Technical Documentation

Abstract

Lugdunum is an open-source 3D engine using the Vulkan API as a backend. Lugudunum's goal is to provide a free, modern, cross-platform (mobile and desktop) 3D engine for everyone.

The team

 Corentin Chardeau

 Nicolas Comte

 Stuart Sulaski

 Yoann Picquenot

 Guillaume-Heritiana Sabatié

 Quentin Buathier

 Guillaume Labey

 Antoine Bolvy

 Alexandre Quivy

 Yoann Long

Document summary

This document is intended for every potential Lugdunum contributor, or for everyone wanting to know a bit more about the internals of the project.

This document is split in two parts: the first is focused on Lugdunum, the 3D rendering engine, and on the other hand, the second is focused on LugBench, the benchmarking product.

In the first part of the document you will find an overview of the Lugdunum project, and details about how we interfaced with the Vulkan API.

Each section will be detailed with examples so that this document may be as simple and straightforward as possible, for developers of all levels. It is however required that you have some background in 3D rendering, and a working knowledge of your own system (git, CMake, etc.) as we will not cover the basics, that are usually well documented on other documents and do not enter in the scope of this manual. When appropriate, useful links and resources will be provided for your convenience.

The document ends with an information section, meant to answer the questions you could have after reading: for example how to report bugs, how to contact us, and other useful links.

In summary, when you finish reading this first part, you should have a rough idea of how Lugdunum's source code is architectured, and you should be able to read through the files without any problems. If anything bugs you, please file an issue and we will be glad to answer any question you may have.

The second part of the document will present the architecture of the API, front-end and desktop application of LugBench, the benchmarking software.

Document description

Title	:	Lugdunum Technical Documentation
Modification date	:	July 12, 2017
Accountant	:	Yoann Long
E-mail	:	lugdunum_2018@labeip.epitech.eu
Subjet	:	Lugdunum – Technical Documentation
Document version	:	2.0

Revisions table

Date	Authors	Version	Modified Section(s)	Comment(s)
2017-03-07	Antoine Bolvy	1.0	All Sections	Creation of the document
2017-03-10	Quentin Buathier, Guillaume Labey, Antoine Bolvy	1.1	Code Guidlelines and Style	Added the guidelines
2017-03-11	Antoine Bolvy, Yoann Long, Guillaume Sabatié	1.2	Building Lugdunum	Added the instructions to build Lugdunum for different platforms
2017-03-11	Guillaume Labey, Quentin Buathier, Antoine Bolvy	1.3	Architecture of Lugdunum	Created the architecture section
2017-04-03	Antoine Bolvy	1.3.1	Architecture of Lugdunum	Fixed the missing sequence diagrams
2017-05-07	Yoann Long, Antoine Bolvy, Nicolas Comte, Guillaume Labey, Yoann Picquenot, Corentin Chardeau, Quentin Buathier	2.0	All Sections	Second version of the Technical documentation and added LugBench

Contents

Part. 1

Lugdunum

Contents

I. Architecture of Lugdunum

The purpose of this section is to introduce you to the internal operation of our 3D engine. We will first talk about the architecture of the renderer. Then we will describe the sequencing of the engine graphic's loop, how each component of the `Renderer::Target` is interacting with the `Render::Window` composed of different `Renderer::View`. Then, we will discuss the GPU & CPU's side operation. We will explain how each buffer is loaded and used by our engine.

1. Renderer Architecture

We decided to be as API independent as possible, i.e. we do not want to be too much dependent on Vulkan itself. This is why we created abstract classes for each type and their Vulkan-equivalent in a separate, API specific directory. This is especially visible in Figure 1.1. Hypothetically speaking, this allows us to be much less dependent on this technology and maybe one day, to derive the implementation for another low-level API, such as D3D12 for example.

The main object of the renderer is the `Render::Target`. A `Render::Target` is any surface on which we can render, e.g. a window or an offscreen image.

A `Render::Target` can have multiple `Render::View`s, each representing a fraction of the `Render::Target`, defined by a `Render::View::Viewport` and a `Render::Scissor` defined as following:

```
1  class Viewport {
2  public:
3     struct {
4        float x;
5        float y;
6     } offset;
7
8     struct {
9        float width;
10       float height;
11    } extent;
12
13    float minDepth;
14    float maxDepth;
15
16    inline float getRatio() const;
17 };
18
19 struct Scissor {
20    struct {
21       float x;
22       float y;
```

```
23    } offset;
24
25    struct {
26       float width;
27       float height;
28    } extent;
29 };
```

Each of the components of `Render::View::Viewport` and `Render::View::Scissor` are defined as percentage values (i.e. a float between 0.0 an 1.0), so it has the same appearance on every size of the `Render::Target`.

A unique `Render::Camera` can be attached to a single `Render::View`, i.e. we cannot have a `Render::Camera` attached to two different `Render::View`s.

`Render::Camera`s contain a `Render::Queue` and have pointer to a `Scene::Scene`, which is created by the user, and can be attached to multiple cameras.

Every frame, the `Render::Queue` is cleared, then filled by the `Scene::Scene` with the objects visible by the `Render::Camera`'s frustrum.

The `Render::Queue` is finally sent to `Vulkan::Render::Technique::Technique::render()`.

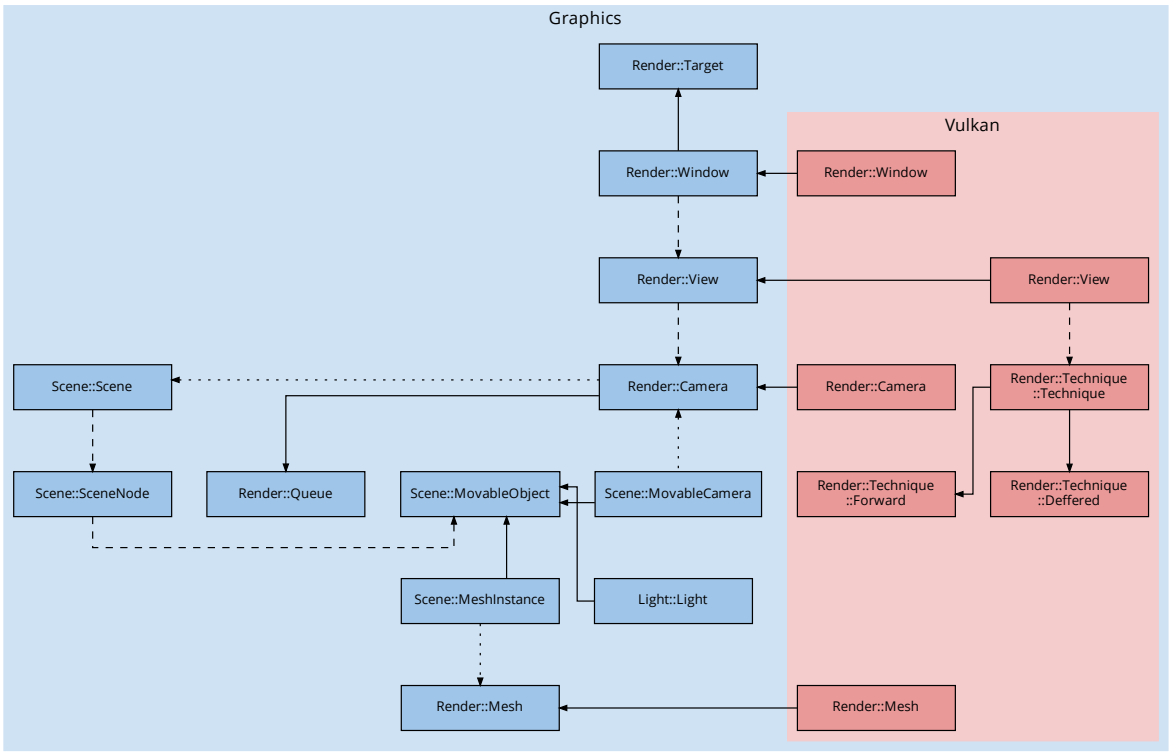

Figure 1.1: Main classes of the renderer

In the diagram Figure 1.1, we are representing the main classes of the renderer and their dependencies.

- Plain line (—>): Inheritance
- Dashed line (- - - >): Contains an instance of the class with ownership
- Dotted line (· · · >): Contains an instance of the class without ownership

The diagram Figure 1.2 shows an example of how classes interact with each other:

- Here we have one `Render::Target`, which contains three `Render::View`s:
  - The render view A
  - The render view B
  - The render view C, which is *disabled*, as each one of these can be enabled and disabled as wished.
- Both render views A and B each have a camera, and each camera has its own render queue.
- Cameras are also linked to a scene, and scenes are linked to each camera's render queues.
- In this particular case, it appears that we have only one scene, so each camera points to the same scene, and the scene points to two render queues.

