Fillwave 7.9.0 - new OpenGL 3.3+ (OpenGL ES 3.0+) graphics engine for C++14

Filip Wasil

November 23, 2017

Abstract

Before you start please ensure your graphics card driver supports at least OpenGL 3.3 and GLSL 330 or OpenGL ES 3.0 and GLSL 300 ES. Also, your C++ compiler must support c++14 standard (Ex. g++>6.1 or clang++>3.3). PC Context examples provided are using GLFW3. Editor provided uses the qt5 QT5 and are editable according to your needs. Android context samples use are using EGL (Java and native). Of course you can use any stub you like (Ex. freeglut, qt or other).

Contents

1 Introduction

1.1 Features

Graphics engine which you are about to use provides extremely, easy, portable, and uses C++14 modern API. It has all the essential functionalities that are needed to create a graphics layer for your application:

- Physics buffers for each model.
- Skybox and terrain generation.
- Renderable textures support.
- Spot and directional light support (Point lights will be available soon).
- Ortographic and Perspective projections.
- Easy to use callbacks mechanism.
- Flexible and easy event system.
- Lots of examples and Doxygen documentation.

Probably you will ask how is Fillwave better than other, more extended engines out there. The answer generally depends on what is your target. With this engine you can easily build a graphics layer to any game without installing any large IDE or lots of libraries. Fillwave provides an abstraction layer to OpenGL API introducing minimum overhead. It does not rely on the OpenGL context you have, so it can be used with GLFW, Freeglut or even with QT as weel. The android example (Using **native app glue** and EGL directly) is also available.

1.2 Code structure

Files in this project are organized in simple manner:

- ”inc” - headers
- ”src” - sources
- ”doc” - documentation
- ”ext” - sources of third party libraries (git submodules)
- ”cmake” - cmake macros
- ”examples” - multiplatform examples
- ”scripts” - building scripts

Engine uses dual namespace design style for modules. Code is splitted into three namespaces: **flw**, **flw:flf**, **flw::flc**. Core layer uses directly OpenGL driver API. **Framework** layer uses the **core** and by design implements a middleware of this project. The highest layer can be found under **fillwave** namespace. It is what we call Fillwave API. Naming convention is following:

- ”p” - shared pointer (Ex. pText)
- ”pu” - unique pointer (Ex. puRenderer)
- ”pw” - weak pointer (Ex. pwCameraPerspective)
- ”e” - enumeration class (Ex. EDebuggerState)
- ”I” - interface (Ex. IDrawable)

1.3 Getting started

The basic application skeleton looks like:

```
#include <fillwave/Fillwave.h>

using namespace flw;

int main(int argc, char* argv[]) {
   ... /* Create OpenGL/OpenGLES context */
   auto fillwave = std::make_unique<Engine>(argc, argv); /*
       Create Engine instance */
   ... /* create and init scene */
   ... /* enter rendering loop */
   exit(EXIT_SUCCESS);
}
```

1.4 Context creation

During the context initialization stage one must provide Fillwave engine a window (surface to draw on) and use **insert** functions in your context input handlers.

```
void onResizeScreen(GLuint width, GLuint height);
void onInput(KeyboardEvent& e);
void onInput(MouseButtonEvent& e);
void onInput(ScrollEvent & e);
void onInput(CharacterEvent& e);
void onInput(CharacterModsEvent& e);
void onInput(CursorEnterEvent& e);
void onInput(CursorPositionEvent& e);
void onInput(TouchEvent& e);
```

Every time when there is an event incoming to you context, (Does not matter if you are using glfw, freegut, QT or other library) and you want Fillwave to handle it you should **insert** a proper event into the engine using **insertEvent** function. Above there is an example using GLFW. The **keyboardCallback** function was previously registered as keyboard callback in GLFW.

```

void ContextGLFW1::keyboardCallback(GLFWwindow* window,
                                    int key,
                                    int scancode,
                                    int action,
                                    int mods) {
  /* Create an event data and fill it */
  flw::flf::KeyboardEventData data {
    action;
    key;
    mods;
    scancode;
  };
  /* Create an event */
  flw::flf::KeyboardEvent event(data);

  /* insert an event */
  mGraphicsEngine->onInput(event);
}
```

