



Technical Documentation n°1

Abstract

Lugdunum is an open-source 3D engine using the Vulkan API as a backend. Lugdunum'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.

In this 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 with hopes to let this document be as simple and straightforward as possible, for developers of all levels. It is however required that you have some background about 3D rendering *and 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 document, you should have a rough idea of how Lugdunum's source code is architected, 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.

Document description

Title	: [2018][TD1] Lugdunum
Modification date	: April 20, 2017
Accountant	: Antoine Bolvy
E-mail	: lugdunum_2018@labeip.epitech.eu
Subjet	: Lugdunum – Technical Documentation n°1
Document version	: 1.3

Revisions table

Date	Authors	Version	Modified Section(s)	Comment(s)
2017-03-07	Antoine Bolvy	1.0	All	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

Contents

About the scope of the EIP	1
Architecture of Lugdunum	2
a. Renderer architecture	2
b. Sequence diagrams	4
c. Vulkan Rendering	7
1. Global	7
2. Forward render technique	8
Code Guidelines and Style	12
a. Header files	12
b. Scoping	14
c. Classes	15
d. Functions	18
e. Other	19
f. Others C++ Features	19
g. Naming	22
h. Comments	24
i. Formatting	27
j. Conclusion	32
Contributing to Lugdunum	33
a. Branching strategy	33
b. Tests	35
Contact us	36
a. Github	36
b. Mailing list	36

About the scope of the EIP

The **E**pitech **I**nnovative **P**roject is the final step in a 5 year computer science program at the Paris Graduate School of Digital Innovation (also know as EPITECH), a program that distinguishes itself from others by its focus on practical projects rather than theory.

The EIP is a group project that spans the 4th and 5th year of the program which aims to have as a result a finished project that has actual lasting value, is well documented and is immediately usable in a professional environment. It is a chance for students to acquire skills and experience in accomplishing a task which aims to emulate the real world working environment.

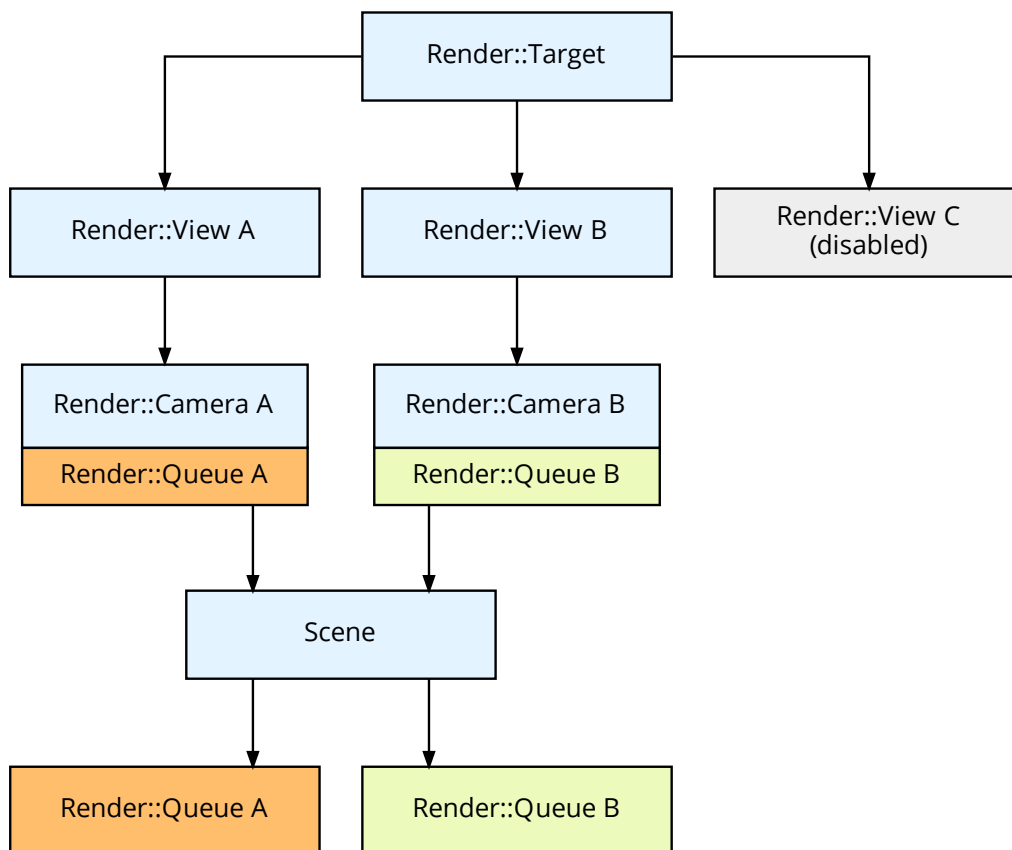
Just like a real large scale project. The students are not only faced with technical problems but they are also tasked with handling marketing, communication, team management as well as writing and maintaining professional documentation, technical or otherwise.

The EIP is the amalgamation of EPITECH's project oriented vision. It aims to symbolize the transition between student and professional life.

