawn of the era of pervasive computing, the concept of *video game streaming* services (also known as *"cloud gaming"*) has just started getting momentum and is likely to grow steadily in the near future. This is all but confirmed by the growing interest of huge companies in the market, like Sony, that bought and incorporated and/or shut down several cloud gaming companies in the last decade, and Microsoft, whose next generation of gaming console is rumored to include a "streaming box" only supporting cloud gaming.

There are two major kinds of cloud gaming: *file-based* and *video-based*. The first kind is a conservative approach: the only change is to allow the player to start playing the game while it is still downloading. This can drastically reduce the initial waiting time, but it can limit the gaming experience until critical parts of the game are received. This kind of approach is already there in several consoles and games of the latest generation. The second kind of cloud gaming, video-streaming based, is more innovative. The idea behind it is to never deliver the full game to the players, instead keeping it on a remote machine (in "the cloud") where it is run and fully rendered. The stream of rendered frames is then encoded and sent to the player as a video stream. This approach has several advantages: the player's gaming device needs not be a powerful machine or even be compatible with the original game's requirements. The game can be fully treated as a service, enabling revenue models such as monthly subscription or pay-as-you-play.

However, this technique is not a panacæa. The streaming of videogame frames incurs all the limitations and problems inherent to multimedia real-time streaming, like global image degradation under bad network conditions. Furthermore, the round-trip time (*RTT*) from the moment an input is received client-side and the moment the updated frame is delivered back is not negligible. Today, the absolute minimum overhead due to the cloud – excluding the network

RTT – is around $100\ \mathrm{ms}$, which is enough to make the gaming experience intolerable for certain game genres.

In this thesis, we explore an approach which is the hybrid between video-streaming based cloud gaming and the traditional "local gaming" model. We make an attempt to split up the rendering pipeline and allow a solution where some stages of the rendering are devolved to the server and some to the client, with the goal of maximizing the data transfer efficiency and video quality robustness.

In our work, we create a distributed system where one machine (the server) contains most of the "world" description whereas a remote machine (the client) presents the final rendered frames after receiving a pre-processed stream of data from the server. As a work looking at the future, we use the new and high-performance–oriented Vulkan graphics API for building the client. We then craft a small demo to put our system to the test in a simple but real case and make our considerations on the results.

This paper is organized as follows. Chapter 1 provides a more detailed overview of the issues in modern gaming and the solutions existing today. Chapter 1.3 presents a review of some related contributions to the problem in literature. In Chapter 2 we described our proposed solution in great detail. Chapter 6 presents our conclusions and our proposed directions where our work may be expanded in the future.

# Chapter 1

# A Shift in Paradigm for Gaming

Videogames have come a long way since their inception, and the age where they were only popular in a limited and isolated niche of enthusiasts is long past (at least with the measuring stick of technology advancement). Today videogames are both a huge global market[1] and an omnipresent reality in the lives of billions of people[2].

More recently, another reality came to light that would radically change the relationship between people and technology: the reality of pervasive and mobile computing. Along with it, the concept of *cloud computing* started rising in popularity and nowadays it is so omnipresent in our lives that we hardly ever even notice.

Today we are witnessing the beginning of a new trend which is the betrothal of the aforementioned two: *cloud gaming*.

## 1.1 Problems Arising in Traditional Gaming

As videogames are now an established mass product (thus bringing forth massive revenues), the fight for leading their market is harsher than ever. The biggest companies are constantly striving for delivering the most stunning and jaw-dropping experiences they can to conquer the interest of their consumers. Today this translates into investing more and more budget for the creation of a single title, resulting in the creation of bigger and bigger games. Modern AAA games are not only "bigger" from a technical standpoint, but also from the much more down-to-earth criteria of computing resources requirements (or, in common gaming lingo, *minimum specs*).

More recent games of course need better CPUs, larger memory and better graphics hardware than dated ones, but also have considerably larger and larger sizes. In the last decade the typical size of a AAA videogame quickly grew from slightly less than $10\,\mathrm{GB}$ to more than $50\,\mathrm{GB}$. This size may easily double or more in the near future, especially for the most ambitious MMORPG games [5, 6].

The ever-growing size of AAA games has a major impact in download times. While broadband availability is growing pretty steadily in developed countries, that is not the case for most

[1] In 2017, the value of videogame market was estimated around 108.9 billion dollars.

[2] In 2015, the amount of gamers worldwide was estimated around 1.8 billions. In 2017, that number grew to around 2.2 billions of people.

developing or underdeveloped ones. Even where broadband is available, a non-negligible wait time must be put on budget when buying a videogame and, while some modern consoles and games offer a "play while you download" option, that is still more the exception than the rule.

Another omnipresent issue in modern videogames are updates. Virtually every game today is getting patched at some point in time, and it is not rare for a multiplayer game to be patched on a weekly basis. While the size of said updates vary wildly, it is usually proportional to the original size of the game, and today many online games deliver patches with sizes measured in the hundreds of megabytes.

Adding together initial downloads and periodic updates for tens of games, waiting for downloads becomes a significant and annoying reality for the player.

## 1.2 A Possible Answer: Cloud Gaming

A possible solution to the issues described above started taking shape at the beginning of the latest decade, in the concept of "cloud gaming". The idea of cloud gaming is to allow players to access a videogame via a *thin client*, while the game itself resides elsewhere (in "the cloud").

There are two main flavours of cloud gaming: *file streaming* cloud gaming and *video streaming* cloud gaming. Both of these techniques require a broadband connection between the client and the cloud gaming service provider.

### 1.2.1 File Streaming

File streaming, also known as "progressive downloading", simply consists in distributing not the entire game at once but just a very small part of it and allow the player to start playing immediately, while the rest of the game is fetched in the background. This technique is used on modern consoles such as PlayStation 4 and XBox One. File streaming reduces the inconvenience of long waiting times, but it comes with some downsides: first of all, an initial waiting time roughly proportional to the total game size still needs to be accounted for. Moreover, several game features may not be available until the download has reached a certain completion threshold. Additionally, the game still takes up all the storage space it would without file streaming and periodic updates are essentially untouched by this solution.

### 1.2.2 Video Streaming

The alternative approach is video streaming. This technique radically changes the approach to the problem by never actually distributing the game files, instead running and rendering everything on a remote machine and only streaming the resulting video frames to a thin client. The high-level overview of this architecture is sketched in Figure 1.

Video streaming based cloud gaming – which we will abbreviate to VSCG from now on – is a powerful technique which enables the fruition of modern and resource-demanding videogames even by less-than-powerful devices, as the client itself only needs the capability to render a video stream[3]. This also enables games originally developed for PC or console to be played

[3] Cloud gaming providers usually employ H.264/MPEG-4 AVC codecs, which is a very common format, more often than not benefitting from hardware-accelerated decoding

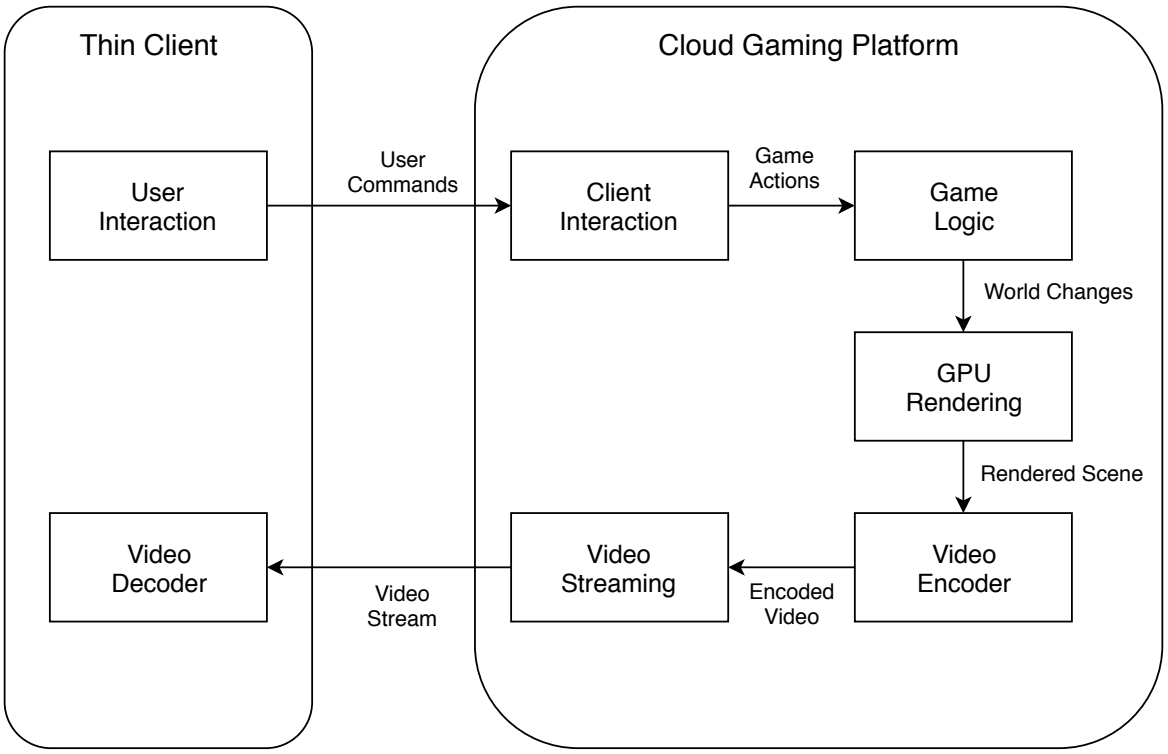


**Figure 1:** Architecture of video streaming based cloud gaming

on mobile device, even if CPU architecture or technical specifications are radically different from the ones originally required [8]. VSCG is definitely an important reality in the future of gaming, as shown by the interest of the major gaming companies worldwide, such as Sony – that bought several newborn cloud gaming services and now runs its own, the *PlayStation Now* service [9, 10, 11] – and possibly Microsoft with its upcoming console [12].

Despite all the advantages it brings, video streaming has three major downsides: interaction delay, video quality robustness and scalability.

### Interaction Delay

"Interaction delay" is the time that passes from the moment the player's input is registered and the moment the player sees the consequences of his/her input on screen. In the case of VSCG, this time is the sum of the following processes:

1. user input is registered by the client;
2. user input is sent via network to the server;
3. the server runs the game simulation for the current frame;
4. the server renders the current frame;
5. the server encodes the frame into the compressed video stream;
6. the video stream is sent via network back to the client;
7. the client decodes the video stream.

All these steps must be performed in a very short time window, which ranges from about $100\,\mathrm{ms}$ to about $1000\,\mathrm{ms}$ depending on the genre of the game being played. Time measurements on popular cloud gaming platforms show that the overhead of the cloud platform is at best around $100\,\mathrm{ms}$, not taking network delay into account [8, 13, 14]. This means that very fast-paced games, such as first person shooters, may offer a substantially degraded experience when played via cloud gaming. This degradation is massively increased if the game being played is a real time online multiplayer game, especially a competitive one.

### Video Quality

Since VSCG uses a multimedia video streaming technique, it is susceptible to degradation of network quality in the same way live streaming and video conference are. Studies conducted on VSCG services such as OnLive and StreamMyGame show that the achieved frame rate drops linearly with the network packet loss, while graphics quality can degrade sensibly for bandwidths smaller than $4\,\mathrm{Mbps}$ [14]. Even on optimal conditions, streaming videogames at higher resolutions require a proportionally higher bandwidth[4] and, in a market where FullHD and even 4K gaming is becoming the norm, providing a resolution capped at 720p is likely to leave a big slice of customers dissatisfied with the service.

### Scalability

Rendering a modern AAA game is a taxing task for a computer. Many recent PC games require a machine worth several thousands euros in order to play at their full potential. Even when rendering more modest targets[5], graphics hardware remains an expensive component ranging from around 150 to more than 1000 euros. The now-defunct cloud gaming service OnLive was reported to spend around $5 million per month on its operation in 2012, mostly for mainteinance of tens of thousands of servers, each supporting only *one* concurrent game session [16]. Clearly, scaling to millions of active users becomes a huge problem when a whole physical server must be deployed for every one or two players.

The common solution to this problem is to put players in a waiting queue when no servers are available, but this is a rather poor solution. If a player gets queued often when he/she wishes to play his/her favourite games, it is unlikely he/she will continue using cloud gaming services. A study conducted on the CloudUnion chinese cloud gaming service showed that during peak hours (18:00 – 22:00) more than *half* of the requests were queued for more than a whopping $500\,\mathrm{s}$ [15]. Whereas the same study claims substantial gain potential from the implementation of their provisioning algorithm, CloudUnion's userbase amounted to only $300{,}000$ users at the time of the experiment. As the cloud gaming userbase is expected to grow significantly in the near future, a combination of better provisioning techniques and higher resource allocation is essential to make VSCG keep up with said expectations.

[4] The cited work reports a $30\,\mathrm{Mbps}$ requirement by StreamMyGame for 1080p quality.
[5] VSCG services often render at 720p resolution to save computational power and bandwidth.

## 1.3 Related Work

Cloud gaming has been subject to several scientific studies over the last years.

Shea, Ngai and Cui (2013) [8] carried a study over the VSCG service OnLive to measure processing times and image quality. They found that the overhead due to the cloud averages around $110\,\mathrm{ms}$ independently on the network delay (for network delays up to $75\,\mathrm{ms}$). As for the image quality, they used the peak signal-to-noise ratio (PSNR) and structural similarity index method (SSIM). Both methods scored a less-than-good result for bandwidths up to $10\,\mathrm{Mbps}$, remaining "acceptable" while over $3\,\mathrm{Mbps}$.

Chen *et al.* (2013) [14] also studied VSCG by comparing OnLive's proprietary service and the software StreamMyGame, which enables setting up a VSCG server on personal computers. They measured network traffic, response delay and image quality for a dozen different games and found OnLive to perform generally better than StreamMyGame. Importantly, they broke up the respose delay into its different components: Network Delay (ND), Processing Delay (PD), Game Delay (GD) and Playout Delay (OD) and measured each component separately. They found that one of the reasons why OnLive performs better is a substantially lower processing delay than StreamMyGame.

Wu, Xue and He (2014) [15] studied the phenomenon of player queueing and server provisioning on the Chinese proprietary VSCG service *CloudUnion*. They then used constrained stochastic optimization theory to devise an algorithm called *iCloudAccess*, which they claim can save more than 30% of provisioning costs by reducing queueing and response delay significantly.

As regards geometry streaming via network, Bischoff and Kobbelt (2002) [20] proposed a method for robust progressive 3D geometry streaming over lossy communication channels based on redundant and order-independent information. They decompose a 3D model into ellipsoids which are efficiently encoded and sent through the network to form a coarse approximation of the model, interleaving them with sampled points used to refine the surface details. Their technique allows to fastly deliver a rough (but still high-fidelity) version of the model even under poor network conditions and refine it later.

Meng and Zha (2003) [19] used a "gaze-guided" method to progressively send incresing LODs of a 3D model with a point-based streaming technique.

Finally, Liao *et al.* (2015) [16] presented an alternative cloud gaming system called *LiveRender* that implements "graphics streaming". Graphics streaming, rather than streaming rendered frames to the client, streams graphics commands (along with their geometry data) which are interpreted by the client and rendered on its GPU. This is in general more bandwidth-costly than video streaming, but LiveRender accomplishes to reach lower traffic by compressing the stream in sophisticate ways closely tailored to graphics commands and geometry. Their approach can be applied to any game thanks to a software layer that intercepts and wraps graphics API calls on the server. Figure 2 shows LiveRender's architecture.

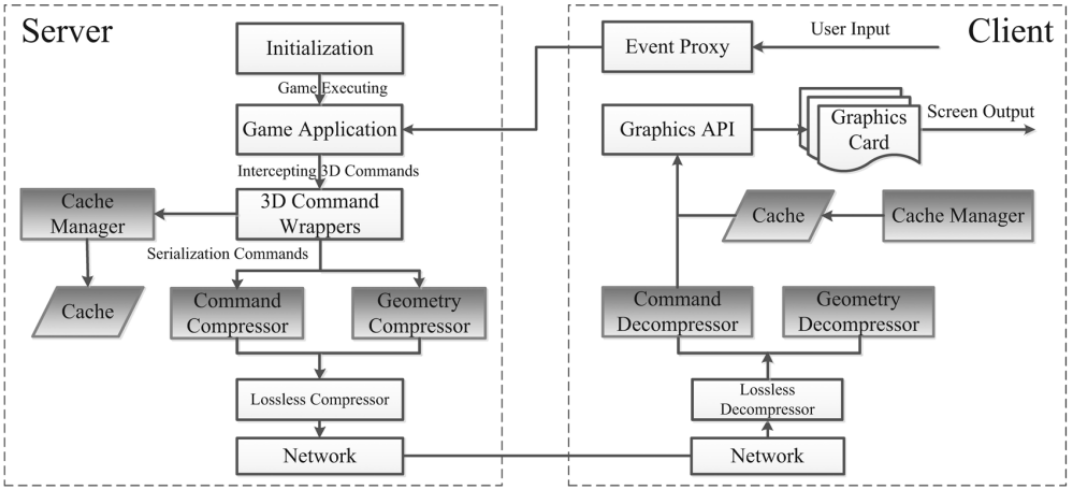


**Figure 2:** Architecture of LiveRender, a graphics streaming cloud gaming software (Figure from [16])

# Chapter 2

# A Mesh-Streaming System Using Vulkan

In this work, we explore an alternative approach to cloud gaming, one based on *geometry streaming* rather than video, file or command streaming. We try to unburden the client from downloading the game assets, thus saving storage and waiting time for the player, and the server from fully rendering the videogame, thus saving computational power and enabling more scalability for the cloud gaming service provider.

In our hybrid approach the server owns all the game assets (3D models, textures and so on), runs the game simulation and performs the first segment of the graphics pipeline's application stage, whereas the client forwards user input to the server while receiving and rendering the raw graphics data streamed by it.

This approach comes with pros and cons: the main downsides are the following:

- the client is not as "thin" as in VSCG: it still needs to have hardware up to the task of rendering a complex scene;
- more in general, the convenience of creating a thin client is diminished, as details like the machine word size and endianness are relevant when deserializing server data, and must be taken into account;
- the amount of graphics data needed to fully render a scene may be very high and, depending on the network conditions, it may take more time to initially start up the game.

On the other hand, we gain the following benefits:

- the computational burden on the server is greatly reduced, enabling greater scalability;
- apart from an initial startup phase, the amount of data transmitted via network "on regime" is much smaller than that required from a video stream;
- the geometric nature of streamed data opens many possibilities for mesh interpolation, decimation, LODding and all kinds of optimizations which may in principle enable high robustness in case of poor network conditions;
- a constant or sudden throttling in bandwidth does not affect video quality at all; it may affect meshes' faithfulness: this is the tradeoff we make in this approach;

- like VSCG, our approach has the welcome side effect of contrasting piracy in a very effective way.

It is important to underline that our work does not propose itself as a drop-in replacement for current cloud gaming solutions: rather, its goal is to try and open the road for a possible upcoming change in the way games are developed and distributed. Our work does not deal with taking existing games and enabling their streaming via network: instead, it is about a new way of creating games with the specific purpose of distributing them according to this model. In other words, this work proposes itself as the basis for a "distributed rendering engine" implementing at least part of the concepts descripted above. The work described in this thesis may in principle become a building block for the more ambitious project to create a fully-fledged *distributed game engine*.

## 2.1 Project Goals and Guidelines

The goal of our work is to design and implement a functioning distributed realtime rendering engine. While creating this project, we had the following goals and guidelines in mind:

- *Future-proofness*: the engine should not rely on old or deprecated technology. On the contrary, since we are building something new and we do not face backward compatibility issues, we prefer using the most modern alternatives at our disposal;
- *Modularity*: the program should be designed and written with the aim of being extended and worked on in the future;
- *Robustness*: we aim to craft a solid basis for whichever expansion may be added afterwards. Therefore, the most fundamental engine features should be solid;
- *Simplicity*: the engine's structure should be kept as simple as possible to keep it intelligible and easily browsable. This encourages frequent and low-cost iterations and refactoring on most parts of the engine;
- *Pragmatism*: the program should be functional to solving real problems, therefore we do not stick with any single programming paradigm chosen *a priori*: instead, we adopt different ones when different situations demand so;
- *Efficiency*: while we do not demand that our program is written in the best possible way, it should still take efficiency in high consideration. This means that every reasonable step to optimize the program along the way is taken and the program itself is often profiled.

These guidelines have the following macroscopic consequences on code design:

- we almost always opt for using small data structures and plain functions rather than creating huge classes with lots of methods;
- we always strive to keep our functions pure[1] whenever it is reasonable to do so;

[1] A "pure" function is a function that has no side effect and makes no use of any global state.

- except possibly for code which is rarely executed, we try to minimize memory allocations and to keep data contiguous in memory; we often employ custom memory allocators for this purpose;
- we sometimes prefer explicit initialization and cleanup over RAII constructs (namely, constructors and destructors);
- we keep our class inheritance as shallow as possible, use as few virtual methods as possible and avoid multiple inheritance save for tiny mix-ins; we also avoid using abstract classes or interfaces;
- we don't make use of complex metaprogramming features; templates are used mostly for container types.

## 2.2 Involved Technologies

### 2.2.1 Graphics Library

Since we're building a rendering engine, the foremost choice we face is which graphics library to use. As of 2018, three major contenders share the mainstream scene of 3D rendering:

**OpenGL** is the oldest and most widespread of the three; it is not an actual library but rather an API specification, with different implementations on different platforms. OpenGL's main advantages are its maturity and its open source nature, which enables its use on a wide variety of platforms and operating systems. However, being a dated API, it also suffers from a significant amount of bloat and historical residues which make it inconvenient to use in some cases, like tiled rendering on mobile devices or rendering from multiple threads.

**Direct3D** is Microsoft's 3D graphics API. Like OpenGL, it is a very mature and widely used API, employed by a large fraction of modern videogames and game engines. The proprietary nature of Direct3D restricts its use to the Windows operating system, which is the main reason why we rejected this choice in the current work.

**Vulkan** is a new graphics API released in 2016. Like OpenGL, Vulkan is an API specification, whose main implementation is LunarG Vulkan SDK [32]. Vulkan aims to be much more low-level than OpenGL or Direct3D[2] and, as such, allows for greater fine-tuning and control over almost every aspect of the application.

As our thesis is a work of research, it seems fit to choose the most modern and innovative of these API, so Vulkan is our library of choice. A more detailed explanation on the difference between Vulkan and older graphics APIs will be presented later in § 4.1.

[2]DirectX 12 introduces a lower level approach comparable to Vulkan, with similar philosophy and functionality. However, it remains a proprietary, non-portable API, which renders it unsuitable to our purpose.

### 2.2.2 Programming Language and Compiler

The choice for the programming language is driven by the following constraints:

- the language should be mature and have a wide library support, as our engine needs to:
  - manipulate several types of assets such as textures and 3D models;
  - perform mathematical computations, especially in the realm of 3D linear algebra;
  - communicate via network;
  - make heavy use of data structures, threads and synchronization;
  - *et cetera*
- the language should run as fast as possible, as we require realtime interaction. This excludes any kind of interpreted language from the list;
- the language must interface easily with the Vulkan SDK;
- since we are undertaking the effort to use Vulkan, we would also like to have the highest possible degree of control over our program. In particular, we want manual memory management rather than a garbage collector;
- efficient network serialization and deserialization, along with the need to use memory as efficiently as possible, make arbitrary pointer manipulation very desirable, so the language must support pointer arithmetics.

Given this combination of constraints, we chose C++[3] as the main development language.

The engine was developed on both GNU/Linux and Windows 10. The compilers used were *gcc*, *Clang* and *MSVC*, using *CMake* as our build system.

### 2.2.3 Auxiliary Libraries and Tools

Third party libraries were used for the following purposes:

- *Assimp*: used for loading 3D models;
- *STB Image*: used for loading images;
- *cfstructs*: fast map/set container types;

We also used the *FlameGraph* tool for profiling, *gdb* and *RenderDoc* for debugging and *cppclean* for code mainteinance [24, 25, 26, 27, 28, 29, 30].

[3]The C++14 standard was used, with no extension.

## 2.3 Overview of the Engine Architecture

The engine consists of two main parts: the server and the client. The server's job is to host the game simulation, manage the raw assets and send to the client the data it needs with the due priority. The client's job is to process the data from the server and render the game world, as well as handling the player input.

The engine project consists of three subprojects: one for the server, one for the client and one for the common code between the two. Common code is compiled and packaged into a static library which the other two parts make use of. Upon building, two executables are produced: *server* and *client*.

## 2.4 The Common Layer

The common part of the project represents the foundation of the engine. In broad terms, it contains general utilities (logging, file handling, code helpers and macros, clocks and timers, hashing procedures), common configuration, the platform independence layer, memory allocators, the basic networking code, the network protocol definitions and the data structures used for resources serialization. Figure 3 gives an overview of the common layer. The following sections will describe the relevant parts in detail.

### 2.4.1 StringId

In programming, a "string" is a sequence of characters that usually represents a name, a word or a sentence. In C++ a plain string is represented as an array of bytes laid out sequentially in memory. Since no information is stored about the string's length, the end of the string is marked by a NUL byte (`'\0'`).

Strings are a very useful data type in any program and are extensively used throughout our engine. However, a plain string is a quite expensive datum in terms of memory requirements and computational cost. A string of length $N$ uses at least $N$ bytes of memory, and that number is usually higher as the compiler pads the character array to the preferred alignment. The two most common operations done on strings, *copying* one or *comparing* two, are significantly more costly than the same operations performed on basic types like integers or even floats. Strings of unknown or variable length cannot be easily allocated on the stack, meaning that passing strings around involves either passing *pointers* (with all the extra care needed to ensure the pointed data still exists) or dynamically allocating and copying memory, which is slow.

To solve these problems in a mostly painless way, we employ a widely used technique in game engines: whenever we only need a string as an internal "name" (which is almost every case, save for logging and storing filesystem paths), we use a *hashed string id* instead of the string itself [1]. The idea is to run a hashing function over the bytes of the original string to obtain an integer value that is used in place of the string everywhere it is needed. Not only this saves memory, using only 4 bytes for every string regardless of its length, but it also enables fast string comparison, copying and very convenient serialization over the network.