1.5 Rendering loop

Last step that has to be done in order to use Fillwave is rendering loop creation. In each iteration a **draw**, **drawLines**, or **drawPoints** function must be called with the **"How many seconds passed since last draw"** parameter. Also there is an extra **drawTexture** function which can be used if a single texture in all You want to see. GLFW example of render loop will look like:

```
void ContextGLFW1::render() {
  while (!glfwWindowShouldClose(mWindow)) {
    float now = glfwGetTime();
    float timeSinceLastFrameInSec = now - mTimeExpired;
    mTimeExpired = now;
    mGraphicsEngine->draw(timeSinceLastFrameInSec);

    /* We were writing to back buffer - make it visible */
    glfwSwapBuffers(mWindow);

    /* evaluate GLFW input events */
    glfwPollEvents();
  }
}
```

Offscreen drawing is possible using **capture** functions instead of **draw**.

```
  void captureFramebufferToFile(const std::string& name);
  void captureFramebufferToBuffer(GLubyte* buffer,
                                  GLint* sizeInBytes,
                                  GLuint format,
                                  GLint bytesPerPixel);
```

If not sure about the format you want you can just leave the default parameters. **captureFramebufferToBuffer** will use **GL_RGBA** with 4 bytes per pixel. This format is also a default one for **captureFramebufferToFile**.

2 Digging into API

2.1 Entity

puEntity is a base draw tree node. You can attach any other entities, models, particle emiters to it. You can move, rotate, and scale each of them. Fillwave uses önly one ownerp̈rinciple and strongly uses unique pointers from C++14.

```
auto entity_parent = pe<Entity();
auto entity_child = make_unique<Entity>();
entity_parent->attach(std::move(entity_child));
entity_parent->attach(make_unique<Entity>());
```

puEntity can be moved, rotated and scaled. The transformation matrix will be computed internally. However if one needs to set it directly (for example if it is computed by physics engine) there is a function provided:

```
void setTransformation(glm::mat4 transformationMatrix);
```

Getting a transformation matrix is also possible:

```
glm::mat4 getTransformation();
```

2.2 Scene

puScene by design is considered to be the root node of your **puEntity** tree. It stores its own **puCameraPerspective** (or **puCameraOrtographic**), **puSkybox** and **puCursor**. It also has an **onHide()** and **onShow()** virtual functions which will be executed during scene change.

```
/* Scene */
engine->setCurrentScene(make_unique<Scene>());

/* Camera */
engine->getCurrentScene()->setCamera(make_unique<CameraOrtographic>());
engine->getCurrentScene()->setCamera(make_unique<CameraPerspective>());
```

2.3 Camera

There are two camera to chose from in Fillwave: **CameraPerspective** and **CameraOrtographic**.Providing empty quaternion results will make the camera look in **-Z** direction. Example camera creation is listed below:

```
/* Perspective and ortographic cameras */
engine->getCurrentScene()->setCamera(make_unique<CameraPerspective>
                (glm::vec3(0.0,0.0,6.0), /* position */
```

```
                    glm::quat(), /* rotation */
                    glm::vec3(0.0,1.0,0.0), /* head up direction */
                    glm::radians(90.0), /* field of view angle */
                    screenWidth/screenHeight, /* screen ratio */
                    0.1, /* projection near plane */
                    1000.0)); /* projection far plane */
engine->getCurrentScene()->setCamera(make_unique<CameraOrtographic>
                       (glm::vec3(0.0,0.0,6.0),
                        glm::quat(), /* rotation */
                        -10.0f, /* x left culling */
                        10.0f, /* x right culling */
                        10.0f, /* y up culling */
                        -10.0f, /* y down culling */
                        0.1f, /* z near culling */
                        1000.0f)); /* z far culling */
```

2.4 Renderers

In current revision (7.0.0) there are 4 types of renderers:

- **RendererPBRP**
- **RendererFR**
- **RendererDR (partially done)**
- **RendererCSPBRP (partially done)**

Renderers are **per Scene** and can be set using **Scene::setRenderer()** function. Do not hesitate to create your own one. Its easy and fun (Just implement IRenderer Interface).