Architecture of Lugdunum

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

a. Renderer architecture



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::Views`, 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 };
```

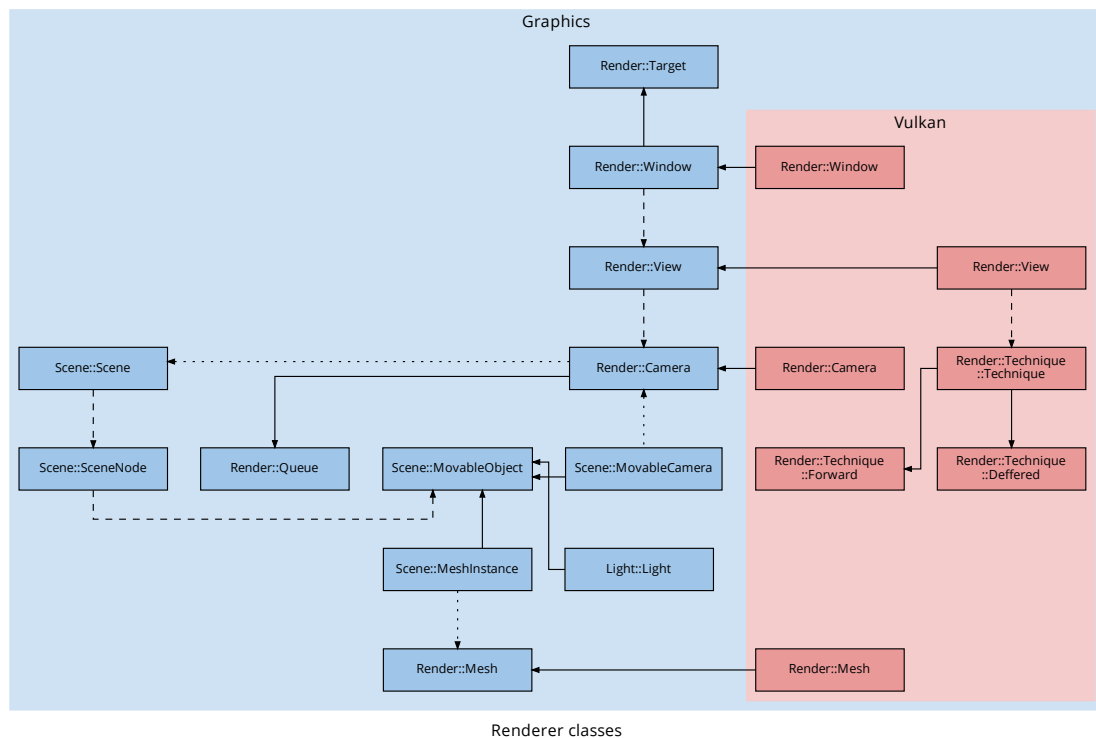
Each of the components of [Render::View::Viewport](#) and [Render::View::Scissor](#) are defined as percentage values (i.e. a float between 0.0 and 1.0), so it has the same appearance on every size of the [Render::Target](#).

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::Views](#).

[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.

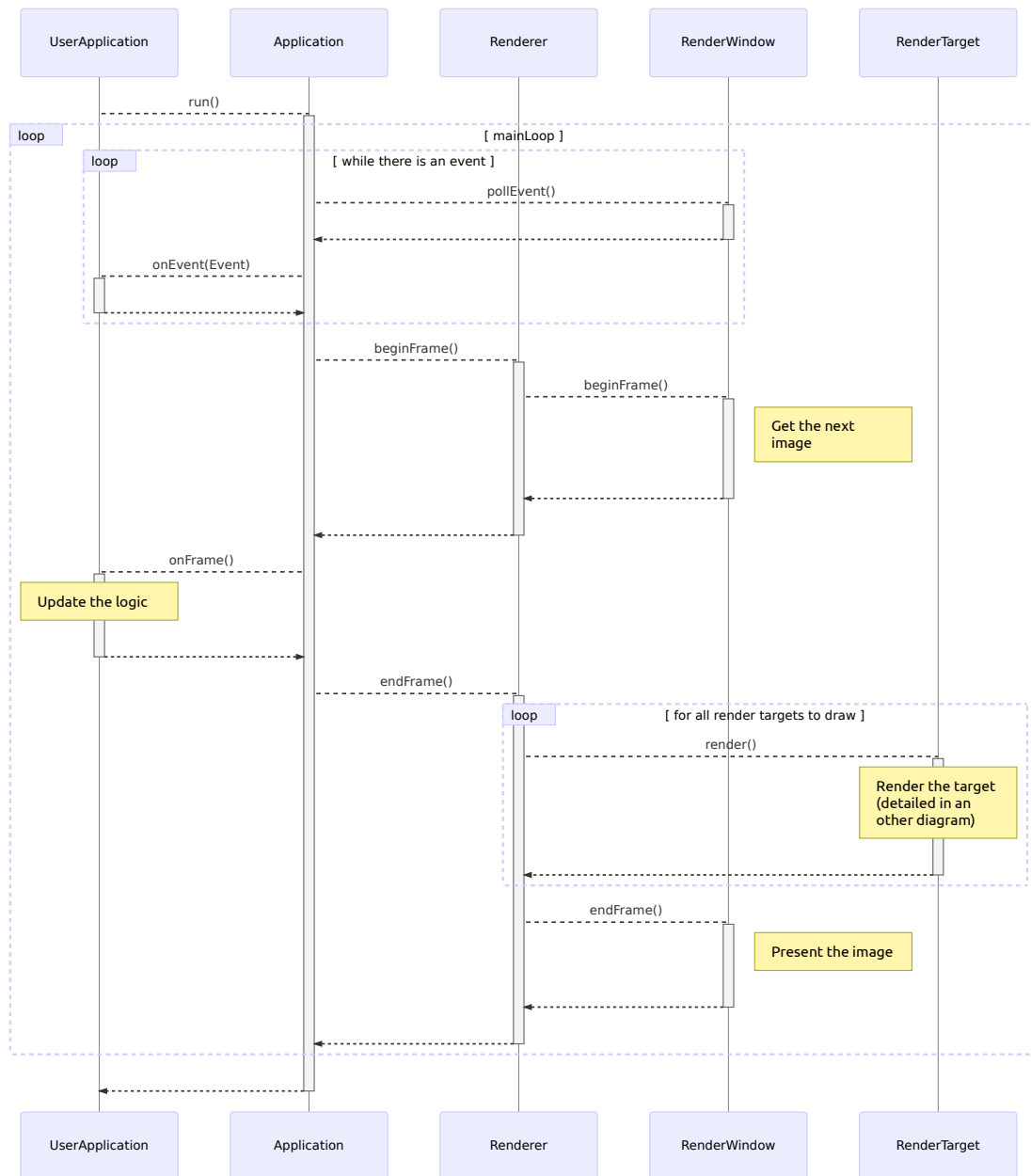
Every frame, the [Render::Queue](#) is cleared, then filled by the [Scene::Scene](#) with the objects visible by the [Render::Camera](#)'s frustum.

The [Render::Queue](#) is finally sent to [Vulkan::Render::Technique::Technique::render\(\)](#).



b. Sequence diagrams

The sequence diagrams below show how a frame renders in our 3D engine. The **Render::Target::render()** method of **Render::Target** has been separated in a different diagram lower to ease readability.



Let us describe this sequence diagram, step by step:

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