For the hashing function, we used the 32-bit version of the Fowler–Noll–Vo function (FNV-1a) [7].

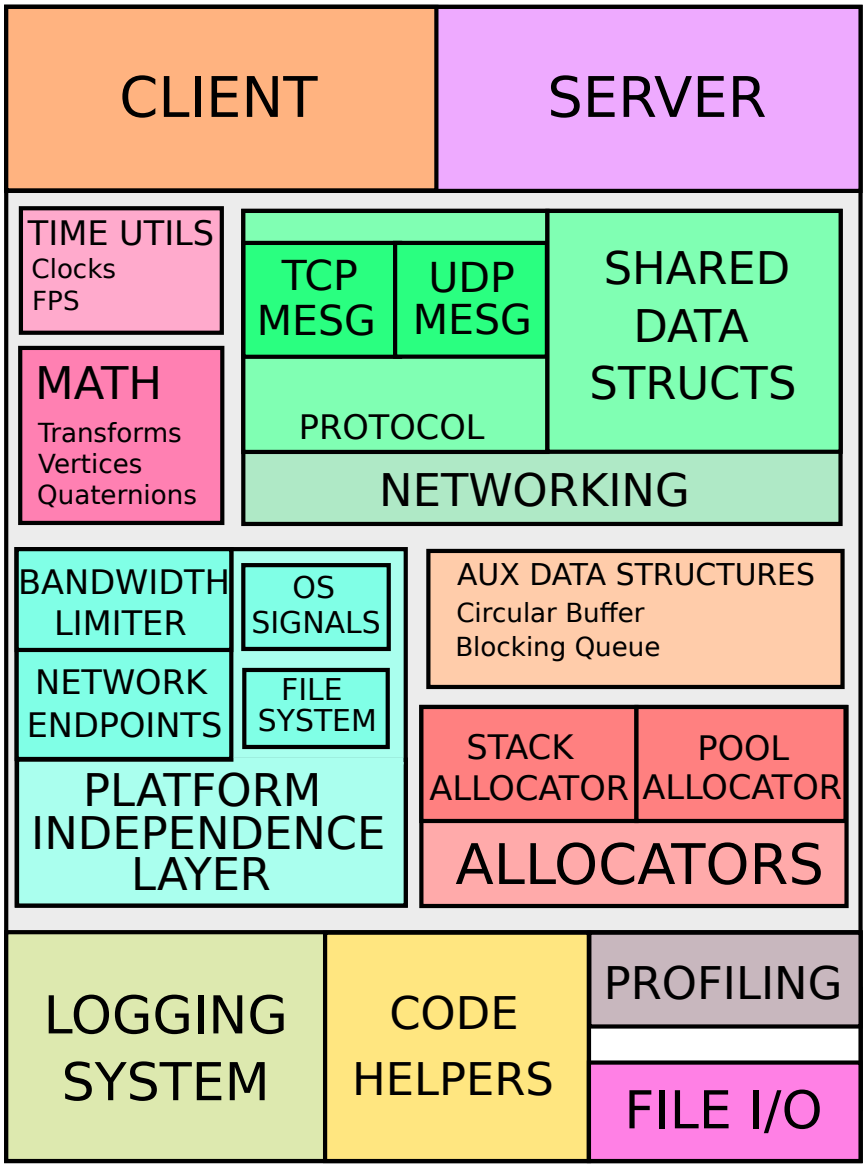


**Figure 3:** Overview of the engine common layer

**Listing 2.1:** StringId

```
// File: hashing.hpp

using StringId = uint32_t;
constexpr StringId SID_NONE = 0;

constexpr uint32_t fnv1a_hash(const char *buffer) {
  constexpr uint32_t fnv_prime32 = 16777619;
  uint32_t result = 2166136261;
  int i = 0;
  while (buffer[i] != '\0') {
    result ^= static_cast<uint32_t>(buffer[i++]);
    result *= fnv_prime32;
  }
  assert(result != SID_NONE);
  return result;
}
```

As the listing 2.1 shows, the `StringId` type is defined as a 32-bit unsigned integer and we reserve the special value `0` as the "null string". We also check that result of the hashing function is not `SID_NONE` for any given input[4].

### 2.4.2 Platform Independence Layer

Our engine needs to work on multiple platforms (as of this work, the engine was built for GNU/Linux and Windows 10), therefore it is very convenient to write a platform independence layer whereon higher layers are built.

The subjects proxied by this layer are filesystem, threads, signal handling and sockets.

Filesystem proxy functions deal with the different OS conventions and system calls for handling files and directories, namely difference in path separators and querying the current working directory.

Signals are used to set up custom exit handler functions by trapping signals such as SIGINT, SIGTERM and SIGPIPE. This is needed as both the client and the server use TCP sockets which need to be closed properly regardless of the program ending successfully or being aborted.

Cross-platform thread support is mostly covered by our use of C++ `std::thread` facility, however we still need platform-specific code for setting a thread's name. The platform independence layer provides a convenient `xplatSetThreadName` function.

The cross-platform sockets API deals with the subtle differences between POSIX sockets and the Windows Socket API (WSA). These differences include the need for explicit initialization

[4]The extremely unlikely cases where an input string resulted in a hash value of 0 would be solved manually by changing the input string, e.g. appending a `1` at the end. This is almost always viable, as we use StringIds only for internal naming.

and cleanup of the WSA system, the different way to retrieve error codes and messages, the type used for socket handles, the different signature of socket functions and the way of closing a socket. Our cross-platform sockets API aliases the socket handle's type to an opaque `socket_t` and exposes several functions for external platform-agnostic use by upper layers.

### 2.4.3 Custom Containers

Throughout the engine code we make extensive use of containers from the standard C++ library, particularly `std::vector`, `std::array` and `std::unordered_map`. We also use third party containers when we need highly efficient and cache-friendly sets and maps (respectively `cf::hashset` and `cf::hashmap`).

However, in some cases we preferred providing our own implementation for specific container types. The containers in question are *circular buffer* and *blocking queue*.

#### CircularBuffer

A circular buffer is a commonly used container type which allows $O(1)$ insertion and deletion and both ends. Our implementation of CircularBuffer is backed by a memory buffer endued with a $startIndex$, $endIndex$, $capacity$ and $numberOfElements$ fields. The memory buffer can be freely resized while the buffer is in use, but it does not self-resize automatically.

CircularBuffer is a type-safe template class that can only contain elements of type `T`. Its interface is also reported in the Listing. Note that we do not allow pushing and popping at both ends, but only a FIFO-style access where the oldest element is the only one that can be popped at any given time. We also allow "overflowing" the buffer by writing more elements than the buffer's capacity: new elements will simply overwrite the oldest ones. If this behaviour is undesired, the user of CircularBuffer must check that $buffer.size() < buffer.capacity()$.

Our CircularBuffer implementation is also iterable. In fact, it provides a pair of methods, `iter_start` and `iter_next`, that allow visiting all the values in the buffer sequentially. We did not bother implementing the iterator traits defined by the C++ standard library, as their overly complicated definition goes against the project guidelines listed in § 2.1 and the benefit they provide hardly justifies their complexity.

#### BlockingQueue

A blocking queue is just a queue allowing concurrent pushing on one end and popping on the other. Following our use case, we implemented our BlockingQueue atop the CircularBuffer. The BlockingQueue changes CircularBuffer's interface by *privately* inheriting it and defining different methods, which are guarded with a mutex and proxied to the parent class. An example of this is reported in Listing 2.2.

### 2.4.4 Base Networking Layer

The ground-level of the networking layer defines an `Endpoint` data structure along with a family of functions which operate on it. This API is used throughout all the engine whenever a network communication must occur.

**Listing 2.2:** BlockingQueue extract

```
// File: blocking_queue.hpp
template <typename T>
class BlockingQueue : private CircularBuffer<T> {

  std::mutex mtx;
  std::condition_variable cv;

public:
  // ... methods omitted ...

  void push(const T& elem) {
    mtx.lock();
    CircularBuffer<T>::push_back(elem);
    mtx.unlock();
    cv.notify_one();
  }

  /** Retrieves the first element of the queue.
   *  If no elements are in the queue, blocks until one is added.
   */
  T pop_or_wait() {
    if (CircularBuffer<T>::elements > 0) {
      std::lock_guard<std::mutex> lock{ mtx };
      return CircularBuffer<T>::pop_front();
    }

    std::unique_lock<std::mutex> ulk{ mtx };
    cv.wait(ulk, [this]() {
      return CircularBuffer<T>::elements > 0;
    });
    return CircularBuffer<T>::pop_front();
  }
};
```

**Listing 2.3:** The Endpoint struct

```
// File: endpoints.hpp

struct Endpoint {
  enum class Type { ACTIVE, PASSIVE };

  socket_t socket = xplatInvalidSocketID();
  std::string ip;
  int port;
  bool connected = false;
};
```

The `Endpoint` structure is shown in Listing 2.3.

The `ACTIVE` and `PASSIVE` enumeration values, used when creating the Endpoint, specify whether the underlying socket will be listening on the given IP/port or will try to connect to it. The boolean field `connected` conveniently keeps track of the connected state of the socket.

Note that the Endpoint structure doesn't map to any particular thread or task operating on the socket itself. We decided to separate the sockets from the application threads that will operate on them so that multiple threads can share a single socket if needed and, conversely, a single thread can in principle operate on multiple sockets. Effectively, the relation $sockets \longleftrightarrow network\_threads$ is a many-to-many relation.

The family of Endpoint functions is summarized in Table 2.1.

### 2.4.5 Bandwidth Limiter

For testing and simulation purposes, our engine has the capability to artificially throttle its own uplink bandwidth to an arbitrary amount of bits per second. This is achieved by hooking the `sendPacket` function with an internal facility called `BandwidthLimiter`. The BandwidthLimiter is a class implementing a token bucket algorithm [31] in a separate thread and providing an interface for requesting tokens from it. When the BandwidthLimiter is active, every request to send $N$ bytes through a socket first requests $N$ tokens from the BandwidthLimiter. If that amount of tokens is available, the packet is sent and the tokens are removed from the BandwidthLimiter's bucket; else, the send is blocked until $N$ bytes become available.

Our implementation of the BandwidthLimiter is as follows: we create a global object `gBandwidthLimiter` of type `BandwidthLimiter`[5]. If the bandwidth limitation is set to active, the main thread initially calls `setSendLimit` on the object; this sets the internal *token rate* of the tocket bucket algorithm. Both the token capacity (also known as *burst size*) and the bucket update interval are set to constant values. The BandwidthLimiter is then started: this operation resets the bucket to contain 0 tokens and starts the bucket filling thread. The filling routine is shown in Listing 2.4.

[5]Since it is designed to work as a global variable, our `BandwidthLimiter` class has neither a constructor nor a destructor, but it is explicitly started and stopped.

**Table 2.1:** Endpoint functions

| Function Name | Description |
|---|---|
| `startEndpoint` | Creates a new Endpoint of specified type (SOCK_STREAM for TCP or SOCK_DGRAM for UDP) and connects or binds (depending on the specified `Endpoint::Type`) its socket to the given IP and port. |
| `closeEndpoint` | Shuts down the given Endpoint's socket and sets `connected` to `false`. |
| `sendPacket` | Sends an array of bytes with a given length through the socket. |
| `receivePacket` | Receives incoming bytes and stores them in a given buffer. |
| `validateUDPPacket` | Checks that the received packet is not older than the previously received ones. |
| `sendTCPMsg` | Sends a header-only TCP message through the socket. |
| `receiveTCPMsg` | Like `receivePacket`, but also decodes the message type according to our TCP protocol. |

The filling routine runs every $updateInterval$ seconds[6], sleeping between one iteration and the next. During each iteration it adds $tokenRate \cdot updateInterval$ tokens to the bucket, up to $maxTokens$. A mutex is used to ensure that all involved variables are not being accessed in the meantime by other threads (which may happen if a thread wants to change the BandwidthLimiter's parameters).

Whenever a thread tries to send a packet, it calls `gBandwidthLimiter.requestTokens` first. This method returns a boolean value telling the thread whether it is allowed to send that packet or not. If the number of tokens currently in the bucket is greater or equal than the requested amount, they are subtracted from the bucket and `true` is returned, so the thread can send the packet immediately. Otherwise, the thread must wait on a semaphore until the bucket is refilled, at which point it can try and request the tokens again. This process is illustrated in Listing 2.5.

The semaphore is implemented via C++'s `std::condition_variable`. A single condition variable field exists in the `BandwidthLimiter` class, which is waited on by every thread that requested more tokens than currently available. At the end of each iteration of `refillTask`, that condition variable is notified to wake all these threads and make them request those tokens again. The process is repeated until all threads have been delivered their required tokens.

### 2.4.6 Memory Allocators

Since one of our project goals is performance, we cannot simply rely on C++'s default heap allocator all the time. The generic strategy for heap allocation is slow and unaware of the ac-

[6]We arbitrarily chose $updateInterval = 0.2$.

**Listing 2.4:** Token Bucket filling routine

```
// File: bandwidth_limiter.cpp
void BandwidthLimiter::refillTask()
{
  using namespace std::chrono;

  while (operating) {
    const auto beginTime = high_resolution_clock::now();
    // Add new tokens (in a thread-safe way)
    {
      std::lock_guard<std::mutex> lock{ mtx };

      auto nTokensToRefill = static_cast<int>(
             tokenRate * updateInterval.count());
      tokens = std::min(maxTokens, tokens + nTokensToRefill);
    }
    cv.notify_all();

    // Sleep
    const auto delay = high_resolution_clock::now()
                             - beginTime;
    std::this_thread::sleep_for(updateInterval - delay);
  }
}
```

**Listing 2.5:** Waiting For Tokens

```
// File: endpoints.cpp
bool sendPacket(socket_t socket,
                const uint8_t* data,
                std::size_t len)
{
  while (!gBandwidthLimiter.requestTokens(len)) {
    std::unique_lock<std::mutex> lock{ gBandwidthLimiter.cvMtx };
    gBandwidthLimiter.cv.wait(lock, [len]() {
      // Waking condition: if not satisfied, keep waiting.
      return !gBandwidthLimiter.isActive() ||
          gBandwidthLimiter.getTokens() >= len;
    });
  }
  // ... Send the packet ...
}
```

**Listing 2.6:** Stack allocation strategy

```
void* allocate(size_t bytes) {
  void* ret = stack_pointer;
  stack_pointer += bytes;
  return ret;
}
```

cess pattern of the underlying memory. The NUMA design of modern computers today means that cache misses are incredibly expensive, at least in relative terms, therefore whenever high performance is desirable one needs to design at least the hot parts of its software to minimize cache misses. Roughly speaking, this is achieved by increasing the locality of data, which in turn is achieved by keeping data as contiguous as possible in memory. This is not the case of a generic heap allocator, which usually tends to allocate chunks in unrelated portions of memory [1].

Among all possible strategies for allocating memory in a cache-efficient way, the most common two consist in the *stack allocator* and the *pool allocator*. Since we use both in our engine, the following sections will describe them in detail.

### Stack Allocator

A stack allocator is one of the simplest design possible when writing a memory allocator. A stack allocator usually starts with a fixed block of free memory along with a "top-of-the-stack" pointer to its lowest address. When an allocation is requested, the stack allocator simply returns the currently pointed address and moves the stack pointer upward of the amount of bytes requested. This is sketched in Listing 2.6.

Stack allocators have two main advantages: they are incredibly easy to implement and they support any size of allocation, as long as it does not exceed their capacity[7]. The main disadvantage is that memory cannot be freed in arbitrary patterns: some stack allocators support deallocating in a LIFO fashion, while others only support deallocating everything at once.

We implemented a stack allocator that:

a) has a fixed-size backing memory buffer, which is owned externally;

b) can allocate arbitrary data types, but internally keeps its stack pointer aligned to the word size;

c) supports individual deallocation by keeping track of all the separate allocations.

One of the most common use cases for stack allocation in our engine is assets loading. When we load a model, texture or shader from a file, a recurring pattern we use is the following:

1. reserve all the remaining allocator's memory as a staging buffer by calling `allocAll()`;
2. load the asset into the buffer;

[7] A careful implementation of a stack allocator will also take memory alignment into account, so some sizes would be rounded up to the nearest multiple of the word size. Our stack allocator aligns every pointer to a multiple of `sizeof(void*)`, i.e. 8 bytes when targeting 64-bit.

3. read the asset's actual size and shrink the allocated buffer to that size. This is done by simply calling `deallocLatest()` and then `alloc(actualAssetSize)`.

This way we avoid the extra step of reading the file's size before loading it without sacrificing any robustness.

#### Pool Allocator

A pool allocator is used to allocate objects of the same size with the option to deallocate them in a random access fashion. A pool allocator typically divides its backing memory buffer in blocks of equal size and only supports allocating or deallocating a single block[8]. Its main advantage is its capability to perform fast (typically $O(1)$) allocation and deallocation in random order.

We implemented a pool allocator in the following way: the allocator is a template class whose type parameter determines the size of the internal blocks. Upon creation, it is handed an externally-owned fixed-size memory buffer which it divides into equally sized blocks. The first $N$ bytes of each blocks (where $N$ is the machine word size) are filled with the memory address of the following block, with the last block pointing to `NULL`. This effectively creates a linked list which needs no external storage, as it is hosted within the allocator's memory itself.

Every time an allocation is requested, the address contained in the first block is looked up to retrieve the address of the first free block. If the address is `NULL`, the allocation fails; else, the address is saved into a variable to be returned, then cast to pointer and dereferenced to obtain the block after it. This new block becomes the first free address by overwriting the old one, and the saved address is returned. This is illustrated in Listing 2.7. Deallocation goes by the same logic.

### 2.4.7 Shared Resources

The client and server portions of the engine share a common set of resources they handle. These resources include 3D models, textures, shaders, cameras and lights. All these "abstract" resources have several concrete representations (typically in the form of plain `structs`), each fit for a specific set of operations.

While for the most part the client's internal representations differ from the server's (since they usually operate in different ways on each resource), some of them are shared between the two. These shared types belong to either of these categories:

i resources that are in a "generic representation" which fits a range of uses for both the client and the server;

ii resources in a serialized state.

An example of the first kind is the `shared::Texture` shown in Listing 2.8.

#### Serialized Data

The other type of shared resource types are serialized data. These are data structures encoding the binary format of the TCP data transmitted through the network. Upon serialization, the

[8] Some pool allocator only support deallocating the whole pool at once.

**Listing 2.7:** Pool allocation/deallocation strategy

```
T* allocate() {
  // ‘pool‘ is the (fixed) pointer to the backing memory.
  // A cast is needed as ‘*pool‘ is of type ‘uintptr_t‘.
  uintptr_t* ret = reinterpret_cast<uintptr_t*>(*pool);
  if (ret == nullptr)
    ABORT("Out of memory");
  uintptr_t next = *ret;
  // Update the first free block, as ‘ret‘ is now in use.
  *pool = next;
  return reinterpret_cast<T*>(ret);
}

void deallocate(T* mem) {
  assert(/* mem belongs to the pool */);
  uintptr_t firstFreeAddr = *pool;
  // Prepend ‘mem‘ to the first free address
  *reinterpret_cast<uintptr_t*>(mem) = firstFreeAddr;
  *pool = reinterpret_cast<uintptr_t>(mem);
}
```

**Listing 2.8:** Shared Representation of a Texture

```
// File: shared_resources.hpp
// Inside namespace shared

enum class TextureFormat : uint8_t { RGBA, GREY, UNKNOWN };

struct Texture {
  /** Size of ‘data‘ in bytes */
  uint64_t size = 0;
  /** Raw pixel data */
  void* data = nullptr;
  /** Format to use when creating a
   *  Vulkan texture out of this data
   */
  TextureFormat format;
};
```

**Listing 2.9:** Serialization/Deserialization Process

```
// ----------- Sender: -----------
shared::Material mat;
mat.name = sid("my_mat_name");
mat.diffuseTex = sid("my_mat_name__diff");
mat.specularTex = SID_NONE;
mat.normalTex = sid("my_mat_name__norm");

uint8_t* packet = reinterpret_cast<uint8_t*>(&mat);
sendPacket(socket, packet, sizeof(shared::Material));


// ----------- Receiver: -----------
std::array<uint8_t, cfg::MAX_PACKET_SIZE> buf;
std::size_t bytesRead;
receivePacket(socket, buf.data(), buf.size(), &bytesRead);

assert(bytesRead == sizeof(shared::Material));
shared::Material mat = *reinterpret_cast<shared::Material*>(buf.data());
```

sender only needs to cast the data structure to a byte array, whereas the receiver simply casts back the received bytes to the correct struct. This process is illustrated in Listing 2.9.

Things get a bit more complex when the TCP message includes a payload, i.e. every time the exchanged resource is of variable length. In this case, the message header needs to be decoded first to retrieve the payload size, then the following packets are read until all data has been received. This process will be described in more details later.

To save as much bandwidth as possible, all the serialized structures are packed with a 1-byte align, so the compiler does not add any padding between fields. The packing is achieved with the `#pragma pack` directive.

## 2.5 Network Protocols and Communication

An overview of the client-server communication logic is schematized in Figure 4.

- First of all, the server starts up and listens on its TCP socket, waiting for a client to connect. Then a remote client starts as well and attempts to connect to the server. Upon connection, a handshake is carried out between the two, after which the application-level connection is considered successful.
- Both the client and the server then open a pair of UDP sockets, one for inbound packets and one for outbound packets.
- The server can now send any initial data it wants to the client. In our demo, we initially send automatically all the lights in the scene.

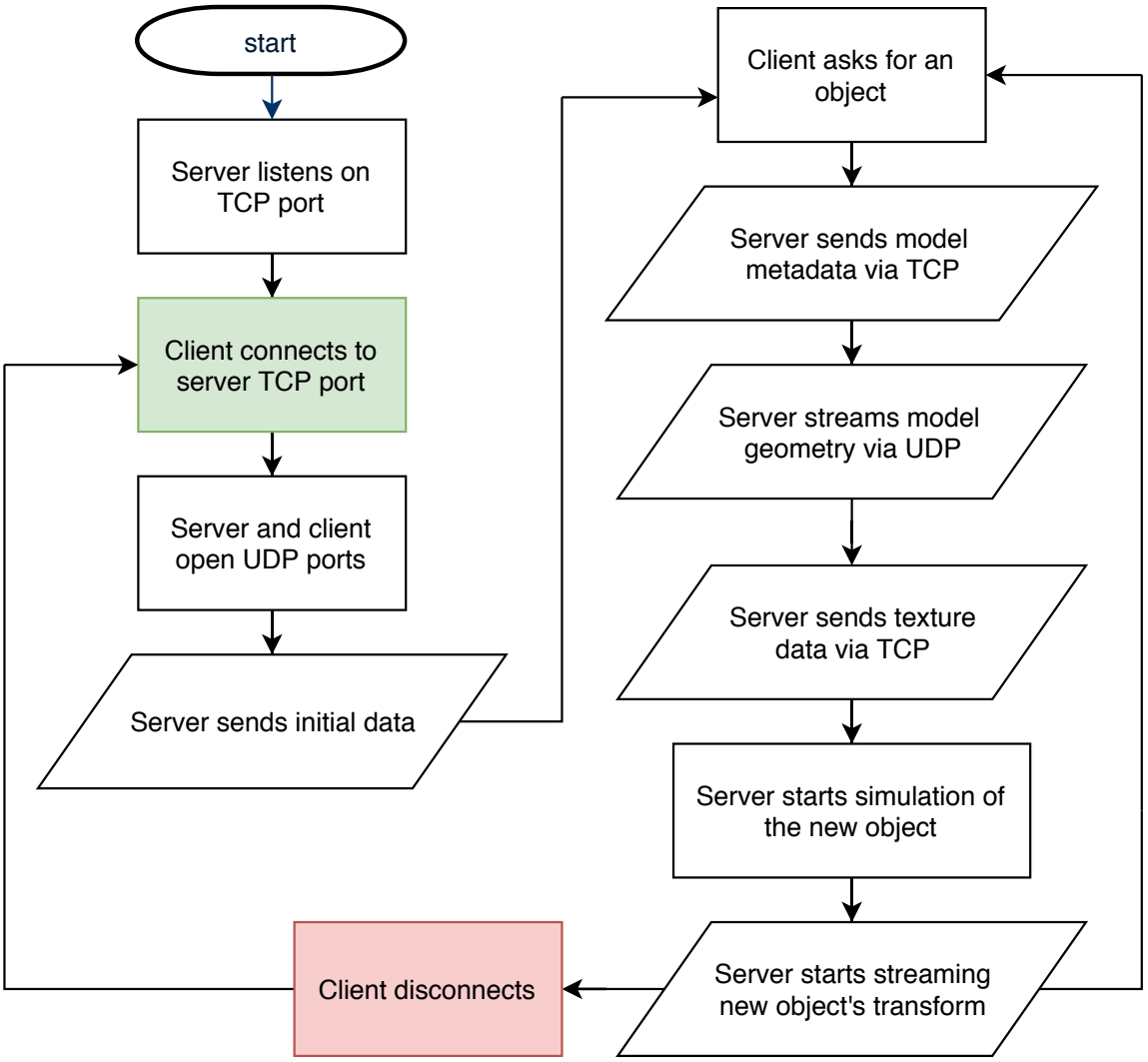


**Figure 4:** Network Communication Logic between Client and Server

- At this point, the client can ask for a particular asset (or set of assets) that it wishes to receive. In our demo we used a "send on demand" logic to better test our infrastructure, but one can think to move the decision on which assets to send to the server. Moreover, in our demo the only type of asset the client asks for are 3D models.
- The server sends the requested model in this order: first, it sends a description of the model's used materials, vertices and indices divided into separate meshes. This description is critical to receive correctly, so it is sent via TCP. Afterwards, the server starts streaming geometry data (vertices and indices) via the UDP channel. Finally, after geometry has been sent, textures are sent via TCP.
- At this point, the server can mark the sent model as "active" and start simulating it in the game loop. In our demo, the server also sends the model's transform data via UDP every time it changes.

