Fillwave 4 - new OpenGL 3.3+ (OpenGL ES 3.0+) graphics engine for C++11

Filip Wasil

March 5, 2016

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++11 standard (Ex. g++>4.7 or clang++>3.3). PC Context examples provided are using GLFW3. 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 **3**
- 1.1 Features . . . 3
- 1.2 Code structure . . . 4
- 1.3 Getting started . . . 4
- 1.4 Context creation . . . 5
- 1.5 Rendering loop . . . 6

2 Digging into API **6**
- 2.1 Entity . . . 6
- 2.2 Scene . . . 7
- 2.3 Camera . . . 8
- 2.4 Renderers . . . 8
- 2.5 Programs and Shaders . . . 8
- 2.6 Store functions . . . 9
- 2.7 Model . . . 10
  - 2.7.1 Direct methods . . . 10
  - 2.7.2 Builders . . . 12
  - 2.7.3 Effects . . . 13
- 2.8 Particles . . . 13
- 2.9 Skybox . . . 16
- 2.10 Terrain . . . 16
  - 2.10.1 Mesh terrain . . . 16
- 2.11 Text . . . 17
- 2.12 Light . . . 17
  - 2.12.1 Spot light . . . 17
  - 2.12.2 Directional light . . . 17
  - 2.12.3 Point light . . . 17
- 2.13 Logging . . . 17
- 2.14 Event system . . . 19
  - 2.14.1 Focus functions and private callbacks . . . 20
  - 2.14.2 Register, unregister and clear functions . . . 20
- 2.15 Easing . . . 21
- 2.16 Physics . . . 21
- 2.17 Extras . . . 22

3 Customization **24**
- 3.1 Events . . . 24
- 3.2 Callbacks . . . 24
- 3.3 Easing . . . 25
- 3.4 Renderers . . . 26

4 Examples **26**

5 Licenses **29**

1 Introduction

1.1 Features

Graphics engine which you are about to use provides extremely, easy, portable, and uses C++11 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: **fillwave**, **fillwave:framework**, **fillwave::core**. 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. pEntity)
- ”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 fillwave;

int main(int argc, char* argv[]) {
   /* Create OpenGL/OpenGLES context */
   Engine* fillwave = new Engine(argc, argv);
   /* Create scene */
   /* enter rendering loop */
   delete engine;
   /* Delete OpenGL/OpenGLES context*/
   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 insertResizeScreen(GLuint width, GLuint height);
void insertInput(KeyboardEvent& e);
void insertInput(MouseButtonEvent& e);
void insertInput(ScrollEvent & e);
void insertInput(CharacterEvent& e);
void insertInput(CharacterModsEvent& e);
void insertInput(CursorEnterEvent& e);
void insertInput(CursorPositionEvent& e);
void insertInput(TouchEvent& e);
void insertInput(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 */
  fillwave::framework::KeyboardEventData data;
  data.action = action;
  data.key = key;
  data.mode = mods;
  data.scanCode = scancode;

  /* Create an event */
  fillwave::framework::KeyboardEvent event(data);

  /* insert an event */
  mGraphicsEngine->insertInput(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 display if a single texture in all You want to see. GLFW example of render loop will look like:

```
void ContextGLFW1::render() {
  while (!glfwWindowShouldClose(mWindow)) {
    GLfloat timeSinceLastFrameInSec, now = glfwGetTime();

    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

pEntity 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.

```
pEntity entity_parent = buildEntity();
pEntity entity_child = buildEntity();
entity_parent->attach(entity_child);
```

pEntity 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

pScenePerspective (or **pSceneOrtographic**) by design is considered to be the root node of your **pEntity** tree. It stores its own **pCameraPerspective** (or **pCameraOrtographic**), **pSkybox** and **pCursor**. It also has an **onHide()** and **onShow()** virtual functions which will be execuded during scene change.