```

1 int main(int argc, char* argv[]) {
2     UserApplication app;
3
4     if (!app.init(argc, argv)) {
5         return EXIT_FAILURE;
6     }
7
8     if (!app.run()) {

```

```

9     return EXIT_FAILURE;
10 }
11
12     return EXIT_SUCCESS;
13 }

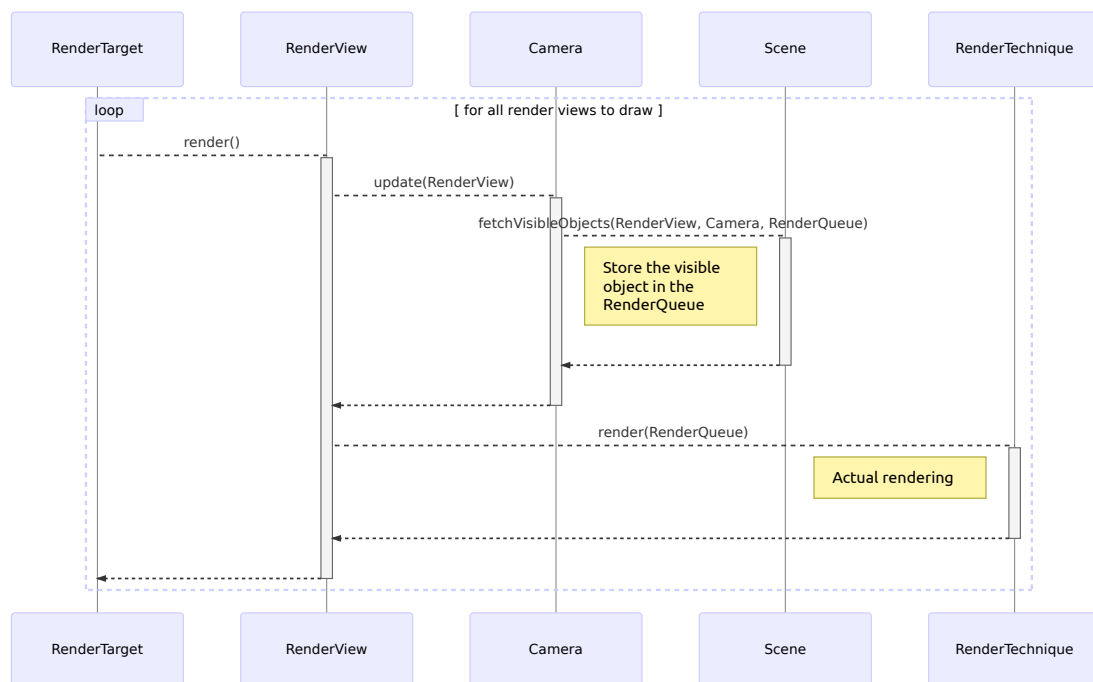
```

The method `lug::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 `lug::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)`.

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

Now, the user can update the logic of the 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.



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

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

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

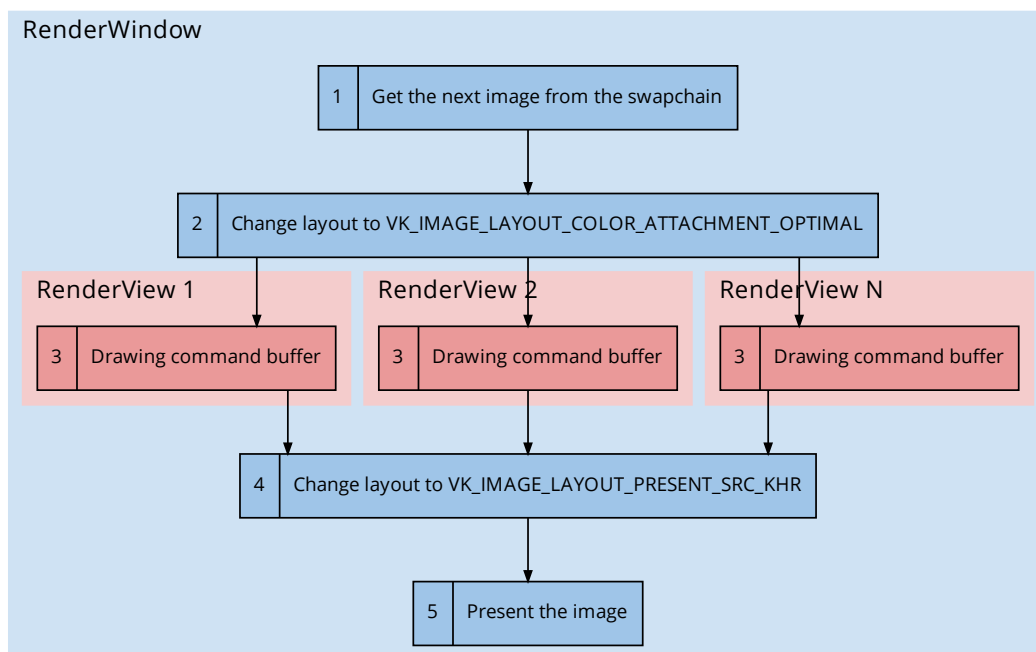
Then the `Render::View` can call the render technique to draw the the elements in the `Render::Queue` (e.g. for Vulkan a class inheriting from `Vulkan::Render::Technique::Technique`).

c. Vulkan Rendering

1. Global

GPU Side

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



