lugdunum

3D engine

User 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

Quentin Buathier

Nicolas Comte

Guillaume Labey

Stuart Sulaski

Antoine Bolvy

Yoann Picquenot

Alexandre Quivy

Guillaume-Heritiana Sabatié

Yoann Long

Document summary

This document is intended for every potential Lugdunum and LugBench user.

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 a brief overview of the Lugdunum project as well as the user documentation, as a guide for using the Lugdunum 3D engine. In this guide, you will find detailed step by step instructions to build your first application with the Lugdunum 3D engine. Detailed examples with code will be provided to illustrate these steps.

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.

This first part 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. When you finish reading this chapter, you should be able to develop your own application using our engine with new Vulkan APIs, without having to suffer from its verbosity. We hope that this guide will be useful to you and serve as a quick reference in the future!

The second part of the document will help the user install a copy of the software on their computer, and teach the user how to use the website.

Document description

Title	:	Lugdunum User Documentation
Modification date	:	July 12, 2017
Accountant	:	Yoann Long
E-mail	:	lugdunum_2018@labeip.epitech.eu
Subjet	:	Lugdunum – User 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	Stuart Sulaski, Nicolas Comte, Corentin Chardeau, Yoann Long	1.1	Quickstart guide	Added quickstart guide
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. Added Lugbench

Contents

1 Lugdunum **1**

I. Quickstart . . . 2

1. Building a project . . . 2
2. Creating a window . . . 3
3. Handling events . . . 4
   - 3.a. Window events . . . 5
   - 3.b. Keyboard events . . . 5
   - 3.c. Mouse events . . . 6
4. Logger . . . 7
   - 4.a. Initialization . . . 7
   - 4.b. Display a message Log . . . 7
   - 4.c. Modules . . . 8
   - 4.d. Choosing a device . . . 8
5. Using lug::Core::Application . . . 9
   - 5.a. Camera . . . 11
   - 5.b. Movable Camera . . . 12
   - 5.c. Lights . . . 12
   - 5.d. Handling Time . . . 13

II. Contact us . . . 13

1. Github . . . 13
2. Mailing list . . . 14

2 Lugbench **15**

I. The LugBench Desktop Application . . . 16

1. Installing . . . 16
   - 1.a. Arch Linux . . . 16
   - 1.b. Android . . . 16
2. How to use . . . 16

II. LugBench website . . . 16

1. How to use . . . 16

Part. 1

Lugdunum

Contents

I. Quickstart . . . 2
1. Building a project . . . 2
2. Creating a window . . . 3
3. Handling events . . . 4
4. Logger . . . 7
5. Using lug::Core::Application . . . 9

II. Contact us . . . 13
1. Github . . . 13
2. Mailing list . . . 14

I. Quickstart

Welcome to Lugdunum's quickstart tutorial! Here we will show you through a number of examples how to use our 3D engine for your own projects.

As a reminder, please note that classes mentioned here are linked (in blue) to our external Documentation[1].

We wish you a good reading!

1. Building a project

To build the project with Lugdunum, you need a CMakeLists.txt like the following:

```
cmake_minimum_required(VERSION 3.1)

# project name
project(test_project)

# define the executable
set(EXECUTABLE_NAME "test_project_executable")

# find vulkan
find_package(Vulkan)

# Check only VULKAN_INCLUDE_DIR because the vulkan library is loaded at runtime
if (NOT VULKAN_INCLUDE_DIR)
   message(FATAL_ERROR "Can't find vulkan headers")
endif()

include_directories(${VULKAN_INCLUDE_DIR})

# find fmt
find_package(Fmt)

if (NOT FMT_INCLUDE_DIR)
   if (NOT EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/../../thirdparty/fmt")
      message(FATAL_ERROR "Can't find fmt, call 'git submodule update --recursive'")
   endif()

   set(FMT_INCLUDE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/../../thirdparty/fmt/include)
   message(STATUS "Found Fmt: ${FMT_INCLUDE_DIR}")
endif()