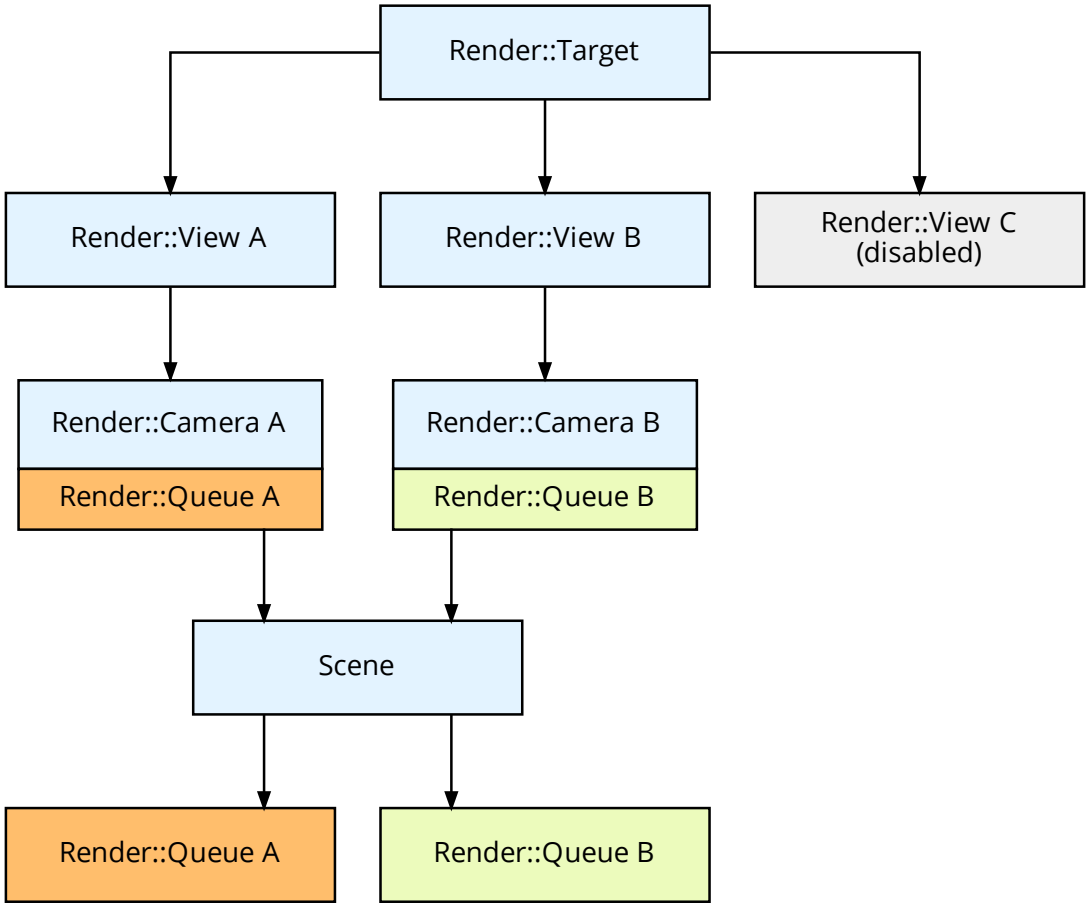

Figure 1.2: Example of a possible usage of the render views

2. Sequence diagrams

In this section will be presented the rendering of a single frame with the help of two sequence diagrams, Figure 1.3 and Figure 1.4. The second is a subset of the first, as they have been separated to ease readability.

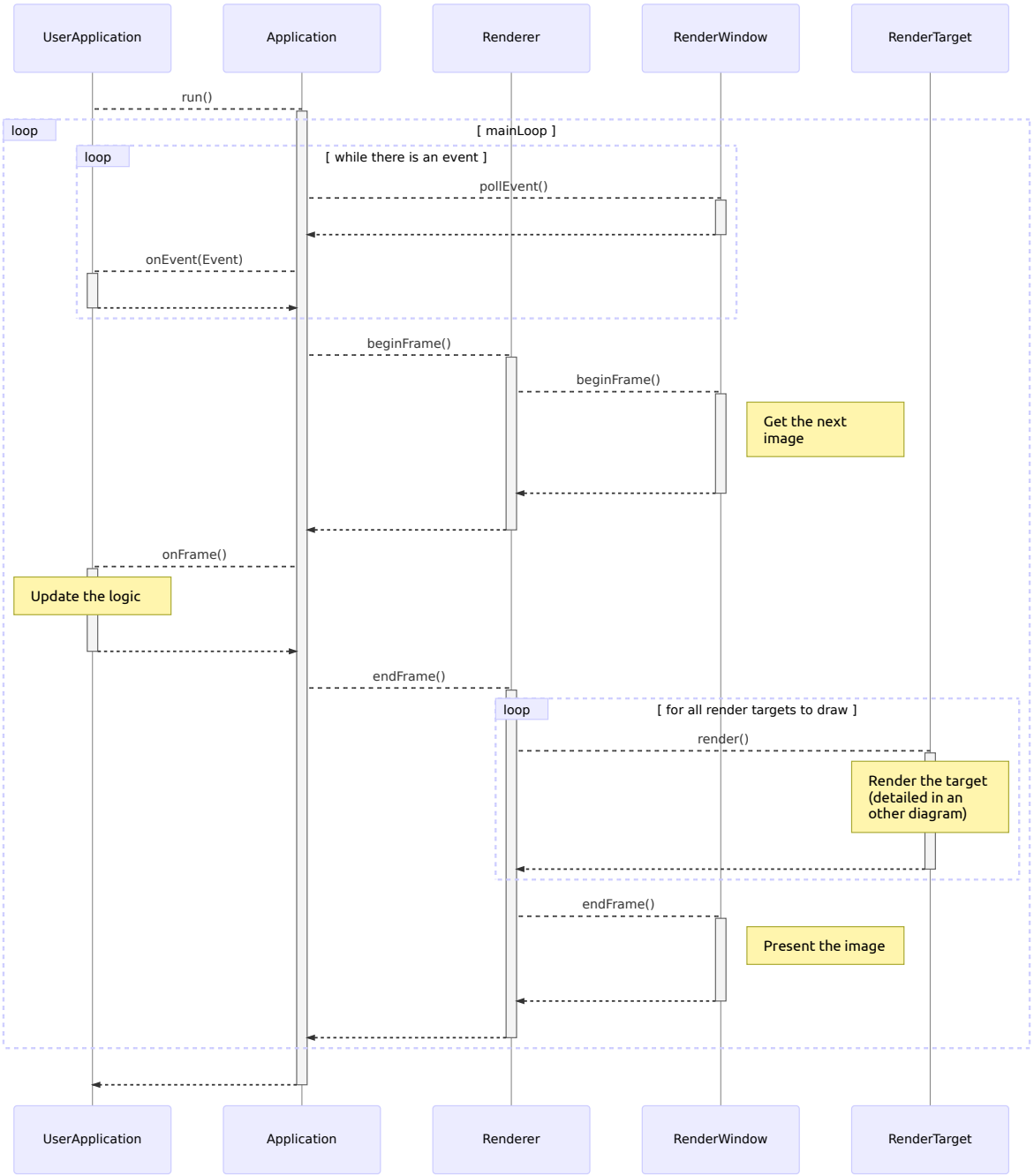

Figure 1.3: Rendering of a frame (part. 1)

Let us describe this sequence diagram, step by step:

First, `UserApplication` is the user-defined class that inherits from `lug::Core::Application` and defines the methods `onEvent` and `onFrame`. `Application::run()` is called (and must be) by the user like in this example:

```
int main(int argc, char* argv[]) {
   UserApplication app;

   if (!app.init(argc, argv)) {
      return EXIT_FAILURE;
   }

   if (!app.run()) {
      return EXIT_FAILURE;
   }

   return EXIT_SUCCESS;
}
```

The method `Core::Application::run()` is the main loop of the engine which polls the events from the window and renders everything correctly. As expected, we can see that the `Core::Application` is polling all the events from the `Render::Window` and sending them to the `UserApplication` through the method `UserApplication::onEvent(const lug::Window::Event& event)`.

Then, `Core::Application` is calling the method `Renderer::beginFrame()` which call itself the method `Render::Window::beginFrame()` to notify the `Render::Window` that we are starting a new frame.

Finally, the user can update the logic of their application in the method `UserApplication::onFrame(const lug::System::Time& elapsedTime)`.

At the end of the frame, the method `Renderer::endFrame()` is called and will call the method `Render::Target::render()` for all `Render::Target` to draw and will finish the frame by calling the method `Render::Window::endFrame()` to notify the `Render::Window` that we are ending this frame.

