osre_doc

Kim Kulling

Apr 08, 2022

CONTENTS:

1 Introduction **1**
- 1.1 My idea . . . 1

2 Build **3**
- 2.1 Installation Prerequisites . . . 3
- 2.2 Build for Windows . . . 3
- 2.3 Build for Linux . . . 4

3 Write your first Hello-World-App **5**
- 3.1 Prepare a workspace . . . 5
- 3.2 Walkthrough . . . 7

4 The Platform Abstraction Layer **9**

5 The Event-System **11**
- 5.1 Introduction . . . 11
- 5.2 Using Event-Handler . . . 11
- 5.3 Engine-Messaging . . . 11

6 The Render System **13**
- 6.1 Multithreaded rendering - The idea . . . 13
- 6.2 The Render-Graph . . . 13
- 6.3 Render-Backend-Service . . . 14
- 6.4 Supported Render-API's . . . 14

7 The Scene **15**
- 7.1 The hierarchical Representation . . . 15
- 7.2 Nodes and more . . . 15
- 7.3 Culling and Picking . . . 15

8 Indices and tables **17**

CHAPTER

ONE

INTRODUCTION

1.1 My idea

Since 1986 I am interested in computer graphics. I started with simple stuff on C16 and lateron on an Anmiga, went to PC's and wrote my first 3D-Program based on the Painter's-Algorithm in 1996 (see https://github.com/kimkulling/osre_doc/edit/main/source/Introduction.rst).

In 2003 I help the ZFX-Team with the ZFX-Community-Engine (more info's are here, only german: https://de-academic.com/dic.nsf/dewiki/1547500).

Everytime when I had worked on the engine-stuff I realized that I do not have any plan how to do all the interesting stuff. I missed the deeper knowledge which is most importand for writing 3D-Software, which can b used by other people. So I decided to restart from scratch and started to work on the OSRE: Just another Open-Source-Render-Engine. This is just a side-project for me. I am trying to implement things like

- Writing my own multithreaded renderer
- How shall a multi-platform layer looks like
- Learn OpenGL and Vulkan
- Implement an asset-loader for assimp-based models
- Doing lightning right
- Implement a simple bitmap-font renderer
- Render UI-elements right

and many more. And here is my place to play around with these things.

CHAPTER

TWO

BUILD

2.1 Installation Prerequisites

At first you have to install the following packages:

- git
- CMake Version 3.10 or higher
- For Windows: Visual-Studio 2017 or 2019
- For Linux: gcc or clang

2.2 Build for Windows

- Open a command-prompt
- Checkout the code via:

```
> git clone https://github.com/kimkulling/osre.git
```

- Navigate into your folder which contains the OSRE-Repository
- We are using a cmake-based build to generate the build files. to generate your project-files for your Build Environment via:

```
> cmake CMakeLists.txt
```

  - If you want to use a different environment like eclipe or CLion you can generate them as well: > cmake CMakeLists.txt -G <Your IDE>
  - You can get the list of generators via: > cmake –help
- If you have select the VS-Generator open the solution osre.sln with Visual-Studio or build OSRE via:

```
> cmake --build .
```

- You will find the samples and tests at:

```
> osre\bin\Debug
```

- To run the samples you have to copy the dlls from SDL2 and assimp into your bin-folder:

```
> cd osre\bin\debug
> copy ..\..\contrib\assimp\bin\debug\*dll .
> copy <SDL2-Folder>\libs\x64\*dll .
```

  - This is an open issue which will get fixed as soon as possible.

2.3 Build for Linux

- Open a terminal
- Checkout the code via:

```
> git clone https://github.com/kimkulling/osre.git
```

- Go into the folder of osre
- Open the solution osre.sln with the VS or build OSRE via:

```
> cmake --build .
```

- You will find the samples and tests at:

```
> ose/bin
```

CHAPTER

THREE

WRITE YOUR FIRST HELLO-WORLD-APP

3.1 Prepare a workspace

To get started you need to create a folder for your app andd add a CMakeLists.txt file into it:

```
INCLUDE_DIRECTORIES(
  ${PROJECT_SOURCE_DIR}
  ../
)

 SET ( 00_helloworld_src
   00_HelloWorld/HelloWorld.cpp
   00_HelloWorld/README.md
 )

 ADD_EXECUTABLE( HelloWorld
   ${00_helloworld_src}
 )

 link_directories(
   ${CMAKE_CURRENT_SOURCE_DIR}/../../ThirdParty/glew/Debug
   ${CMAKE_CURRENT_SOURCE_DIR}/../../ThirdParty/glew/Release
 )

 target_link_libraries ( HelloWorld   osre )
```