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 parrallel
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.

Steps 2 and 4 are using prebuilt Vulkan command buffers, one for each image in the swapchain. Step 3 is dependant of the render technique used.

CPU Side

It is up to each method to select semaphores, from a pool, to be used between each step.

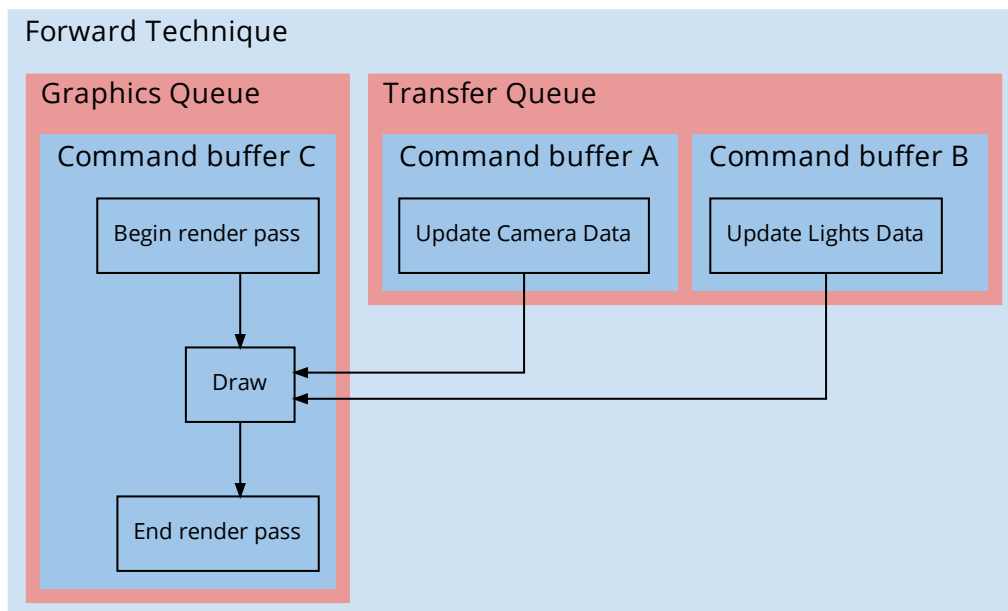
Steps 1 & 2 The method `Vulkan::Render::Window::beginFrame()` is used to accomplish steps 1 and 2. This method chooses one semaphore to be notified when the next image is available and chooses N semaphores to notify each `Vulkan::Render::View` when the image has changed layout. (N being the number of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

2. Forward render technique

GPU Side



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 datas of the `Render::Camera` and `Light::Lights`, each one contained in an uniform buffer `Vulkan::API::Buffer` sent in separate `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:

```

1 // Camera
2 layout(set = 0, binding = 0) uniform cameraUniform {
3     mat4 view;
4     mat4 proj;
5 };
6
7 // Directional light
8 layout(set = 1, binding = 0) uniform lightUniform {
9     vec3 lightAmbient;
10    vec3 lightDiffuse;
11    vec3 lightSpecular;
12    vec3 lightDirection;
13 };
14
15 // Point light
16 layout(set = 1, binding = 0) uniform lightUniform {
17    vec3 lightAmbient;
18    float lightConstant;
19    vec3 lightDiffuse;
20    float lightLinear;
21    vec3 lightSpecular;
22    float lightQuadric;
23    vec3 lightPos;
24 };
25
26 // Spot light
27 layout(set = 1, binding = 0) uniform lightUniform {
28    vec3 lightAmbient;
29    vec3 lightDiffuse;
30    vec3 lightSpecular;
31    float lightAngle;
32    vec3 lightPosition;
33    float lightOuterAngle;
34    vec3 lightDirection;
35 };

```

Each type of lights have a different pipelines 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 :

```

1 layout (push_constant) uniform blockPushConstants {
2     mat4 modelTransform;
3 } pushConstants;

```

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

The "Command buffer C" for the drawing depends on the two command buffers of transfer by the 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.

We do not want to have a lot of allocations, so the pool is just allocating a chunk of memory on the GPU which will contains many `Vulkan::Render::BufferPool::SubBuffers`.

A `Vulkan::Render::BufferPool::SubBuffer` is just part of a bigger `Vulkan::API::Buffer` that we can allocate / free 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 re-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's 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:

```

1 BeginCommandBuffer
2
3 # The viewport and scissor are provided by the render view
4 SetViewport
5 SetScissor
6
7 BeginRenderPass
8
9 # We can bind the uniform buffer of the camera early
10 # It's the same everywhere
11 BindDescriptorSet(Camera)
12
13 # All the lights influencing the rendering (visible to the screen)
14 Foreach Light
15     # Each type of Light have a different pipeline
16     BindPipeline(Light)
17
18     # We can bind the uniform buffer of the light
19     BindDescriptorSet(Light)
20
21     # All the objects influenced by the light
22     Foreach Object
23         # Push the transformation matrix of the Object
24         PushConstant(Object)
25
26         # We use indexed draw, so we need to bind
27         # the index and the vertex buffer of the object
28         BindVertexBuffer(Object)
29         BindIndexBuffer(Object)
30

```

```
31     DrawIndexed(Object)
32     EndForeach
33 EndForeach
34
35 EndRenderPass
36
37 EndCommandBuffer
```

Code Guidelines and Style

a. 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. 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.

2. Headers Guards

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

```
1 #pragma once
```

3. Forward Declarations

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

4. 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:

```
1 #pragma once
2
3 // ...
4
5 namespace lug {
6     namespace System {
7         namespace Logger {
8
9             class LUG_SYSTEM_API Logger {
```



```

10 // ...
11
12 template<typename T, typename... Args>
13 void debug(const T& fmt, Args&&... args);
14
15 // ...
16 };
17
18 #include <lug/System/Logger/Logger.inl>
19
20 } // Logger
21 } // System
22 } // lug

```

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

```

1 // No template opened here
2
3 template<typename T, typename... Args>
4 inline void Logger::debug(const T& fmt, Args&&... args) {
5     // ...
6 }
7
8 // No template closed here either

```

5. 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:

```

1 #include <lug/System/Logger/Logger.hpp>
2
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <unistd.h>
6
7 #include <memory>
8 #include <set>
9 #include <string>
10
11 #include <vulkan.h>
12
13 #include <lug/System/Logger/Handler.hpp>
14 #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 count 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).

b. Scoping

1. 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:

```
1 namespace lug {  
2 namespace System {  
3 namespace Logger {  
4  
5 class Logger {  
6     // ...  
7 };  
8  
9 } // Logger  
10 } // System  
11 } // lug
```

Do not declare anything in the namespace `std`, and do not use inline namespace.

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.

Do not use inline namespaces, except for very, very specific use-cases.

2. 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.

3. 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.