```
/* Build scene */
pISceneOrtographic sceneO = buildSceneOrtographic();
pIScenePerspective speneP = buildScenePerspective();

/* Build camera */
pCameraOrtographic cO = std::make_shared<CameraOrtographic>();
pCameraPerspective cP = std::make_shared<CameraPerspective>();

/* Attach camera */
sceneO->setCamera(cO);
speneP->setCamera(cP);

/* Attach scene */
engine->setCurrentScene(sceneP);
```

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 */
pCameraPerspective cameraP = std::make_shared<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 */
gCameraOrthographic cameraO = std::make_shared<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 (4.2.1) there are 4 types of renderers:

- **RendererPBRP**
- **RendererFR**
- **RendererDR (still in progress)**
- **RendererCSPBRP (still in progress)**

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(gEngine);
pProgram default = loader.getDefault();
pProgram animation = loader.getDefaultBones();
```

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 the shaders providing source file path */
pShader storeShaderFragment(const std::string& path);
pShader storeShaderVertex(const std::string& path);
pShader storeShaderGeometry(const std::string& path);
pShader storeShaderTesselationControl(const std::string& path);
pShader storeShaderTesselationEvaluation(const std::string& p);
/* Store the shaders providing the source directly */
pShader storeShaderFragment(const std::string&, std::string&);
pShader storeShaderVertex(const std::string&, std::string&);
pShader storeShaderGeometry(const std::string&, std::string&);
pShader storeShaderTesselationControl(const std::string&,
    std::string&);
pShader storeShaderTesselationEvaluation(const std::string&,
    std::string&);
pProgram storeProgram(const std::string& , std::vector<pShader>);
pTexture storeTexture (const std::string&, const GLuint&);
pTexture2DRenderableDynamic
                    storeTextureDynamic (const std::string&
                        fragmentShaderPath);
pTexture3D 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);
pLightSpot storeLightSpot(glm::vec3, glm::vec4, pEntity);
pLightPoint storeLightPoint(glm::vec3, glm::vec4, pEntity);
pLightDirectional storeLightDirectional(glm::vec4, glm::vec3);
pText storeText(std::string,std::string,GLfloat,GLfloat,GLfloat);
pCursor storeCursor(pTexture, GLfloat);
```

2.7 Model

Fillwave provides different methods to build a model. You can use **buildModel** functions or direct constructors.

2.7.1 Direct methods

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

pModel model = buildModel(engine, program, "model.obj");

pModel model = std::make_shared<framework::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 framework::Shape<core::VertexBasic>
 */

framework::Sphere sphere(1.0,10.0,10.0);

pModel model = buildModel(engine,
                          program,
                          sphere,
                          diffuseMap,
                          normalMap,
                          specularMap,
                          material);

pModel model = std::make_shared<framework::Model>(engine,
                          program,
                          sphere,
                          diffuseMap,
                          normalMap,
                          specularMap,
                          material);
```

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

pModel model = buildModel(engine,
                    program,
                    "model.obj",
                    "relativePathToDiffuseMap",
                    "relativePathToNormalsMap",
                    "relativePathToSpecularMap");

pModel model = std::make_shared<framework::Model>(engine,
                     program,
                     "model.obj",
                     "relativePathToDiffuseMap",
                     "relativePathToNormalsMap",
                     "relativePathToSpecularMap");

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

pModel model = buildModel(engine,
                    program,
                    "model.obj",
                    diffuseMapTexture,
                    normalMapTexture,
                    specularMapTexture,
                    material);
pModel model = std::make_shared<Model>(engine,
                    program,
                    "model.obj",
                    diffuseMapTexture,
                    normalMapTexture,
                    specularMapTexture,
                    material);
```

2.7.2 Builders

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

```
/* BuilderModelExternalMaps uses custom texture maps */

/* First method */
BuilderModelExternalMaps builder1 (engine,
                              modelPath,
                              pProgram program,
                              diffusePath,
                              normalPath,
                              specularPath);

pModel m = builder1.build();