The following sections will describe the messages format for both the UDP and TCP chan-

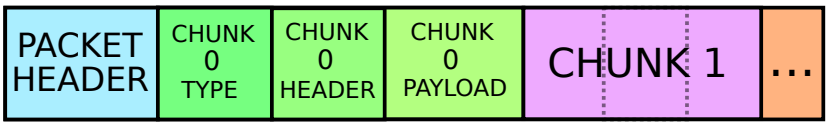


**Figure 5:** Format of our UDP Packets

nels.

### 2.5.1 UDP Messages

We pass messages via UDP whenever it is acceptable that such messages either

a) are received out-of-order, possibly not on the first try, or

b) are not received at all.

For the first category we use *acknowledged UDP packets*: these messages are marked with a serial number and the receiver needs to send back an UDP *ACK* message with the serial of each received packet. The sender will periodically resend all packets that were not acknowledged yet.

For the latter, we do not require any kind of acknowledgement. The UDP packet is simply sent once with the hope it does not get lost, but that possibility is tolerated.

Currently, there are 4 types of messages that fit the two conditions described above:

1. messages carrying geometry data: these may be received in any order and it is not critical they are received in a strict interval; these messages are currently the only acknowledged ones;
2. the acknowledge messages themselves;
3. transform updates: these messages get overwritten pretty frequently, so we deem a better choice to wait for the next update rather than ensuring that every single update is delivered;
4. light updates: these are used to change the properties of existing dynamic lights, such as color and attenuation. For reasons akin to transform updates, these messages are not acknowledged either.

ACK packets are currently the only type of UDP message sent by the client to the server: all the other types are server-to-client messages.

#### Server-To-Client Messages

UDP packets sent by the server to the client have the format depicted in Figure 5.

Every packet starts with a common UDP header containing information about the "packet generation" and the actual content size. The packet generation is a sequentially increasing number assigned to each group of packet sent during the same thread iteration and used to discard old packets. Discarding old packets is important for the non-acknowledged messages, as they should not be overwritten by packets received out of order. The actual content size is used to

determine how many bytes should be kept from the incoming packet, as each UDP packet has a fixed size of $512\,\mathrm{B}$ but not all of those bytes are necessarily used.

The data structures encoding a generic UDP packet are shown in Listing 2.10.

A packet's payload consists of one or more "chunks", each encoding a separate message. Every chunk has itself a header and possibly a payload, depending on its type. For example, a chunk carrying geometry data consists in the header reported in Listing 2.11 plus a payload containing raw vertices or indices data.

Chunks with a payload are of variable size, while header-only chunks are of fixed size. If the size is not fixed, it can always be retrieved by the chunk header upon deserialization.

#### Client-To-Server Messages

Since ACK packets are only sent by the client and are the only type of UDP message the client sends, they are not wrapped as chunks of a generic packet like server messages are, but are directly serialized as a specialized data structure shown in Listing 2.12.

As one can imagine, ACKs are not sent individually, but packed together as much as possible in order to minimize the number of packets sent. Each ACK consists in an integer equal to the serial ID of the acknowledged packet.

### 2.5.2 TCP Messages

Messages are passed via TCP in all cases where

a) they need to be received reliably *and*

b) they are "one-shot" messages, which do not need to be repeatedly sent over time (like e.g. objects' transforms).

The types of TCP messages are more numerous than their UDP counterparts, as revealed by their enumeration in Listing 2.13.

These messages are roughly divided into the following categories:

1. handshake messages;
2. connection keepalive/termination;
3. resource exchange;
4. client requests.

Handshake messages are pretty straightforward: they are header-only messages used to carry out the initial connection phase. The (very simple) handshake logic is shown in Figure 6.

The `KEEPALIVE` message is sent periodically by the client to tell the server it is still connected. When the server has received no keepalive messages for a while, it drops the client. Conversely, the `DISCONNECT` message is used to terminate the connection immediately. It can be sent from either the client or the server to tell the other party to close all its endpoints now.

Resource exchange messages are the most interesting type. Whenever the server wants to send one or more resources to the client (a texture, a material, a model and so on), it first sends

**Listing 2.10:** UDP Packet

```
// File: udp_messages.hpp

#pragma pack(push, 1)

struct UdpHeader {
  /** Sequential packet "generation" id.
   *  Used to discard old packets.
   */
  uint32_t packetGen;

  /** How many bytes of the payload are
   *  (as there may be garbage at the end).
   *  Must be equal to the sum of all the
   *  chunks' size (type + header + payload).
   */
  uint32_t size;
};

struct UdpPacket {
  UdpHeader header;

  /** Payload contains chunks
   *  (each consisting of chunk header + payload)
   */
  std::array<
      uint8_t,
      cfg::PACKET_SIZE_BYTES - sizeof(UdpHeader)
  > payload;
};

// ...
#pragma pack(pop)
```

**Listing 2.11:** Header of chunks containing geometry data

```
// File: udp_messages.hpp

/** Update vertex/index buffers of existing model */
struct GeomUpdateHeader {
  /** Unique ID of the packet, needed to ACK it */
  uint32_t serialId;

  /** What model we are updating */
  StringId modelId;

  /** Whether vertices or indices follow */
  GeomDataType dataType;

  /** Starting vertex/index to modify */
  uint32_t start;
  /** Amount of vertices/indices to modify */
  uint32_t len;
};
```

**Listing 2.12:** UDP Packet containing ACKs

```
// File: udp_messages.hpp

struct AckPacket {
  /** Must be UdpMsgType::ACK */
  UdpMsgType msgType;

  /** How many ACKs does this packet carry */
  uint32_t nAcks;

  /** The serial IDs to acknowledge */
  std::array<
    uint32_t,
    (cfg::PACKET_SIZE_BYTES - sizeof(UdpMsgType)
      - sizeof(uint32_t)) / sizeof(uint32_t)
  > acks;
};
```

**Listing 2.13:** Enumeration of all TCP message types

```
// File: tcp_messages.hpp

enum class TcpMsgType : uint8_t {
  /** Handshake */
  HELO = 0x01,
  HELO_ACK = 0x02,
  /** Ready to receive UDP data */
  READY = 0x03,
  /** Keep the connection alive */
  KEEPALIVE = 0x04,
  /** Announce own disconnection */
  DISCONNECT = 0x05,
  /** Open a resource exchange */
  START_RSRC_EXCHANGE = 0x06,
  /** Acknowledge either a START_RSRC_EXCHANGE
   *  or the complete reception of a resource.
   */
  RSRC_EXCHANGE_ACK = 0x07,
  RSRC_TYPE_TEXTURE = 0x08,
  RSRC_TYPE_MATERIAL = 0x0A,
  RSRC_TYPE_MODEL = 0x0B,
  RSRC_TYPE_POINT_LIGHT = 0x0C,
  RSRC_TYPE_SHADER = 0x0D,
  /** Close a resource exchange */
  END_RSRC_EXCHANGE = 0x1F,
  /** Client asks the server to send a specific model.
   *  Follows a 2 bytes payload with the "model number"
   *  (an arbitrary index into some model list on the server)
   */
  REQ_MODEL = 0x22,
  UNKNOWN,
};
```

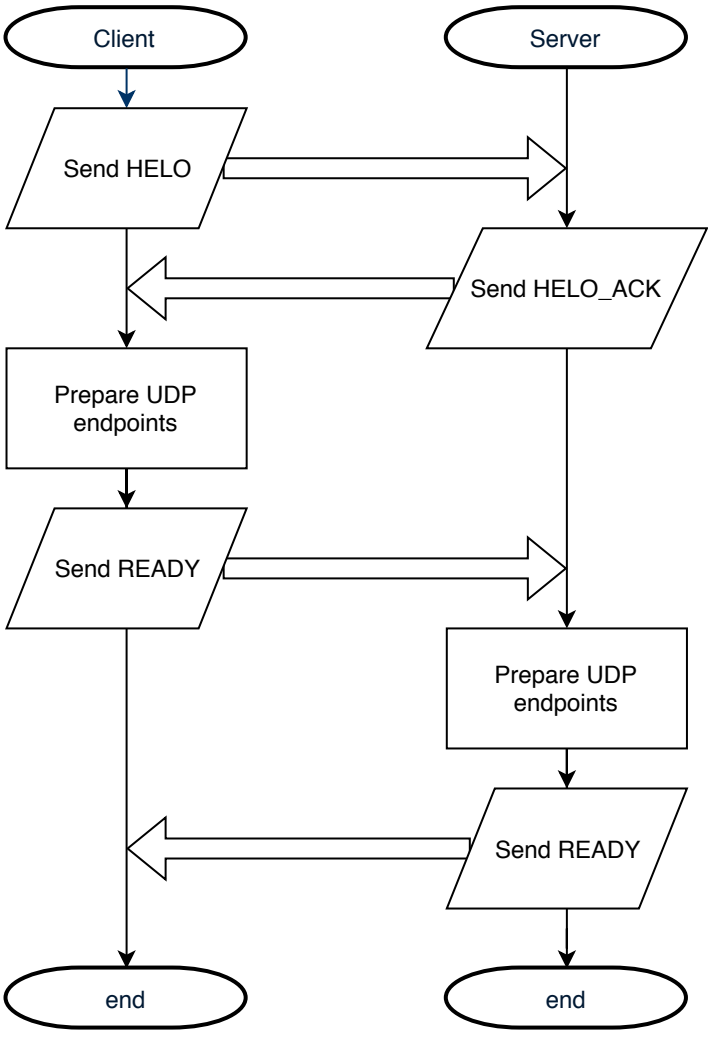


**Figure 6:** Handshake Process

**Listing 2.14:** TCP resource packet

```
// File: tcp_messages.hpp

/** Template for the TCP messages used to send resources.
 *  'ResType' is the actual content of the message.
 */
template <typename ResType>
struct ResourcePacket {
  TcpMsgType type;
  ResType res;
};
```

a `START_RSRC_EXCHANGE` message. This gives the client the opportunity to prepare a memory buffer to store incoming data. When the client is ready to receive the resources, it sends a `RSRC_EXCHANGE_ACK` message back to the server, which then proceeds to send the actual resources. Every resource type is sent as a TCP packet that, like for UDP, consists in a header and, possibly, a payload. The header is different for each resource type, while the payload is only present if the resource is of variable size. Like in the UDP case, all resource packets are wrapped in a common data structure which merely prepends the type of the resource to the actual header. This structure is shown in Listing 2.14.

In this case, we chose to use a template to embed the specific resource header into the wrapper struct. This is because resources are not serialized into chunks like UDP messages, but are sent individually. This approach was taken for simplicity, at the expense of some network overhead. However, since most TCP resources take up more than a single packet anyway, it is not nearly a primary issue.

The actual structs used as `ResType` are the serialization types described in § 2.4.7.

### Models Sending On Demand

The `REQ_MODEL` message is tailored to our specific demo, where the client explicitly asks the server to send models on demand. We implemented a system where each model in a predefined set is numbered; when the user presses a digit button on its keyboard, a `REQ_MODEL` message is sent along with the pressed digit as a payload[9]. The server decodes these messages and serves the corresponding model if the client had not received it already.

[9] Our demo only uses a maximum of 7 models, so a single digit input is enough.

# Chapter 3

# Server Side

The server side of the engine has quite a few tasks to perform. It has to handle game assets, receive and dispatch client's commands, run the game loop, send data to the client and manage its connection and disconnection, all while keeping a coherent state of the information shared with the client. The coarse-grained scheme of the client's and server's respective tasks is reported in Figure 7.

In order to accomplish its tasks, the server uses 3 sockets: and inbound UDP socket, an outbound UDP socket and a bidirectional TCP socket. These network endpoints are used by various threads in different ways. More specifically, the server uses a total of 6 threads:

1. the **main thread**, responsible for starting the server and running the main "game loop"[1];
2. the **UDP active thread**, which sends geometry data, object transforms and dynamic lights' updates to the client;
3. the **UDP passive thread**, which receives UDP ACKs from the client and tells the UDP active thread to stop sending the acknowledged packets;
4. the **TCP active thread**, that is the one pushing resources to the client (as described in § 2.5.2);
5. the **TCP passive thread**, that receives all TCP packets coming from the client and dispatches them to the appropriate threads;
6. the **keepalive listening thread**, which periodically checks for the latest time a `KEEPALIVE` message was received and drops the client if that time is beyond a certain threshold.

## 3.1 Overview of the Server Implementation

The main actor of the server-side implementation is the `Server` struct. It works as the aggregator of all the main pieces that together form the server-side architecture. Listing 3.1 reports the Server struct in its entirety.

[1] In our demo, this is more of a mockery of an actual game loop, as it does nothing more than moving objects around and changing lights' parameters. In an actual game, this would also run gameplay logic, animations, player actions and so on.

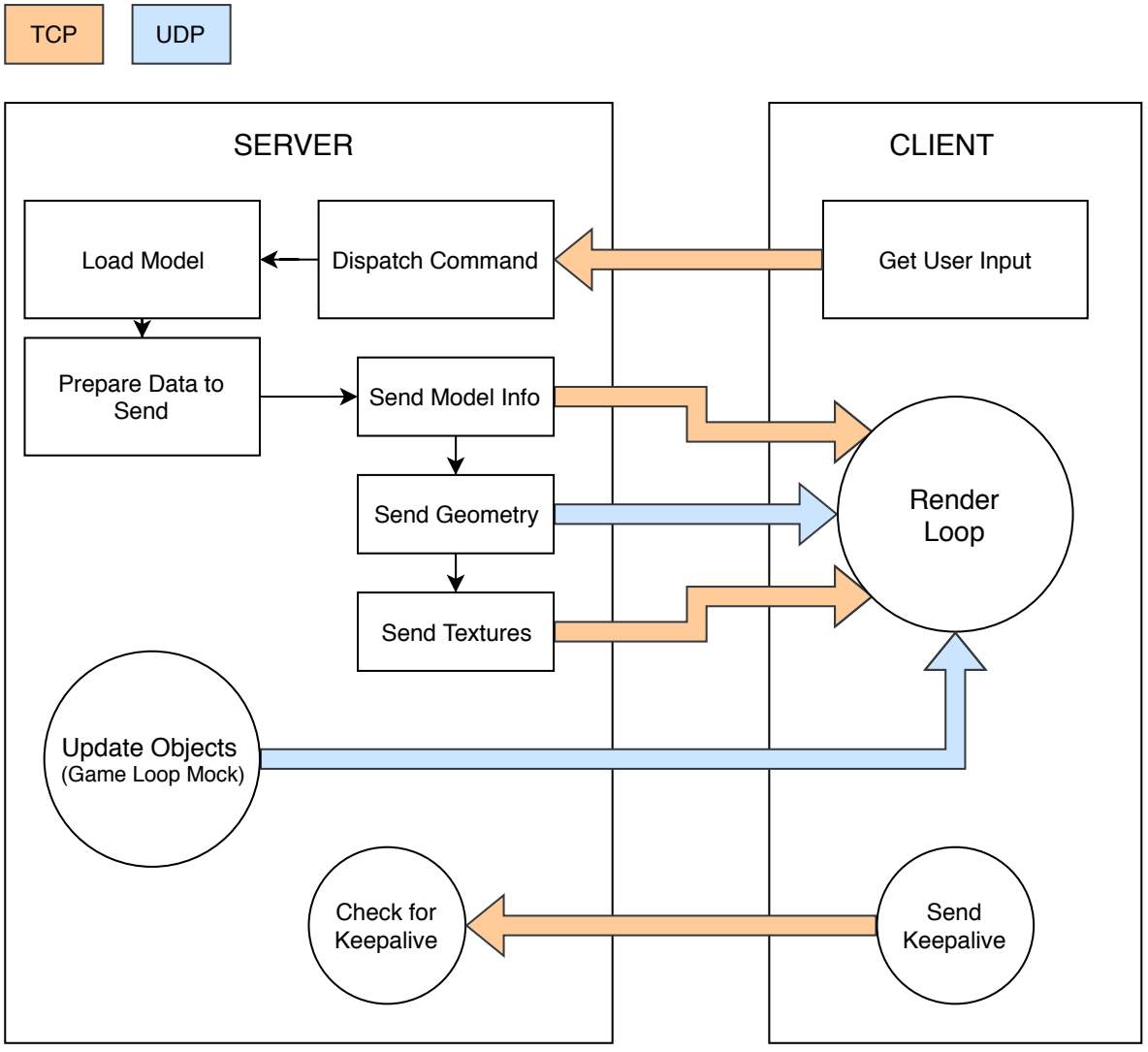


**Figure 7:** Scheme of client's and server's tasks and their relation

**Listing 3.1:** The Server struct

```
// File: server.hpp

struct Server {
  std::vector<uint8_t> memory;
  StackAllocator allocator;

  struct {
    Endpoint udpActive;
    Endpoint udpPassive;
    Endpoint reliable;
  } endpoints;

  struct {
    std::unique_ptr<UdpActiveThread> udpActive;
    std::unique_ptr<UdpPassiveThread> udpPassive;
    std::unique_ptr<TcpActiveThread> tcpActive;
    std::unique_ptr<KeepaliveListenThread> keepalive;
    std::unique_ptr<TcpReceiveThread> tcpRecv;
  } networkThreads;

  std::string cwd;

  ClientToServerData fromClient;
  ServerToClientData toClient;

  ServerResources resources;
  Scene scene;
  /** Keeps track of resources sent to the client */
  cf::hashset<StringId> stuffSent;

  BlockingQueue<TcpMsg> msgRecvQueue;

  /** Constructs a Server with `memsize` internal memory. */
  explicit Server(std::size_t memsize);
  ~Server();

  void closeNetwork();
};
```

The Server struct owns the main memory buffer used by the whole server and its submodules. This memory is reserved upon Server creation and disposed automatically when the Server object is destroyed. A stack allocator (see § 2.4.6) is used to manage this memory and to allocate the proper fraction of it to each component that needs it.

Since all the components using server memory have the same lifetime as the Server object itself, a stack allocator is the ideal choice for memory management.

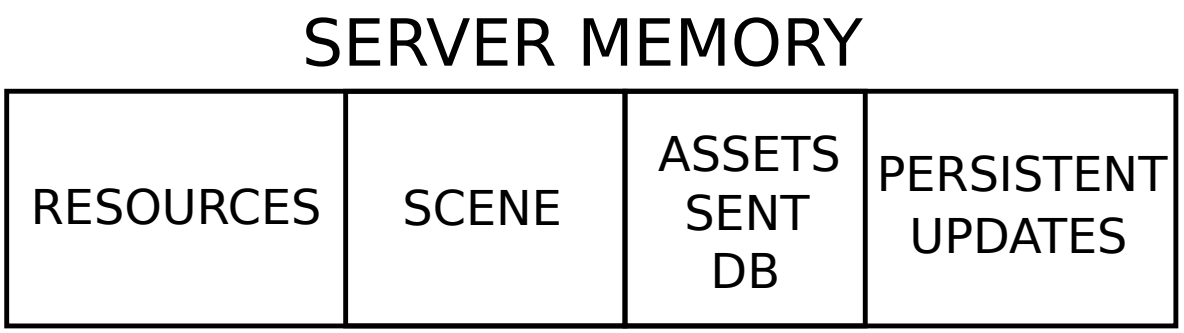


**Figure 8:** Server Memory Subdivision (chunks not to scale)

As reported in Figure 8, the server's main memory is divided into 4 chunks:

1. the first chunk is the memory area where all resources are loaded and stored. These resources are 3D models, textures and shaders; this is the biggest chunk, accounting for 2/3 of the total memory;
2. the second chunk stages the memory used by the server's scene tree, used to spatially organize loaded objects as nodes with a given transform, name and type;
3. the third chunk is used as the backing memory for a hash set containing the names (stored as `StringIds`) of all objects that were already sent to the client. This is used to avoid sending assets multiple times;
4. the fourth chunk is the backing memory for a hash map where UDP messages that require acknowledgement by the client (known as "persistent" messages) are stored until their ACK message is received.

After the main memory and its allocator, the following fields in the Server struct are the network endpoints and threads. Network threads are implemented as RAII classes whose thread and routine start upon creation, so they must be stored as pointers in the Server struct in order to be correctly created at the proper time.

For convenience, the Server structs also stores the path to the current working directory, i.e. the directory containing the server's executable.

**Listing 3.2:** ClientToServerData

```
// File: server.hpp

struct ClientToServerData {
  std::vector<uint32_t> acksReceived;
  std::mutex acksReceivedMtx;
};
```

The `fromClient` and `toClient` fields contain several fields related respectively to server-to-client and client-to-server communication. The `ClientToServerData` struct merely contains the list of ACKs received, which is the way the UDP passive thread passes that information to the UDP active thread so it can unqueue all the acknowledged messages (see Listing 3.2).

The `ServerToClientData` struct is more complex and will be described later along with the details of UDP threads.

Following these fields, the Server structs also contains the structure managing all resources, the one managing the scene, the hash set with the assets sent and finally a queue where all received TCP messages are staged before being decoded[2].

The Server struct only has three methods: the constructor, which allocates server memory and splits it into chunks as described above; `closeNetwork`, which terminates all network threads and closes all the sockets, and the destructor, which does nothing more than calling `closeNetwork`.

## 3.2 Server Resources

The following sections will describe all the resource types managed by the server and their internal representation and handling.

### 3.2.1 Models

A fundamental task of the server is loading 3D models and keeping them in memory in a suitable format as long as needed. For the low-level parsing of model formats such as OBJ or COLLADA we use the library *Assimp*. *Assimp* is a powerful library capable of parsing a multitude of model formats and providing information about their used materials.

#### Material

A "material" is a grouping of information that specify how to shade a particular object. Materials can contain myriad information, but for this thesis we limit our scope to the following properties:

[2] An analogous queue for UDP messages is not required, as currently only the UDP passive thread receives and processes those messages, with no demuxing needed.

**Listing 3.3:** Server's Material definition

```
// File: model.hpp

struct Material {
  StringId name = SID_NONE;

  std::string diffuseTex;
  std::string specularTex;
  std::string normalTex;
};
```

- the diffuse texture, used to determine the object's base color;
- the specular texture, used to map the way different parts of the object reflect the light more or less intensely;
- the normal map texture, used to simulate details, etchings and reliefs on a topologically flat surface.

Therefore, our definition of Material is very simple, as depicted in Listing 3.3. Like for most assets we handle, we give materials a name, which is used to identify it univocally. The material's name is simply the hashed string id of its original name, as specified in the model definition.

It should be noted that the `Material` structure reported in Listing 3.3 is not the only definition of a "material" in our engine, but only the server's internal definition. In fact, both the client and the common layer define their own version of Material in a way that is more convenient for their use case.

### The Model Struct

We now present how a model is described server-side. Unsurprisingly, we have a `Model` structure containing all data relevant to a single model, including vertices, indices, materials and meshes. The relation between models, materials and textures is shown in Figure 9.

Since models are an important part of our engine, we need to take particular care on their implementation, lest we slow down every process that involves them and, with them, the entire engine.

To this avail, we logically divide a model into two parts: model *data* and model *information*. The model's data is the heavyweight payload consisting of geometry (i.e. vertices and indices), meshes and materials information. The model's information is the lightweight "meta-data" consisting in the model's name, number of vertices and indices as well as the *pointers* to the model's data. However, this separation is still too rough: the so-called model data comprises pieces of information that are accessed with very different patterns. While we need fast and sequential access to a model's geometry (e.g. for building the UDP geometry messages to send when the model is initially loaded), we rarely need to access the model's meshes or materials. Therefore we classify these two components as "cold data" and store them separately from both the geometry and the Model struct.

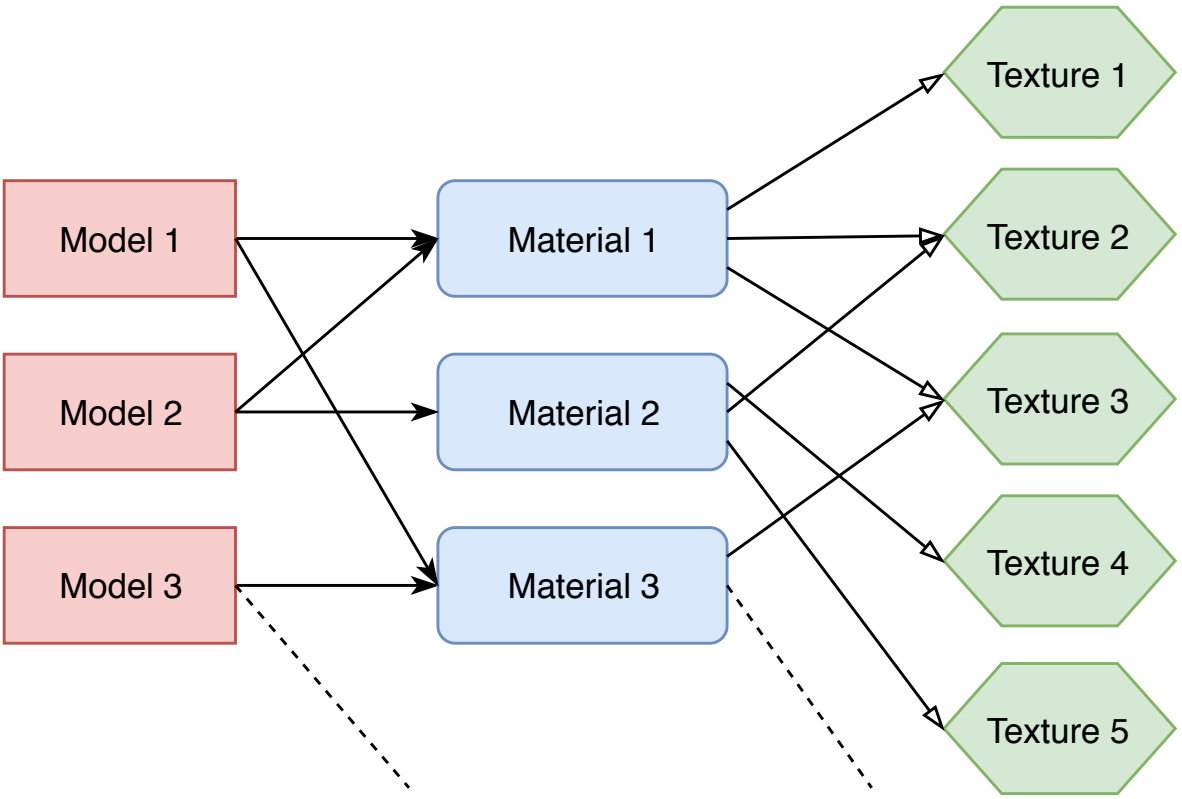


**Figure 9:** Relation between models, materials and textures

In the end, a single model is stored in 3 different memory locations: the Model struct itself, the geometric data and the cold data. The Model structure contains pointers to the other two memory locations, so one only needs an instance of the lightweight Model struct to be able to completely manage a model. Listing 3.4 shows the actual model implementation.