4. Local Variables

Do not separate variable declaration from its initialization.

```
1 int x = 40; // Good
2 int y;
3
4 y = 2; // Bad (initialization separated from declaration)
```

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

5. 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.

c. Classes

1. Constructors

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

Example:

```
1 // ...
2 // ...
3
4 namespace lug {
5 namespace Graphics {
6
7 // ...
8
9 class LUG_GRAPHICS_API Camera : public Node {
10 // ...
11
12 protected:
13     Scene* _scene{nullptr};
14     RenderQueue _renderQueue;
15     RenderView* _renderView{nullptr};
16
17     float _fov{45.0f};
18     float _near{0.1f};
19     float _far{100.0f};
20
21 private:
22     // ...
23
24     Math::Mat4x4f _projMatrix{Math::Mat4x4f::identity()};
25     Math::Mat4x4f _viewMatrix{Math::Mat4x4f::identity()};
26
27     bool _needUpdateProj{true};
28     bool _needUpdateView{true};
29 };
30
31 } // Graphics
32 } // lug
```

2. 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 meaningful 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     // ...
16 };
17
18 } // Vulkan
19 } // Graphics
20 } // lug

```

3. 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](#).

```

1 namespace lug {
2 namespace Graphics {
3 namespace Vulkan {
4
5 class LUG_GRAPHICS_API Camera final : public lug::Graphics::Camera {
6 public:
7     Camera(const std::string& name);
8
9     Camera(const Camera&) = delete;
10    Camera(Camera&&) = default;
11
12    Camera& operator=(const Camera&) = delete;
13    Camera& operator=(Camera&&) = default;
14 };
15
16 } // Vulkan
17 } // Graphics
18 } // lug

```

4. Structs vs. Classes

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

5. 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 subclass 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;

```
1 namespace lug {
2 namespace Graphics {
3
4 class LUG_GRAPHICS_API Light : public MovableObject {
5 public:
6     virtual ~Light();
7     virtual void* getData(uint32_t& size) = 0;
8     // Virtual destructor and method because overridden in PointLight
9
10    // ...
11 };
12
13 class LUG_GRAPHICS_API PointLight final : public Light {
14     // Use final here because no class inherit from PointLight
15
16 public:
17     ~PointLight() override final;
18     void* getData(uint32_t& size) override final;
19     // Use final here because no subclass of PointLight will override getData() or the destructor
20     // Also use override to override getData and Light destructor
21
22     // ...
23 };
24
25 } // Graphics
26 } // 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.

6. Interfaces

All methods must be pure virtual (ends with = 0).

The interface must declare a virtual destructor.

```
1
2 class Foo {
3 public:
4     virtual ~Foo() = default;
5     // virtual destructor
6
7     virtual method() = 0;
8     // Pure virtual method
9 };
```

7. 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)

8. 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)
- Constructors and assignment operators
- Destructors
- Methods, including static methods
- Data properties

d. Functions

1. 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.

2. 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.

3. 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. Function Overloading

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

5. Default Arguments

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

6. 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).

e. Other

1. 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`.

f. Others C++ Features

1. Rvalue References

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

2. 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.

3. Exceptions

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

4. Run-Time Type Information (RTTI)

Avoid using Run Time Type Information (RTTI).

5. 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 (see `const`).
- 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. 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 `lug::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.

7. Preincrement and Predecrement

Always use the prefixed form.

8. 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;
```

9. Integer Types

Always use fixed-size integer types from `<stdint.h>` 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.

10. 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 maintainability of the codebase. Be smart, and keep them simple!

11. 0 and nullptr/NULL

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

12. sizeof

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

Example:

```
1 SomeType data;  
2 memset(&data, 0, sizeof(data)); // GOOD  
3 memset(&data, 0, sizeof(SomeType)); // BAD
```

13. 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.

14. Braced Initializer List

Prefer to use Braced Initializer List when possible.

15. Lambda expressions

Use lambda expressions when appropriate, e.g. to pass a short comparison 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.

16. Template metaprogramming

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

17. std::hash

Do not define specializations of std::hash.

18. C++14

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

19. Nonstandard Extensions

Only use standard extensions, exceptionnally 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 ;)

20. Aliases

g. Naming

1. General Naming Rules

2. 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 should not contain separators such as spaces, dashes or underscores.

Header files must head with the .hpp extension, inline header files must end 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:

```
1 .
2   src
3     Foo
4       Bar
5         MyClass.hpp
6         MyClass.inl
7     ...
8   include
9     Foo
10      Bar
11        MyClass.cpp
12    ...
```

3. Type Names

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

4. Variable Names

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

Private and protected class members should start with an underscore.

Example:

```
1 class Foo
2 {
3   // ...
4 public:
5     int barPublic;
6
7 protected:
8     int _barProtected;
9
10 private:
11     int _barPrivate;
12 };
```

5. Constant Names

Same as variable names

6. Function Names

Function names should 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. 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:

```
1 namespace lug {  
2 namespace std { // BAD  
3     // ...  
4 }  
5 }
```

8. Enumerator Names

See variable names.

9. Macro Names

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

h. Comments

1. Comment Style

Use // for single-line comments and /**/ for multiline comments outside of function blocks. Small blocks of multiline text can be written as multiple // lines, see an example [here](#).

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 // mutiline comment, but this is allowed // Good
```