2.5 Programs and Shaders

Default programs can be built using **ProgramLoader** class using **getDefault** and **getDefaultBones** functions. See the example below:

```
/* Create loader, and use it to create programs */
loader::ProgramLoader loader(mEngine);
auto d = loader.getProgram(Eprogram::basic);
auto a = loader.getProgram(Eprogram::basicAnimated);
```

2.6 Store functions

Use **"store"** functions to create OpenGL objects which will be also stored by internal managers, and which will be internally, reloaded and reused if needed. Use store functions everywhere where possible.

```
/* Store shader using source directly */
mEngine->storeShader<GL_FRAGMENT_SHADER>("fancy_name",
     "shader code here");
/* Store shader providing file path */
mEngine->storeShader<GL_FRAGMENT_SHADER>("fillwave_default.frag");

/* Store program */
flc::Program* storeProgram(const std::string& ,
    std::vector<flc::Shader*>);
/* Store textures */
flc::Texture* storeTexture (const std::string&, const GLuint&);
flc::Texture2DRenderableDynamic*
                    storeTextureDynamic (const std::string&
                        fragmentShaderPath);
flc::Texture3D* storeTexture3D(const std::string& path,
                       const std::string& path,
                       const std::string& path,
                       const std::string& path,
                       const std::string& path,
                       const std::string& path);
auto storeLightSpot(glm::vec3, glm::vec4, pEntity);
auto storeLightPoint(glm::vec3, glm::vec4, pEntity);
auto storeLightDirectional(glm::vec4, glm::vec3);
auto storeText(std::string,std::string,GLfloat,GLfloat,GLfloat);
```

2.7 Model

Fillwave provides different methods to build a model. You can use make function to create unique_unique or builder classes instead.

2.7.1 Direct methods

```
/*
 * When the appropriate map paths are available
 * together with your model asset file.
 */

auto model = make_unique<Model>(engine, program, "model.obj");

/*
 * When the appropriate map paths are available in your
 * file and you want to draw Your custom shape derived
 * from flf::Shape<flc::VertexBasic>
 */

auto model = make_unique<Model>(engine,
                      program,
                      flf::Sphere(1.0,10.0,10.0),
                      diffuseMap,
                      normalMap,
                      specularMap,
                      material);


/*
 * When we want to explicitily provide texture paths
 * but stll use the model asset from file.
 */

auto model = make_unique<Model>(engine,
                    program,
                    "model.obj",
                    "relativePathToDiffuseMap",
                    "relativePathToNormalsMap",
                    "relativePathToSpecularMap");

/*
 * When we want to use previously created texture
 * and material objects.
 */
```

```
39
40 auto model = make_unique<Model>(engine,
41                     program,
42                     "model.obj",
43                     diffuseMapTexture,
44                     normalMapTexture,
45                     specularMapTexture,
46                     material);
```