This subdivision makes the Model struct, which is the only part of a model we pass around frequently, very cheap to create and copy.

A 3D model is frequently made up by multiple meshes. A mesh is nothing more than a subset of the model's vertices along with a material used to render it. Our Mesh definition is reported in Listing 3.5. This definition is shared between server and client, and it is the one used to serialize a mesh for network transmission.

### Model Loading

Loading a model is done via the function `loadModel`, whose signature is reported in Listing 3.6.

This function internally uses *Assimp* to load the given file into a data structure containing the vertices (divided into meshes) and materials. It then deduplicates vertices, creating an array of indices in the process. The loaded vertices and indices are stored contiguously in the given memory buffer and materials' relevant information are saved along with meshes into a separate memory area. Then a Model object is created, made to point to the appropriate memory locations and returned.

Once loaded, a model is stored in the `ServerResources` data structure. This is one of the

**Listing 3.4:** Server's Model definition

```
// File: model.hpp

struct ModelColdData {
  std::vector<shared::Mesh> meshes;
  std::vector<Material> materials;
};

struct Model {
  StringId name = SID_NONE;

  /** Unowning pointer to the model's vertices */
  Vertex* vertices = nullptr;
  /** Unowning pointer to the model's indices */
  Index* indices = nullptr;
  /** Unowning pointer to the model's cold data */
  ModelColdData* data = nullptr;

  uint32_t nVertices = 0;
  uint32_t nIndices = 0;

  bool operator==(const Model& other) const {
    return name == other.name;
  }
};
```

**Listing 3.5:** Mesh definition

```
// File: shared_resources.hpp
// Inside namespace shared

/** A Mesh represents a group of indices into the parent model.
 *  These indices all use the same material.
 */
struct Mesh {
  /** Offset into the parent model's indices */
  uint32_t offset;
  /** Amount of indices */
  uint32_t len;
  /** Index into parent model's materials. */
  int16_t materialId = -1;
};
```

**Listing 3.6:** loadModel function

```
// File: model.hpp

/** Loads a model's vertices and indices into 'buffer'.
 *  'buffer' and 'coldData' must be pointers to initialized memory.
 *  Upon success, 'buffer' gets filled with [vertices|indices]
 *  (indices start at offset 'sizeof(Vertex) * nVertices') and
 *  'coldData' is filled with a pointer to the model's cold data.
 *  @return a valid model, or one with null vertices and indices
 *          if there were errors.
 */
Model loadModel(const char* modelPath,
  /* inout */ void* buffer,
  /* inout */ ModelColdData* coldData,
  std::size_t bufsize);
```

components that uses the server's main memory as shown in Figure 8. This memory is managed by another stack allocator and used to store the heavyweight part of loaded assets; more specifically, it stores the models' geometry, the textures' raw data and the shaders' code. These data, stored sequentially in memory due to the stack allocation strategy, are accessed through pointers contained in the lightweight representations of the resources themselves, e.g. the `vertices` and `indices` pointers in the `Model` struct. These lightweight structs are stored either in hash maps or arrays depending on their prevalent access pattern. When hash maps are used, the keys always consist in the resource's name (which is just the hashed string id of their filesystem path). The ServerResources data structure is illustrated in Listing 3.7.

Some aspects of the `ServerResource` struct need a clarification:

- the struct derives from `ExternalMemoryUser`, which is just a convenient base class for all types that have a pointer to an unowned memory buffer which they can use (but not free or reallocate); this explains why there is no explicit field pointing to the server's main memory;
- the `modelsColdData` field contains the cold data for every model in `models`; these pointers are owned by ServerResources and freed in its destructor;
- we use different types of maps for the various resources due to their different access patterns. In general, we use `cf::hashmap` whenever we need the most performance on both random and linear access to the map's data, while we use `std::unordered_map` when we do not bother seeking performance. This is because `cf::hashmap` requires us to manually manage its underlying memory, which adds complexity to the program;
- the point lights stored in ServerResources are the only field that needs no extra memory: point lights are just simple self-contained structs.

Figure 10 illustrates the memory access patterns explained above.

**Listing 3.7:** ServerResources

```
// File: server_resources.hpp

struct ServerResources : public ExternalMemoryUser {
  /** Allocator managing the staging memory
   *  for the resources data.
   */
  StackAllocator allocator;

  /** Map { resource name => resource info } */
  cf::hashmap<StringId, Model> models;
  std::vector<ModelColdData*> modelsColdData;
  std::unordered_map<StringId, shared::Texture> textures;
  std::unordered_map<StringId, shared::SpirvShader> shaders;
  std::vector<shared::PointLight> pointLights;

  Model loadModel(const char* file);
  shared::Texture loadTexture(const char* file);
  shared::SpirvShader loadShader(const char* file);

  ~ServerResources();

private:
  void onInit() override;
};
```

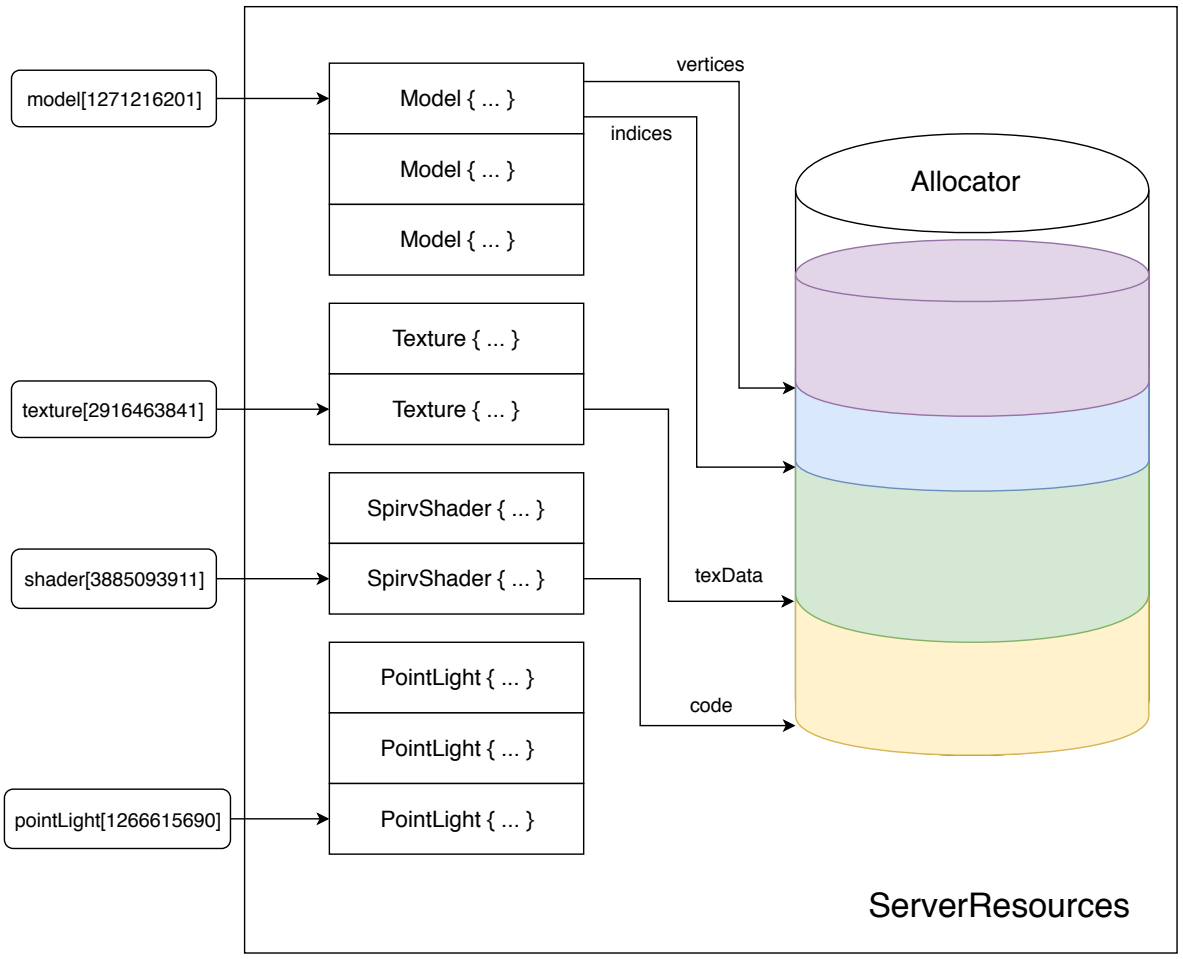


**Figure 10:** ServerResources

### 3.2.2 Textures

As far as the server is concerned, loading and managing textures is a much simpler task than models. That is because the server never needs to actually process or even decode textures, as that is a task performed by the client. All the server has to do with textures is loading them in memory, sending them via network (along with some metadata about them) and unload them as soon as possible to save memory.

The definition of the Texture data structure is reported in Listing 2.8. It merely consists in a pointer to the texture raw data, the size of said data and an enumeration value telling whether the texture uses full RGBA or just the grey channel.

The process of loading a texture is very simple:

1. load the texture file into memory;
2. create a `Texture` object and fill its `size` and `data` fields properly;
3. store it into ServerResources.

The information about color channels is given *a priori*, based on the knowledge of the intended usage for the texture: every texture used as a diffuse or normal map is RGBA, while all specular maps are monochrome. Of course, the actual textures file should reflect this convention[3].

Since textures are only needed during their network transfer, unlike models they are not just loaded and retained in memory "forever". Instead, they are only loaded one at a time just before sending them and unloaded immediately after (again, the stack allocator fits this pattern perfectly).

### 3.2.3 Shaders

Shaders are treated for the most part exactly like textures are. The server does not really need to understand the shaders' format, instead treating them like opaque binary files with a size, a pointer to their data and a type. The shader type, akin to the texture color channel information, is transmitted along with the shader data and used by the client to determine what type of shader it just received: a vertex shader, a geometry shader and so on. Along with the shader type information, we also send information about the *pass* when the shader is supposed to run, since the client's graphics pipeline is composed of multiple passes (more on that later).

Although our engine has the capability to push shaders to the client, in our demo we do not use this feature, instead relying on the client's default set of shaders, which is enough for our scope.

### 3.2.4 Scene

Assets are not the only kind of object that the server has to handle. Aside from loading and sending resources, the server also needs to run a simulation over a set of objects, which are

[3] In a more realistic use case than our demo, one would probably pack all textures in a binary blob along with their metadata, so no "guess" would be taken for their format, resulting in much more robustness.

**Listing 3.8:** Node structs definitions

```
// File: spatial.hpp

enum class NodeType { EMPTY, MODEL, POINT_LIGHT };

enum NodeFlags {
  /** A "static" node's transform cannot be changed. */
  NODE_FLAG_STATIC = 1 << 0,
};

/** A Node is a generic entity in the world with a 3D transform */
struct Node {
  /** In the case of resource-backed nodes, such as
   *  models and point lights, this name points
   *  to a resource in server.resources.
   */
  StringId name;
  NodeType type;
  Transform transform;
  uint8_t flags = 0;

  Node* parent = nullptr;
};
```

not necessarily in a one-to-one relationship with said resources. To do this, we need a separate representation of "abstract objects" that we can treat independently from assets and resources.

For this purpose, we introduce Nodes and the Scene tree. A Node represents a generic entity living in the 3D simulated world. We give a very minimalistic set of properties to a Node, as reported in Listing 3.8.

In the scope of our work, we only define 3 types of Nodes: a pure abstract node, a node representing a 3D model and one representing a point light. One can imagine to expand the `NodeType` enumeration to include more types of objects as needed.

Each node is identified, as usual, by a unique name. If such node is a "concrete" one (i.e. in our demo, one representing a model or a point light) this name is the same name as its backing resource stored in ServerResources; otherwise, it can be an arbitrary name, as long as it is not the same as any other node.

The main usefulness of a Node is that it has a *transform* representing its position, orientation and scale in the 3D simulated world. This field can be manipulated on the server side to make the client-side rendered scene change in some way.

As hinted by the `parent` field, Nodes are stored in a tree structure used to define a parent–child relationship between objects. A Node's transform is combined with all its parents' transform in a hierarchical way. This tree structure, along with some helper fields and methods, is

stored in the `Scene` struct listed in Listing 3.9. The Scene struct is another component using the server's main memory as shown in Figure 8.

The Scene struct provides methods to add, destroy and retrieve a Node by name. It has a single root node which is an abstract Node of type `EMPTY` where other nodes branch from. Additionally to the tree, it also keeps two helper data structures: a linear array of pointers to all Nodes, used to conveniently iterate on all nodes, and a hash map from nodes' names to their index in this array. This is used to accelerate the `getNode` and `destroyNode` methods.

Since all Nodes have the same size and may be allocated and freed in arbitrary order, the Scene struct uses a pool allocator (§ 2.4.6) to manage its portion of server memory where Nodes are allocated from.

## 3.3 Server Threads

The next sections will describe the inner workings of the various server threads and their relationship. This relationship is displayed in Figure 11.

The picture is as follows. Upon starting, the main thread launches the TCP Active thread, that listens on the TCP socket. This socket, which is independent on the client, stays active until the whole server is shut down. When a client connects, the TCP Active thread spawns the four threads marked as "client-specific" in Figure 11. These threads all use sockets linked to the client's IP address, therefore they exist only as long as the client is connected. If the client is dropped, all these threads are stopped, their sockets closed, and the TCP Active thread gets back to listening.

From this description alone, one may wonder why the TCP Active thread is in fact named "Active". The reason behind this choice is that, as soon as a client connects, the TCP Active thread stops listening on the socket and enters a loop called the "TCP Active Loop", which is going to be described in the following sections. At this point, listening on the TCP socket connected with the client is devolved to the TCP Passive thread, which acts as a message demultiplexer.

### 3.3.1 Main Thread

The main thread has four main tasks to perform in sequence:

1. process the command line arguments;
2. install the exit handler;
3. start the TCP Active thread;
4. run the "game loop".

The main options that may be given via command line are the IP address to use, the verbosity level and the bandwidth limitation. Logging in our engine uses a classic strategy of "log levels", with the predefined levels being (from most verbose to least): Verbose, Debug, Info, Warn and Error. By default, the log level is set to Info. Bandwidth limitation, as described in § 2.4.5, is only enabled if the `-b` flag is passed, its argument being the maximum bytes per second to allow.

The exit handler is enabled through the API exposed by the platform independence layer (as seen in § 2.4.2); its job is to close all sockets even if the program terminates in an unconventional

**Listing 3.9:** The Scene struct

```
// File: spatial.hpp

/** A Scene is a graph of Nodes.
 *  Nodes are allocated from a pool which uses the server's memory
 *  (helper data structures are allocated independently).
 */
struct Scene : public ExternalMemoryUser {

  /** Allows fast iteration on all nodes */
  std::vector<Node*> nodes;

  /** The scene tree's root */
  Node* root = nullptr;

  /** Adds node `name` of type `type` */
  Node* addNode(StringId name, NodeType type, Transform transform);

  /** Deallocates node `name` and removes it from the scene. */
  void destroyNode(StringId name);

  /** @return A pointer to node `name` or nullptr
   *  if that node is not in the scene.
   */
  Node* getNode(StringId name) const;

  void clear();

private:
  PoolAllocator<Node> allocator;
  /** Allows random access to nodes. Maps
   *  node name => node idx in the `nodes` array.
   */
  std::unordered_map<StringId, uint64_t> nodeMap;

  void onInit() override;
};
```

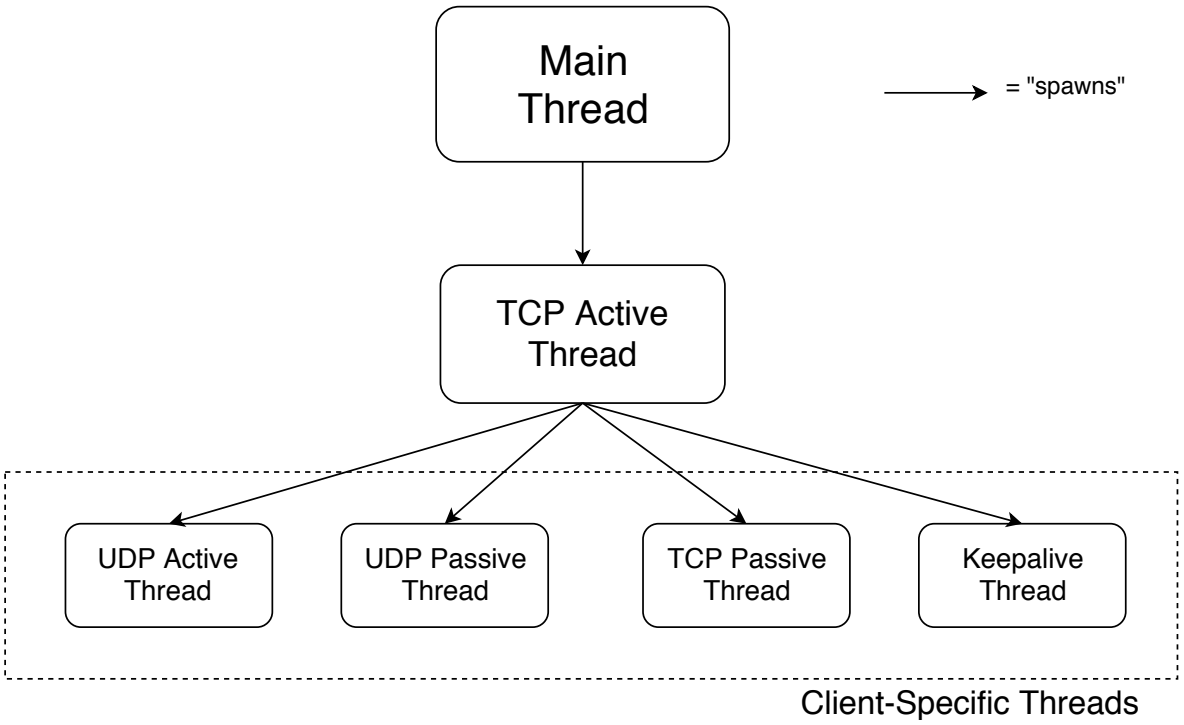


**Figure 11:** Relationship between server threads

way, which prevents `Server`'s destructor to run and normally close the network. Listing 3.10 shows the (pretty straightforward) exit handler installation process.

Launching the TCP Active thread is simply a matter of creating the TCP endpoint and the `TCPActiveThread` object. We do not need extra steps since their disposal is already taken care of by the Server struct and the exit handler.

Finally, the main thread launches the game loop mock. We call this loop the "application-stage loop" as, in use cases more complex than our demo, it also may contain the first part of the graphics pipeline's application stage, most notably geometry culling via spatial acceleration structures and similar optimizations. This loop is capped to a fixed framerate (typically 30 or 60 frames per second).

### 3.3.2 TCP Active Thread

The TCP Active thread has two major tasks:

1. accept a client's connection and, upon connection,
2. serve the client TCP resources.

In this thesis we imposed the limitation of a one-to-one relationship between the client and the server, so using a single thread for both tasks is the solution that makes most sense. This limitation also makes sense in a broader way: using multiple server processes to serve multiple clients would probably be much more convenient than having a monolithic process with complex internal multiplexing logic.

**Listing 3.10:** Enabling the exit handler

```
// File: server_main.cpp
const auto atExit = [&server]() {
  // Ensure we close the sockets even if we terminate abruptly (signals
  // like SIGKILL cannot be trapped, so in some cases the sockets will
  // still fail to close, but this cannot be fixed).
  gBandwidthLimiter.stop();
  server.closeNetwork();
  if (xplatSocketCleanup())
    info("Successfully cleaned up sockets.");
  else
    warn("Error cleaning up sockets: ", xplatGetErrorString());
  std::exit(0);
};

if (!xplatEnableExitHandler()) {
  err("Failed to enable exit handler!");
  return EXIT_FAILURE;
}
xplatSetExitHandler(atExit);
```

To accomplish its tasks, the TCP Active thread uses two nested loops with the logic shown in Listing 3.11.

The internal message loop has three main tasks itself:

1. if the client asked for a model (remember from § 2.5.2 that in our demo the client explicitly asks for model via the `REQ_MODEL` message), enqueue that model in the list of resources to send;
2. if the list of resources to send is not empty, send all of its entries and empty the list;
3. if the UDP geometry data for a model was completely sent, start sending its texture data.

These tasks all operate on different data structures. Checking for model requests is done on the received message queue, which is fed by the TCP Passive thread and consumed by the TCP Active thread. Messages are popped from the queue and for each `REQ_MODEL` message found the model enqueuing procedure is called. This procedure checks if the model was already sent and, if not, loads it in memory and adds it to the list of resources to send.

This list is implemented as an instance of the `ResourceBatch` structure listed in Listing 3.12. Such instance is a data member of the `TCPActiveThread` class. This data structure contains hash sets where the "lightweight part" of several types of resources are stored. Since the large portion of their data is only stored by pointer inside these lightweight parts, we can move resources around as values, which is convenient, without incurring in significant performance penality. Note that textures are not present in the ResourceBatch struct: they are enqueued separately for reasons that will become clear shortly.

**Listing 3.11:** TCP Active thread logic

```
while (server is running) {
  client_socket = accept(tcp_socket);

  if (client_socket is not ok)
    continue;

  perform handshake;
  start client-specific threads;

  while (client_socket is connected)
    push messages to client;

  cleanup client resources;
}
```

**Listing 3.12:** The ResourceBatch structure

```
// File: server_resources.hpp

/** A collection of (unowned) references to existing resources. */
struct ResourceBatch {
  std::unordered_set<Model> models;
  std::unordered_set<shared::SpirvShader> shaders;
  std::unordered_set<shared::PointLight> pointLights;

  std::size_t size() const { /* ... */ }
  void clear() { /* ... */ }
};
```

The ResourceBatch collection is the one which the task of sending resources operates on. The size of this collection is checked during each iteration and all resources contained in it are sent via TCP. There is one caveat here: whereas shaders and point lights are immediately sent in their entirety, only the model's metadata are sent at first. The heavyweight parts, i.e. geometry and textures, are sent separately: geometry is sent by the UDP Active thread, while textures are still sent by the TCP Active thread, but are treated as low priority resources. This is what the third task in the enumeration above is about.

When a model's metadata are sent, the paths to the textures used by the model are saved in a separate queue. This queue is then checked by the TCP Active thread at each iteration, but its elements are only sent while *no geometry is being sent simultaneously*. This *de facto* prioritizes geometry data over textures, which is our way to minimize the waiting times before displaying images client-side. In poor network conditions, we want the client to start rendering an approximate view of the scene, which it can do even before the textures are received. The player will then see the models appear with short delay, even if using default textures; they will then get updated with their actual textures after they are received too, which may possibly take a long time. This sending and updating is done on a *per mesh* basis, which makes client update its rendering accuracy much more often than it would per model. Figure 12 illustrates the process.

This approach allows for further improvements: while not present in our work, a second layer of prioritization may be added where low-resolution textures are sent before the original ones, in which way the player would see a better approximation of the actual models before receiving the highest fidelity version of the textures.

Whenever the TCP Active thread is not performing any of the aforementioned tasks, it waits on a condition variable which is notified every time any of the data structures listed above acquire new elements. In this way no CPU power is wasted in periodic polling or busy loops.

### 3.3.3 TCP Passive Thread

The job of the TCP Passive thread is very straightforward: it listens to the TCP socket connected to the client and pushes all received messages in the server's TCP message queue. Two types of messages receive a special treatment: `DISCONNECT` and `KEEPALIVE`. These messages are not pushed to the queue but rather processed immediately in the following ways:

- if the `DISCONNECT` message is received, the TCP Passive thread terminates its loop and notifies its parent thread, the TCP Active thread. As soon as the TCP Active thread notices the TCP Passive thread's termination, it kills all client-specific threads and drops the client.
- if a `KEEPALIVE` message is received, a $latestPing$ variable of type `time_point` is updated with the current time. This is the variable checked periodically by the Keepalive thread.

The TCP message queue is implemented as a `BlockingQueue` (described in § 2.4.3), so no data races occur when this thread attempts to push into it at the same time as the TCP Active thread pops elements out.