3.1.1 Your first Hello-World-Application

Here the code:

```
#include <osre/App/App.h>
#include <osre/Common/Logger.h>
#include <osre/RenderBackend/RenderBackendService.h>
#include <osre/Scene/MeshBuilder.h>
#include <osre/Scene/Camera.h>

using namespace ::OSRE;
using namespace ::OSRE::App;
using namespace ::OSRE::RenderBackend;
```

(continues on next page)

(continued from previous page)

```
// To identify local log entries we will define this tag.
static const c8 *Tag = "HelloWorldApp";

///
/// The example application, will create the render environment and render a simple␣
↪triangle onto it
///
class HelloWorldApp : public AppBase {
    /// The transform block, contains the model-, view- and projection-matrix
    TransformMatrixBlock m_transformMatrix;
    /// The entity to render
    Entity *mEntity;
    /// The keyboard controller to rotate the triangle
    Scene::AnimationControllerBase *mKeyboardTransCtrl;

public:
    /// The class constructor with the incoming arguments from the command line.
    HelloWorldApp(int argc, char *argv[]) :
            AppBase(argc, (const char **)argv),
            m_transformMatrix(),
            mEntity(nullptr),
            mKeyboardTransCtrl(nullptr) {
        // empty
    }

    /// The class destructor.
    ~HelloWorldApp() override {
        // empty
    }

protected:
    bool onCreate() override {
        if (!AppBase::onCreate()) {
            osre_error(Tag, "Error while creating application basics.");
            return false;
        }

        AppBase::setWindowsTitle("Hello-World sample! Rotate with keyboard: w, a, s, d,␣
↪scroll with q, e");
        World *world = getActiveWorld();
        mEntity = new Entity("entity", *AppBase::getIdContainer(), world);
        Scene::Camera *camera = world->addCamera("camera_1");
        ui32 w, h;
        AppBase::getResolution(w, h);
        camera->setProjectionParameters(60.f, (f32)w, (f32)h, 0.001f, 1000.f);

        Scene::MeshBuilder meshBuilder;
        RenderBackend::Mesh *mesh = meshBuilder.allocTriangles(VertexType::ColorVertex,␣
↪BufferAccessType::ReadOnly).getMesh();
        if (nullptr != mesh) {
            mEntity->addStaticMesh(mesh);
```

(continues on next page)

(continued from previous page)

```
            world->addEntity(mEntity);
            camera->observeBoundingBox(mEntity->getAABB());
        }
        mKeyboardTransCtrl =␣
↪AppBase::getTransformController(DefaultControllerType::KeyboardCtrl, m_
↪transformMatrix);

        return true;
    }

    void onUpdate() override {
        RenderBackendService *rbSrv = getRenderBackendService();
        mKeyboardTransCtrl->update(rbSrv);

        rbSrv->beginPass(PipelinePass::getPassNameById(RenderPassId));
        rbSrv->beginRenderBatch("b1");

        rbSrv->setMatrix(MatrixType::Model, m_transformMatrix.m_model);

        rbSrv->endRenderBatch();
        rbSrv->endPass();

        AppBase::onUpdate();
    }
};

/// Helper function to generate the main function.
OSRE_MAIN(HelloWorldApp)
```

3.2 Walkthrough

Now let's take a deeper look what is going on in the code. We need to include some basic stuff for our first render-experiment.

3.2.1 Lets start with the headers:

```
#include <osre/App/App.h>
#include <osre/Common/Logger.h>
#include <osre/RenderBackend/RenderBackendService.h>
#include <osre/Scene/MeshBuilder.h>
#include <osre/Scene/Camera.h>
```

To initialize the OSRE-rendersystem you can use the AppBase class. By including **App/App.h** the class and all dependencies will be included.

To make your application more verbose we want to log some messages. This is the reason to use the **Common/Logger.h** header.

We want to render a single triangle. The **RenderBackend/RenderBackendService.h** will provide the API to add triangles. **Scene/MeshBuilder.h** offers you a simple way to create triangle. So we need this include as well.

And we want to look onto the scene. **Scene/Camera.h** provides an inferface for that.