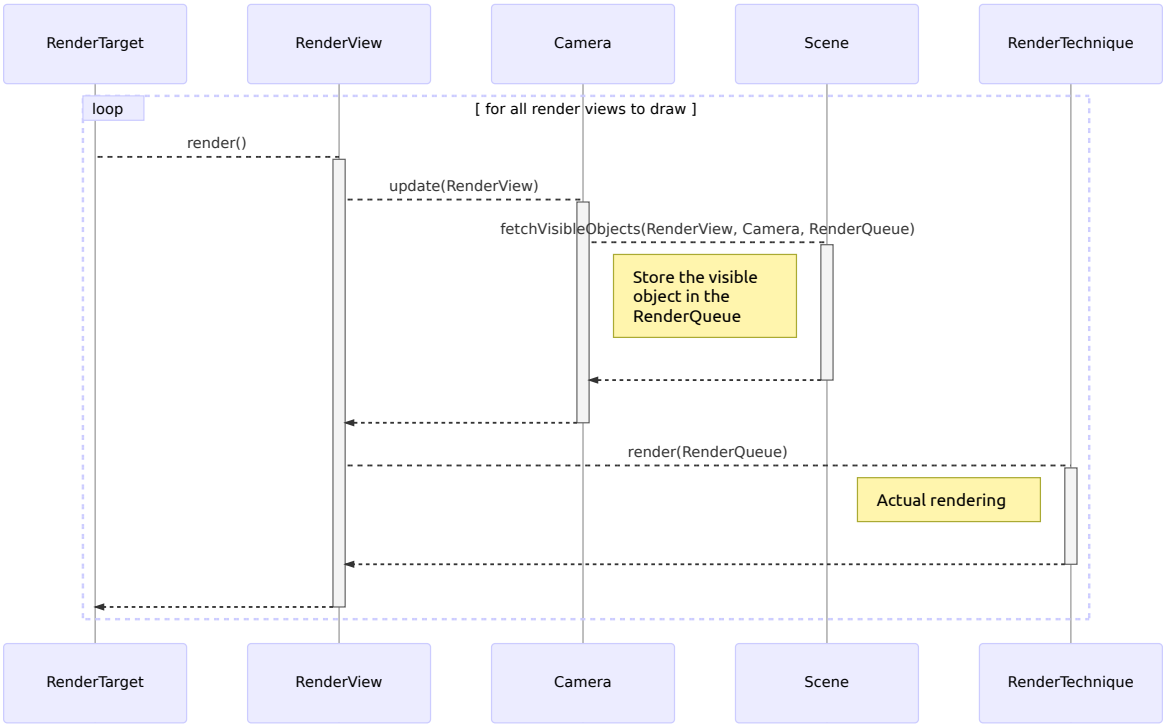

Figure 1.4: Rendering of a frame (part. 2)

In the method `Render::Target::render()`, the `Render::Target` is calling the method `Render::View::render()` for each enabled `Render::View`.

To be rendered, `Render::View` needs to update its `Render::Camera` which will fetch all the elements in its `Render::Queue` from the scene with `Scene::fetchVisibleObjects()`.

So the `Render::Queue` will contain every elements needed to render the `Scene::Scene`, meshes, models, lights, etc.

Then the `Render::View` can call the render technique to draw the the elements in the `Render::Queue` (e.g. for Vulkan a class inheriting from `Vulkan::Render::Technique::Technique`).

3. Vulkan Rendering

3.a. Global

GPU Side

The `Vulkan::Render::Window` and the `Vulkan::Render::Views` of Lugdunum are pretty straightforward. For simplicity's sake we have split this process into five steps:

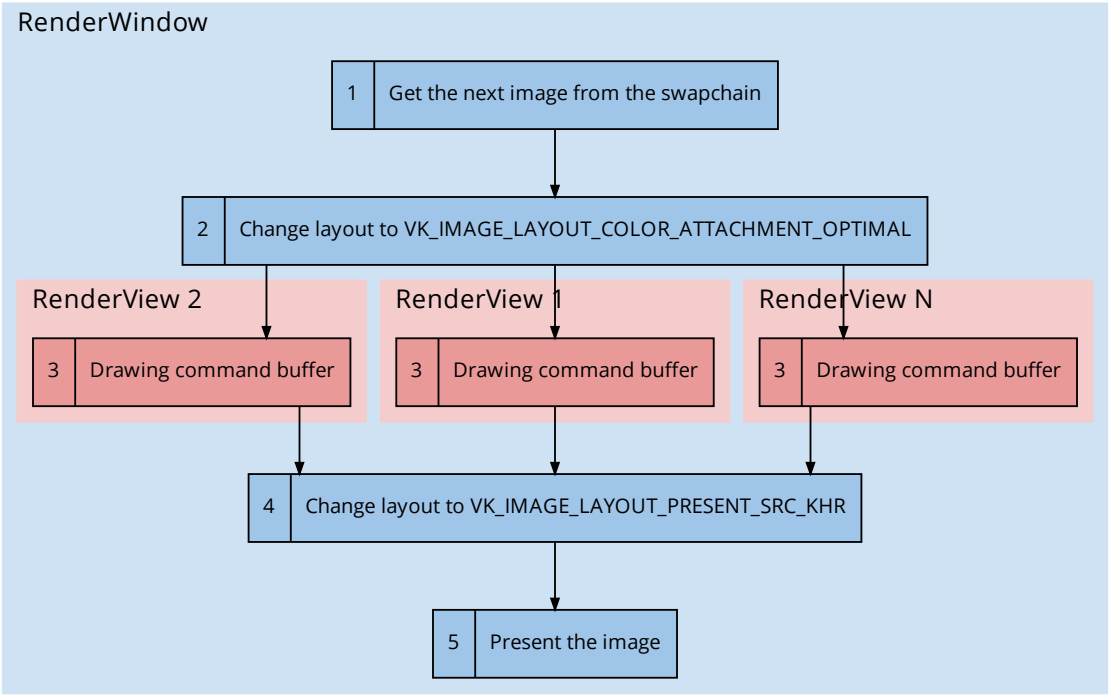

Figure 1.5: Swapchain image acquisition and synchronization

Each arrow represents a Vulkan semaphore for synchronization purpose.

1. We get an available image from the swapchain
2. We change the layout of this image to `VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL`
3. We render each `Vulkan::Render::View` in parallel
4. We change the layout of this image to `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`
5. We add the image to the presentation queue of the swapchain.

For steps 2 and 4 we are using one Vulkan command buffer per image in the swapchain. Each of the command buffers are built beforehand, therefore we don't need to rebuild them each frame.
Step 3 is dependent on the render technique used.

CPU Side

Since our semaphores are stored in a pool, we let each method (`beginFrame()`, `endFrame()`, ...) select their own semaphore(s) to use.

Steps 1 & 2

The method `Vulkan::Render::Window::beginFrame()` is used to accomplish steps 1 and 2. This method chooses one semaphore to be notified when the next image is available and chooses N semaphores to notify each `Vulkan::Render::View` when the image has changed layout. (N being the number of `Vulkan::Render::View` in the `Vulkan::Render::Window`)

Step 3

The method `Vulkan::Render::Window::render()` is used to accomplish step 3. This method uses the N previous semaphores, one for each call to `Vulkan::Render::View::render()`. Each `Vulkan::Render::View` has a semaphore which is signaled when the view has finished rendering. We will explain how the render technique works in the next part.

Steps 4 & 5

The method `Vulkan::Render::Window::endFrame()` is used to accomplish steps 4 and 5. This method retrieves all the semaphores from the `Vulkan::Render::View` and chooses one semaphore to be notified when the image has changed layout.