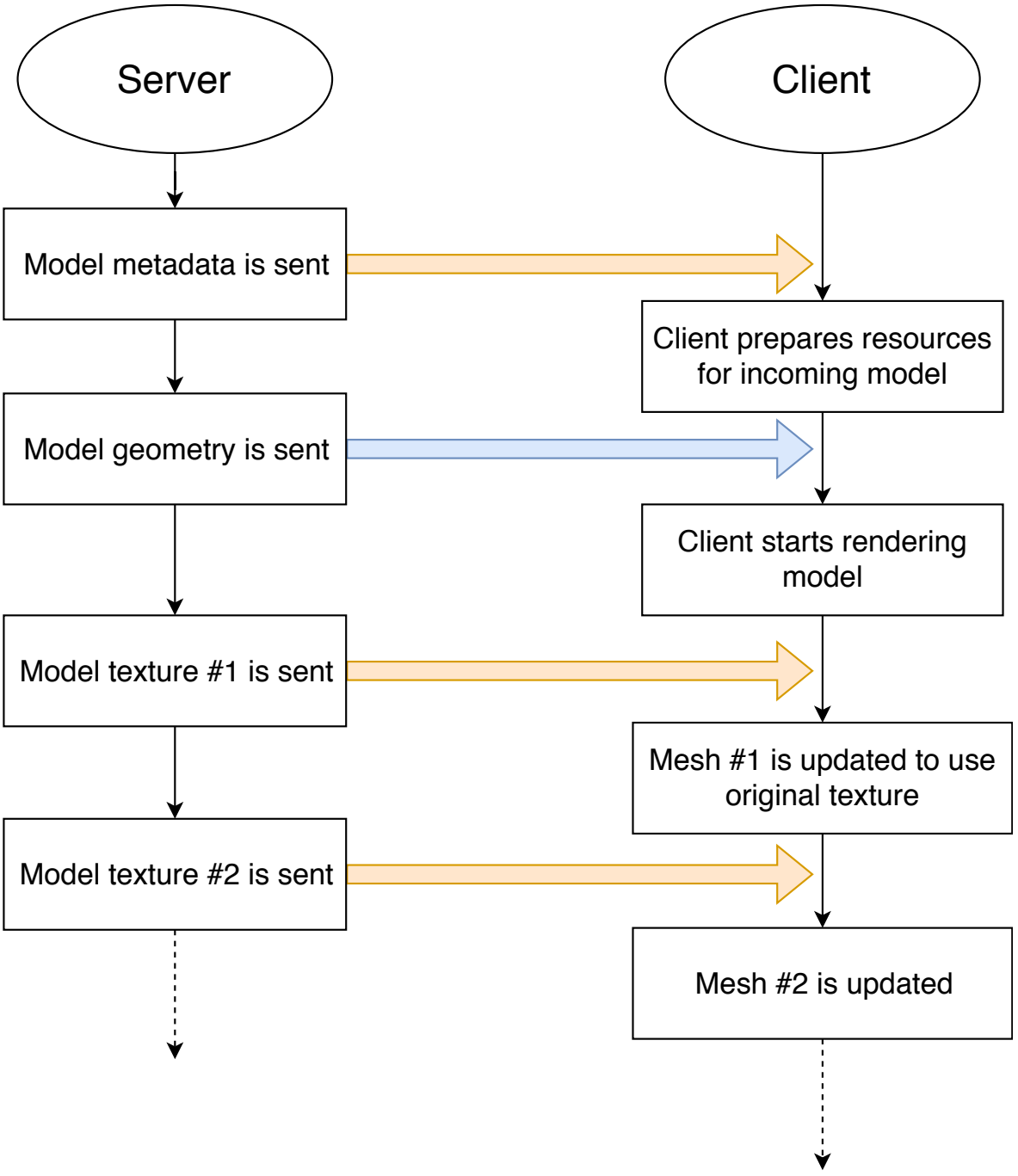


**Figure 12:** Gradual sending of resources for models, enabling earlier visual feedback for the client.

### 3.3.4 Keepalive Listening Thread

The Keepalive Listening thread is a very straightforward one as well: its job is to periodically check the time when the latest `KEEPALIVE` message (also called a *ping*) was received from the client and, if too much time passed since that moment, drop the client.

The mechanism used to drop the client is the same as that invoked by the TCP Passive thread upon receiving a `DISCONNECT` message: the Keepalive thread breaks from its loop, sets a boolean variable and notifies the TCP Active thread, that will terminate all client-specific threads and resume listening on its socket.

The one thing worth examining here is the following: since we want all client-specific threads to be interruptible at any moment (as they must be killed as soon as the client disconnects), we do not use a plain `sleep` instruction to time the loop iterations. Instead, we use a convenient `wait_for` method provided by `std::condition_variable`, which blocks until either the condition variable is notified or the given timeout expires. When we want to terminate the thread immediately, we notify the condition variable to wake it from its wait.

### 3.3.5 UDP Active Thread

The task of the UDP Active thread is to send UDP messages to the client. These messages are always meant to *update* some state on the client, so we also call them "updates" rather than "messages". Recall from § 2.5.1 that updates are divided into ones which need acknowledgement and ones that do not. We name the first kind "persistent" updates and the second kind "transitory".

UDP updates are managed via a data structure named `UpdateList`, reported in Listing 3.13. The "transitory" and "persistent" nomenclature of updates refers to their lifetime in this data structure: updates are pushed into the UpdateList by the application-stage loop running on the main thread and consumed by the UDP Active thread. For this reason, access to the transitory and persistent collections are guarded by a mutex.

Due to their different usage, the transitory and persistent collections are of different type: transitory updates only need to be browsed sequentially and are all deleted at once, while persistent updates need to be both iterated sequentially and removed by key from their list. For this reason, we use a `cf::hashmap` for them, which provides both constant access time by key and fast iteration, due to it being backed by a contiguous memory buffer.

Like in the case of the TCP Active thread, we wish to make the UDP Active thread pause whenever no updates are in the list, as well as make it resume as soon as new updates are available. We use UpdateList's field *cv* to this avail: this condition variable is notified every time a bunch of updates are added to any of the two lists.

The UDP Active thread's main loop goes as follows:

1. ensure the UpdateList has elements; otherwise, wait on *cv* until it does;
2. lock the UpdateList, copy the transitory updates on thread-owned storage, clear the original transitory collection and unlock the UpdateList. We do not operate directly on the original collection as the following sending loop may take a relative long time to execute, and this would also block the application-stage loop;
3. send all transitory updates through the UDP channel;

**Listing 3.13:** The UpdateList structure

```
// File: server.hpp

struct UpdateList {
  /** Updates in this list get wiped out every appstage loop */
  std::vector<QueuedUpdate> transitory;
  /** Updates in this list must be ACKed by the client
   *  before they get deleted.
   */
  cf::hashmap<uint32_t, QueuedUpdate> persistent;

  /** Mutex guarding updates */
  std::mutex mtx;

  /** Notified whenever there are updates to send to the client */
  std::condition_variable cv;

  std::size_t size() const {
    return transitory.size() + persistent.size();
  }
};
```

4. lock the UpdateList again. Then check for ACKs we may have received in the meantime and delete all ACKed updates from the persistent list;

5. if the persistent list reaches zero after this deletion, notify the TCP Active thread so it can wake up and start sending textures if it was waiting;

6. otherwise, send all persistent updates. Differently from the transitory updates, we do not copy the collection this time; the reason for this is that persistent updates (i.e. geometry updates) are not added at every application-stage iteration, but only rarely, so we hardly ever risk to block other threads. Persistent updates are not sent during every iteration, but have a minimum delay between consecutive sends to avoid throttling the network;

7. unlock the UpdateList.

This is likely the most complex thread loop in the engine. This complexity arises from the fact it has to interface with three other threads (the main thread, the TCP Active thread and the UDP Passive thread) and handle different types of messages and containers. The data flow among these threads is depicted in Figure 13. The application-stage loop produces and pushes UDP updates into the UpdateList; from here, transitory updates are read by the UDP Active thread and consumed, while persistent updates are first filtered depending on the ACK messages received in the meantime: every persistent update whose serial ID matches a received ACK is removed from the UpdateList, while others are sent. The ACK messages themselves are produced by the UDP Passive thread and are consumed as soon as a matching update is found in (and deleted from) the UpdateList. Despite its complexity, the UDP Active task procedure accounts for around 130 lines of code, which is still a very manageable size. Listing 3.14 reports the points described above in a pseudocode format for ease of comprehension.

From Listing 3.13 we can see that the individual elements of the UpdateList are of type `QueuedUpdate`. This type is a small data structure containing the minimal information needed to build any type of UDP message. The rationale is the following: since UDP updates are generated by a different thread than the one sending them, they need to be kept in memory and passed around through data structures like UpdateList. Building the actual full update packets when they are generated would be inefficient, as they would need to be stored somewhere to be passed to the UDP Active thread. Because UDP packets may be relatively big and variable in size, they would have to be allocated on the heap (or via some custom memory allocator), passed by pointer, and freed after use. It is much more convenient and simple to store and pass around values and build the real packet only when needed.

The way we store "generic" information while keeping a fixed size of the QueuedUpdate struct is by using an union type, like shown in Listing 3.15. This union contains any of the "concrete" QueuedUpdate structs. An enumeration value is stored alongside the union to retrieve the correct type of the QueuedUpdate.

QueuedUpdates map one-to-one with the chunks inside a UDP packet (see Figure 5). The `addUpdate` function listed in Listing 3.16 is responsible for converting a QueuedUpdate in a full chunk and adding it to a buffer. QueuedUpdates are added one by one until no space is left for fitting the next chunk, at which point the buffer is sent as a UDP packet.

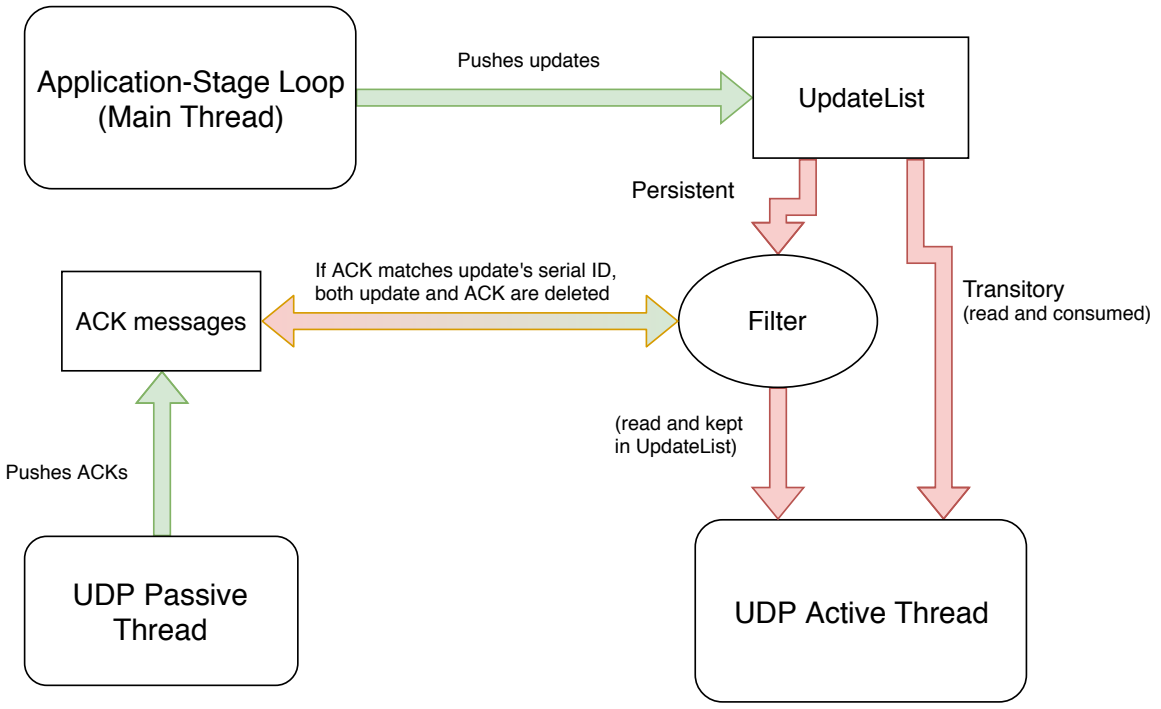


**Figure 13:** Data flow involving UDP Active thread

### 3.3.6 UDP Passive Thread

The last thread to cover is the UDP Passive thread. The job of this thread is to receive UDP messages from the client and process them. As of this work, the only client-to-server UDP message type consists in ACKs, therefore the processing required is very simple.

The UDP Passive loop has 3 steps:

1. listen on the UDP passive endpoint until a packet is received;
2. ensure the packet is of type ACK;
3. deserialize it and copy all the ACKed serial ids into a shared memory area where the UDP Active thread can pop them from.

The third step requires a bit of attention as the `acksReceived` array is shared between both UDP threads and its access must be synchronized.

**Listing 3.14:** Logic of the UDP Active loop

```
while (thread is running) {
  // Note: ‘updates‘ is the UpdateList
  if (updates.size() == 0) {
    wait on the cv until updates.size() > 0
        or thread was terminated;
  }

  // Transitory
  lock updates;
  transitory = updates.transitory.copy();
  updates.transitory.clear();
  unlock updates;

  for (each transitory) {
    send transitory update via UDP;
  }

  // Persistent
  if (now() - time_of_latest_persistent_send > SEND_INTERVAL) {
    lock updates;
    if (updates.persistent.size() > 0) {

      remove all ACKed persistent updates;

      if (updates.persistent.size() == 0) {
        notify TCP Active thread;
      } else {
        for (each persistent) {
          send persistent update via UDP;
        }
      }
    }
    unlock updates;
    time_of_latest_persistent_send = now();
  }

  ++packetGeneration;
}
```

**Listing 3.15:** QueuedUpdate structs

```
// File: queued_update.hpp

struct QueuedUpdateGeom {
  GeomUpdateHeader data;
};

struct QueuedUpdatePointLight {
  // Only need to save which light changed
  StringId lightId;
};

struct QueuedUpdateTransform {
  // Only need to save which object changed
  StringId objectId;
};

/** Each QueuedUpdate will transformed into a
 *  single udp Chunk by the UDP Active thread.
 */
struct QueuedUpdate {
  enum class Type {
    UNKNOWN,
    GEOM,
    POINT_LIGHT,
    TRANSFORM
  } type = Type::UNKNOWN;

  union {
    QueuedUpdateGeom geom;
    QueuedUpdatePointLight pointLight;
    QueuedUpdateTransform transform;
  } data;
};
```

**Listing 3.16:** Converting a QueuedUpdate to a chunk

```
// File: udp_serialize.hpp

/** @returns the offset where to write the next chunk. */
std::size_t addUpdate(
  // Buffer containing the UDP packet
  uint8_t* buffer,
  std::size_t bufsize,
  // Offset into `buffer` where the chunk should be added
  std::size_t offset,
  const QueuedUpdate& update,
  const Server& server)
{
  switch (update.type) {
    using T = QueuedUpdate::Type;
  case T::GEOM:
    return addGeomUpdate(buffer, bufsize, offset,
        update.data.geom.data, server.resources);

  case T::POINT_LIGHT:
    // ...
    return addPointLightUpdate(buffer, bufsize, offset, light);

  // ...

  }

  err("Unknown QueuedUpdate type: ", update.type);
}

/// Usage (in UDP Active loop):

std::array<uint8_t, cfg::BUFFER_SIZE_BYTES> buffer {};
int offset = 0;
while (/* there are more updates */) {
  QueuedUpdate update = /* (retrieve next update) */
  auto written = addUpdate(buffer.data(), buffer.size(),
      offset, update, server);
  if (written > 0) {
    offset += written;
    /* (advance iterator) */
  } else {
    // not enough room: send the packet and start a new one.
    sendPacket(socket, buffer.data(), buffer.size());
    /* (reset buffer) */
  }
}
```

# Chapter 4

# Client Side

The client side of the engine has two major tasks: forward user input to the server and render the scene the server sends back. Like the server, the client uses three network endpoints: a UDP active one, a UDP passive one and a TCP bidirectional one. These endpoints are shared between a total of five threads:

1. the **main thread**, which is the one performing the initial connection, starting the other threads and running the render loop. In our demo, the `REQ_MODEL` and `DISCONNECT` TCP messages are also sent directly from the main thread for convenience;
2. the **TCP passive thread**, which receives TCP messages from the server;
3. the **keepalive thread**, that periodically pings the server to prevent it from dropping the client;
4. the **UDP active thread**, that pushes ACK messages to the server;
5. the **UDP passive thread**, that receives UDP update messages.

The main thread is by far the most complex and important one, as the following sections will show.

## 4.1 Overview of Vulkan

To give a better understanding of the following sections, we will now briefly introduce the Vulkan library and its differences with the previous graphics APIs. The main problems Vulkan was designed to solve are the following [17]:

- older graphics APIs were initially designed in the era when the graphics pipeline was for the most part limited to configurable fixed functions. Whenever new functionality was added to the graphics cards, those API had to integrate those new features without breaking the old functionality, which resulted in less-than-ideal abstractions and proliferation of "API extensions". As a result, drivers for those APIs are now very complex and stratified, sometimes even inconsistent between vendors;

- older APIs try to lift the programmer from many bookkeeping aspects of graphics programming and do significant parts of the work behind the curtain. While this results in them being more approachable and easy to use, it also removes control from the application programmer, which in some cases has to work around the way the driver works in order to achieve specific features or optimizations;
- older APIs were born long before mobile GPUs, which have different architectures and best practices than desktop ones. Since older APIs give the programmer relatively little control over the application, it becomes difficult to obtain the most out of these GPUs without working around drivers significantly;
- older APIs have limited multithreading support. In our era, when CPU cores are fundamentally not getting any faster, instead getting more parallel, this hinders the possibility to take advantage of the increasing number of cores and can result in CPU-side bottlenecks.

Vulkan was designed taking all these aspects in consideration. In particular, it is designed to give programmers much more control over every aspect of the application by having fundamentally no implicit state; moreover, it has explicit multithreading support and uses a standard bytecode format for shaders. This comes at the expense of a more verbose API, which requires the programmer to explicitly initialize every aspect of the rendering process even for the simplest program.

### 4.1.1 Steps and Elements of the Rendering Process

Differently from state-machine based APIs like OpenGL that operate on an implicit "global state", Vulkan explicitly conveys all state through data structures that are passed to every function that need it. These can either be handles to opaque underlying structures or actual structs passed around by pointer.

The starting point of a Vulkan application is `VkInstance`, which describes the application itself along with its used extensions, API version and validation layers. Validation layers are a handy Vulkan feature that, if enabled, makes Vulkan check for incorrect API usage and reports any error found at runtime. As this additional check slows down the driver, they are usually only kept enabled on debug builds and disabled in release.

After creating the instance, the list of physical devices is queried and the appropriate one is selected[1]. From the physical device, a handle to the corresponding "logical device" is created. This handle is what is passed to most Vulkan functions to convey state and it stores information about the used device features.

In Vulkan, the application delivers commands to the GPU by pushing "command buffers" to the device's *queues*. A queue is a communication channel between the application and the GPU that only deals with one type of command, like graphics, compute or transfer[2]. Queues used by the application are retrieved at this point from the device.

[1] Multiple devices may be used in the same application, but in our work we only use one.

[2] A single queue may actually serve multiple purposes, but from a logical standpoint they are treated like separate queues.

**Listing 4.1:** The rendering process in Vulkan

```
prepare Vulkan resources;
record command buffer;
while (window is open) {
  poll events;

  if (need to update resources) {
    re-record command buffer;
  }

  submit draw commands;
  await frame completed;
  present frame to the swap chain;
}
```

If the application needs to present rendered image to a window, like in our case, a `VkSurface` must be created and linked to the handle provided by the windowing library. A *swap chain* is then created from the surface. A swap chain is a collection of images where the final frames are rendered and presented to the window one at a time. For each image in the swap chain, a `VkImageView` is created along with a `VkFramebuffer`, which "decorate" images with additional information needed to use them.

Before starting the actual rendering, we need to create a *render pass*. A render pass is Vulkan's way of describing the set of resources that will be used during the drawing process as well as their binding throughout the pipeline and their dependencies between pipeline stages. All this information is specified in advance so the driver can better optimize its management of resources and memory. Moreover, validation layers will complain if the programmer fails to follow the render pass description when doing the actual resource binding later.

Next, a `VkGraphicsPipeline` is created. This describes the graphics card's configurable state such as viewport size, cull mode, drawing primitive and shaders used. Once created, the state of a pipeline is fixed, so the pipeline must be recreated every time we want to change part of its configuration[3].

In order to submit commands to the graphics card, a *command pool* must be created, where command buffers are allocated from. Command buffers contain directives like "bind this vertex buffer", "draw N vertices" and so on. It is important to note that command buffers are not executed immediately: instead, they are "recorded" into a command buffer that must then be submitted to a queue: only then the graphics card will start executing the given commands. This process is asynchronous with the application, so a synchronization mechanism is required to properly wait for a frame to be rendered before presenting it to the swap chain.

The application's rendering logic therefore looks like the pseudocode in Listing 4.1.

[3]Actually there is a small number of states that can be set dynamically, like the viewport and scissor size.

## 4.2 Overview of the Client Architecture

Due to the enormous amount of state required by the rendering process and all its subtasks, the client is structured as a big central class making use of numerous external submodules. This class, called `VulkanClient`, contains the state of the application and it is responsible for its initialization and cleanup. Conversely, the external modules are mostly implemented as families of small data structures and "stateless" procedures, which are passed the needed state by the VulkanClient.

The program high-level flow goes as follows:

- initialize the application and its main parts;
- try to connect to the server;
- if the handshake is successful, initialize Vulkan resources;
- run the main loop;
- upon disconnection, clean up the resources and terminate.

The first initialization part sets up the basic Vulkan components and creates the application window. We use GLFW as cross-platform windowing library. Then, the TCP endpoint starts and attempts to connect to the server. If the connection is successful, the UDP endpoints are started as well and the remaining Vulkan resources are created. The main loop is then started and the application begins to render to the window. Of course, initially nothing is actually shown on screen as the client has received no assets yet.

In our demo, the client waits for the user to press a digit button. When a digit is pressed, the client sends a model request to the server, which it answers as described in § 3.3.

### 4.2.1 Deferred Shading

When a 3D scene is rendered, it must be *shaded*, meaning that the lights in the scene are made to interact with the objects in the scene to yield the final color of each pixel. Various shading techniques exist today, each with its pros and cons (usually the pros being better performance and scalability, the cons being increased complexity and implementation effort). The most used techniques include forward shading, deferred shading and their tile-based versions.

#### Forward Shading

**Strategy** Run lighting equations per object, then discard overlapping fragments by Z order.

✓ Straightforward to implement, easy to handle transparent objects.

✗ Scales like $N_{objects} \times N_{lights}$, cannot have many dynamic lights.

#### Deferred Shading

**Strategy** In the first pass save materials information in a "G-buffer" (discarding overlapping fragments by Z order); in the second pass, feed that information to the lighting equations to shade every pixel.

✓ Scales like $N_{pixels} \times N_{lights}$, which does not depend on the scene complexity. Therefore can handle much more dynamic lights than forward shading.

✗ Cannot handle transparency; uses more memory than forward shading due to the G-buffer.

**Forward Tiled Shading (or *Forward+*)**

**Strategy** Like forward shading, but does a pre-pass where lights are culled via a screen subdivision in tiles.

✓ Can handle even more dynamic lights than deferred shading while using less memory; handles transparency.

✗ Newer and less-documented, slightly harder to implement than deferred shading. If not done properly, the light culling pass can end up taking a lot of time, degrading performance.

We refer to [18] for an in-depth explanation and comparison of the mentioned shading techniques.

In this work, we opted to go with deferred rendering as a good compromise between implementation effort and performance. Our rendering process therefore has two passes: the G-buffer pass and the lighing pass.

### 4.2.2 Graphics Pipeline and Descriptor Sets

In this section we will briefly discuss how we handle the graphics pipeline and its resources. The Vulkan pipeline is shown in Figure 14. We can see that a pipeline must be fed multiple pieces of information in order to function: mainly *buffer bindings*, *descriptor sets* and *framebuffers*.

Buffer binding is performed via plain commands inside a command buffer, such as `vkCmdBindIndexBuffer`. As such, every time our vertex or index buffers change, the corresponding command buffer must be freed and re-recorded.

Descriptor sets work in a more convoluted way:

1. first, a *descriptor set layout* is created for each descriptor set with a distinct structure[4];
2. then, the actual descriptor sets are created, each referring to an already-defined descriptor set layout. The concrete resources contained in these descriptor sets (such as uniform buffers or textures) are bound to each descriptor set during its creation;
3. finally, descriptor sets are bound inside the command buffers just like plain buffers, with commands such as `vkCmdBindDescriptorSets`. More than one descriptor set can be bound at the same time, provided they have different binding indices.

In other words, descriptor sets are created independently from the pipeline, then plugged in and out during command recording. The pipeline's shaders will get their data from the currently bound descriptor sets. A shader can specify which descriptor set it grabs data from via the `set` layout specifier, like shown in Listing 4.2.

[4]For example, if two descriptor sets both contain one uniform buffer with the same binding point they may share the same layout. A descriptor set layout is to a descriptor set what a class is to a class instance.

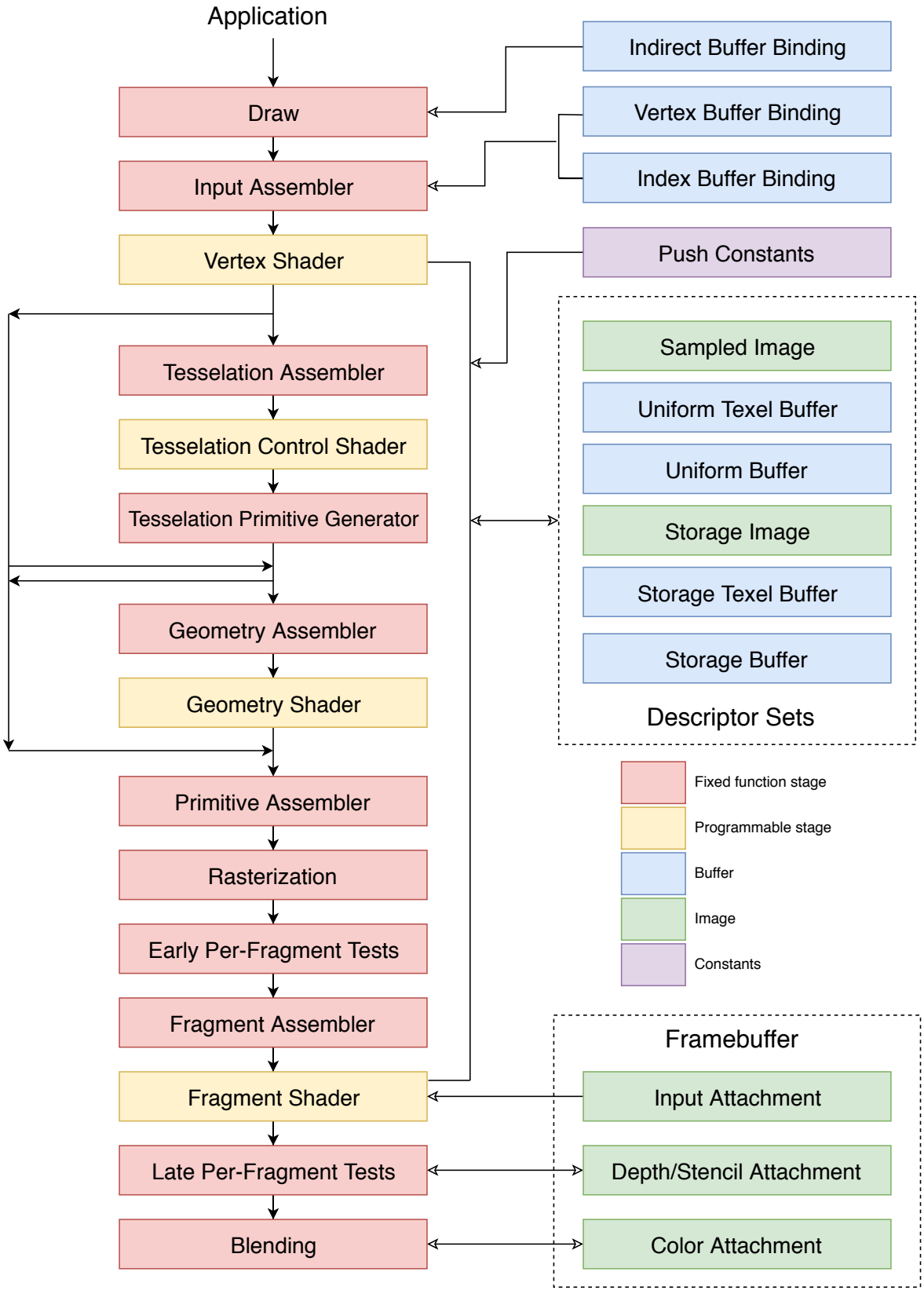


**Figure 14:** Vulkan graphics pipeline

**Listing 4.2:** A shader using multiple descriptor sets

```
// ...

// Data for this buffer will be taken from binding #0 in set #0.
layout (set = 0, binding = 0) uniform ViewUniformBuffer { ... };

// This will be taken from set #2, and so on
layout (set = 2, binding = 0) uniform sampler2D texDiffuse;
layout (set = 2, binding = 1) uniform sampler2D texSpecular;

// ...
```

The possibility of binding multiple descriptor sets is very handy, as it allows splitting the uniform buffers into different descriptor sets on an update frequency basis. Uniform buffers are then organized in a hierarchical way to minimize the data rebinding, for example:

- the uniform buffer(s) containing per-view resources (e.g. the camera position) are stored in descriptor set 0;
- the one(s) containing per-shader resources are in descriptor set 1;
- the one(s) containing per-material resources are in set 2;
- the one(s) containing per-object resources are in set 3...