2. 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.
7  */
8  class LUG_GRAPHICS_API Camera : public Node {
9      // ...
10 }
```

3. 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:

```
1 class LUG_GRAPHICS_API Graphics {
2     /**
3      * @brief Initializes the application with the informations filled in @p initInfo structure.
4      *
5      * @param[in] initInfo The initialize information.
6      *
7      * @return @p true if the initialization was successful.
8      */
9     bool init(const InitInfo& initInfo);
10 }
```

4. 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:

```
1 struct foo
2 {
3     int bar; ///< This is a data member comment.
4 };
```

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](#).

5. 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:

```
1 for (std::size_t i = 0; i < renderQueue.getLightsNb(); ++i) {
2
3     // Blend constants are used as dst blend factor
4     // Now the depth buffer is filled, we can set the blend constants to 1 to enable blending
5     if (i == 1) {
6         const float blendConstants[4] = {1.0f, 1.0f, 1.0f, 1.0f};
7         vkCmdSetBlendConstants(static_cast<VkCommandBuffer>(cmdBuffer), blendConstants);
8     }
9 }
```

Trivial code:

```
1 // Increment i // Bad
2 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:

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

6. 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.

7. TODO Comments

Add a TODO comment before code that is incomplete or need review.

The name of the person who wrote the TODO should appear inside parenthesis after the TODO, so we can ask them for more precisions.

Example:

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

i. Formatting

1. Line Length

A line should not have more than 120 characters.

2. 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"äôée"`.

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.

3. 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.

4. Function Declarations and Definitions

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

Example:

```
1 void Node::lookAt(const Math::Vec3f& targetPosition) {
2     // ...
3 }
```

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:

```
1 void Node::lookAt(
2     const Math::Vec3f& targetPosition, // 4 spaces indent
3     const Math::Vec3f& localDirectionVector,
4     const Math::Vec3f& up,
5     TransformSpace space) {
6     // ...
7 }
```

5. 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:

```
1 auto toUpper = [&foo](char c) {
2     return static_cast<char>(toupper(c));
3 };
```

6. Function Calls

Splitting arguments in function calls should respect the same rules as in [function declarations](#).

Example:

```
1 void main(int ac, char* av[]) {
2     // ...
3     callFooWhichIsALongFunctionAndTakesManyArguments(
4         andNo,
5         your,
6         functionNames,
7         should,
8         reallyNot,
9         beThatLong);
10 }
```

7. 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:

```
1 VkDescriptorSetLayoutCreateInfo createInfo{
2     createInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,
3     createInfo.pNext = nullptr,
4     createInfo.flags = 0,
5     createInfo.bindingCount = bindingCount,
6     createInfo.pBindings = layoutBindings
7 };
8
9 lug::Graphics::Vulkan::Image::Extent extent = {width, height};
```

8. 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:


```
1 if (condition) {
2     // 4 spaces indent
3     // ...
4 } else if (condition) {
5     // ...
6 } else {
7     // ...
8 }
```

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

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

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

```
1 if (!condition) // Bad
2     return false;
3
4 if (!condition) { // Good
5     return false;
6 }
```

9. 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:

```
1 for (uint32_t i = 0; i < 5; ++i) // Bad
2     std::cout << i << std::endl;
3
4 for (uint32_t i = 0; i < 5; ++i) { // Good
5     std::cout << i << std::endl;
6 }
```

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

Example:

```
1 switch (enumVal) {
2     case VK_SUCCESS: // 4 spaces indent
3         return "Success"; // 8 spaces indent
4     case VK_NOT_READY:
5         return "A fence or query has not yet completed";
6 }
```

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:

```
1 // Bad
2 switch (type) {
3     case Light::Type::Directional: return std::make_unique<Light::Directional>(name);
4     case Light::Type::Point:
5         return std::make_unique<Light::Point>(name);
6     case Light::Type::Spot: return std::make_unique<Light::Spot>(name);
7 }
```

```

7 }
8
9 // Good
10 switch (type) {
11     case Light::Type::Directional: return std::make_unique<Light::Directional>(name);
12     case Light::Type::Point: return std::make_unique<Light::Point>(name);
13     case Light::Type::Spot: return std::make_unique<Light::Spot>(name);
14 }

```

10. 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:

```

1 int foo;
2 int* bar; // Good
3 int * x; // Bad
4
5 bar = &foo;

```

11. 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:

```

1 if (!_pipelines[Light::Type::Directional] ||
2     !_pipelines[Light::Type::Point] || // 4 spaces indent
3     !_pipelines[Light::Type::Spot]) {
4     // ...
5 }

```

12. Return Values

The use of parenthesis around the return value is disallowed:

```

1 return (5); // Bad
2 return 5; // Good

```

The only exception is for complex expressions:

```

1 return (longExpressionA &&
2         longExpressionB);

```

13. Variable and Array Initialization

Prefer using `{}` than `()`.

There is no spaces around and inside the `{}` or `()`.