```

[1]external Documentation: https://lugdunum3d.github.io/doc/doxygen/index.html

```
include_directories(${FMT_INCLUDE_DIR})

# find Lugdunum
find_package(LUG REQUIRED ${THIS_DEPENDS})

# source files of your application
set(SRC
   src/Application.cpp
   src/main.cpp
)

# include files of your application
set(INC
   include/Application.hpp
)

# create target
if(LUG_OS_ANDROID)
   add_library(${EXECUTABLE_NAME} SHARED ${SRC} ${INC})

   set(ANDROID_PROJECT_PATH ${CMAKE_SOURCE_DIR}/android/${EXECUTABLE_NAME})
   set(ANDROID_PROJECT_ASSETS ${ANDROID_PROJECT_PATH}/src/main/assets)
   set(ANDROID_PROJECT_SHADERS ${ANDROID_PROJECT_PATH}/src/main/shaders)
else()
   add_executable(${EXECUTABLE_NAME} ${SRC} ${INC})
endif()

lug_add_compile_options(${EXECUTABLE_NAME})

# use lugdunum
include_directories(${LUG_INCLUDE_DIR})
target_link_libraries(${EXECUTABLE_NAME} ${LUG_LIBRARIES})
```

2. Creating a window

Let’s see how simple it is to create a window with the Lugdunum Engine.

```
int main() {
   auto window = lug::Window::Window::create(800, 600, "Default Window", lug::Window::Style::
       Default);

   // ...

```

```
    return 0;
}
```

The first and second argument are the size of the window's width and height of the window.
The third argument is the name you wish to give to your application.
The fourth argument is the style of the window. It allows you to choose which decorations and features you wish to enable. You can use any combination of the following styles:

Style Flag	Description
`lug::Window::Style::None`	No decoration at all
`lug::Window::Style::Titlebar`	The window has a titlebar
`lug::Window::Style::Resize`	The window can be resized and has a maximize button
`lug::Window::Style::Close`	The window has a close button
`lug::Window::Style::Fullscreen`	The window is shown in fullscreen mode
`lug::Window::Style::Default`	It is a shortcut for `Titlebar | Resize | Close`

3. Handling events

Now that the window is created you may want to handle events.
To do this, it is as simple as looping while the window is open and retrieving events that have been received.

```
int main() {
    auto window = lug::Window::Window::create(800, 600, "Default Window", lug::Window::Style::
        Default);

    while (window->isOpen()) {
        lug::Window::Event event;
        while (window->pollEvent(event)) {

            // ...
        }
        // ...
    }
}
```

```
while (window->isOpen()) {

}
```

This line ensures that our application keeps running while our window is open. If the window's state ever changes then our application ends the loop and ends.

```cpp
lug::Window::Event event;
while (window->pollEvent(event)) {

}
```

To retrieve events we need a `Event struct` that we pass to `window—>pollEvent(...)`.
Each time `pollEvent(...)` is called, the window returns the next available `Event` and discards it from its queue.
If there are no more events left to be handled, then it returns `false`.

3.a. Window events

Here is a simple example where the user detects a `lug::Window::Event::Type::Close` events and then call the window's `window—>close()` function which ends our application by exiting the while loop.

Note: Even if you do not care about events, you still need an event loop to ensure that the window works as intended.

```cpp
int main() {
   auto window = lug::Window::Window::create(800, 600, "Default Window", lug::Window::Style::
       Default);

   while (window->isOpen()) {
      lug::Window::Event event;
      while (window->pollEvent(event)) {

         if (event.type == lug::Window::Event::Type::Close) {
            window->close();
         }

      }
   }
}
```

3.b. Keyboard events

This is pretty much the same example but in addition to detecting window events, the user also detects a keyboard event which leads to the same result.

```cpp
int main() {
   auto window = lug::Window::Window::create(800, 600, "Default Window", lug::Window::Style::
       Default);