...and so on [21]. In our engine, we take advantage of this by dividing our resources into four separate descriptor sets.

Finally, framebuffers are basically collections of images that the pipeline can read from and/or write to. An image (called an "attachment" in this context) can be written to by one render pass and read from by another one. The `VkRenderPass` structure is used in Vulkan to specify access dependencies and barriers between different render passes. This synchronization is required as the driver may perform the inner steps of a render pass out of order to improve rendering speed.

Vulkan also allows to divide a single render pass into multiple *subpasses*. A subpass is a Vulkan abstraction which aims to minimize writing back to memory when the content of some attachments are only needed during the mid passes of the rendering. To take deferred shading as an example, the "traditional" approach would be:

1. do the geometry pass and output information to the G-Buffer;
2. write the G-Buffer into device memory;
3. start the lighting pass;
4. read the G-Buffer back from memory;
5. do the lighting and write to the swap chain backbuffer;
6. discard the G-Buffer information.

Using subpasses enables bypassing the G-Buffer writing and reading back from device memory by keeping it in what Vulkan calls "transient memory". This can substantially speed up the rendering process [22]. Access dependencies between subpasses can be specified inside a `VkRenderPass` struct just as those between render passes. The main restriction is that all subpasses must write and read at the same resolution, which fits our use case just fine. In our engine, we use a single render pass with two subpasses: one for the geometry pass and one for the lighting pass. Each of these subpasses uses its own graphics pipeline, so we have two `VkPipeline` objects as well.

## 4.3 Client Submodules and Data Structures

This section will describe the main submodules that are used by the client and the data structures they use.

First of all, we defined an `Application` structure designed to give access to most of the client's parts; we carry around our application's state by passing the Application object as the first argument to functions that need it. Listing 4.3 gives an idea of what makes up this state. Names prefixed with `Vk-` are bare Vulkan types, those prefixed with `GLFW-` are GLFW types, while all the other ones are custom types we defined.

The following sections will describe our user-defined data structures.

### 4.3.1 Validation

The `Validation` class is a wrapper for handling Vulkan's validation layers. As we mentioned, validation layers are an opt-in mechanism to make the Vulkan API report incorrect usage during runtime. Validation layers are referred to by name and can be toggled on or off individually with very fine granularity. In our engine we simply enable all default validation layers when building in debug mode. This is done by enabling the `"VK_LAYER_LUNARG_standard_validation"` layer.

When enabling a validation layer, one must also provide its handler function to be called when a validation error occurs. Our function is a pretty simple "print to stderr" function, however we take an extra step beyond simply reporting the literal error string. By themselves, error strings raised by validation layers offer virtually no hint about the code location where the error generated from. Thus, every time we create a Vulkan resource we add a call to our custom method `validation.addObjectInfo`; this method adds the newly created Vulkan handle to a dictionary mapping the handles to their creation site. This way, when an error occurs, we can parse the error string and inject the saved locations into it to aid debugging significantly. Listing 4.4 illustrates the process.

We take advantage of the preprocessor macros `__FILE__` and `__LINE__` to know the exact location of the object creation. Unfortunately, this does not tell us where the error *occurred*, but at least it points us to the procedure that created the possibly misused resource. An example validation error message follows; words surrounded by double square brackets were injected into the plain message by our custom reporting function.

**Listing 4.3:** The Application struct

```
// File: application.hpp

struct Application {

  GLFWwindow* window = nullptr;
  GLFWmonitor* monitor = nullptr;

  VkInstance instance = VK_NULL_HANDLE;
  VkSurfaceKHR surface = VK_NULL_HANDLE;

  Validation validation;

  VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
  VkDevice device = VK_NULL_HANDLE;

  struct {
    VkQueue graphics;
    VkQueue present;
  } queues;
  std::vector<VkCommandBuffer> commandBuffers;

  VkCommandPool commandPool = VK_NULL_HANDLE;
  VkDescriptorPool descriptorPool = VK_NULL_HANDLE;

  SwapChain swapChain;
  GBuffer gBuffer;
  Buffer screenQuadBuffer;

  VkSampler texSampler = VK_NULL_HANDLE;
  VkSampler cubeSampler = VK_NULL_HANDLE;

  Resources res;

  VkPipelineCache pipelineCache = VK_NULL_HANDLE;
  VkRenderPass renderPass = VK_NULL_HANDLE;

  void init();
  void cleanup();
};
```

**Listing 4.4:** Augmenting validation errors

```
//// Creating a Vulkan resource:
VkBuffer buffer;
VLKCHECK(vkCreateBuffers(..., &buffer));
// Add the location information. This is no-op in release mode.
validation.addObjectInfo(buffer, __FILE__, __LINE__);

//// In validation.cpp:
void Validation::addObjectInfo(
    void* handle,
    const char* file,
    int line) const
{
#ifndef NDEBUG
  std::stringstream ss;
  ss << file << ":" << line;
  // Note: Vulkan handles are typedef'd as uint64_t.
  objectsInfo[reinterpret_cast<uint64_t>(handle)] = ss.str();
#endif
}
```

```
validation layer |Validation|: Object: 0x88 [[multipass.cpp:521]] (Type
   = 23) | Descriptor set 0x88 [[multipass.cpp:521]] bound as set #2
  encountered the following validation error at vkCmdDrawIndexed()
  time: Descriptor in binding #1 at global descriptor index 1 is being
   used in draw but has not been updated.
```

### 4.3.2 Buffers

Buffers are one of the most commonly used Vulkan resource types. They are basically just portions of GPU memory, which live in one of the various GPU heaps available. Some types of memory are faster but cannot be accessed outside the GPU while others are slower but can be mapped to host-accessible memory through a pointer. Using the correct type of backing memory for both buffers and images is paramount for achieving performance.

Buffers are used to store various types of data, like vertex or index data, or as a staging area during data transfers, for example when loading textures.

Throughout our engine's client code, buffers are usually referred to via the user-defined `Buffer` struct shown in Listing 4.5. This struct contains the buffer's Vulkan handle, the handle to its backing memory object, its size in bytes, its offset in its backing memory and an optional pointer to its host-mapped memory area.

Creating a Vulkan buffer involves the following steps:

1. allocate memory for the buffer, or pick an existing `VkDeviceMemory` object to use;

**Listing 4.5:** The Buffer struct

```
// File: buffers.hpp

struct Buffer {
  VkBuffer handle;
  VkDeviceMemory memory;
  VkDeviceSize size;
  /** Offset in the underlying memory */
  VkDeviceSize offset = 0;
  /** Host-mapped pointer, if any */
  void* ptr = nullptr;
};
```

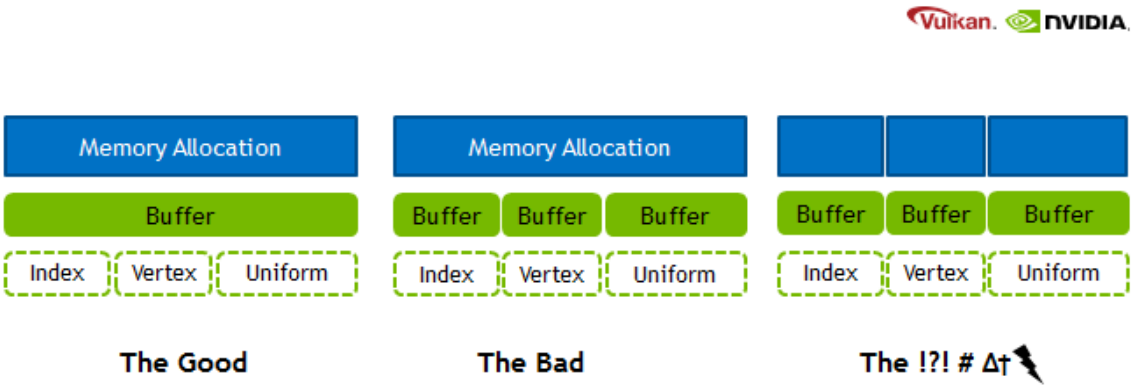


**Figure 15:** Preferred way of allocating buffers in Vulkan

2. create the buffer object;
3. bind the newly created buffer to the chosen memory object (properly setting its offset if needed);
4. (optional): if the picked memory type allows for it, map the buffer's memory to a host-accessible pointer.

Since allocating and freeing memory is in general an expensive operation, on the GPU like on the host, allocating a separate VkDeviceMemory object per buffer is a very poor strategy. Instead, the preferred method is to minimize memory allocations by sharing a single memory object with as many buffers as possible. This is made very clear by Figure 15, taken from the NVIDIA documentation for Vulkan.

To abide by these guidelines, we use two helper facilities: the *BufferAllocator*, which allocates multiple buffers using the minimum possible number of VkDeviceMemory allocations, and the *BufferArray*, that uses the least possible number of Vulkan buffers to accomodate multiple

**Listing 4.6:** Creating, mapping and destroying Buffers

```
/// Creating the buffers:
BufferAllocator bufAlloc;
Buffer vertexBuf, indexBuf, uniformBuf;
bufAlloc.addBuffer(vertexBuf, /* ... properties ... */);
bufAlloc.addBuffer(indexBuf, ...);
bufAlloc.addBuffer(uniformBuf, ...);
bufAlloc.create(app);

/// Host-map a buffer:
mapBuffersMemory(app.device, { &uniformBuf });
// Memory can now be accessed via ‘uniformBuf.ptr‘.

/// Unmap and destroy the buffers:
unmapBuffersMemory(app.device, { uniformBuf });
destroyAllBuffers(app.device, { vertexBuf, indexBuf, uniformBuf });
```

“logical buffers”. Referring to Figure 15, the first facility brings us from the rightmost image to the central one, while the latter brings us from the central image to the leftmost one.

### 4.3.3 Buffer Allocator

The `BufferAllocator` class is used to create many Buffers at once using as few memory allocation as possible. The way it works is the following:

1. an arbitrary number of buffers is scheduled for creation by calling the `addBuffer` method; this saves the buffers’ properties internally to the BufferAllocator class;
2. after scheduling all desired buffers, the `create` method is called. This looks up all the saved buffers, figures out how many different memory types are required to accomodate all of them, allocates the appropriate memory objects, creates the individual buffers and finally binds them all to the allocated memory objects. Every buffer sharing the same memory type is bound to the same memory object, at different offsets.

Allocating buffers through this class greatly reduces the number of memory allocations performed on the device. The only caveat is that all these buffers must be destroyed together to ensure all backing memory is freed exactly once. This is done via a handy `destroyAllBuffers` function. An example usage of the functionality described above is depicted in Listing 4.6.

### 4.3.4 Buffer Array

The `BufferArray` is a quite complex class used to treat $N$ actual Vulkan buffers as if they were $M \geq N$ buffers. This means we can have both the convenience of treating the “virtual” buffers as separate entities and the performance gain due to actually instancing only a few real buffers.

The BufferArray presents the interface reported in Listing 4.7. Let us go through it and explain all its parts.

First, we defined a `SubBuffer` structure which is just a Buffer endued with an additional field reporting the offset in bytes of the SubBuffer into the underlying *Buffer*. This is analogous to the *offset* field, but refers to the beginning of the Buffer rather than the memory object. The BufferArray is basically a container of SubBuffers, that can be created via the `addBuffer` method. Each SubBuffer is given a name so it can be conveniently referred to in a second moment, in order to be either retrieved from the BufferArray or removed from it.

A look at the BufferArray's constructor reveals that it can only create one single type of buffer. If a buffer with a different combination of usage flags or properties needs to be created, a different BufferArray must be used. This is not a big problem though, as the main usage of BufferArray in our engine is linked to managing uniform buffers, which all share the same usage and properties.

The way SubBuffers are allocated by the BufferArray is by first creating a Vulkan buffer, then fitting all requested SubBuffers inside it (taking care of alignment requirements). As soon as a SubBuffer does not fit the available space inside the backing buffer, another "real" buffer is created with a size of

$$max\{minBufferSize, requiredBufferSize\}$$

When a SubBuffer is deleted via `rmBuffer`, its space is marked as 'free'. All free spaces are then candidate for accomodating future allocations: the space is chosen via a closest-fit algorithm, which selects the smallest hole with enough space to fit the requested allocation. The BufferArray also takes care to merge adjacent holes every time a SubBuffer is removed. Figure 16 illustrates the process.

### 4.3.5 Images

Images are another commonly used resource in any rendering engine. Vulkan images, like buffers, are backed by a device memory object, but their memory is accessed in a different (and less direct) way. First, images cannot have their memory directly mapped to the host through a pointer: to copy data into or from an image one must employ a "staging buffer" as a transfer proxy between the host and the device. Moreover, images are manipulated throughout the graphics pipeline via *image views*, opaque objects that know how to interface with the image data.

Like in the case of Buffers, we wrap the bare Vulkan images into our custom type, shown in Listing 4.8. The `Image` struct is very similar to `Buffer`, but instead of the size and pointer fields it has an optional *view* and a *format*.

Like for buffers, images can be created atop the same backing memory object and it is recommended to do so. For this purpose, we have an *ImageAllocator* class analogous to the BufferAllocator described above.

### 4.3.6 Textures and Texture Loader

Images are commonly used for storing textures. A texture is basically a Vulkan image along with a `VkSampler` object used to sample it. As such, it needs no additional user-defined type: we just

**Listing 4.7:** BufferArray interface

```
// File: buffer_array.hpp

struct SubBuffer : public Buffer {
  /** Offset in the underlying Buffer */
  VkDeviceSize bufOffset;
};

class BufferArray {
  // ... omitted private fields ...
public:
  explicit BufferArray(
      VkBufferUsageFlags usage, VkMemoryPropertyFlags properties)
    : usage{ usage }, properties{ properties }
  {}

  void initialize(
    const Application& app, VkDeviceSize minBufferSize = 0);
  void cleanup();

  /** Allocates a backing buffer which is
   *  at least 'initialSize' bytes.
   */
  void reserve(VkDeviceSize initialSize);

  /** Maps all currently and future
   *  allocated buffers to host memory.
   */
  void mapAllBuffers();
  /** Unmaps all currently allocated buffers
   *  and stops mapping future ones.
   */
  void unmapAllBuffers();

  /** Adds a logical buffer to the array and returns it.
   *  If the buffer fits the already-allocated buffer(s),
   *  it will be part of one of them;
   *  else, a new backing Buffer will be allocated.
   */
  SubBuffer* addBuffer(StringId name, VkDeviceSize size);
  SubBuffer* getBuffer(StringId name) const;
  void rmBuffer(StringId name);
};
```

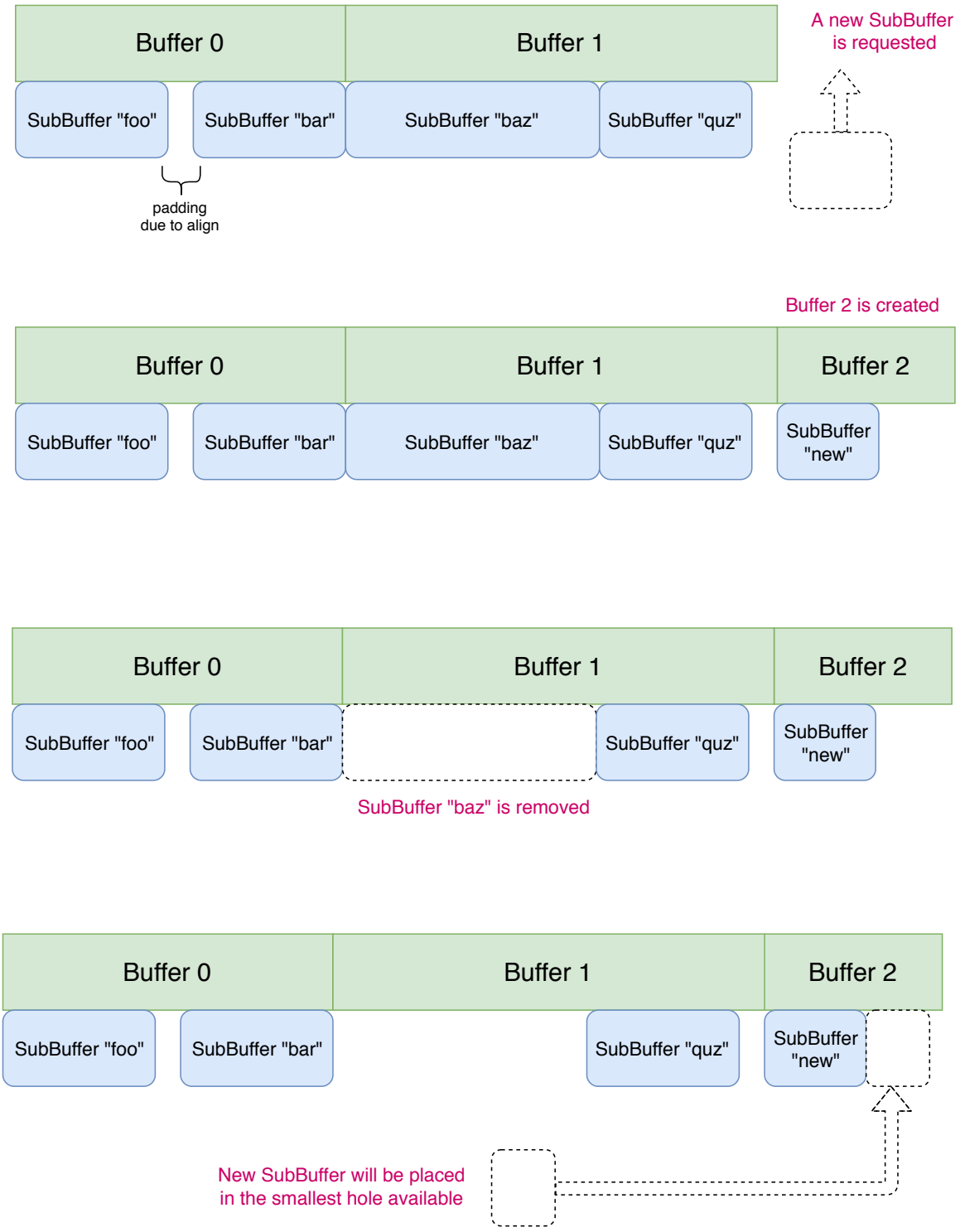


**Figure 16:** SubBuffer allocation and deallocation in BufferArray

**Listing 4.8:** The Image struct

```
// File: images.hpp

struct Image {
  VkImage handle = VK_NULL_HANDLE;
  VkDeviceMemory memory = VK_NULL_HANDLE;
  VkDeviceSize offset;
  VkImageView view = VK_NULL_HANDLE;
  VkFormat format;
};
```

use `Image` to represent textures. However, texture have an additional requirement other than being allocated – which we do via our ImageAllocator: once allocated, they must be filled with the texture data.

To this purpose, we introduce the `TextureLoader` class, listed in Listing 4.9. The TextureLoader has a very similar design to BufferAllocator and ImageAllocator: first, textures are scheduled to be loaded with an `addTexture` method, then they are all created at once by calling the `create` method.

Calling `addTexture` causes the texture data to be loaded from either a file or a portion of memory where they reside and decoded[5]. Moreover, the desired texture properties (such as the format) are stored alongside the Image pointer they will be bound to.

Calling `create` creates the needed backing Images (through an ImageAllocator), binds them to the pointers passed to `addTexture` and copies the loaded texture data inside them.

The `addTexture` method requires two explanations: first, it has two overloads: one is used to load a texture from a local file, the other is used to load a texture from a memory buffer. This second overload is used to decode textures passed via network.

Second, both overloads have an `Async` version. The `addTextureAsync` methods load and decode the image data asynchronously and return a `std::future` object. This is a "promise" object that needs to be awaited on before calling `create`. Decoding images asynchronously provides great speedup to texture loading, so we almost always use these variants of the `addTexture` method.

### 4.3.7 Swap Chain

The `SwapChain` struct, shown in Listing 4.10 contains resources related to the swap chain. Aside from the Vulkan handle to the swap chain itself, it contains its extent, format, the underlying images along with their views (that Vulkan requires to operate on images), the associated framebuffers and the depth buffer image.

As the Listing shows, the SwapChain has a `destroy` method but no "initialize" one. That is because the various parts of the SwapChain are created by separate functions in different moments, as they depend on a specific creation order of Vulkan resources.

[5]We use the *stb_image* library to decode images.

**Listing 4.9:** TextureLoader extract

```
// File: textures.hpp
class TextureLoader {
  struct ImageInfo {
    VkFormat format;
    uint32_t width;
    uint32_t height;
  };

  /** Buffer used to transfer image data from
   *  host to device memory
   */
  Buffer& stagingBuffer;
  std::size_t stagingBufferOffset = 0;

  std::vector<ImageInfo> imageInfos;
  std::vector<Image*> images;

public:
  explicit TextureLoader(Buffer& stagingBuffer)
    : stagingBuffer{ stagingBuffer }
  {}

  /** Load a texture from raw data pointed by 'texture'.
   *  Upon 'create()', 'image' will be bound to a valid Image.
   */
  bool addTexture(Image& image, const shared::Texture& texture);
  /** Load a texture from file with given format.  */
  bool addTexture(Image& image, const std::string& texturePath,
          shared::TextureFormat format);

  /** Like 'addTexture', but asynchronous.
   *  Returns a 'future' which must be waited for
   *  before calling 'create'. It contains the success state.
   */
  std::future<bool> addTextureAsync(Image& image,
          const shared::Texture& texture);

  std::future<bool> addTextureAsync(Image& image,
          const std::string& texturePath,
          shared::TextureFormat format);

  void create(const Application& app);
};
```

**Listing 4.10:** The SwapChain struct

```
// File: swap.hpp

struct SwapChain {
  VkSwapchainKHR handle = VK_NULL_HANDLE;
  VkExtent2D extent;
  VkFormat imageFormat;

  std::vector<VkImage> images;
  std::vector<VkImageView> imageViews;
  std::vector<VkFramebuffer> framebuffers;
  Image depthImage;

  void destroy(VkDevice device);
};
```

### 4.3.8 G-Buffer

The `GBuffer` struct groups the resources related to the G-Buffer, namely the images where the data gathered from the first shading pass are stored. The type of information one can preserve for the shading pass, i.e. the *format* of the G-Buffer, varies from case to case. In our work, we used a very simple G-Buffer format consisting in three framebuffers:

1. a **position** buffer, containing in each fragment the 3D world position of the nearest object. This is stored as a vector of three 32-bit floats;
2. a **normal** buffer, containing in each fragment the 3D direction of the surface normal of the nearest object. This too is stored as a vector of three 32-bit floats;
3. an **albedo-specular** buffer, containing in each fragment the diffuse color of the nearest object and its specular exponent. This is stored as a vector of four 8-bit floats.

Figure 17 shows a visual representation of the final image along with its separate channels.

As shown in Listing 4.11, the GBuffer struct contains the three needed images and a `destroy` method, analogous to the SwapChain. In this case it also contains a method that creates the three attachments at once, using the ImageAllocator to minimize the number of required memory allocations.