3.b. Forward render technique

GPU Side

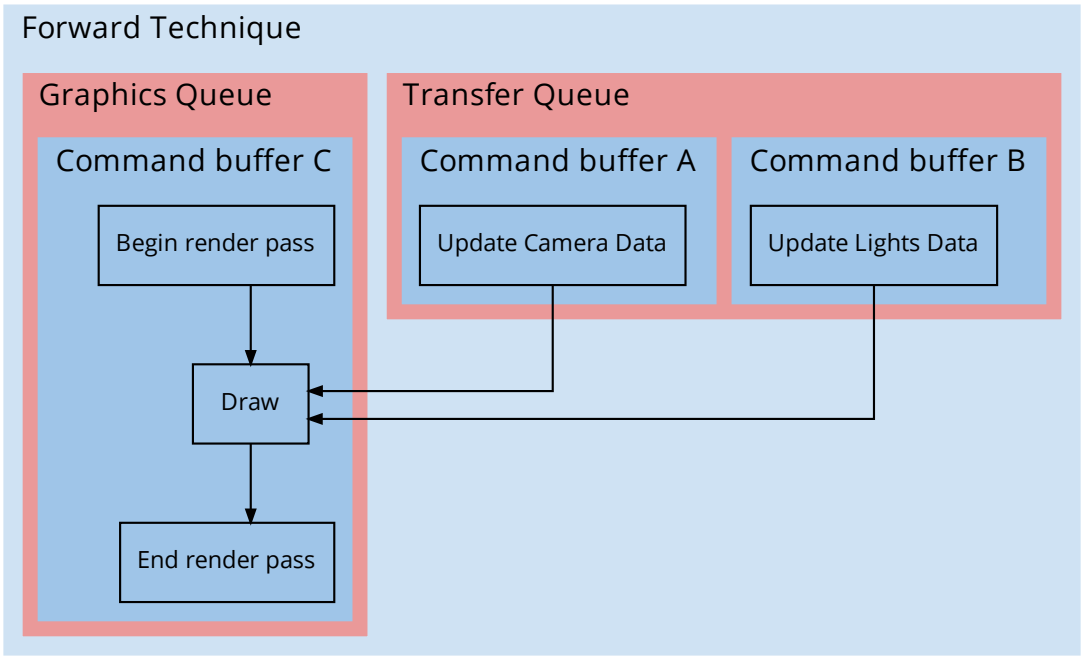

Figure 1.6: Forward technique

The `Vulkan::Render::Technique::Forward` has two different `Vulkan::Render::Queue`, i.e. one transfer and one graphics.

The transfer `Render::Queue` is responsible for updating the data of the `Render::Camera` and `Light::Light`s, each of which is contained in a uniform buffer `Vulkan::API::Buffer` which is sent through different `Vulkan::API::CommandBuffer`s (i.e. “Command buffer A” and “Command buffer B” in the above schema). These `Vulkan::API::CommandBuffer`s are then sent to the transfer `Render::Queue`.

Here is the structure of the uniform buffers for the camera and the lights:

```
// Camera
layout(set = 0, binding = 0) uniform cameraUniform {
   mat4 view;
   mat4 proj;
};
```

```

// Directional light
layout(set = 1, binding = 0) uniform lightUniform {
   vec3 lightAmbient;
   vec3 lightDiffuse;
   vec3 lightSpecular;
   vec3 lightDirection;
};

// Point light
layout(set = 1, binding = 0) uniform lightUniform {
   vec3 lightAmbient;
   float lightConstant;
   vec3 lightDiffuse;
   float lightLinear;
   vec3 lightSpecular;
   float lightQuadric;
   vec3 lightPos;
};

// Spot light
layout(set = 1, binding = 0) uniform lightUniform {
   vec3 lightAmbient;
   vec3 lightDiffuse;
   vec3 lightSpecular;
   float lightAngle;
   vec3 lightPosition;
   float lightOuterAngle;
   vec3 lightDirection;
};
```

Each type of light has a different pipeline using different fragment shaders (That's why all the light uniforms are using the same binding point in the above code sample).

To pass the transformation matrix of the objects we are using pushconstant:

```
layout (push_constant) uniform blockPushConstants {
   mat4 modelTransform;
} pushConstants;
```

The graphics `Render::Queue` is responsible for all the rendering.

The "Command buffer C" for the drawing depends on the two command buffers of transfer by means of semaphores at different stages of the pipeline, `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT` for the camera

and VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT for the lights.

CPU Side

Buffer Pool

The allocation of the uniform buffers is managed by a Vulkan::Render::BufferPool, one for the camera and one for the lights.

As we do not want to perform lots of allocations, we mitigate this using the pool which will allocate a relatively large chunk of memory on the GPU, that will itself contain many Vulkan::Render::BufferPool::SubBuffers.

A Vulkan::Render::BufferPool::SubBuffer is a portion of a bigger Vulkan::API::Buffer that can be allocated and freed from the pool and bind with a command buffer without worrying about the rest of the Vulkan::API::Buffer.

Triple buffering

Because we are using triple buffering, we need a way to store data for a specific image. For that we have Vulkan::Render::Technique::Forward::FrameData that contains all we need to render one specific frame (command buffers, depth buffer, etc.). To avoid using a command buffer already in use, we are synchronizing their access with a fence.

To share Vulkan::Render::BufferPool::SubBuffer across frames, e.g. if the camera does not move, we have a way to reuse the same Vulkan::Render::BufferPool::SubBuffer. We associate the Vulkan::Render::BufferPool::SubBuffer with the object (camera or light), and test at the beginning of the frame if we can use a previous one (if the object has not changed from the update of this Vulkan::Render::BufferPool::SubBuffer).

If it is not possible to use a previously allocated buffer we are allocating a new one from the Vulkan::Render::BufferPool.

Drawing Command Buffer

Here is the pseudo code that we are using to build the command buffer of drawing:

```
BeginCommandBuffer

# The viewport and scissor are provided by the render view
SetViewport
SetScissor

```

```
BeginRenderPass

# We can bind the uniform buffer of the camera early
# It is the same everywhere
BindDescriptorSet(Camera)

# All the lights influencing the rendering (visible to the screen)
Foreach Light
   # Each type of Light has a different pipeline
   BindPipeline(Light)

   # We can bind the uniform buffer of the light
   BindDescriptorSet(Light)

   # All the objects influenced by the light
   Foreach Object
      # Push the transformation matrix of the Object
      PushConstant(Object)

      # We use indexed draw, so we need to bind
      # the index and the vertex buffer of the object
      BindVertexBuffer(Object)
      BindIndexBuffer(Object)

      DrawIndexed(Object)
   EndForeach
EndForeach

EndRenderPass

EndCommandBuffer
```

II. Code Guidlelines and Style

1. Header files

Each `.cpp` file should have an associated `.hpp` file.
Place the definitions for templates and inline functions in separated `.inl` files, alongside the corresponding header file in which it is included.

If applicable, include the `.inl` file inside the namespaces of the header file, as to not repeat these namespace in the inline file and include it at the end of the header file, just before the namespaces' closing brackets.

1.a. Self-contained Headers

Header should be "self-contained", i.e. they must include all their dependencies, and the user should not have to worry about them.

1.b. Headers Guards

To protect headers against double inclusion, headers must start with:

```
#pragma once
```

1.c. Forward Declarations

Preferably avoid forward declarations when possible, include the necessary files when possible, but do not feel restrained by this rule.

1.d. Inline Functions

Inline functions should be implemented in `.inl` files.
Simple getters and setters should be inlined, as well as other short functions (usually less than 10 lines).

Typically, do not inline functions with loops, switch statements and others (unless if, in the common case, the loop or switch statement is never executed), as in this case, inlining the function might not be cost-effective.

Example:

lug/System/Logger/Logger.hpp:

```
#pragma once

// ...

namespace lug {
namespace System {
namespace Logger {

class LUG_SYSTEM_API Logger {
    // ...

    template<typename T, typename... Args>
    void debug(const T& fmt, Args&&... args);

    // ...
};

```

```
#include <lug/System/Logger/Logger.inl>

} // Logger
} // System
} // lug
```

Corresponding inline file in lug/System/Logger/Logger.inl:

```
// No namespace opened here

template<typename T, typename... Args>
inline void Logger::debug(const T& fmt, Args&&... args) {
   // ...
}

// No namespace closed here either
```

1.e. Names and Order of Includes

Inclusion should happen in this order, each section separated by a new line and sorted in alphabetic order:

1. Related header (in a `.cpp` file, this is the corresponding `.hpp` header)
2. C library headers
3. C++ library headers
4. Other libraries' headers
5. Project headers

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory shortcuts `.` (the current directory) or `..` (the parent directory).
These headers should be included as "system" headers, with angle brackets instead of double quotes, because it looks better in our opinion. Deal with it ;)

Example:

```
#include <lug/System/Logger/Logger.hpp>

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <memory>
#include <set>
#include <string>

```

```
#include <vulkan.h>

#include <lug/System/Logger/Handler.hpp>
#include <lug/System/Logger/Message.hpp>
```

You should include all the headers that define the symbols you rely upon, except in the unusual case of forward declaration. If you rely on symbols from `bar.hpp`, don't rely on the fact that you included `foo.hpp` which (currently) includes `bar.hpp`: include `bar.hpp` yourself, unless `foo.hpp` explicitly demonstrates its intent to provide you the symbols of `bar.hpp`. However, any includes present in the related header do not need to be included again in the related `.cpp` (i.e., `foo.cpp` can rely on `foo.hpp`'s includes).

2. Scoping

2.a. Namespaces

All namespaces should be terminated by a comment after the end bracket specifying the name of the corresponding namespace. A namespace doesn't imply another level of indentation, see below for an example.

Namespaces should be used also in the `.cpp` files to avoid repetition.