```

```cpp
    while (window->isOpen()) {
        lug::Window::Event event;
        while (window->pollEvent(event)) {

            if (event.type == lug::Window::Event::Type::Close) {
                window->close();
            }

            if (event.type == lug::Window::Event::Type::KeyPressed && event.key.code == lug::
                Window::Keyboard::Key::Escape) {
                window->close();
            }

        }
    }
}
```

3.c. Mouse events

Finally here is an example that includes handling mouse events.
In this example, clicking the left mouse button does not do anything, but we are sure you will be able to come up with some creative ways to use this!

```cpp
int main() {
    auto window = lug::Window::Window::create(800, 600, "Default Window", lug::Window::Style::
        Default);

    while (window->isOpen()) {
        lug::Window::Event event;
        while (window->pollEvent(event)) {

            if (event.type == lug::Window::Event::Type::Close) {
                window->close();
            }

            if (event.type == lug::Window::Event::Type::KeyPressed && event.key.code == lug::
                Window::Keyboard::Key::Escape) {
                window->close();
            }

            if (event.type == lug::Window::Event::Type::ButtonPressed && event.button.code ==
                lug::Window::Mouse::Button::Left) {
            // ...
            }
        }
    }
}
```

4. Logger

4.a. Initialization

First, you need to initialize an instance of Logger, with the name of the instance (you can choose any string you want here: it allows you to know where the logs come from in the future).

```
lug::System::Logger::makeLogger("myLogger");
```

When you have an instance of the logger, you have to attach a handler for each output to which you want to log. For example, if you need the standard output:

```
LUG_LOG.addHandler(lug::System::Logger::makeHandler<lug::System::Logger::StdoutHandler>("
    stdout"));
```

If you need to log on Android device:

```
LUG_LOG.addHandler(lug::System::Logger::makeHandler<lug::System::Logger::LogCatHandler>("
    logcat"));
```

Below is a table that describes what handlers are available to you and in which circumstances you may use them.

Class	Description
LogCatHandler	Android
StdoutHandler	Standard output
StderrHandler	Error output
FileHandler	File

4.b. Display a message Log

To use the logger, you have to use one of this following methods:

Display message type	Example usage
`debug`	`LUG_LOG.debug("Debug info: {}", debugValue);`
`info`	`LUG_LOG.info("Starting the app");`
`warn` (warning)	`LUG_LOG.warn("Something wrong happened");`

Display message type	Example usage
`error`	`LUG_LOG.error("An error occured");`
`fatal` (fatal error)	`LUG_LOG.fatal("Fatal error");`
`assrt` (assert)	`LUG_LOG.assrt("Assert level logging");`

4.c. Modules

The Vulkan API defines many different features (more information on the Khronos group website[2]) that must be activated at the creation of the device.

Lugdunum abstracts these features with the use of *modules*.

Modules are a bunch of pre-defined set of mandatory features required by the renderer to run specific tasks.

For example, if you want to use tessellation shaders, you have to specify the module `tessellation` in the structure `InitInfo` during the initialization phase.

You can specify two type of modules: `mandatoryModules` and `optionalModules`. The former ensures the initialization fails if any mandatory module is not supported by the device, whereas the later treats them as optionals, as the name implies.

Note: You can query which optional modules are active with lug::Graphics::getLoadedOptionalModules()

4.d. Choosing a device

Usually, calling `Application::init()`, lets Lugdunum decide which device to use. The engine prioritises a discrete device that supports all the mandatory modules. If the engine is unable to find such a device, the call fails.

Alternatively the user can decide which device he or she wishes to use, as long as the device meets the minimum requirements for all the mandatory modules. For that, you can use two others methods named `Application::beginInit(int argc, char *argv[])` and `Application::finishInit()`.