### 4.3.9 Resources

The `Resources` struct is a key-value container for most of the auxiliary resources used during the rendering process which are not linked to a particular submodule. The types of such resources are the following:

- *pipelines*, the Vulkan objects representing the graphics pipelines;

**Listing 4.11:** The GBuffer struct

```
// File: gbuffer.hpp

struct GBuffer {
  Image position;
  Image normal;
  Image albedoSpec;

  void createAttachments(const Application& app);

  void destroy(VkDevice device);
};
```

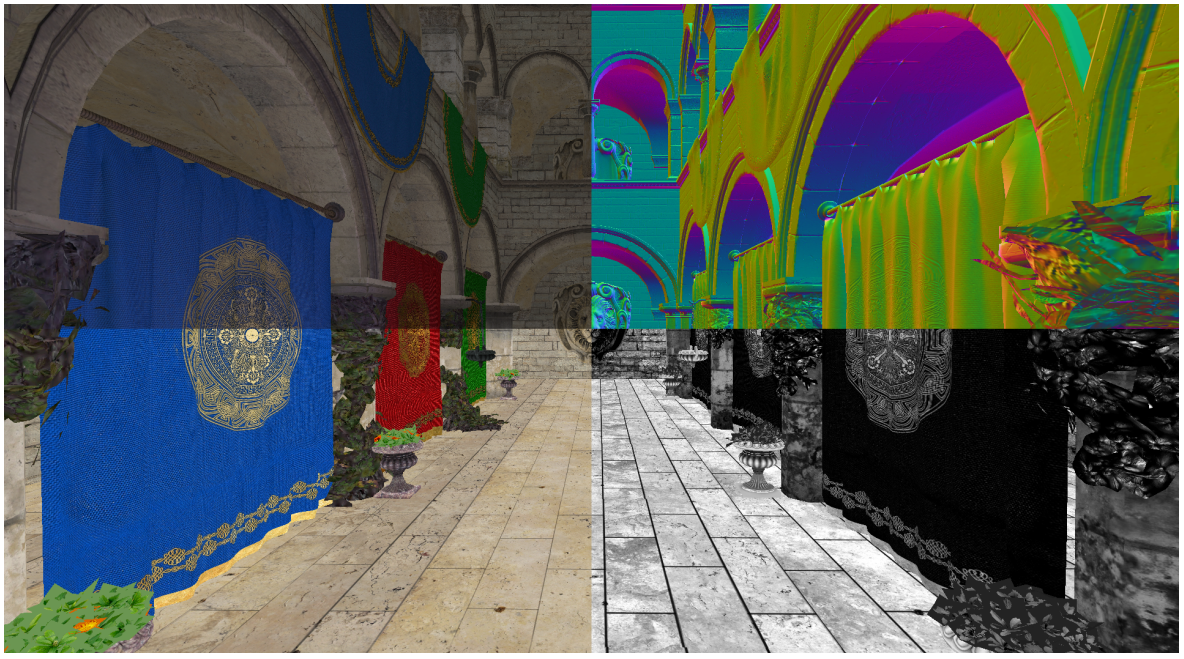

**Figure 17:** Visual representation of the G-Buffer content. From top-left to bottom-right: final result, normals, albedo and specular.

**Listing 4.12:** Accessing Resources

```
// Adding a resource to the Resources struct:
// (app.res is an instance of Resources)
app.res.pipelines->add(sid("my_pipeline"), myPipelineHandle);

// Retrieving a resource by name:
VkDescriptorSet descSet = app.res.descriptorSets->get(sid("view"));

// Disposing of all resources:
app.res.cleanup();
```

- *pipeline layouts*, used to describe a graphics pipeline's binding layout;
- *descriptor sets*, collections of resources such as textures and uniform buffers, that are bound to a pipeline during the rendering process;
- *descriptor set layouts*, containing the "blueprint" used to create a descriptor set;
- *semaphores*, Vulkan objects used to synchronize device operations with the host.

The Resources struct contains a separate key-value map for each of these types. The keys are `StringId`, allowing for a convenient naming of the individual resources. Listing 4.12 shows how these resources are accessed.

### 4.3.10 Geometry

The `Geometry` struct is a container for geometry data on the device. It is used to accomodate all models' geometry data inside a single pair of buffers (a vertex buffer and an index buffer). Alongside those buffers, the Geometry struct contains a map between model names and the location of their geometry, i.e. the offset of vertices and indices into the main buffers. The Geometry struct is shown in Listing 4.13.

Every time new models are loaded Geometry must be updated by fitting new vertices and indices inside the existing buffers. If any of the buffers is too small to fit all new vertices or indices, it is expanded by doubling its size. In this case, all existing vertices or indices must be copied to the new buffer, but their locations remain unchanged. Figure 18 shows how geometry data are stored and referenced inside Geometry.

### 4.3.11 Network Resources

When a resource is received via network, depending on its type, it may require some processing in order to be used by the client. For example, textures and shaders need to be decoded and loaded into device memory; materials need to collect the Vulkan handles to their referenced textures, and so on. The `NetworkResources` struct simply stores all the post-processed versions of network-received resources inside dynamic arrays or maps. It also contains the default textures, used as placeholders when the actual textures are not available for some material.

**Listing 4.13:** The Geometry struct

```
struct Geometry {
  /** Single buffer containing all vertices for all models */
  Buffer vertexBuffer;

  /** Single buffer containing all indices for all models */
  Buffer indexBuffer;

  /** Offsets in bytes of the first vertex/index
   *  inside the buffers for each model
   */
  struct Location {
    VkDeviceSize vertexOff;
    VkDeviceSize vertexLen;
    VkDeviceSize indexOff;
    VkDeviceSize indexLen;
  };

  /** Maps modelName => location into buffers */
  std::unordered_map<StringId, Location> locations;
};
```

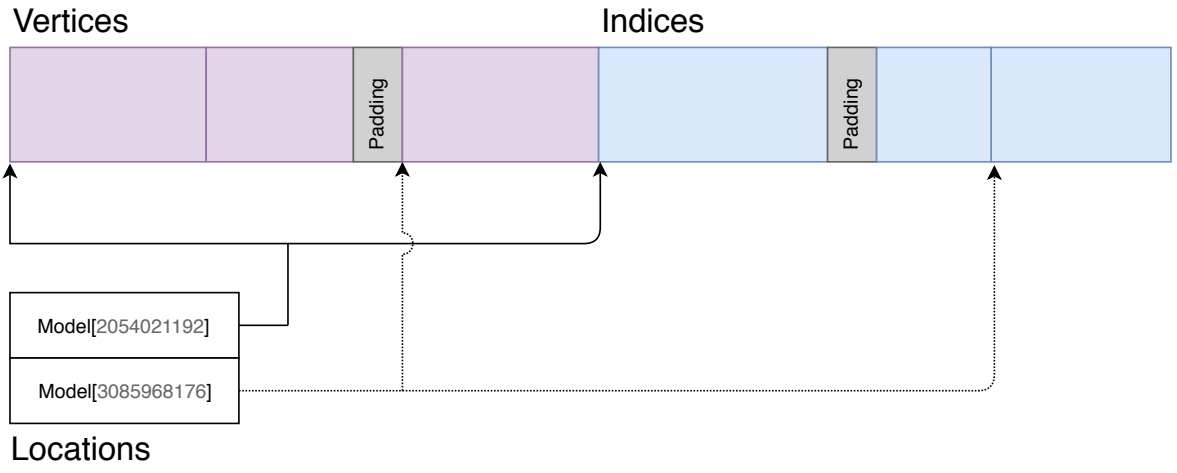


**Figure 18:** How data is stored and referenced inside Geometry

### 4.3.12 Object Transforms

We call "objects" all assets that have a 3D position, orientation and scale in the simulated world. In our engine, objects are either models or point lights. The client needs to keep track of all objects' position, orientation and scale in order to properly draw them. This is obtained through a dictionary called *objTransforms* that maps objects' names to their transform matrix, stored as a `glm::mat4` type.

### 4.3.13 Memory Monitor

As a debugging aid, we created a class called `MemoryMonitor`, that keeps track of memory allocations and deallocations on the GPU.

In "debug" mode only, immediately after every allocation or free, we call the `newAlloc` or `newFree` method on the MemoryMonitor, which causes it to save the memory allocation data internally. These data can be retrieved by calling `report`, which presents a summary of all types of memory that were used, how many allocations and frees happened so far and the maximum device memory size reached by the application.

This small debugging utility proved very useful while optimizing device-allocating portions of code.

## 4.4 Client Threads

The next sections, analogously to 3.3, will describe the inner workings of the client threads and their relationship. As Figure 19 shows, the client threads are almost the exact mirror of the server threads, except they are all directly spawned and "owned" by the main thread and the TCP Active thread is missing. This is for two reasons:

1. all client threads have the same lifetime as the client's main process: they are all terminated together. On the other hand, the server has to differentiate between client-specific and non–client-specific threads;
2. the client can only send trivial TCP messages, so dedicating a whole thread to TCP message sending seemed excessive to us.

The client's behaviour is the following: upon start, the client tries to connect to the server (from the main thread). If the connection is successful, it starts the UDP threads and sends a `READY` message. When a `READY` is received back from the server, the TCP Passive thread and the Keepalive thread are started as well.

### 4.4.1 UDP Active Thread

The task of the UDP Active thread is to send `ACK` messages upon receiving geometry update packets. ACKs are pushed into a queue by the main thread and popped and sent by this thread. The UDP Active loop is pretty simple, as shown in Listing 4.14.

Like most thread loops, it first checks if there are data to operate on and, if there are not, it waits on a condition variable until data is available. Then it groups all ACKs into packets and sends them one at a time.

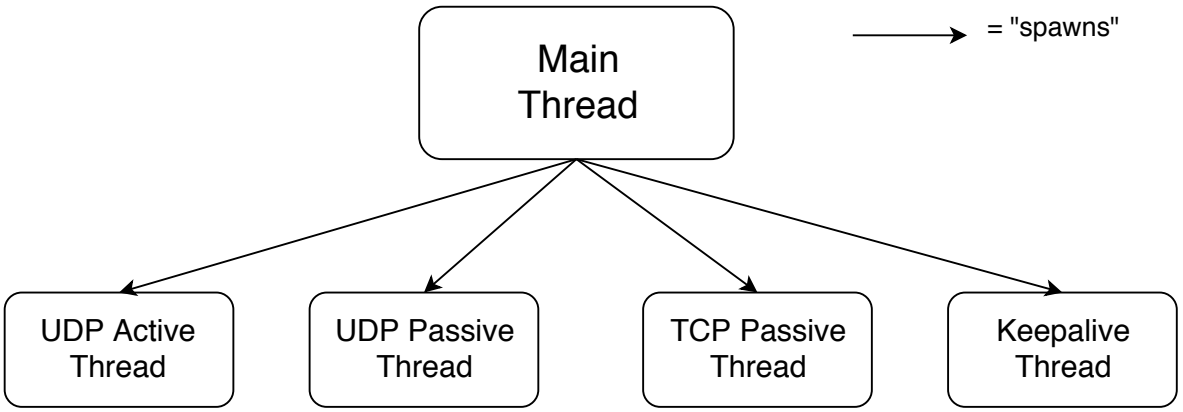


**Figure 19:** Relationship between client threads

**Listing 4.14:** UDP Active loop (pseudocode)

```
// File: client_udp.cpp:
while (connected) {
  wait on condition_variable until acks.list is not empty;

  AckPacket packet;
  foreach (ack in acks.list) {
    add ack to packet;
    if (packet is full) {
      send packet;
      reset packet;
    }
  }

  clear acks.list;
}
```

### 4.4.2 UDP Passive Thread

The design of the UDP Passive thread is simple as well. We want to ensure that as few incoming packets as possible are lost, therefore we offload the packets decoding part to the main thread. The UDP Passive thread just has to validate the incoming packets and copy the received data from its local buffer to a buffer shared with the main thread. This way the UDP Passive thread can be calling `receivePacket` very often and drop little to no packets under normal conditions.

UDP packets have the format shown in Figure 5. The UDP Passive thread, after validating a packet, merely copies the entire payload (omitting the packet header) into a buffer where the main thread will read from to decode it.

**Listing 4.15:** The Keepalive loop

```
static void keepaliveTask(const Endpoint& ep,
    std::condition_variable& cv)
{
  std::mutex mtx;
  while (ep.connected) {
    std::unique_lock<std::mutex> ulk{ mtx };

    // Using a condition variable instead of sleep_for
    // since we want to be able to interrupt it.
    const auto r = cv.wait_for(ulk, std::chrono::seconds{
        cfg::CLIENT_KEEPALIVE_INTERVAL_SECONDS
        });
    if (r == std::cv_status::no_timeout && !ep.connected) {
      info("keepalive task: interrupted");
      break;
    }
    if (!sendTCPMsg(ep.socket, TcpMsgType::KEEPALIVE))
      warn("Failed to send keepalive.");
  }
}
```

### 4.4.3 Keepalive Thread

The Keepalive thread is responsible for pinging the server at regular intervals in order to prevent it from dropping the client for timeout. The Keepalive loop has just two steps:

1. wait for a fixed duration, and
2. send the `KEEPALIVE` message via TCP.

The waiting is accomplished through `std::condition_variable::wait_for`, rather than a plain sleep, to make the task interruptible from outside. Listing 4.15 shows the Keepalive loop in its entirety.

### 4.4.4 TCP Passive Thread

The client's TCP Passive thread has the job to receive incoming TCP messages and handle them. TCP messages and their receiving loops are structured in a hierarchical way, with some messages leading from one loop to another. Each loop only accepts a predefined set of messages, and invalid messages are ignored. Figure 20 shows the hierarchy of TCP message loops.

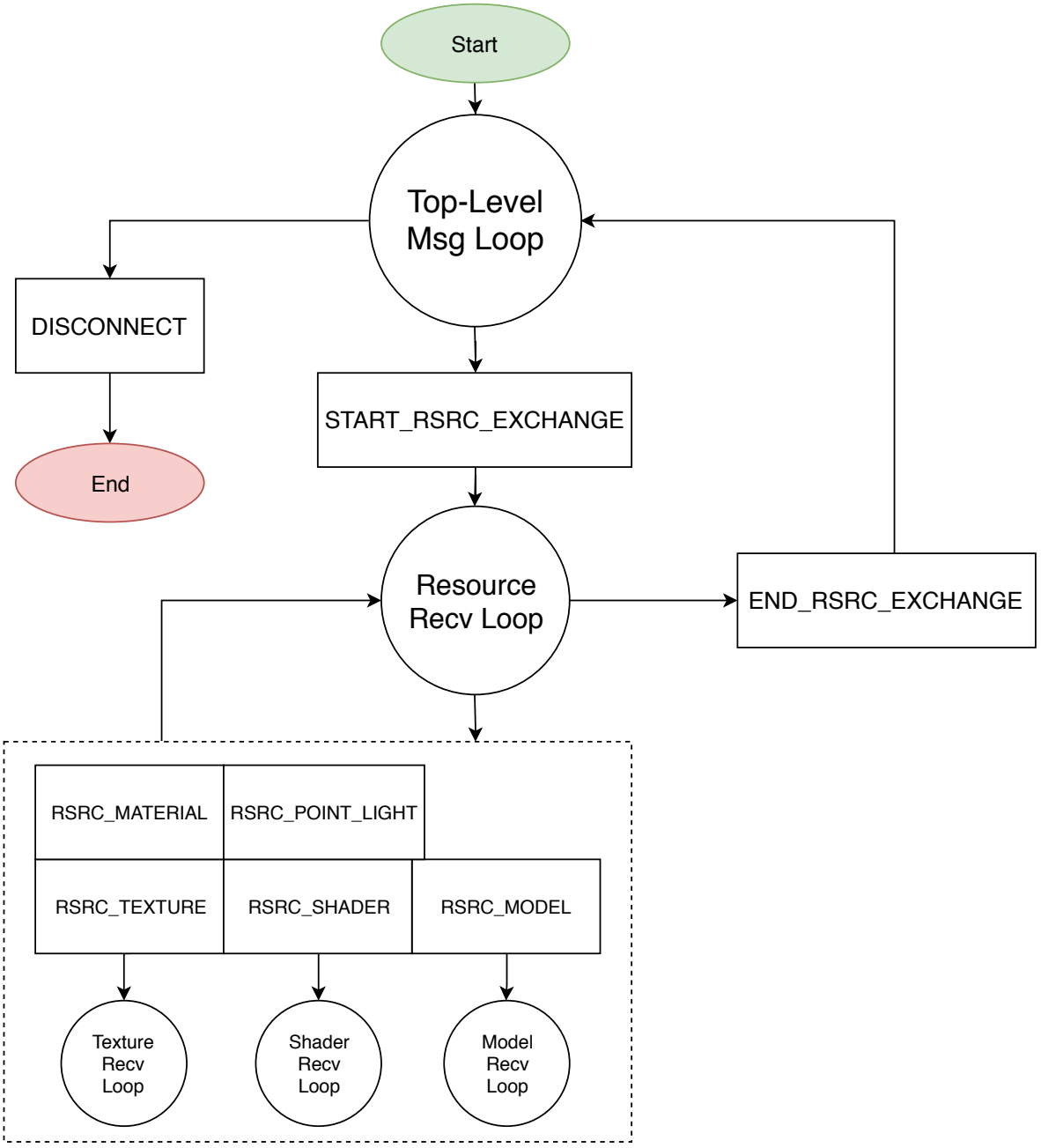


**Figure 20:** Structure and hierarchy of TCP message receiving loops in the client

The top-level message loop accepts two message types: `DISCONNECT`, which makes the client drop the connection and quit, and `START_RSRC_EXCHANGE`, that starts a resource exchange session. This latter message marks the passage to the inner message loop that receives resources.

The resource receiving loop accepts either a `RSRC_*` message, followed by a payload containing the resource data, or an `END_RSRC_EXCHANGE` message, that signals the end of the exchange session and brings the state back to the top-level loop.

There are currently five types of resource messages: models, materials, textures, point lights and shaders. Materials and point lights have their payload "inline", meaning their payload is al-

ways fully contained inside the first packet. Models, textures and shaders, being of variable size, may be spread across multiple packets, so each subroutine receiving these types of resources have themselves an inner message receiving loop that reads packets until the resource's payload is exhausted. Pseudocode in Listing 4.16 illustrates this process.

The `TcpMsgThread` class, which is the one running the TCP Passive loop, stores all received resources into a class field, which the main thread polls during each iteration. If the resources are marked as "available", i.e. they are not empty and an exchange session is not currently active, the main thread retrieves them for processing and the TCP thread's internal field is emptied.

### 4.4.5 Main Thread

The client's main thread performs the following tasks:

1. it initializes the client resources and cleans them up on termination;
2. it connects to the server and performs the initial handshake;
3. it runs the event and rendering loop;
4. it updates the graphic resources when new assets are received from the server;
5. it sends TCP messages to the server when needed.

The initialization part was already covered in § 4.2 and the connection to the server was covered in § 2.5, so we are going to skip over those points.

The main loop is shown in Listing 4.17.

The `LimitFrameTime` object is just a convenient class implementing frame limiting through RAII; frame limiling can be toggled on or off with a boolean flag.

Before running the loop iteration, we check for disconnection of the TCP endpoint and break from the loop if the connection was dropped. Then the events are polled via GLFW, the frame is run and some per-frame time statistics are calculated.

The big part of the work happens inside the `runFrame` method. The steps the client runs during a frame are the following:

1. it checks if network TCP resources are available and processes them if needed;
2. it checks for UDP update packets and applies them if needed;
3. it enqueues ACK messages for received UDP packets if needed;
4. it updates the uniform buffers;
5. it updates the local camera by reading user input;
6. it draws the frame.

In our demo we run no game logic, audio, animation or any other "real-world" engine subsystems: we simply draw what the server tells us to draw.

**Listing 4.16:** Deserializing TCP resources (pseudocode)

```
// (Actual routines are in file tcp_deserialize.cpp)
procedure receiveResource(packet) {
  switch (packet.header.type) {
  case POINT_LIGHT:
    receivePointLight(packet);
    return;
  case TEXTURE:
    receiveTexture(packet);
    return;
  // ...
  }
}

procedure receivePointLight(packet) {
  // Deserializing an inline resource requires little more
  // than a cast:
  pointLight := cast_to<PointLight>(packet.payload);
  validate(pointLight);
}

procedure receiveTexture(packet) {
  // Deserializing a variable-size resource in general
  // requires reading multiple packets:
  textureInfo := cast_to<TextureInfo>(packet.header);
  totSize := textureInfo.size;
  buffer := new byte_array[size];
  copy packet.payload into buffer;
  bytesRead := size_of(packet.payload);
  while (bytesRead < totSize) {
    packet := receivePacket();
    copy packet.payload into buffer;
    bytesRead += size_of(packet.payload);
  }
}
```

**Listing 4.17:** Client main loop

```
// File: client.cpp

while (!glfwWindowShouldClose(app.window)) {
  LimitFrameTime lft{ RENDER_FRAME_TIME };
  lft.enabled = gLimitFrameTime;

  // Check for disconnection
  if (!endpoints.reliable.connected) {
    warn("RelEP disconnected");
    break;
  }

  glfwPollEvents();
  runFrame();
  calcTimeStats(fps, beginTime);
}
```

### 4.4.6 TCP Resource Updating

When the main thread finds that resources are available from the TCP Passive thread, it retrieves them by locking a mutex and grabbing the pointer to the memory area where those resources were stored.

First, the client iterates on all the received materials and collects information about all their referenced textures. Since textures are sent separately from the material referencing them, the client needs to keep track of which textures are required by which material in order to properly update those materials later. This information is stored in a dictionary whose keys are the textures' names and values are the lists of materials that reference them[6].

Once texture information are gathered, the client proceeds to process the other resources. Model information are used to insert new entries in the *objTransforms* dictionary and reserve space in the `Geometry` struct described in § 4.3. Point lights are also inserted into *objTransforms* with a default transform (like for models, their actual transforms will be received via UDP update packets). Both are also saved into the `NetworkResources` struct without further processing.

Next, shaders are copied into NetworkResources as well; textures are decoded asynchronously and loaded into device memory via the `TextureLoader` struct. Finally, materials are converted from their network format to the client internal format and saved into NetworkResources.

At this point, the main thread releases the lock so the TCP Passive thread can proceed receiving resources if needed. Once all the new resources have been gathered, the main thread performs the following steps:

[6] Since in general multiple materials can reference the same textures, as shown in Figure 9, the same key may be associated with multiple values

1. if new models were received, update the Geometry struct to make space for new vertices and indices; then, create a new uniform buffer containing some object information such as its transform. This buffer is created from a BufferArray containing all uniform buffers;
2. if either new models or new materials were received, create new Vulkan descriptor sets: one per material and one per model;
3. if new textures were received, regenerate all materials which reference those textures, along with their descriptor sets;
4. if new shaders were received, regenerate the graphics pipelines;
5. reset the Vulkan command pool and re-record the command buffers to include the new resources.

### 4.4.7 UDP Resource Updating

The updating of resources via UDP works a bit differently. As of the current work, there are three types of UDP updates: transform updates, point light updates and geometry updates. All these updates involve writing a new value into some data structure: transform updates will write into the *objTransforms* dictionary, point light updates into the NetworkResources structure and geometry updates into the Geometry structure. Transforms and point light parameters are copied from their respective storage structures to some uniform buffers during each frame using device-mapped host-visible pointers. An example of this is reported in Listing 4.18.

The main thread checks for UDP updates during each frame. If chunks are available from the UDP Passive thread, they are copied into a temporary buffer and decoded one by one. According to the chunk type reported in the header of each chunk, the proper amount of bytes is extracted from the buffer and handed to a specific function that decodes it and outputs a data structure of type `UpdateReq`.

The UpdateReq structure is the client counterpart of `QueuedUpdate` described in Listing 3.15: it contains a generic update request in the shape of an union type of all possible update types along with a *type* field containing the actual type.

Once all the chunks have been deserialized, an array of UpdateReq structs is returned to the client which passes it to a processing function that applies all the updates. During this process, ACKs for the geometry updates are generated and added to the UDP Active thread's "to-send" list.

**Listing 4.18:** Per-frame updating of the lights' uniform buffer

```
// File: client.cpp

void VulkanClient::updateLightsUniformBuffer()
{
  auto lightBuf = uniformBuffers.getBuffer(sid("lights"));

  // This pointer is mapped to device memory
  auto ubo = reinterpret_cast<LightsUBO*>(lightBuf->ptr);

  ubo->nPointLights = netRsrc.pointLights.size();
  for (unsigned i = 0; i < netRsrc.pointLights.size(); ++i) {
    const auto& pl = netRsrc.pointLights[i];
    ubo->pointLights[i] = UboPointLight{
      // Light position
      glm::vec3{ objTransforms[pl.name][3] },
      // Light attenuation
      pl.attenuation,
      // Light color
      { pl.color.r, pl.color.g, pl.color.b },
      // Padding
      0
    };
  }
}
```

# Chapter 5

# Experiments

In this section we analyse the performance of our engine. We set up a demo scene and measure the following parameters:

1. the amount of traffic between the client and the server;
2. the response delay under different network conditions;
3. the client frame rate under various workloads.

The experiments setup is as follows:

- we build both the client and the server in "release" mode;
- we employ two machines, **D** and **L**, whose specifications are reported in Table 5.1;
- **D** and **L** are connected to the same LAN;
- on the client machine, we press the keyboard digits to send a model request to the server;
- we measure the frame rate on stable regime, i.e. while no heavyweight TCP data is being transferred. That is because, during TCP resource updating, both the client and the server's frame rates jitter due to spikes in either I/O or computation and this effect pollutes the measurements;
- we measure the response delay in two different steps: the first step, called "geometry", is the interval passing from the moment we press a digit button to the moment we see the full mesh on the client; the second step, called "full", starts at the same time as geometry but only ends when we see the fully textured mesh.

**Table 5.1:** Specs of the machines used for experiments

| *Name* | *Type* | *Operating System* | *GPU* |
|---|---|---|---|
| **D** | Desktop | Debian Buster/Sid | NVIDIA GeForce 1060 |
| **L** | Laptop | Windows 10 | NVIDIA GeForce 960M |

**Table 5.2:** 3D models used during the demo. In the *Texture types* column, D, S and N mean respectively "diffuse", "specular" and "normal"; – means "untextured". *Size* refers to the uncompressed size in memory.

| *Name* | *Abbreviated* | *Vertices* | *Indices* | *Texture types* | *Size (KiB)* |
|---|---|---|---|---|---|
| Wall | **w** | 24 | 36 | D, N | 1 |
| Cat | **c** | 4135 | 11862 | D | 272 |
| Nanosuit | **nano** or **n** | 14262 | 57174 | D, S, N | 1003 |
| Table | **t** | 197355 | 766628 | – | 13787 |
| Robot | **r** | 208295 | 844056 | – | 14688 |
| Sponza Palace | **sponza** or **s** | 239670 | 786759 | D, S, N | 16180 |
| Mecha | **m** | 267899 | 559764 | – | 16837 |

## 5.1 Client FPS

For benchmarking the client framerate we downloaded a set of free 3D models and put them together in a scene along with a variable number of dynamic point lights. The models used are reported in Table 5.2.