Example:

```
namespace lug {
namespace System {
namespace Logger {

class Logger {
   // ...
};

} // Logger
} // System
} // lug
```

Do not declare anything in the namespace `std`, and do not use inline namespace, except for very, very specific use-cases.

using-directive and *namespace-aliases* are prohibited in header files, only use them in `.cpp` files or in some particular cases in internal-only namespaces.

2.b. Unnamed Namespaces and Static Variables

Use of `static` variables and *unnamed namespaces* is encouraged in `.cpp` files for all code that does not need to be referenced elsewhere. Do not use that in header files.

2.c. Nonmember, Static Member and Global Functions

Do not use global functions, always put them in a namespace. Do not use class as a namespace for some functions, use a namespace for that.

Static methods should generally be closely related to instances of the class or the class's static data.

2.d. Local Variables

Do not separate variable declaration from its initialization.

```
int x = 40; // Good
int y;

y = 2; // Bad (initialization separated from declaration)
```

Declare variables in the lowerest scope and as close as possible of the first use.

2.e. Static and Global Variables

Prefer POD (plain old data) when using static and global variables (except some very particular cases, one example would be the global Logger object, which is a static member of lug::System::Logger::Logger).

Preferably, do not use static and global variables at all.

3. Classes

3.a. Constructors

Where applicable, initialize members in the class definition (in the `.hpp` file).

Example:

```
// ...

namespace lug {
namespace Graphics {

// ...

class LUG_GRAPHICS_API Camera : public Node {
    // ...

protected:
    Scene* _scene{nullptr};
    RenderQueue _renderQueue;
```

```
14    RenderView* _renderView{nullptr};
15
16    float _fov{45.0f};
17    float _near{0.1f};
18    float _far{100.0f};
19
20 private:
21    // ...
22
23    Math::Mat4x4f _projMatrix{Math::Mat4x4f::identity()};
24    Math::Mat4x4f _viewMatrix{Math::Mat4x4f::identity()};
25
26    bool _needUpdateProj{true};
27    bool _needUpdateView{true};
28 };
29
30 } // Graphics
31 } // lug
```

3.b. Implicit conversions and User defined conversions

Do not define implicit conversions, use the `explicit` keyword for conversion operators and single-argument-constructors.

Even with the `explicit` keyword, only use user defined conversions when it's meaningfull in some particular cases. In Lugdunum, we use them to convert types from our Vulkan abstraction to native Vulkan types.

Example:

```
1  // ...
2
3  namespace lug {
4  namespace Graphics {
5  namespace Vulkan {
6
7  class Device {
8  public:
9     // ...
10
11    explicit operator VkDevice() const {
12       // ...
13    }
14
15    // ...
```

```
};

} // Vulkan
} // Graphics
} // lug
```

3.c. Copyable and Movable Types

All classes should define a default move and copy constructor and a default move and copy assignment operator using = `default`. If the move/copy operations are not useful for your class, you should disable them with = `delete`.

```
namespace lug {
namespace Graphics {
namespace Vulkan {

class LUG_GRAPHICS_API Camera final : public lug::Graphics::Camera {
public:
    Camera(const std::string& name);

    Camera(const Camera&) = delete;
    Camera(Camera&&) = default;

    Camera& operator=(const Camera&) = delete;
    Camera& operator=(Camera&&) = default;
};

} // Vulkan
} // Graphics
} // lug
```

3.d. Structs vs. Classes

`struct` are only for passing “inactive” data or Plain Old Data. They don’t have constructors, destructors, functions. Everything else is a `class`.

3.e. Inheritance and multiple Inheritance

All methods should be private, except for methods that need to be accessed in subclasses which have to be protected.
When a method need to be override, define it as virtual in the base class and use the key word `override`.
If no class inherited from the sublclass override the method, the key word `final` must be used. The key word

`final` must also be used for the inheritance itself, if no class inherit from the subclass.
Make your base class destructor virtual;

```
namespace lug {
namespace Graphics {

class LUG_GRAPHICS_API Light : public MovableObject {
public:
   virtual ~Light();
   virtual void* getData(uint32_t& size) = 0;
   // Virtual destructor and method because overridden in PointLight

   // ...
};

class LUG_GRAPHICS_API PointLight final : public Light {
   // Use final here because no class inherit from PointLight

public:
   ~PointLight() override final;
   void* getData(uint32_t& size) override final;
   // Use final here because no subclass of PointLight will override getData() or the
       destructor
   // Also use override to overide getData and Light destructor

   // ...
};

} // Graphics
} // lug
```

Multiple inheritance is discouraged and is only allowed if all base classes are interfaces or if the base classes are abstract classes (but discouraged). The diamond inheritance is disallowed.

3.f. Interfaces

All methods must be pure virtual (ends with = `0`).
The interface must declare a virtual destructor.

```
class Foo {
public:
   virtual ~Foo() = default;
   // virtual destructor

```

```
6     virtual method() = 0;
7     // Pure virtual method
8 };
```

3.g. Operator Overloading

Do not abuse of operator overloading, only use them if their meaning is obvious.
The operators should be defined in the same namespace and class as the type it overloads, except from the binary operators (taking two parameters) that is encouraged to be declared in a non-member function (however in the same namespace as the class)

3.h. Declaration Order

The declaration order in a class should be `public:`, followed by `protected:`, then `private:`. The methods and data members should have different blocks.
Each declaration block should be separated with a blank line. It's encouraged to separated with a blank line inside the blocks too for each methods/properties that can be grouped.

In each declaration block, the order in the following:

- Using declarations, typedefs and enums
- Constants (static const)
- Constuctors and assignment operators
- Destructors
- Methods, including static methods
- Data properties

4. Functions

4.a. Parameter Ordering

When a function takes inputs and outputs (i.e. an output pointer), it should list inputs first, then outputs.

Inputs are usually values or `const` references, whereas inputs and input/output should be pointers to non-`const` variables.

4.b. Write Short Functions

It seems obvious, but prefer dividing large functions in "atomic" functions, dedicated to one specific task. As a rule of thumb, a function is considered as large when it has more than 50 lines.

Short functions improve code maintainability and readability.

4.c. Reference Arguments

All parameters passed by reference must be labeled const.

Example:

```
1 void foo(const string &in, std::string *out);
```

If you want to pass a `null`-able value as a parameter, you can use a raw pointer instead.

4.d. Function Overloading

Function overloading can add complexity to the code and make it less readable. You can use function overlading but ask yourself first if there is not a better, more readable option available.

4.e. Default Arguments

Default arguments are allowed on non-virtual functions when the default is guaranteed to always have the same value accross possible overloaded functions. For the same reasons detailed function overloading, be careful when using default arguments.

4.f. Trailing Return Type Syntax

C++11 introduced a new syntax for function return types.

Old:

```
1 int foo(int x);
```

New (C++11 only):

```
1 auto foo(int x) -> int;
```

The difference is that in the new syntax, the type is declared in the function's scope.

Do continue to use the older style of function declaration where the return type goes before the function name. Use the new trailing-return-type form only in cases where it's required (such as lambdas).

5. Other

5.a. Ownership and Smart Pointers

Object ownership is represented by a `std::unique_ptr` or a `std::shared_ptr`, a.k.a. smart pointers. Consider that you never have ownership on raw pointers, so you must never free or delete a raw pointer.

Do not use shared ownership without a very good reason to back it up.

Never use `std::auto_ptr`. Instead, use `std::unique_ptr`.

6. Others C++ Features

6.a. Rvalue References

Use rvalues only in constructors and movement operators/constructors, or to do perfect forwarding (with `std::forward`).

6.b. Friends

Use of `friend` is not strictly forbidden but you should avoid it as possible. You can use friend if it allows to remove a public access that is only used by the friended class.

Friends should be defined in the same file as much as possible.

6.c. Exceptions

Use exceptions sparsely, only when another option such as return status/code or asserts is not available.

6.d. Run-Time Type Information (RTTI)

Avoid over using Run Time Type Information (RTTI).

Using the type of an object at run-time is in general a problem of architecture. And it's also hard to maintain if you have decision trees or switch statements scattered throughout the code which all need to be updated when making changes.

6.e. Casting

Even if the C++-style cast syntax (with `static_cast<>`) is more verbose, always use it over old C-style casts.

- Use brace initialization to convert arithmetic types (e.g. int64{x}). This is the safest approach because code will not compile if conversion can result in information loss. The syntax is also concise.
- Use `static_cast` as the equivalent of a C-style cast that does value conversion, when you need to explicitly up-cast a pointer from a class to its superclass, or when you need to explicitly cast a pointer from a superclass to a subclass. In this last case, you must be sure your object is actually an instance of the subclass.
- Use `const_cast` to remove the const qualifier, avoid using it too frequently.
- Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.