Between the two you can set what we call *preferences*, and choose the device that you want that way.

```cpp
if (!lug::Core::Application::beginInit(argc, argv)) {
   return false;
}

lug::Graphics::Renderer* renderer = _graphics.getRenderer();
lug::Graphics::Vulkan::Renderer* vkRender = static_cast<lug::Graphics::Vulkan::Renderer*>(
    renderer);

```

[2]on the Khronos group website: https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html#features-features

```cpp
auto& chosenDevice : vkRender->getPhysicalDeviceInfos();

for (auto& chosenDevice : vkRender->getPhysicalDeviceInfos()) {
   if (...)
   {
      vkRender->getPreferences().device = chosenDevice;
      break;
   }
}

if (!lug::Core::Application::finishInit()) {
   return false;
}
```

In the example shown above the user retrieves the list of all available device, decides which one they wishes to use and then sets that one as the device to use by default, setting it with the line `vkRender—>getPreferences().device = chosenDevice;`.
If the device does not meet the minimum requirements `Application::finishInit()` fails.

5. Using lug::Core::Application

In order to simplify the use of Lugdunum, the engine provides an abstract class named `Application` that helps you get started quicker.

Note: The use of this class is not mandatory but strongly recommended

The first thing to do is to create your own class that inherits from it:

```cpp
class Application : public ::lug::Core::Application {
   // ...
}
```

As `Application` is an abstract class, you have to override two methods as well as the destructor:

```cpp
void onEvent(const lug::Window::Event& event) override final;
void onFrame(const lug::System::Time& elapsedTime) override final;
~Application() override final;
```

Each time an event is triggered by the system, the method `onEvent()` is called. In this *callback* you can check the `event.type` which is referenced in the enum `Window::Event::Type`

```cpp
void Application::onEvent(const lug::Window::Event& event) {
   if (event.type == lug::Window::Event::Type::Close) {
      // ...
   }
}
```

The second method to override is `Application::onFrame()`. This method is called each loop, and you can do whatever pleases you in it, e.g. your application's logic. The `elapsedTime` variable contains the elapsed time since the last call to `onFrame()`.

```
void Application::onFrame(const lug::System::Time& elapsedTime) {
   _rotation += (0.05f * elapsedTime.getMilliseconds<float>());

   float x = 20.0f * cos(lug::Math::Geometry::radians(_rotation));
   float y = 20.0f * sin(lug::Math::Geometry::radians(_rotation));

   if (_rotation > 360.0f) {
      _rotation -= 360.0f;
   }

   auto& renderViews = _graphics.getRenderer()->getWindow()->getRenderViews();

   for (int i = 0; i < 2; ++i) {
      renderViews[i]->getCamera()->setPosition({x, -10.0f, y}, lug::Graphics::Node::
          TransformSpace::World);
      renderViews[i]->getCamera()->lookAt({0.0f, 0.0f, 0.0f}, {0.0f, 1.0f, 0.0f}, lug::
          Graphics::Node::TransformSpace::World);
   }
}
```

The constructor of `Application` takes a structure `Info` defined as follow:

```
struct Info {
   const char* name;
   Version version;
};
```

You can use the initializer list to make it easier:

```
lug::Core::Application::Application{{"triangle", {0, 1, 0}}} {
   // ...
}
```

After the constructor phase, you have to call `Application::init(argc, argv)`. This method processes two main initialization steps.

First, it will initialize `lug::Graphics::Graphics _graphics;` with these default values:

```
lug::Graphics::Graphics::InitInfo _graphicsInitInfo{
   lug::Graphics::Renderer::Type::Vulkan, // type
   { // rendererInitInfo
      "shaders/" // shaders root
   },
   { // mandatoryModules
      lug::Graphics::Module::Type::Core
   },
   {}, // optionalModules
};
```

- `Renderer::Type` is set to `Vulkan` by default, as it the only supported renderer at this moment.
- `Module::Type::Core` defines basic default requirements for Vulkan (see Modules)

Finally, it will initialize `lug::Graphics::Render::Window* _window{nullptr};` as `Application` manages itself all the window creation. It uses these default values:

```
lug::Graphics::Render::Window::InitInfo _renderWindowInitInfo{
   { // windowInitInfo
      800, // width
      600, // height
      "Lugdunum - Default title", // title
      lug::Window::Style::Default // style
   },

   {} // renderViewsInitInfo
};
```