Example:

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

14. Preprocessor Directives

Preprocessor directives follow a separate indentation 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.

```
1 int main(int ac, char* av[]) {
2     uint16_t foo = 21;
3
4     #if defined(MACRO_A)
5         if (ac > 2) {
6             // Code run only if macro A
7             foo += 21;
8         }
9         #if defined(MACRO_B)
10            // Code run only if macro A and macro B
11            foo -= 42;
12        #endif
13    #endif
14
15    return foo;
16 }
```

15. 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:

```
1 Camera::Camera(const std::string& name) : Node(name) { // Short constructor
2     // ...
3 }
4
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
7     // ...
8 }
```

16. Vertical Whitespace

Try to limit the use of blank lines, but you can use them sparsely to split logically independent code sections and help readability.

Each file should end with a new line (`\n`).

j. 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!

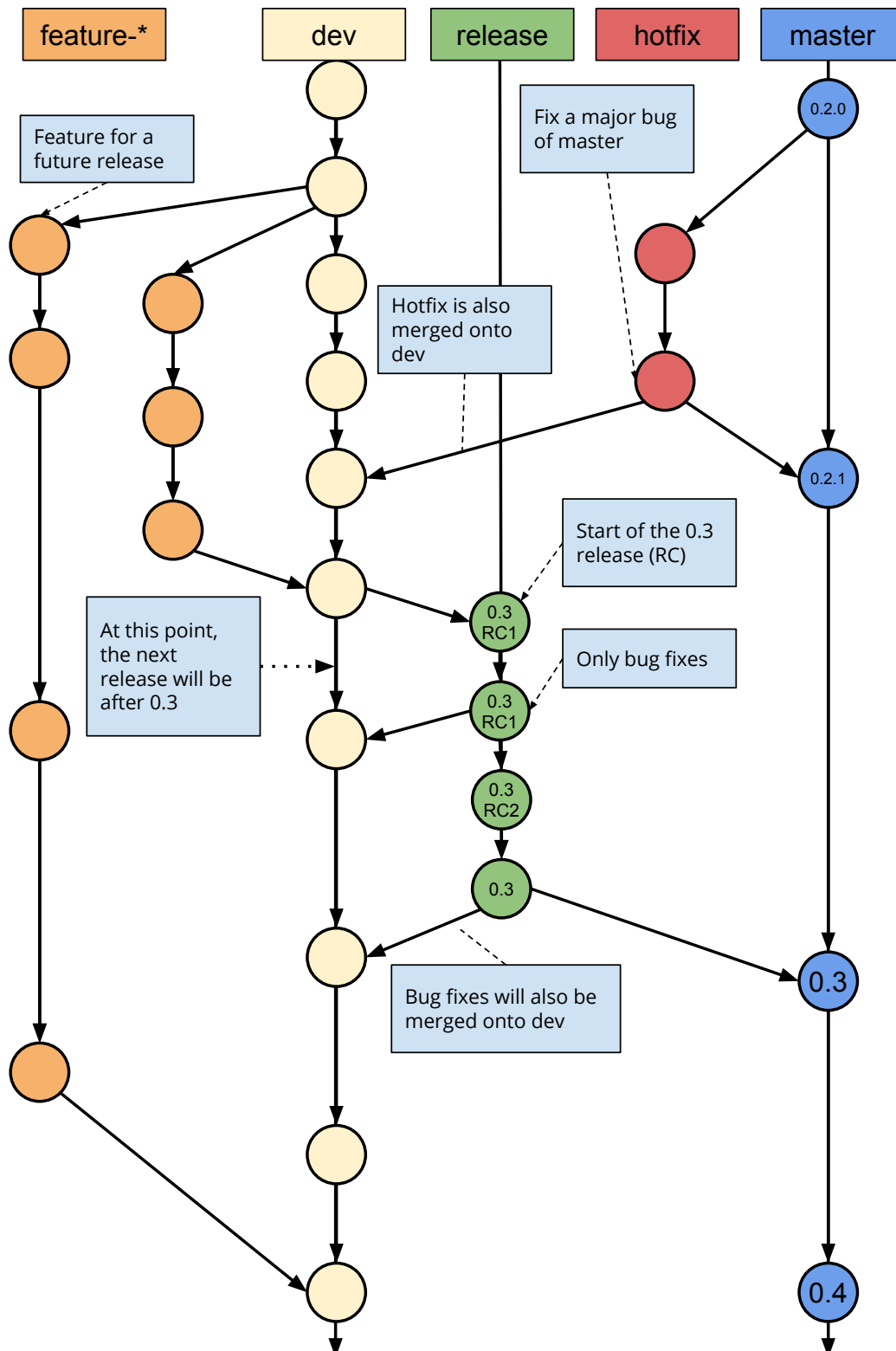
Contributing to Lugdunum

a. Branching strategy

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

- *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 is 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 committed 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 functionality 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 sufficiently 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 branches must be opened onto *dev* only.

Below is a quick schematic synthetizing all this information:





b. Tests

Each commit pushed on each branch is compiled and tested by [CircleCI](#) and [AppVeyor](#).

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!

Contact us

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

a. Github

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

b. Mailing list

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