2.7.2 Builders

Fillwave also provides two builders classes. You can use **BuilderModelExternalMaps** or **BuilderModelManual** described below.

```
1  /* BuilderModelExternalMaps uses custom texture maps */
2
3  /* First method */
4  BuilderModelExternalMaps builder1 (engine,
5                                modelPath,
6                                flc::Program* program,
7                                diffusePath,
8                                normalPath,
9                                specularPath);
10
11 auto m = builder1.build();
12
13 /* Second method */
14 BuilderModelExternalMaps builder1(engine);
15
16 auto m = builder1.setModelPath(modelPath).
17                 setProgram(program).
18                 setdiffusePath(diffuseMap).
19                 setNormalMapPath(normalsMap).
20                 setSpecularMapPath(specularMap).
21                 setMaterial(material).
22                 build();
23
24
25
26 /* BuilderModelManual uses custom textures and material *
27 /
28 /* First method */
29
30 BuilderModelManual builder2 (engine,
```

```
                          modelPath,
                          program,
                          diffuseMap,
                          normalsMap,
                          specularMap,
                          material);
auto m = builder2.build();

/* Second method */

BuilderModelManual builder2 (engine);

auto m = builder2.setModelPath(modelPath).
                setProgram(program).
                setDiffuseMapTexture(diffuseMap).
                setNormalMapTexture(normalsMap).
                setSpecularMapTexture(specularMap).
                setMaterial(material).
                build();
```

In each case animations will be also loaded. You can check how many of them are available, and activate one You are interested in. Default value for active animation in each model is set to "FILLWAVE_DO_NOT_ANIMATE".

```
void setActiveAnimation(GLint animationID)
GLint getAnimations();
```

2.7.3 Effects

Fillwave provides **Effects** objects which can be added to each Model. You can use built in effects: **Fog**, **BoostColor**, **ClockwiseDrawEffect**, **Painter** and **TextureOnly**. You can also create Your own one by inheriting from **Effect** class and implementing all necessary methods. Remember that during the effect execution, the models program is already used, so You can call **uniformPush** function. Effects uses shared pointer policy. They are shared between models and callbacks by design.

```

auto f std::make_shared<Fog>());
auto b std::make_shared<BoostColor(10.0);
auto c std::make_shared<ClockwiseDrawEffect();
auto p std::make_shared<Painter();
```

```
auto t std::make_shared<TextureOnly();

model->addEffect(f);
```

2.8 Particles

Particles system entry in Fillwave is in fact two (but powerfull) classes: **EmiterPointCPU** and **EmiterPointGPU**. The **EmiterPointGPU** particle emiter is computed entirely on GPU and uses Texture3D noise as a seed to generate random positions and velocities. It is slower but gives better robustness factors. **EmiterPointCPU** emiter particles are precomputed on CPU. They are faster but the factors are less robust.

```
EmiterPointCPU::EmiterPointCPU(Engine* engine,
                               GLfloat emitingSurfaceRadius,
                               GLfloat robustness,
                               GLint howMany,
                               glm::vec4 color,
                               glm::vec3 acceleration,
                               glm::vec3 velocity,
                               glm::vec3 distance,
                               flc::Texture* texture,
                               GLfloat lifetimeInSec,
                               GLfloat pointSize,
                               GLboolean dephTest,
                               GLfloat alphaCutOff)

EmiterPointGPU::EmiterPointGPU(Engine* engine,
            GLfloat emitingSourceRate,
            GLuint howMany,
            glm::vec4 color,
            glm::vec3 acceleration,
            glm::vec3 startVelocity,
            glm::vec3 robustnessVelocity,
            glm::vec3 startPosition,
            glm::vec3 robustnessPosition,
            GLfloat startSize,
            GLfloat lifetime,
           flc::Texture* texture,
           GLenum blendingSource,
           GLenum blendingDestination,
           GLboolean dephTest,
           GLfloat alphaCutOff);
```

```
31
32 /* Change the blending function if needed */
33 /* Default blending source is GL_SRC_ALPHA */
34 /* Default blending destination is GL_ONE_MINUS_SRC_ALPHA*/
35 void setBlendingFunction (GLenum sourcePixel, GLenum destPixel);
```

EmiterPointCPU emits particles using a round surface source. You can set the radius of this surface (**emitingSurfaceRadius**), and emiting **robustness**. **Robustness = 0** will make the particles flow perpendicular to the emiting surface. Parameter **dephTest** is critical. Using the depth test is slower but it guarantees that particles will stay visible only when they should be. Giving up the depth test will make them look much nicer and rendered faster, but they will be visible **always** which can make scene look not natural. AlphaCutOff parameter privides additional feature to discard all pixels with alpha value less than alphaCutOff.