You can also manually change these values:

```
getRenderWindowInfo().windowInitInfo.title = "Foo Bar";
```

Warning: You have to change these value before calling `Application::init(argc, argv)`

5.a. Camera

You use `createCamera()` to create a camera and give it a name.
A `Camera` has the following attributes which can be changed via getters and setters. They essentially constitute the frustrum of the camera:

Attributes	Description
Fov	Field of view
Near	Near plane

Attributes	Description
Far	Far plane
ViewMatrix	View matrix (computed from the previous attributes)
ProjectionMatrix	Projection Matrix (computed from the previous attributes)

```
// Create a camera
std::unique_ptr<lug::Graphics::Render::Camera> camera = _graphics.createCamera("camera");
```

5.b. Movable Camera

Once the camera is created, you have to create a node from the scene in order to obtain a movable camera with a position.

```
// Add camera to scene
{
   std::unique_ptr<lug::Graphics::Scene::MovableCamera> movableCamera = _scene->
       createMovableCamera("movable camera", camera.get());
   _scene->getRoot()->createSceneNode("movable camera node", std::move(movableCamera));
}
```

5.c. Lights

Our 3D engine has three different types of light:

Types of light	Description
Directional	Light that is being emitted from a source that is infinitely far away. All shadows cast by this light are parallel, an ideal choice for simulating sunlight.
Point	Emits light from a single point, as a real-life bulb. It emits in all directions.
Spotlight	Emits light in a cone shape, as a flashlight, or a stage light.

Let us assume that you want to set a `Directional` light, here is a sample:

```
{
   std::unique_ptr<lug::Graphics::Light> light = _scene->createLight("light", lug::Graphics::
       Light::Light::Type::Directional);
   std::unique_ptr<lug::Graphics::Scene::Node> lightNode = _scene->createSceneNode("light
       node");

   light->setDiffuse({1.0f, 1.0f, 1.0f});
   static_cast<lug::Graphics::Light::Light*>(light.get())->setDirection({0.0f, 4.0f, 5.0f});
```

```

    lightNode->attachMovableObject(std::move(light));
    _scene->getRoot()->attachChild(std::move(lightNode));
}
```

Here is what you need to do, step by step:

1. First create the light of type `Graphics::Light`, and give as first parameter of `createLight()` the name of the light, and as second parameter the type of light (cf. array above).
2. Then create a `lug::Scene::Node` which will be the movable object in the scene and attach the light to it so you can move the light in the scene.
3. Now you can set the diffusion of the light.
4. Next set the direction of the light. Be aware that you have to `static_cast<>` the pointer with the good type of light *(in this case `lug::Graphics::Light::Directional*`)*
5. Finally attach the light to the scene.

5.d. Handling Time

With Lugdunum, the time is in microseconds, and it is stored in a `int64_t`.
The `Time` class represents a time period, the time that elapses between two event.

A Time value can be constructed from a microseconds source.

```
lug::System::Time time(10000);
```

You can get the time in different formats.

```
int64_t timeInMicroseconds = time.getMicroseconds();
float timeInMilliseconds = time.getMilliseconds();
float timeInSeconds = time.getSeconds();
```

II. 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[3], and report specific problems or questions directly by filing a new issue.

[3]Lugdunum3D: https://github.com/Lugdunum3D

2. Mailing list

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

Part. 2

Lugbench

Contents

I. The LugBench Desktop Application 16
  1. Installing 16
  2. How to use 16
II. LugBench website 16
  1. How to use 16

I. The LugBench Desktop Application

1. Installing

In order to simplify the process of getting Lugbench, precompiled binaries are available on the LugBench website[1].

However depending on you operating system, you can:

1.a. Arch Linux

```
pacaur -S lugbench
```

1.b. Android

Search and install `LugBench: Vulkan Benchmarking` on the Play Store.