3.2.2 Define your own application class

```
class HelloWorldApp : public AppBase {
    /// The transform block, contains the model-, view- and projection-matrix
    TransformMatrixBlock m_transformMatrix;
    /// The entity to render
    Entity *mEntity;
    /// The keyboard controller to rotate the triangle
    Scene::AnimationControllerBase *mKeyboardTransCtrl;

public:
    /// The class constructor with the incoming arguments from the command line.
    HelloWorldApp(int argc, char *argv[]) :
            AppBase(argc, (const char **)argv),
            m_transformMatrix(),
            mEntity(nullptr),
            mKeyboardTransCtrl(nullptr) {
        // empty
    }

    /// The class destructor.
    ~HelloWorldApp() override {
        // empty
    }
...
```

We want to create a triangle. And we want to rotate this with our keyboard. So we need an attribute from the type **TransformMatrixBlock** . This class provides a simple API to create a matrix with translation, scaling and rotating.

To manage the triangle in our scene we need an attribute from type **Entity**.

And last but not least we need an attribute to accss the keyboard input and control the animation of our triangle from the type **Scene::AnimationControllerBase**.

CHAPTER

FOUR

THE PLATFORM ABSTRACTION LAYER

CHAPTER

FIVE

THE EVENT-SYSTEM

5.1 Introduction

5.2 Using Event-Handler

5.3 Engine-Messaging

CHAPTER

SIX

THE RENDER SYSTEM

6.1 Multithreaded rendering - The idea

In classic engines the rendering and the game logic will be done in the same thread: the main thread. So all the logic must share the time to get their tasks done like:

- Input handling
- User interaction
- Scene-updates
- And, last but to least, the rendering itself

So when you want to get a frame rate from 60 you have to work in a timeframe from 1/60 -> 1.67ms. Not too much. To encouble this a little bit I the OSRE-Rendering will be done in a separate render-thread. Each update will be done one frame before. This helps to get the render logic encapsulated from the rest and get more resources for a smooth render experience. Of course new render-API's will be able to instrument multible threads for the rendering. To implement this logic a separate render-thread is an advantage as well. There is only one place where you have to look at. All the rendering will be managed in a separate task. The main-thread can communicate with the back-end aka the render-thread about the Render-system.

6.2 The Render-Graph

The rendering is managed by a render-graph: - In each frame all the passes were iterated - For each pass all the render-batches will be iterated

- A batch iteration will set the uniform parameter
- A batch iteration will set the material
- A batch iteration will do all render calls.

It look like:

6.3 Render-Backend-Service

We have a class called the RenderBackend-Service which is the fascade for the user to the render-backend. If you want to create a render window or you want to add a new mesh to the scene you have to do this via the RenderBackend-class:

```
class RenderBackendService {
public:
    RenderBackendService();
    virtual ~RenderBackendService();
    void setSettings(const Properties::Settings *config, bool moveOwnership);
    const Properties::Settings *getSettings() const;
    void sendEvent(const Common::Event *ev, const Common::EventData *eventData);
    PassData *getPassById(const c8 *id) const;
    PassData *beginPass(const c8 *id);
    RenderBatchData *beginRenderBatch(const c8 *id);
    void setMatrix(...);
    void setUniform(UniformVar *var);
    void setMatrixArray(const String &name, ui32 numMat, const glm::mat4 *matrixArray);
    void addMesh(Mesh *geo, ui32 numInstances);
    void addMesh(const CPPCore::TArray<Mesh *> &geoArray, ui32 numInstances);
    void updateMesh(Mesh *mesh);
    bool endRenderBatch();
    bool endPass();
    void clearPasses();
    void attachView();
    void resize(ui32 x, ui32 y, ui32 w, ui32 h);
    void syncRenderThread();
};
```

To work with this you have to configure it and open the access to it:

```
auto *rbService = new RenderBackendService();
rbService->setSettings(mySettings, false);
if (!m_rbService->open()) {
    // Error handling
}
```

6.4 Supported Render-API's

At this moment the following render-backends are implemented:

- OpenGL
- Vulkan (in progress)

CHAPTER

SEVEN

THE SCENE

7.1 The hierarchical Representation

If you want to build a virtal model of the world you can use trees. If you want to create a table with a glass staing on it you can model this like: - The floor where is the table stands on is the root-node - On the floor there is the table. So the table is a child of the floow- - On the table there is the glass standing on it. So the glass can be modelled as the child of the table.

The advantage of this kinde of modelleing gets clearer ifd you want to shake the floor by an earthquake. For a planned animation you need to know which objects get affected by the earchquake. And here the hierachical tree from your scnene gan hekp you: you just need to traverse all children from the floor.

7.2 Nodes and more

So how can we describe the scene by using a tree. OSRE is using nodes. Each node can have one single parent node and a couple of children nodes:

```` class Node {

Node mParent; ::CPPCore::TArray<Node*> mChildren;

### 7.2.1 };

## 7.3 Culling and Picking
````

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search