2.9 Skybox

To create a skybox in fillwave You just need to provide texture paths as shown below.

```
1
2 auto texture=flc::Texture3D*(new Texture3D("textures_right.png",
3                                            "textures_left.png",
4                                            "textures_ceil.png",
5                                            "textures_floor.png",
6                                            "textures_front.png",
7                                            "textures_back.png"));
8 scene->setSkybox(make_unique<Skybox>(engine, texture));
```

2.10 Terrain

Terrain in Fillwave can be generated using a quad chunks. This method provides mechanism for terrain generation.

```
1 auto fs =
    engine->storeShader<GL_FRAGMENT_SHADER>("default.frag");
2 auto vs = engine->storeShader<GL_VERTEX_SHADER>("default.vert");
3 auto program = buildProgram(fs + vs);
4 auto terrain = buildTerrainVoxel(engine,
5                                  program,
6                                  "textures/test.png",
7                                  new MountainConstructor(),
8                                  5);
9 sene->attach(std::move(terrain));
```

You may have noticed that some code mensions also a **voxel terrain** feature. This is an old legacy feature and it will be replaced by more generic solution. Stay tuned.

2.10.1 Mesh terrain

To create a terrain Mesh you should create a class derived from **TerrainConstructor** class, and implement a **calculateHeight** method. The method should take x and z coordinates in the range of (-1,1) in and return Y position.

2.11 Text

To create a 2D on screen text using ttf fonts You can use the **storeText** function. Texts and HUD uses shared pointers policy. They are shared between the engine and the user by design.

```
auto text = engine->storeText( "Hello Fillwave",/* content */
                               "FreeMono",/* font to use */
                               -0.95, /*left bottom y start (-1,1)*/
                               -0.80, /*left bottom x start (-1,1)*/
                               100.0, /* text size */
                               ETextEffect::none); /* text effect */
```

Fillwave will look for the font in the directory relative to Your binary directory. If it will not find it, it will search the **/usr/share/fonts/truetype/freefont/** directory. Next, it will create a texture and save its metadata. Finally this texture will be used as an atlas.

2.12 Light

There are three Possible light types which can be created in Fillwave. These types are: point, spot and directional lights.

2.12.1 Spot light

Spot lights have position, intensity (RGBA) and entity parameters. When the entity is provided, the light will follow the entity whatever happens and do not consider the **position**. When there is no entity provided, spot light will keep its position as set in constructor. Spot light generates perspective shadows into the scene.

2.12.2 Directional light

Difference between spot and directional lights is a projection type. Directional lights will have an ortographic projection. It is perfect for light sources which gives constant size shadowing (Sun for example).

2.12.3 Point light

Point lights emits the light in all directions. In current revision this kind of light does not generate any shadowing effect.

2.13 Logging

All objects in Fillwave have a **log** function which prints most of the objects data to standard output. There are also predefined macros ready to use:

- **fLogU** - free to use.
- **fLogC** - checks OpenGL errors.
- **fLogI** - prints **log** function information.
- **fLogD** - reserved for internal debug info.
- **fLogE** - called in case of internal engine error.
- **flogF** - just like **fLogE** but also calls abort(). It indicates blocking errors like: "Shaders not found". If such error occurs, and the reason is not trivial then it needs further investigation by the author. Do not hesitate to contact me in such case.