2. How to use

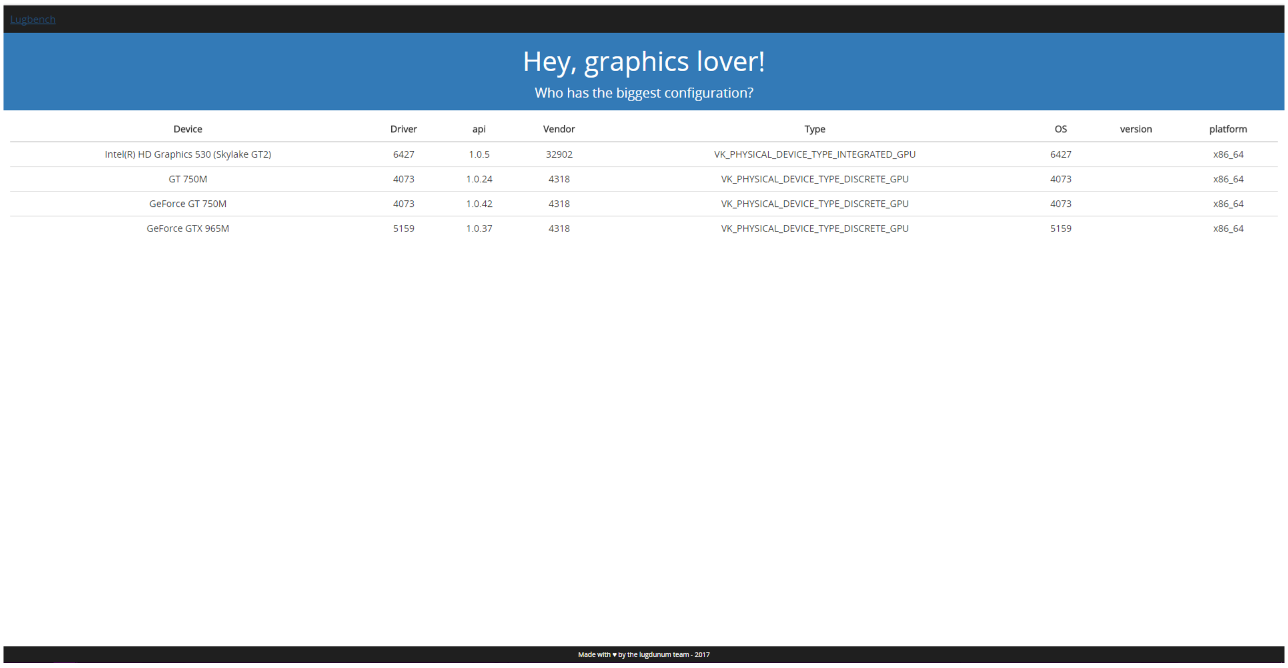
At this moment, the lugbench application only query vulkan hardware capability and send them to the server. The application doesn't start any 3D scene yet, therefore no GUI is yet to be seen from the user.

II. LugBench website

1. How to use

First of all, you want to open your favorite browser. Next, you can go on our website[2].

[1]LugBench website: `http://163.5.84.217/`
[2]on our website: `http://163.5.84.217/`



Hey, graphics lover!

Who has the biggest configuration?

Device	Driver	api	Vendor	Type	OS	version	platform
Intel(R) HD Graphics 530 (Skylake GT2)	6427	1.0.5	32902	VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU	6427		x86_64
GT 750M	4073	1.0.24	4318	VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU	4073		x86_64
GeForce GT 750M	4073	1.0.42	4318	VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU	4073		x86_64
GeForce GTX 965M	5159	1.0.37	4318	VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU	5159		x86_64

Made with ♥ by the lugdunum team - 2017

Figure 2.1: Front page of the LugBench website

You can access the list of all the lugbench user's configurations, and sort it by any field you want.

Don't worry about the minimalist aspect of the website. For now, it is still a work in progress. If you want, feel free to open an issue to present any idea you may have about a new cool feature. We are open to any suggestion but refer to us before start developing it as we may work on it as well.