lugdunum
game engine

User Documentation n°1

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 user.

In this document you will find a brief overview of the Lugdunum project as well as the user documentation 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 basic 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 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!

Document description

Title	:	[2018][UD1] Lugdunum
Modification date	:	March 12, 2017
Accountant	:	Yoann Long
E-mail	:	lugdunum_2018@labeip.epitech.eu
Subjet	:	Lugdunum – User Documentation n°1
Document version	:	1.1

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	Stuart Sulaski, Nicolas Comte, Corentin Chardeau, Yoann Long	1.1	Quickstart guide	Added quickstart guide

Contents

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.

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 mentionned here are linked (in blue) to our external documentation:

External Documentation

We wish you a good reading!

a. 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

b. Handling events

Now that the window is created you may want to handle events.
To do this it's 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 will keep running while our window is still open. If the window's state ever changes then our application ends the loop and ends.

```
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 will return the next available Event and the discard it from it's queue.
If there are no more events left to be handled then it returns false.

1. 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 don't care about events, be they window, keyboard or mouse related. You still need an event loop to ensure that the window functions as it should.

```
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();
            }

        }
    }
}
```

2. Keyboard events

Now 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.

```
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();
        }

      }
    }
}
```

3. Mouse events

Finally here is an example that includes handling mouse events.
In this case clicking with the left mouse button won't do anything but I'm sure you all will be able to come up with ways to use this.

```
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) {
            // ...
            }
        }
    }
}
```

c. Logger

1. initialization

First, you need to initialize an instance of Logger, with the name of the instant (you can choose what you want here, but it will be useful to know where the logs come from afterwards).

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

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

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

or if you need to log on Android device:

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

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

2. Display a message Log

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

```
LUG_LOG.info("Starting the app!");`
```

Display message type
debug
info
warn (warning)
error
fatal (fatal error)
assert (assert)

3. Modules

The Vulkan API defines many different features (more info here) 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 in order 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 will fail the initialization if any mandatory module is not supported by the device, whereas the later won't.

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

4. Choosing a device

Instead of calling Application::init(), which will let Lugdunum decide which device to use. The engine will prioritise a discrete device that supports all the mandatory modules. If the engine is unable to find a device with one of each then it will simply fail.

Alternatively the user can decide which device he 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);

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 seen above the user retrieves the list of all available device, decides which one he wishes to use and then sets that one as the device to use by default setting it with this line vkRender−>getPreferences().device = chosenDevice;.
If the device doesn't meet the minimum requirements [Application::finishInit()](#lug::Core::Application ::finishInit()) will fail.

d. Using lug::Core::Application

In order to simplify the use of Lugdunum, we are providing an abstract class called Application that help you to 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;
```

As I just mentioned, you have to override the Application::onEvent() method. Each time an event is triggered by the system, the method onEvent() is called. Therefore you can check the event.type which is referenced in the enum lug::Window::EventType::

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

The second method to override is Application::onFrame(). This method is called each loop, and you can do whatever you want in it. The elapsedTime variable contain the elapsed time since the last call to onFrame().

```cpp
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:

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

You can use the initializer list to make it easier:

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

After the constructor phase, you have to call Application::init(argc, argv))

This method will process two main initialization steps.

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

```cpp
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 change these value:

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

Note: Obviously, you have to change these value before calling Application::init(argc, argv)

1. Camera

You use createCamera() to create a camera and give it a name.
A camera has these attributes which can be changed via Getter/Setter:

Attributes
FoV
Near
Far
ViewMatrix
ProjectionMatrix

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

2. 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));
}
```

3. Lights

Our 3D engine has three different types of light:

Types of light
Directional
Point
Spotlight

Let's say that you want to set a lug::Light::Type::DirectionalLight, here is a sample:

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

    light->setDiffuse({1.0f, 1.0f, 1.0f});
    static_cast<lug::Graphics::DirectionalLight*>(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::SceneNode 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::Graphic::DirectionalLight*>)*
5. Finally attach the light to the scene.

4. Handling Time

With Lugdunum, the time is in microseconds. 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();
```

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`.