6.f. Streams

Use streams only when they actually are the best tool for the job. Stream formatting and performance is not that good so think of the available alternatives when using streams.

Do not use `std::cout` or `std::cerr` for logging purpose, use `System::Logger::Logger` instead, which supports custom types, and other useful features such as easy-to-use formatting and cross-platform handlers/sinks.

Overload << as a streaming operator for your type only if it represents a value and writes a human readable representation of that value. Do not expose implementation details in the output of <<. Such overloaded types are de-facto supported by Lugdunum's logger.

6.g. Preincrement and Predecrement

Always use the prefixed form.

6.h. Use of const

Always use `const` where applicable, and use `constexpr` when you are defining true constants, i.e. fixed at compile time. When writing code, put the `const` keyword before the type:

```
1 const int* foo;
```

6.i. Integer Types

Always use fixed-size integer types from <cstdint> such as `int32_t`, `int16_t`, `uint32_t`, etc. instead of `unsigned`, `long`, `unsigned int`, etc.

When applicable, always use `size_t` or `ptrdiff_t` to hint at the actual purpose of the variable.

6.j. Preprocessor Macros

Avoid preprocessor macros, prefer `constexpr` values, inline functions, or even lambdas.

X macros are a special case and are not as much discouraged, but do weigh the advantages of the code lightness versus the readability disadvantage induced by X macros. X macros are very hard to read for inexperienced programmers, can quickly become too complicated and can really hurt the maintanability of the codebase. Be smart, and keep them simple!

6.k. 0 and nullptr/NULL

Use `0` for integers, `0.0` for reals, `nullptr` (do *not* use `NULL`) for pointers, and `'\0'` for chars.

6.l. sizeof

Prefer `sizeof(varname)` to `sizeof(type)` as it improves code maintainability.

Example:

```
SomeType data;
memset(&data, 0, sizeof(data)); // Good
memset(&data, 0, sizeof(SomeType)); // Bad
```

6.m. auto

Use `auto` to avoid type names that are noisy, obvious, or unimportant - cases where the type doesn't aid in clarity for the reader. Continue to use manifest type declarations when it helps readability. However, do not use a `auto` variable with initializer lists.

Only use auto on local variables.

6.n. Braced Initializer List

Prefer using Braced Initializer List where possible.

6.o. Lambda expressions

Use lambda expressions when appropriate, e.g. to pass a short comparaison function to an std algorithm.

Always use explicit captures by specifying which variables do you want, and make sure that the lifetime of the variable is longer than the lifetime of the lambda when capturing by reference or capturing a pointer.

Keep unnamed lambdas short and without a lot of captures.

Specify the return type of the lambda explicitly only when it is not obvious to the readers.

6.p. Template metaprogramming

Think twice before using template metaprogramming, prefer a simpler technique if possible.

6.q. std::hash

Do not define specializations of `std::hash`, as writing hash functions is difficult and error-prone, even for experts. Due to the high risk of ending up with a broken hash function, it has been decided to forbid specializing `std::hash` for your types.

6.r. C++14

Always use C++14 libraries and features if possible, but keep it compatible with all the project supported compilers.

6.s. Nonstandard Extensions

Only use standard extensions, exeptionnally where at least widely used and available on all the project supported compilers. Be smart and don't introduce non-maintainable code in the codebase ;)

7. Naming

7.a. File and Folder Names

File names must match the class defined inside and the file must be placed in a directory structure matching the namespace for the class.

Therefore, filenames should be in UpperCamelCase and shoud not contain separators such as spaces, dashes or underscores.

Header files must head with the `.hpp` extension, inline header files must hend with `.inl` and must be placed alongside the classic header files, and finally, source files must end with the `.cpp` extension.

Source files must be placed in the `./src/` folder, whereas header and inline source files must be located in the `./include/` folder.

Example: `Foo::Bar::MyClass` should have the following directory structure:

```
.
|-- src
| |-- Foo
| | +-- Bar
| | |-- MyClass.hpp
| | +-- MyClass.inl
| +-- ...
+--include
   |-- Foo
   | +-- Bar
   | +-- MyClass.cpp
   +-- ...
```

7.b. Type Names

Type names should be in UpperCamelCase (with no underscores): `MyClass`.
This applies for classes, structs, type aliases, enums and type template parameters.

7.c. Variable Names

Variable names should be in lowerCamelCase (with no underscores): `myVariable`.

Private and protected class members should start with an underscore.

Example:

```
class Foo {
// ...
public:
   int barPublic;

protected;
   int _barProtected;

private:
   int _barPrivate;
};
```

7.d. Constant Names

Refer to Variable Names above.

7.e. Function Names

Function names shoul be written the same way as variable names, in lowerCamelCase, with no underscores: `myFunction()`.

When there is an acronym, it should be capitalized: write `sendUDP()`, not `sendUdp()`.

7.f. Namespace Names

Nested namespaces should be in UpperCamelCase and the top-level namespace should be in lowerCamelCase, with no underscores, e.g.: `lug::Graphics`.

Do not use nested namespaces that would match top-level namespaces:

```
namespace lug {
namespace std { // Bad
   // ...
}
}
```

7.g. Enumerator Names

Refer to Variable Names above.

7.h. Macro Names

Macro names should be written in upper case with underscore between words: `MY_MACRO`.
Keep in mind that macros are not recommanded (See Preprocessor Macros).

8. Comments

8.a. Comment Style

Use `//` for single-line comments and `/* */` for multiline comments outside of function blocks. Small blocks of multiline text can be written as mutliple `//` lines, see an example in the implementation section.

A comment should always start with an upper case letter, and there should be a space after the opening comment syntax.

Example:

```
1  //comment // Bad
2
3  /* This is single-line a comment */ // Bad
4
5  // This is a single-line comment // Good
6
7  /**
8   * This is a multiline
9   * comment, that spans three lines of
10  * text.
11  */ // Good
12
13 // This is also a small
14 // mutliline comment, but this is allowed // Good
```

8.b. Class Comments

Each class should be described with a block preceding the class declaration, in accordance with the Doxygen format (with `@`, not `\`, i.e. `@brief` instead of `\brief`).

Example:

```
1  /**
2   * @brief Class for camera.
3   *
4   * This class represents a Camera in the 3D engine. A scene can be attached
5   * to a Camera. Only one scene can be attached to a Camera.
6   * A Camera can be attached to only one Render::View.
```

```
*/
class LUG_GRAPHICS_API Camera : public Node {
   // ...
}
```

8.c. Function Comments

Same as classes, function declarations should be preceded with a block defining the function purpose, params, and return values. The block is also in accordance with the Doxygen format (with `@`, not `\`, i.e. `@brief` instead of `\brief`).

Example:

```
class LUG_GRAPHICS_API Graphics {
   /**
    * @brief Initializes the application with the informations filled in @p initInfo
       structure.
    *
    * @param[in] initInfo The initialize information.
    *
    * @return @p true if the initialization was successful.
    */
   bool init(const InitInfo& initInfo);
}
```

8.d. Variable Comments

Data member

Comments of members of struct, union, class, or enum should be written after the variable declaration, with `///<` (in accordance to the Doxygen format) instead of the usual comment syntax.

Example:

```
struct foo {
   int bar; ///< This is a data member comment.
};
```

The comment should describe, in less than one line, the purpose of the data. This comment will be present in the generated API documentation.

However, no comment is needed if the type and name of the data member are self explanatory.

Global variables

The comment style of global variables is the same as single-line comments, described in Comment Style.

8.e. Implementation Comments

If a block is tricky or too complicated to understand it by reading the code, a comment can be written before it.

At Lugdunum, we prefer well written and readable code over over-commented, unreadable blocs of code.

For example, you should not comment trivial operations.

Example:

```
for (std::size_t i = 0; i < renderQueue.getLightsNb(); ++i) {

   // Blend constants are used as dst blend factor
   // Now the depth buffer is filled, we can set the blend constants to 1 to enable blending
   if (i == 1) {
      const float blendConstants[4] = {1.0f, 1.0f, 1.0f, 1.0f};
      vkCmdSetBlendConstants(static_cast<VkCommandBuffer>(cmdBuffer), blendConstants);
   }
}
```

Trivial code:

```
// Increment i // Bad
i += 1;
```

The same applies for single lines, however, if you feel like you have to comment everything, maybe you should rethink your code first ;)

Example:

```
// All the lights pipelines have the same renderPass
API::RenderPass* renderPass = _pipelines[Light::Light::Type::Directional]->getRenderPass();
```

8.f. Punctuation, Spelling and Grammar

Comments should have good punctuation, spelling and grammar, like narrative texts.

Comments can sometimes be less formal, like for short comment describing a data member.

8.g. TODO Comments

You should generally add a TODO comment before any code that is incomplete or needs review and or particular attention. This allows temporary quircks and hacks to be grouped and easily searched (e.g. in an IDE) in order to be correctly addressed before any merging is done to a definitive branch or version.