To print a debug info in a certain source file You should define a module name and debug flags with macro **FLOGINIT**. Examples below:

```
1
2 #define FLOGINIT_DEFAULT()
3 #define FLOGINIT_NONE()
4 #define FLOGINIT_MASK(FERROR | FFATAL | FDEBUG | FDEBUG | FUSER)
5 #define FLOGINIT("My module", FERROR | FFATAL | FDEBUG | FDEBUG
    | FUSER)
```

2.14 Event system

There are two basic types of callback functions:

- hierarchy callbacks
- private callbacks

Difference between the **hierarchy** and **private** is that hierarchy callback executes synchronously just before the draw when the scene is drawn. As opposite, the private one is called asynchronously when the particular event is introduced into the engine (Ex. Mouse button click, or Key press). Most commonly used **private callbacks** are **TimedCallback** classes:

```
TimedCallback(GLfloat timeToFinish,
   EasingFunction easing = eEasing::None);
TimedScaleCallback(Moveable* entity,
   glm::vec3 normalizedScaleVec,
   GLfloat lifetime,
   EasingFunction easing);
TimedRotateCallback(Moveable* entity,
   glm::vec3 axis,
   GLfloat angle,
   GLfloat lifeTime,
   EasingFunction easing);
TimedMoveCallback(Moveable* entity,
   glm::vec3 endPosition,
   GLfloat lifeTime,
   EasingFunction easing);

puEntity entity = make_unique<Entity>();

entity->registerHierarchyCallback(
   make_unique<TimedMoveCallback>(
      entity.get(),
      glm::vec3(0.0f,0.0f,1.0f),
      10.0f);
```

TimedCallback by itself stands only for a time delay. **TimedScaleCallback**, **TimedRotateCallback**, and **TimedMoveCallback** on the other hand can be used to modify the model scale/position/rotation in time with current easing described by std::function **EasingFunction**. Default easing for all of the callbacks is **LinearInterpolation**.

2.14.1 Focus

Focus functionality and hierarchy callbacks were introduced to enable executing particular callbacks in particular entity without iterating over the whole scene tree. To set an entity which will receive a callback from chosen input **Engine::attachCallback** function should be used.

```

/* Second parameter means that the IFocusable interface will be
    notified to engine so that it will be removed after during
    model destruction */
engine->attachCallback(
     make_unique<AnimationKeyboardCallback>(beast.get(),
         EEventType::eKey),
     beast.get());
```

To attach/detach an item callback to/from an entity:

```
void Entity::attachHierarchyCallback(Callback* c);
```

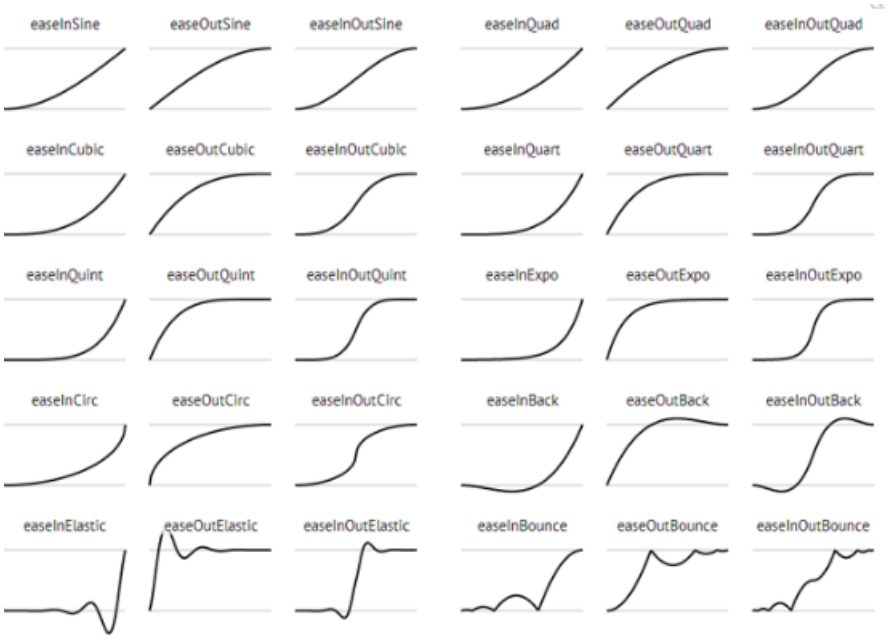
2.14.2 Register, unregister and clear functions