/* Second method */
BuilderModelExternalMaps builder1(engine);

pModel m = builder1.setModelPath(modelPath).
               setProgram(program).
               setdiffusePath(diffuseMap).
               setNormalMapPath(normalsMap).
               setSpecularMapPath(specularMap).
               setMaterial(material).
               build();



/* BuilderModelManual uses custom textures and material *
/
/* First method */

BuilderModelManual builder2 (engine,
                         modelPath,
                         program,
                         diffuseMap,
                         normalsMap,
                         specularMap,
                         material);
pModel m = builder2.build();

/* Second method */

```

```
BuilderModelManual builder2 (engine);

pModel 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.

```

pIEffect fog(new Fog());
pIEffect boost(new BoostColor(10.0));
pIEffect ccw(new ClockwiseDrawEffect());
pIEffect paint(new Painter());
pIEffect textureOnly(new TextureOnly());

model->addEffect(fog);
```

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,
                               pTexture texture,
                               GLfloat lifetimeInSec,
                               GLfloat pointSize,
                               GLboolean dephTest,
                               GLfloat alphaCutOff)

pEmiterPointGPU::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,
           pTexture texture,
           GLenum blendingSource,
           GLenum blendingDestination,
           GLboolean dephTest,
           GLfloat alphaCutOff);

/* Change the blending function if needed */
/* Default blending source is GL_SRC_ALPHA */
/* Default blending destination is GL_ONE_MINUS_SRC_ALPHA*/
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

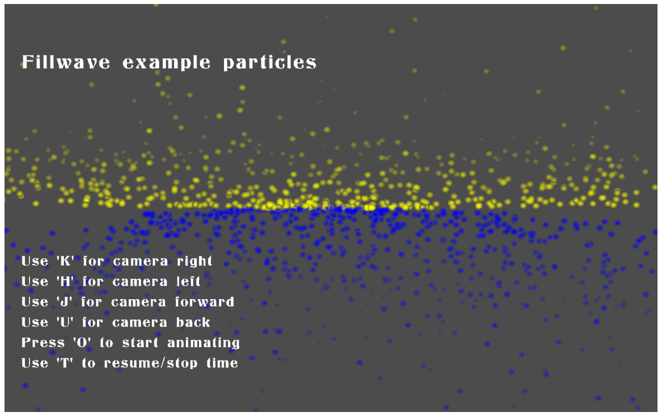


Figure 1: Particles with depth test active

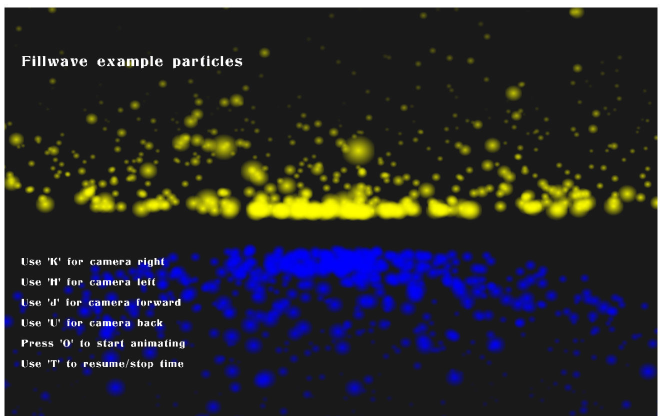


Figure 2: Particles without depth test active

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.

```

pTexture3D texture=pTexture3D(new Texture3D("textures_right.png",
                                           "textures_left.png",
                                           "textures_ceil.png",
                                           "textures_floor.png",
                                           "textures_front.png",
                                           "textures_back.png"));
pSkybox skybox = buildSkybox(engine,
                             texture);
scene->setSkybox(skybox);
```

2.10 Terrain

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

```
pShader fs = engine->storeShaderFragment("default.frag");
pShader vs = engine->storeShaderVertex("default.vert");
pProgram program = buildProgram(fs + vs);
pTerrain terrain = buildTerrainVoxel(engine,
                                  program,
                                  "textures/test.png",
                                  new terrain::MountainConstructor(),
                                  5);
gScene->attach(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.