Many models we use purposefully have a very high polygon count (much higher than the typical models used in realtime rendering). This way we can achieve a high total amount of geometry without having to build a complex scene with hundreds of models. All the following FPS measurements refer to a fullscreen scene rendered at 1920x1080 (FullHD) resolution, with framerate locking disabled, using **D** as the client and **L** as the server.

Figure 21 shows the average frame rate for various mix-ups of models and point lights. The only combination that achieved less than 60 frames per second was all the models combined (for a total of $3{,}026{,}279$ indices) with 200 dynamic lights. As of this work, our client does not perform any light or model culling, so all these results may be improved significantly by future work.

Figure 22 shows how framerate varies in fixed scenes with an increasing number of lights. We used both the "Nanosuit" model, which has an average number of polygons, and the "Sponza Palace" model, which has a high polygon count.

## 5.2 Response Delay

For the response delay measurements we started the server on **L** and the client on **D**. We measured the network bandwidth between the two and found it to be about $68\ \mathrm{Mbps}$: we took this as our "baseline" bandwidth and measured the response delay under these conditions. Next, we artificially throttled the bandwidth using the engine's BandwidthLimiter described in § 2.4.5. The results of the measures are reported in Table 5.3 and Figure 23.

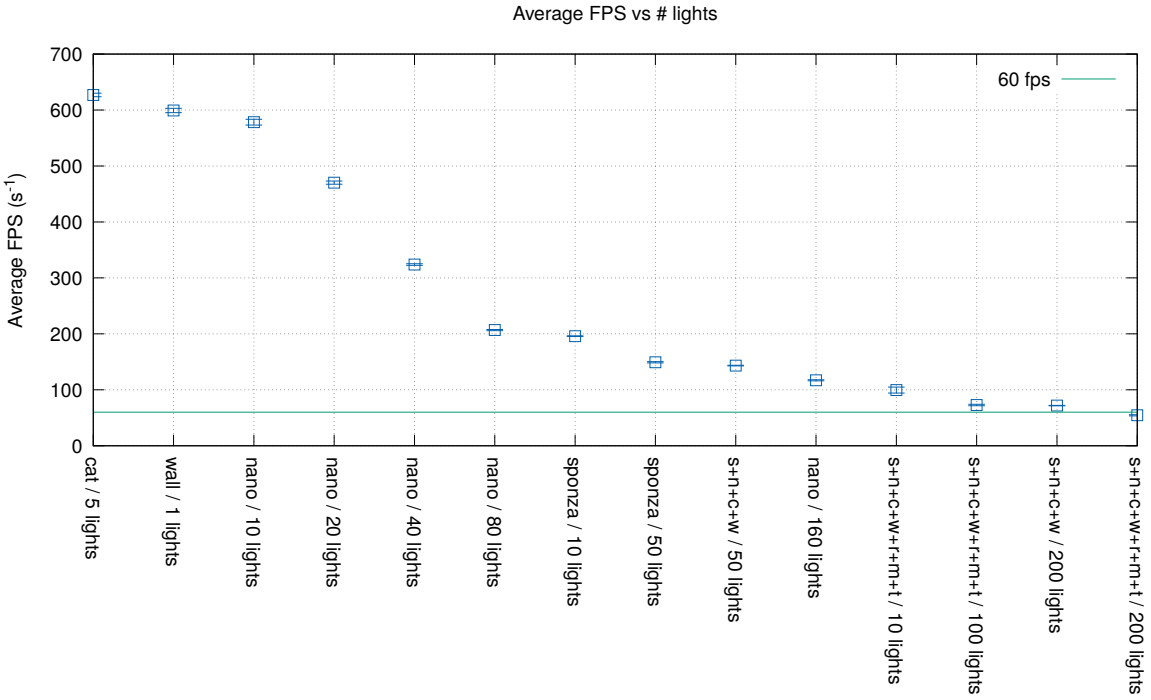


**Figure 21:** Framerate measurements for various scene combinations

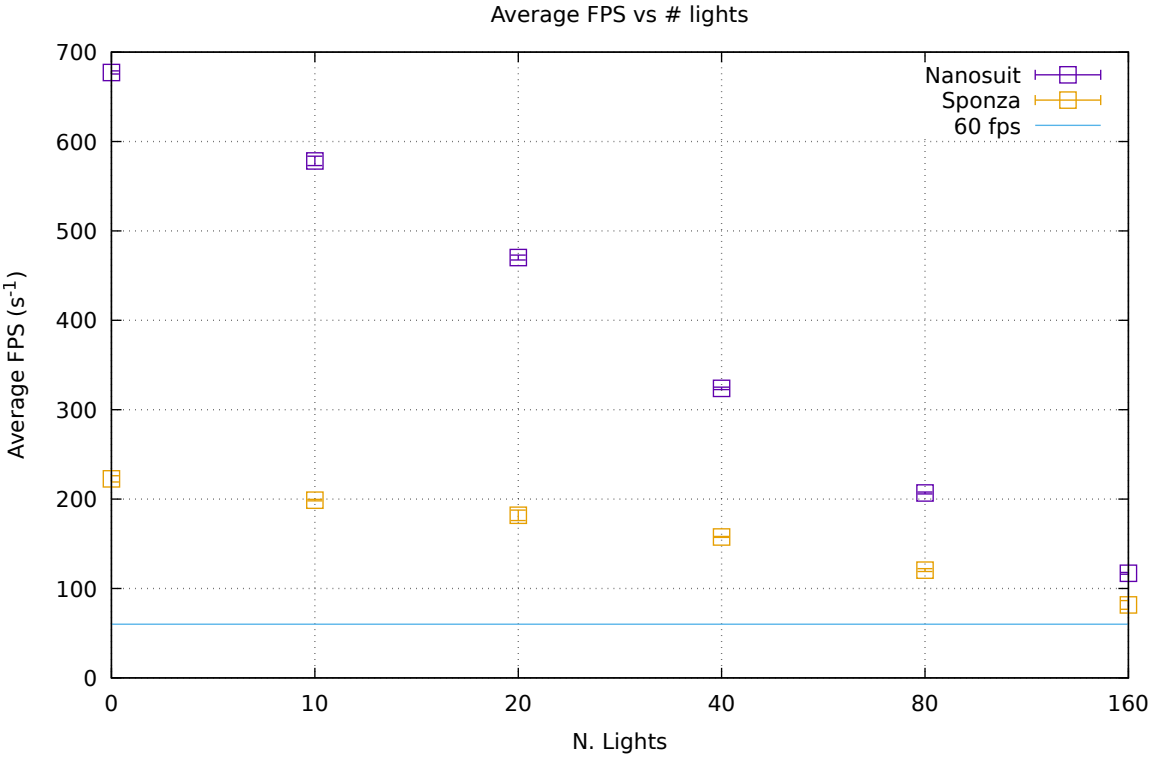


**Figure 22:** Average FPS with varying number of lights

**Table 5.3:** Response delay with varying bandwidths

| *Bandwidth* | *nano/geo* | *nano/full* | *sponza/geo* | *sponza/full* |
|---|---|---|---|---|
| 68 Mbps | $< 0.5$ s | $(1.8 \pm 0.2)$ s | $(1.3 \pm 0.2)$ s | $(5.0 \pm 0.2)$ s |
| 40 Mbps | $< 0.5$ s | $(4.2 \pm 0.3)$ s | $(5.4 \pm 0.2)$ s | $(32.0 \pm 0.3)$ s |
| 24 Mbps | $< 0.5$ s | $(7.9 \pm 0.3)$ s | $(8.4 \pm 0.2)$ s | $(52.0 \pm 0.3)$ s |
| 16 Mbps | $< 0.5$ s | $(11.5 \pm 0.2)$ s | $(8.7 \pm 0.2)$ s | $(52.0 \pm 0.3)$ s |

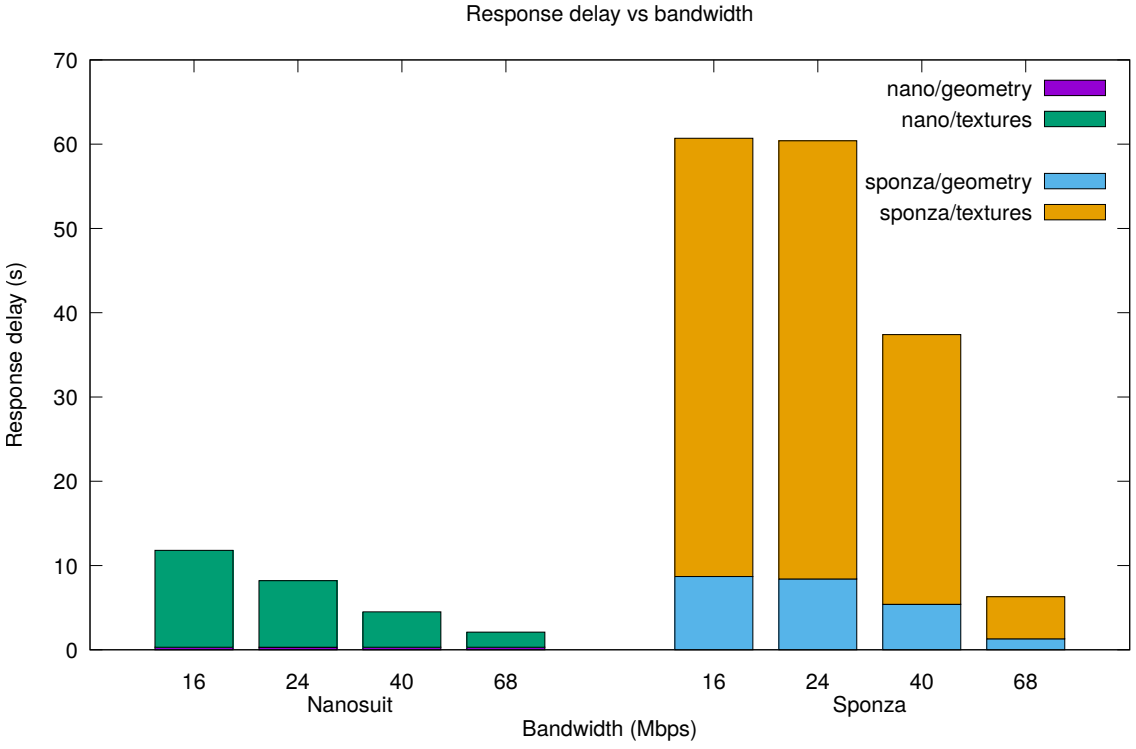


**Figure 23:** Response delay with varying bandwidth

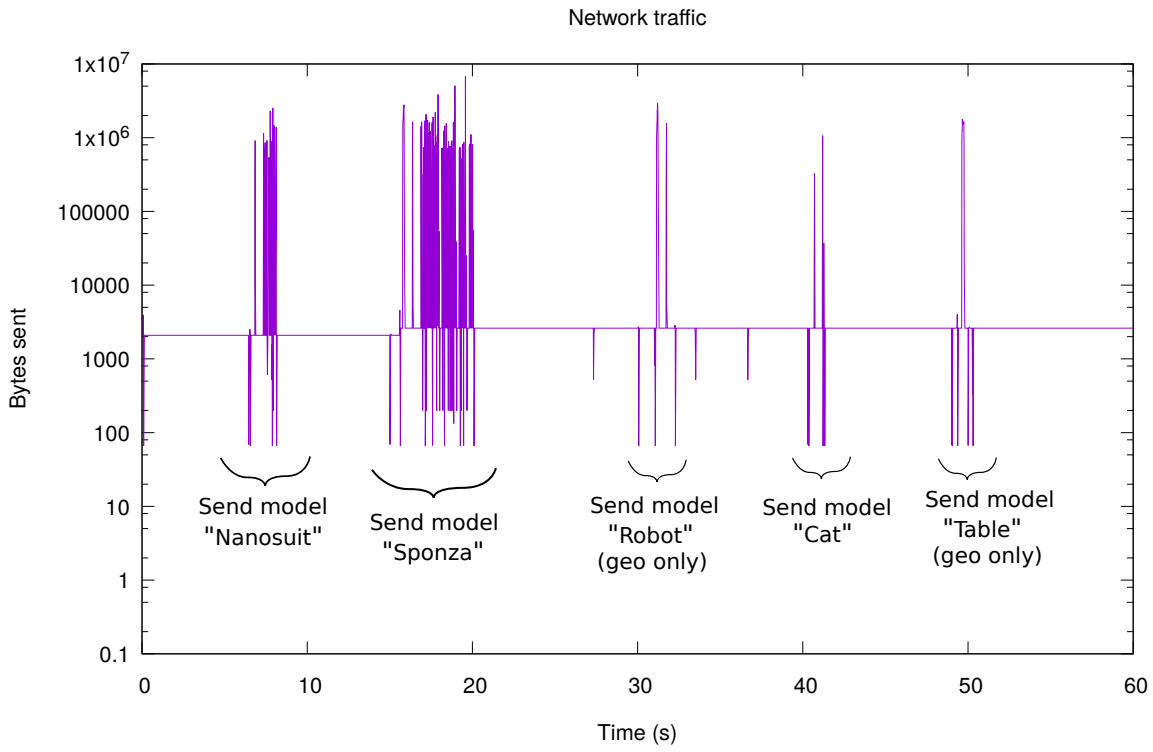


**Figure 24:** Network traffic over time (client downlink)

Most of the time making up the response delay consists in textures exchange. This is expected, as on average textures have a much bigger size than the models' geometry.

## 5.3 Network Traffic

For measuring the network traffic we used *tcpdump* to capture all packets sent by the server to the client, using a time resolution of $0.1\,\mathrm{s}$. For consistency with the previous measurements, we kept the server on **L** and the client on **D**. Results are shown in Figures 24 and 25. Unsurprisingly, the downlink traffic is much higher than the uplink one, as it includes full model geometry and textures, while the uplink traffic pretty much only includes UDP ACKs (client outbound TCP messages are negligible in size).

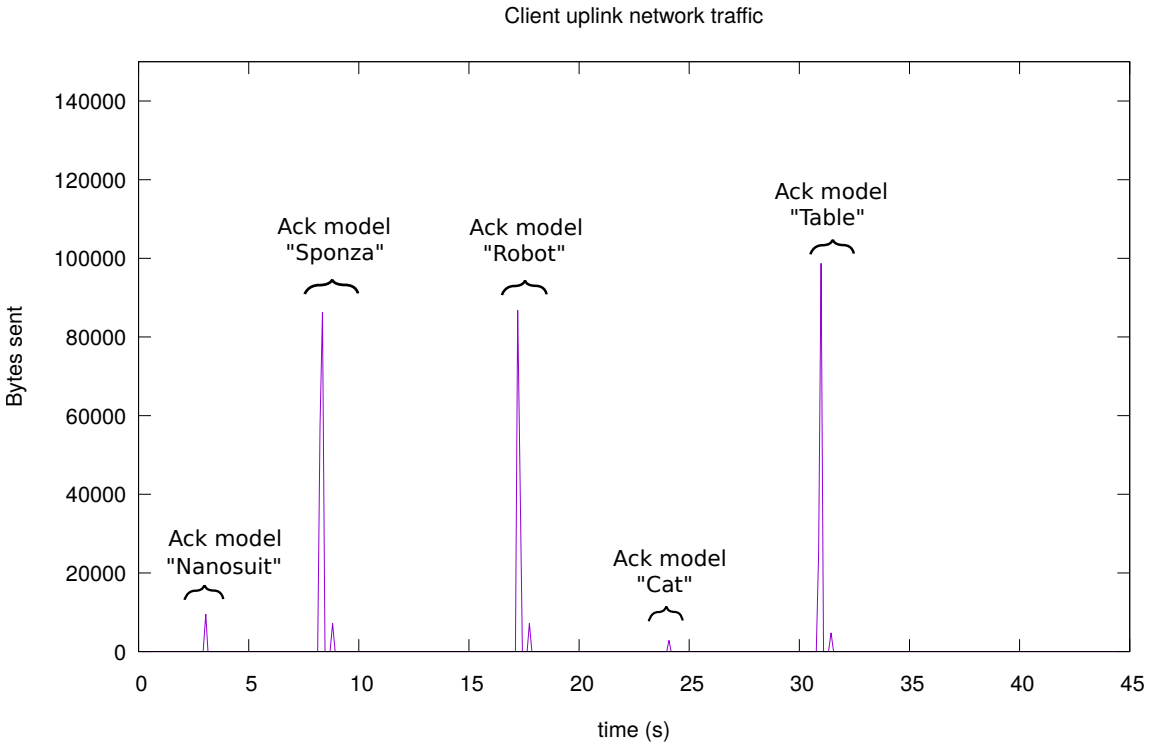


**Figure 25:** Network traffic over time (client uplink)

# Chapter 6

# Conclusions and Future Work

## 6.1 Vulkan-Based Distributed Rendering Engine

In this work, we successfully designed and implemented a realtime rendering system where virtually all assets are retrieved from a remote server on demand. The client is able to render a complex scene with a high number of polygons and dynamic lights at a satisfying frame rate.

The initial retrieval of assets is the most time-consuming portion of our connection and rendering pipeline: at the current state of the engine, a full high-complexity scene would take some minutes to complete with a $100\,\mathrm{Mbps}$ connection. However it is apparent that this number would be shrunk down significantly by employing different levels of texture quality and models LODding. Sending millions vertices and indices only takes a few seconds over a broadband connection, which means the client can start rendering a low-fidelity version of the scene after a very brief wait and improve it step by step in the following seconds or minutes.

Our engine was built with modularity and expandability in mind, so it can effectively work as a basis to expand upon or a module to integrate into a bigger project. In particular, our demo "game loop" on the server may be replaced by a more sophisticate loop updating not only the objects' transforms and the dynamic lights' parameters but also several proper subsystems such as an animation system, an audio system, a physics system and so on. Our game loop mock is not tightly bound to the rest of the rendering engine, so such a replacement is certainly feasible.

We successfully employed Vulkan to implement a rendering pipeline using the deferred shading technique via a single render pass, applied many application-level optimizations and crafted a system to effectively forward network-received data into the rendering process in real time. We also implemented several debugging utilities for both the client and the server, mainly in the form of the Validation class, the MemoryMonitor, the BandwidthLimiter and various helper macros.

## 6.2 Future Improvements

During this thesis, we definitely reached our initial goals to provide a concrete proof of concept to back our idea for a new paradigm in remote gaming. However, we are aware of many aspects of our work that could be improved, refined or extended in one way or another, that we could

**Figure 26:** Some samples of the models rendered by our engine (nanosuit and Sponza belong to Crytek.)

**Table 6.1:** Compressing UDP server data (offline)

| *Command Used* | *Original Size* | *Compressed Size* | *Compression Ratio* | *Time Taken* |
|---|---|---|---|---|
| `gzip udp.bin` | 18 MB | 11 MB | 1.63 | 1.5 s |
| `bzip2 udp.bin` | 18 MB | 11 MB | 1.63 | 1.71 s |
| `xz -0 udp.bin` | 18 MB | 8.1 MB | 2.22 | 1.63 s |
| `xz udp.bin` | 18 MB | 6.6 MB | 2.72 | 6.4 s |

not afford to undertake due to time constraints. The following sections will present a brief review of some of those aspects that we believe to be of major interest.

### 6.2.1 Data Compression

In our work we did not implement any compression on exchanged data. However, geometry and transform data are likely to greatly benefit even from a data-agnostic compression algorithm such as DEFLATE or LZMA. To test this, we tried dumping the UDP data sent by the server for some seconds to a file and running *gzip*, *xz* and *bzip2* on the file. The results are reported in Table 6.1. We observe a very interesting compression ratio for all algorithms, meaning that implementing online compression at the engine level would reduce the initial waiting time significantly. This would make a noticeable difference especially for narrower bandwidth connections.

TCP data would not benefit from data compression, as the largest portion of such data consists in texture raw data which is already compressed.

### 6.2.2 Application-Stage LODding

Currently, the server always sends the entire geometry for every requested model. An optimization strategy may consist in only sending a rough approximation of the model and prioritize upgrading the level of detail in areas which the player can currently see rather than uniformly. Even without gaze-oriented prioritization, using different LODs for models would probably yield an additional speed-up to the initial scene loading phase.

However, geometry is not the primary candidate for optimization. Measurements show that texture exhange is by far the most time-consuming part of the initial loading, therefore a much bigger reduction in waiting times would likely be achieved by creating LOD chains for textures rather than models. By sending all low-resolution textures first, an initial scene of much higher fidelity could be presented with just a few seconds wait.

Another possibility would be adding a “vertex color” attribute to vertex data and use that to present a flat-colored version of all models even before receiving any texture data (although this would increase the size of each vertex, with possible negative consequences on UDP traffic size).

### 6.2.3 Asynchronous Client Resources Updates

As described in previous chapters, the client performs TCP resources updating on the main thread. This is done for convenience and simplicity, but has the downsize to briefly freeze the screen while Vulkan resources are recreated[1]. Adding a new client thread that performs all updates in a "resources back buffer" which gets swapped in atomically when updating is complete may reduce the stuttering at the expense of higher memory usage. Alternatively, updates may be saved in a "to-do" list by the client and only applied in group at lower frequency than individual updates: this would reduce the stuttering as well, at the price of not seeing all updates immediately.

[1] The freezing only happens when heavyweight resources are added to the scene, namely models and textures.

# Bibliography


[1] Gregory, J. *Game Engine Architecture, Second Edition*. A.K. Peters, Ltd. Natick, MA, USA 2014

[2] McDonald, E. (2017, Apr 20). *The Global Games Market Will Reach $108.9 Billion in 2017 With Mobile Taking 42%*. Retreived from https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/

[3] Skaugen, K. (2015, Aug 18). *The Game Changer*. Retreived from https://blogs.intel.com/technology/2015/08/the-game-changer/

[4] McDonald, E. (2017, Jun 20). *Newzoo's 2017 Report: Insights into the $108.9 Billion Global Games Market*. Retreived from https://newzoo.com/insights/articles/newzoo-2017-report-insights-into-the-108-9-billion-global-games-market/

[5] Sims, D. (2015, Apr 15). *The problem with growing download sizes*. Retreived from https://www.pcgamer.com/the-problem-with-growing-download-sizes/

[6] Chalk, A. (2015, Mar 11). *Star Citizen client expected to be around 100GB*. Retreived from https://www.pcgamer.com/star-citizen-client-expected-to-be-around-100gb/

[7] G. Fowler, L. C. Noll, K. Vo, D. Eastlake, T. Hansen. *The FNV Non-Cryptographic Hash Algorithm* (2018, Jun 12). Retreived from https://tools.ietf.org/html/draft-eastlake-fnv-15#section-2

[8] R. Shea, J. Liu, E. C.-H. Ngai, Y. Cui. *Cloud Gaming: Architecture and Performance* (IEEE, 2013)

[9] Sakr, S. (2012, Jul 2). *Sony buys Gaikai cloud gaming service for $380 million*. Retreived from https://www.engadget.com/2012/07/02/sony-buys-gaikai/

[10] Lowensohn, J. (2015, Apr 2). *Sony buys streaming games service OnLive only to shut it down*. Retreived from https://www.theverge.com/2015/4/2/8337955/sony-buys-onlive-only-to-shut-it-down

[11] Sony. *PlayStation Now*. Retreived from https://www.playstation.com/en-us/explore/playstation-now/

[12] Fingas, J. (2018, Jul 23). *Microsoft's next Xbox could have a cloud-only counterpart*. Retreived from https://www.engadget.com/2018/07/23/xbox-scarlett-cloud-console/

[13] S. Chen, Y. Chang, P. Tseng, C. Huang, C. Lei. *Cloud Gaming Latency Analysis: OnLive and StreamMyGame Delay Measurement*. Retreived from http://www.iis.sinica.edu.tw/ swc/onlive/onlive.html

[14] K. Chen, Y. Chang, H. Hsu, D. Chen, C. Huang, C. Hsu. *On the Quality of Service of Cloud Gaming Systems* (IEEE, 2014)

[15] D. Wu, Z. Xue, J. He. *iCloudAccess: Cost-Effective Streaming of Video Games from the Cloud With Low Latency* (IEEE, 2014)

[16] X. Liao, L. Lin, G. Tan, H. Jin, X. Yang, W. Zhang, B. Li. *LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming* (IEEE, 2016)

[17] Overvoorde, A. *Vulkan Tutorial*. Retreived from https://vulkan-tutorial.com/Overview

[18] Van Oosten, J. (2015, Sep 4). *Forward vs Deferred vs Forward+ Rendering with DirectX 11*. Retreived from https://www.3dgep.com/forward-plus/

[19] F. Meng, H. Zha. *Streaming Transmission of Point-Sampled Geometry Based on View-Dependent Level-of-Detail* (IEEE, 2003)

[20] S. Bischoff, L. Kobbelt. *Streaming 3D Geometry Data over Lossy Communication Channels* (IEEE, 2002)

[21] Kubisch, C. (2016, Jan 28). *Vulkan Shader Resource Binding*. Retreived from https://developer.nvidia.com/vulkan-shader-resource-binding

[22] Garrard, A. (2016, May). *Vulkan Subpasses or The Frame Buffer is Lava*. Retreived from https://www.khronos.org/assets/uploads/developers/library/2016-vulkan-devday-uk/6-Vulkan-subpasses.pdf

[23] ARM Software (2016). *Deferring shading with Multipass*. Retreived from https://arm-software.github.io/vulkan-sdk/multipass.html

[24] *The Open-Asset-Importer-Lib*. Retreived from http://www.assimp.org/

[25] S. T. Barrett. *STB*. Retreived from https://github.com/nothings/stb

[26] T. Herzog. *cfstructs*. Retreived from https://github.com/karroffel/cfstructs

[27] B. Gregg. *FlameGraph*. Retreived from https://github.com/brendangregg/FlameGraph

[28] *RenderDoc*. Retreived from https://renderdoc.org/

[29] *cppclean*. Retreived from https://github.com/myint/cppclean

[30] The GNU Project. *gdb*. Retreived from https://www.gnu.org/software/gdb/

[31] Tanenbaum, A. S. & Wetherall, D. (2014). *Computer Networks (Fifth edition, Pearson New international edition.)*. New Delhi: Dorling Kindersley (India) Pvt. ltd.

[32] LunarG. *Vulkan SDK*. Retreived from https://www.lunarg.com/vulkan-sdk/