To register/unregister a callback in Fillwave use following functions:

```
void Entity::attachHandler(std::function<void(const Event&)>&&
    e, eEventType t)
void detachHandlers();
```

2.15 Easing

Handlers can be used to modify model transformation (scale, rotation and position) in time with particular easing. You can choose one of following easings define by **EasingFunction**:



2.16 Physics

To synchronize Your graphics with physics engine just use the **setTransformation** function which is available for each entity. it overwrites all other transformations for a model.

```
1 void Entity::setTransformation(glm::mat4 modelMatrix)
```

If You have a light attached to Your model, the light will be moved together with its entity. However, only translation will be updated. If You want the light to keep the same rotation as its entity, You should use **updateParentRotation** function explicitily.

```
1 void Entity::updateParentRotation(glm::quat rotationQuaternion)
```

There is a **PhysicsMeshBuffer** defined. It can be used by physics engine to generate a collision object from a mesh polygons. Example usage of this

buffer can be found in Fillwave car racing demo - **Waveracer**. To get physics buffer from asset file use:

```
PhysicsMeshBuffer Engine::getPhysicalMeshBuffer(const
    std::string& shapePath)
```

2.17 Extras

To change the background color use:

```
void Engine::configBackgroundColor (glm::vec3 color);
```

To apply the time factor to in Fillwave engine use:

```
void Engine::configTime(GLfloat timeFactor); /* 1.0f as default
    */
```

To get the current executable directory use:

```
std::string Engine::getExecutablePath()
```

To set/reset reset file logging use:

```
void Engine::configFileLogging(std::string fileName = "");
```

empty or not valid file name will disable the file logging.

There are few texture generators built-in in Fillwave. To use them just pass one of the patterns as a texture path in **Model** constructor or **storeTexture** function:

```
/* [R]_[G]_[B].color - for color texture */
/* [R]_[G]_[B].checkboard - for color checkboard texture */
/* "" - Black texture */

pModel model = buildModel(engine,
        programDefault,
        "model.obj",
        "255_0_0.checkboard", /* Red checkboard diffuse texture */
        "", /* black normal map */
        "255_255_255.color"); /* white specular map */
```

Debugger related API is provided to enable simple debugging of depth maps from each spot light, and to enable viewing the pickable objects if there are any of them registered in the scene. debugger can be configured using one of the following enum constants. **toggleState** is a special value which will just iterate over the possible debugger configurations.

```
enum class EDebuggerState {
   lightsSpot,
   lightsSpotColor,
   lightsSpotDepth,
   lightsPoint,
   lightsPointDepth,
   lightsPointColor,
   pickingMap,
   off,
   toggleState
};

void Engine::configureDebugger(EDebuggerState state);
```

2.18 Renderers

By implementation of custom renderer class, one can simply manage the rendering approach of Fillwave engine. This is the most powerfull feature, because it provides a lot of flexibility. The simplest renderer example RendererFR. The most complex one we have here is RendererDR.

```
/*! \class CustomRenderer
 * \brief Base for all renderers.
 */

class CustomRenderer {
public:
   CustomRenderer();
   virtual ~CustomRenderer();

   /* Add renderable item to your container */
   void update(IRenderable* renderable) override;

   /* Iterate over your container passing and perform the draw on
       each of them */
   void draw(ICamera& camera) override;

   /* Reset the renderers state */
   void reset(GLuint width, GLuint height) override;

   /* Clear the container */
   void clear() override;
private:
   /* Container which will keep your renderable elements */
   std::vector<IRenderable*> mContainer;
};

} /* flf */
} /* flw */
```