The name of the person who added such comment should appear inside parenthesis, right after the TODO. As such, the person resonsible for the comment remains easily tracked and also accountable for the TODO.

Example:

```
// TODO(saveman71): replace opening file with something more global
std::ifstream shaderCode(file, std::ios::binary);
```

9. Formatting

9.a. Line Length

A line should not be more than 120 characters. This greately code enhance readability and prevents editor auto-wrapping that usually isn't smart enough to split the line(s) at the right position(s).

9.b. Non-ASCII Characters

Although Non-ASCII characters should be rare, because applications should be localized externally, they must use u8 prefix to ensure that the string literal uses UTF-8 encoding: u8"äôéè".

Don't use `char16_t` or `char32_t` because they are not for UTF-8 character storage. Don't use the Windows type `wchar_t`, unless you are working with the Windows API in implementation specific files, or regions delimited by preprocessor directives.

9.c. Spaces vs. Tabs

Never use tabs, only use spaces.
Indentation is only with 4 spaces, so configure your editor to correctly indent with 4 spaces.

9.d. Function Declarations and Definitions

The return type, function name and parameters should be on the same line.

Example:

```
void Node::lookAt(const Math::Vec3f& targetPosition) {
   // ...
}
```

If the line is longer than the maximum line length, you should write each parameter on one, separated line. The last parameter has to contain the closing parenthesis and the opening bracket of the function's scope.

Example:

```
void Node::lookAt(
   const Math::Vec3f& targetPosition, // 4 spaces indent
   const Math::Vec3f& localDirectionVector,
   const Math::Vec3f& up,
   TransformSpace space) {
   // ...
}
```

9.e. Lambda Expressions

Lambda expressions are to be formatted the same way as functions.
There is no space between the capture mode and the variable captured.

Example:

```
auto toUpper = [&foo](char c) {
   return static_cast<char>(toupper(c));
};
```

9.f. Function Calls

Splitting arguments in function calls should respect the same rules as in function declarations.

Example:

```
void main(int ac, char* av[]) {
   // ...
   callFooWhichIsALongFunctionAndTakesManyArguments(
      andNo,
      your,
      functionNames,
      should,
      reallyNot,
      beThatLong);
}
```

9.g. Braced Initializer List Format

Splitting arguments in braced initializer lists should respect the same rules as in function declarations except that the closing curly brace should be on his own line.

There is no space after the opening and the closing curly braces {}.

Examples:

```
VkDescriptorSetLayoutCreateInfo createInfo{
   createInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
   createInfo.pNext = nullptr,
   createInfo.flags = 0,
   createInfo.bindingCount = bindingCount,
   createInfo.pBindings = layoutBindings
};

lug::Graphics::Vulkan::Image::Extent extent = {width, height};
```

9.h. Conditionals

The conditions should have no spaces after the opening parenthesis (and before the closing parenthesis), and there should be one space between the condition keyword and the opening parenthesis (.
The `else` keyword should be on the same line as the closing bracket } of the previous condition.

Example:

```
if (condition) {
   // 4 spaces indent
   // ...
} else if (condition) {
   // ...
} else {
   // ...
}
```

For short single-line conditions, it's OK to put the condition on one line, but only if it improves readability:

```
if (!condition) return false; // Ok
```

If the short single-line condition is not on one line, you must wrap the body with curly braces:

```
if (!condition) // Bad
   return false;

if (!condition) { // Good
   return false;
}
```

9.i. Loops and Switch Statements

As for conditions, you must always wrap the body for loops statements with curly braces, even if it's only one line long.

Example:

```
for (uint32_t i = 0; i < 5; ++i) // Bad
   std::cout << i << std::endl;

for (uint32_t i = 0; i < 5; ++i) { // Good
   std::cout << i << std::endl;
}
```

Switch brackets `{}` follow the same rules as function brackets.
You should not use brackets `{}` around `case` keyword.

Example:

```
switch (enumVal) {
   case VK_SUCCESS: // 4 spaces indent
      return "Success"; // 8 spaces indent
   case VK_NOT_READY:
      return "A fence or query has not yet completed";
}
```

It's OK to put case on the same line if it enhances readability. However, all the `cases` should one line long, as to keep consistency among each `switch` statement.

Example:

```
// Bad
switch (type) {
   case Light::Type::Directional: return std::make_unique<Light::Directional>(name);
   case Light::Type::Point:
      return std::make_unique<Light::Point>(name);
   case Light::Type::Spot: return std::make_unique<Light::Spot>(name);
}

// Good
switch (type) {
   case Light::Type::Directional: return std::make_unique<Light::Directional>(name);
   case Light::Type::Point: return std::make_unique<Light::Point>(name);
   case Light::Type::Spot: return std::make_unique<Light::Spot>(name);
}
```

9.j. Pointer and Reference Expressions

When declaring a pointer, the * should be placed on the type, i.e. there is no space before the * or &.

Example:

```
int foo;
int* bar; // Good
int * x; // Bad

bar = &foo;
```

9.k. Boolean Expressions

Spaces around boolean operators are obligatory.
If a boolean expression is longer than the maximum line length, you should write each expression on separate lines, with the boolean operators at the end of each lines.

Example:

```
if (!_pipelines[Light::Type::Directional] ||
   !_pipelines[Light::Type::Point] || // 4 spaces indent
   !_pipelines[Light::Type::Spot]) {
   // ...
}
```

9.l. Return Values

The use of parenthesis around the return value is disallowed:

```
return (5); // Bad
return 5; // Good
```

The only exception is for complex expressions:

```
return (longExpressionA &&
      longExpressionB);
```

9.m. Variable and Array Initialization

Prefer using {} than ().
There is no spaces around and inside the {} or ().

Example:

```
int foo(5); // Good
int foo{5}; // Better
```

9.n. Preprocessor Directives

Preprocessor directives follow a separate intentation scheme:

- Each preprocessor directive starts with a # on the first character of the line
- Nested conditions should have their content indented with one and only space per indent level.
- Again, preprocessor directive are not dependent on the indentation of the code they currently are located, and in the same way, indentation of code located inside preprocessor directives should not be disturbed.

A good way to remember this is that final, preprocessed code, should have the correct indentation.

```
int main(int ac, char* av[]) {
   uint16_t foo = 21;

#if defined(MACRO_A)
   if (ac > 2) {
      // Code run only if macro A
      foo += 21;
# if defined(MACRO_B)
      // Code run only if macro A and macro B
      foo -= 42;
# endif
   }
#endif

   return foo;
}
```

9.o. Constructor Initializer Lists

If the constructor line is longer than the maximum line length, all the code after : should be written to a new line.

Example:

```
Camera::Camera(const std::string& name) : Node(name) { // Short constructor
   // ...
}

```

```
5 Image::Image(VkImage image, const Device* device, const Extent& extent, bool swapchainImage,
      VkImageAspectFlags aspect) :
6     _image(image), _device(device), _swapchainImage(swapchainImage), _aspect(aspect), _extent
          (extent) { // Long constructor, 4 spaces
7     // ...
8 }
```

9.p. Vertical Whitespace

Try to limit the use of blank lines, but you can use them sparsely to split logically independant code sections and help readability.
Each file should end with a new line (`\n`).

10. Conclusion

This style guideline is quite complete, but still missing some details. If you find an edge-case that this guideline does not cover, feel free to report any issue or contribute to this guideline.

As a general rule of thumb, your code should be the most readable possible, and it is always possible to flex some rules, if it makes your code better.

Good luck, have fun coding with us!

III. Contributing to Lugdunum

1. Branching strategy

In order to have an efficient workflow, we chose to create different branches, each with its own responsability:

- `master`: the *master* branch points to the latest stable release of the 3D engine. It is protected, which means that only trusted contributors can accept a pull-request to this branch. This branch guaranteed (up to a certain level) to be stable, and this is the only branch officially supported.
- `hotfix`: this branch is dedicated to urgent bug fixes of the *master* branch. Emergency fixes will be commited to this branch directly, and a pull-request will be opened to allow a really quick code-review before pushing the changeset to *master*.
- `release`: this branch contains changes that one day will reside on *master*. They are present to allow users to test out new functionnality before it is officially supported and bug-free.
- `dev`: this is the unstable, working branch. Changes on this branch may not be quite stable yet, and they might not work correctly on every platform. Once *dev* is sufficiantly stable, it will be merged onto *release* (or cherry-picked).
- `feature-*`: these branches are feature branches, usually used by one or more developers working on a new feature. Pull-requests from these branch must be opened onto *dev* only.

An example is show in Figure 1.7, to demonstrate the utility of each branch, with a real-world scenario. This branching strategy is applicable across all Lugdunum's projects and must be respected. As such, the branches `master` and `dev` are *protected* on Github, which means that only administrators have push access to these branches, and that pull-requests with complete, passing tests must be opened in order to have changes implemented in these branches.

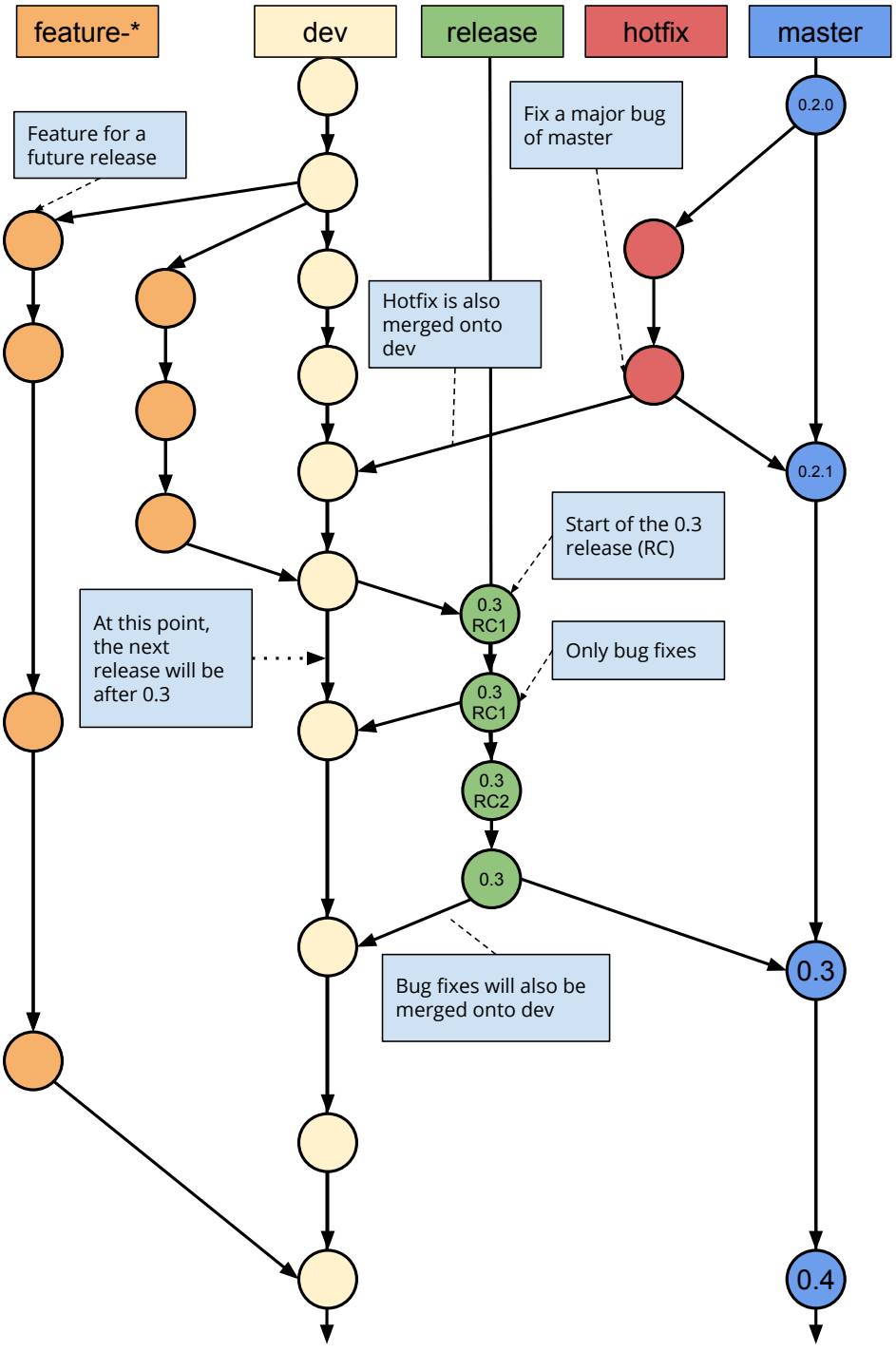

Figure 1.7: Brancing strategy

IV. Testing architecture

Each commit pushed on each branch is compiled and tested by CircleCI[1] and AppVeyor[2].

You are encouraged to write tests for your code. Broken build will not be allowed in any case in a pull-request, so be careful!

1. Introduction

All our sensible code is covered by unit tests. We use the Google-Test[3] framework which is considered as a third party module of our project. It is bound with Google-Mock[4].

All the written tests can be found in the test folder of the Lugdunum's repository[5] in the dev branch.

All the tests included in the folder `test` are executed when you run the tests with cmake, and are executed as well in CircleCI.

2. How to add new tests

If you want to add your tests, we recommend you to create a new folder in the `test` folder and put all your `*.cpp` in it. The structure of a test file should be like following :

```
#include <gtest/gtest.h>

TEST(myTestPool, myTest) {
   bool toto = true;
   EXPECT_EQ(toto, true);
}
```

To be compiled with other tests, each tests directory should have a CMakelists.txt. In a `Math` directory, this file will have the following format:

```
# Tests directory path
set(SRC_ROOT ${PROJECT_SOURCE_DIR}/Math)

# Define *.cpp tests
set(SRC
   ${SRC_ROOT}/Geometry/Transform.cpp
   ${SRC_ROOT}/Matrix2x2.cpp
   ${SRC_ROOT}/Matrix3x3.cpp
```

[1] CircleCI: https://circleci.com/gh/Lugdunum3D/Lugdunum
[2] AppVeyor: https://ci.appveyor.com/project/Lugdunum/lugdunum
[3] Google-Test: https://github.com/google/googletest/tree/master/googletest
[4] Google-Mock: https://github.com/google/googletest/tree/master/googlemock
[5] Lugdunum's repository: https://github.com/Lugdunum3D/Lugdunum/tree/dev/test

```
9     ${SRC_ROOT}/Matrix4x4.cpp
10    ${SRC_ROOT}/Quaternion.cpp
11 )
12 source_group("src" FILES ${SRC})
13
14 # Add tests to compilation
15 lug_add_test(Math
16           SOURCES ${SRC}
17           DEPENDS lug-math
18 )
```

Note: `source_group` on line 12 is a special CMake directive used for grouping source files in IDE project generation, for example groups in Visual Studio. More information is available on the official CMake documentation[6].

3. Build tests

When using CMake, you need to add the command line argument `-DBUILD_TESTS`.
It will create one project for each test directory. In the previous example, it will create a `runMathUnitTests` project.

V. Contact us

The development team is available through a wide range of channels if you want to reach out to us:

1. Github

You can find our repositories on Github, at Lugdunum3D[7], and report specific problems or questions directly by filing a new issue.

2. Mailing list

If you want to write us an email, you can totally do so at `lugdunum_2018@labeip.epitech.eu`.

[6] on the official CMake documentation: `https://cmake.org/cmake/help/v3.0/command/source_group.html`
[7] Lugdunum3D: `https://github.com/Lugdunum3D`

Part. 2

Lugbench

Contents

1. Homepage

The homepage is located at `http://lugbench_url/gpus`.

2. Project architecture

2.a. Configuration

The project contains some configuration files.
Here is the list.

Files	Description
package.json	The definition of dependencies, used by `npm` when installing the project.
gulpfile.js	Configuration of different Gulp tasks.
tsconfig.json	TypeScript configuration file.
tslint.json	TypeScript linter configuration file.
conf/*.js	Configuration of additional modules used by the project.

2.b. Sources

All the sources files are located in the `src` folder.
The initialization page is located at the root of this `src` folder.
Then, all the components and models are located in the `src/app` folder.

I. API documentation

1. List of endpoints

Method	Route	Description
GET	`/api/v1/gpus`	Returns all GPUs present in the database.
GET	`/api/v1/gpus/:id`	Returns the GPU with the id ":id" if present in the database.
PUT	`/api/v1/gpus`	Add or edit a GPU if present in the database.

Note: The details of the object to pass in the payload is available online on the API's repository[1].
The object has to be formatted in json.

2. Response codes

Here is the response codes returned by the back-end.

[1] online on the API's repository: `https://github.com/Lugdunum3D/LugBench-API/blob/dev/v1/models/gpu/index.js`

Response code	Description
200	Success - Request returned without any problem.
201	Creation success - Object inserted in the database without any problem.
400	Bad request - Some headers or fields are missing.
500	Server error - Please open an issue or contact us.

II. Unit tests

Our API is covered by unit tests. We will use Mocha[2], a feature-rich JavaScript testing framework running on Node.js.
All creation and retrieving of data are tested.

[2] Mocha: https://mochajs.org/