```
pText 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:

- **FLOG_USER** - free to use.
- **FLOG_CHECK** - checks OpenGL errors.
- **FLOG_INFO** - prints **log** function information.
- **FLOG_DEBUG** - reserved for internal debug info.
- **FLOG_ERROR** - called in case of internal engine error.
- **FLOG_FATAL** - just like FLOG_ERROR 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:

```

#define FLOGINIT_DEFAULT()
#define FLOGINIT_NONE()
#define FLOGINIT_MASK(FERROR | FFATAL | FDEBUG | FDEBUG | FUSER)
#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(pEntity entity,
                   glm::vec3 normalizedScaleVec,
                   GLfloat lifetime,
                   EasingFunction easing);
TimedRotateCallback(pEntity entity,
                    glm::vec3 axis,
                    GLfloat angle,
                    GLfloat lifeTime,
                    EasingFunction easing);
TimedMoveCallback(pEntity entity,
                  glm::vec3 endPosition,
                  GLfloat lifeTime,
                  EasingFunction easing);
```

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 functions and private callbacks

Focus functionality and private 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 **setFocus** functions should be used.

```

void setFocus(eEventType eventType, pEntity entity);
```

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

```
void Entity::attachHierarchyCallback(Callback* c);
void Entity::attachPrivateCallback(Callback* c);
void Entity::detachHierarchyCallback(Callback* c);
void Entity::detachPrivateCallback(Callback* c);
```

2.14.2 Register, unregister and clear functions

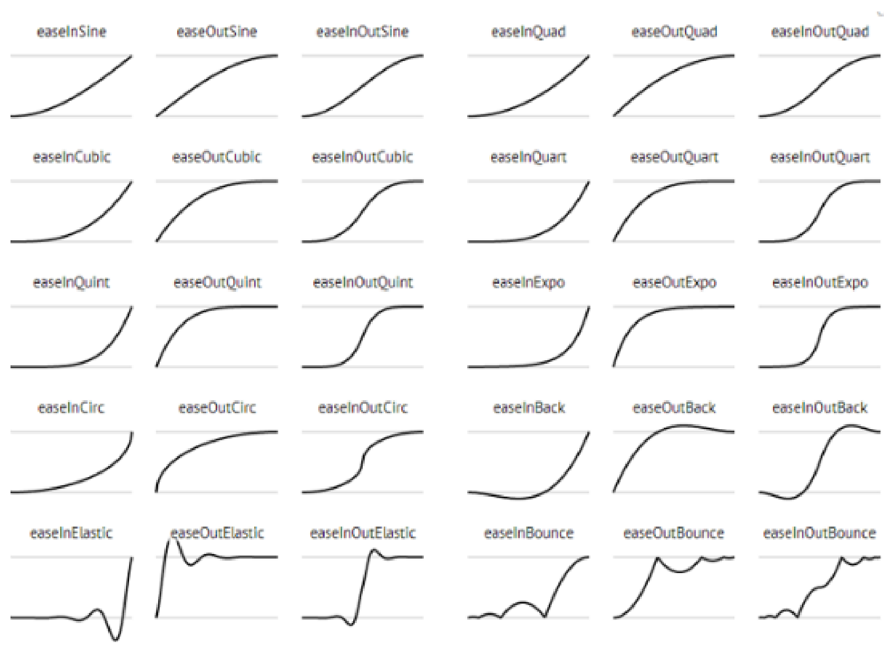
To register/unregister a callback in Fillwave use following functions:

```
void registerCallback(Callback* c);
void unregisterCharacterModsCallback(Callback* c);
```

Note, that there is no need to **delete** any objects. This happens inside the callbacks.

2.15 Easing

Timed Callbacks 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.

```
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.

```
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::configureBackgroundColor (glm::vec3 color);
```

To apply the time factor to in Fillwave engine use:

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

To get the current executable directory use:

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

To set/reset current "frames per seconds" counter in right left corner use:

```
void Engine::configureFPSCounter(std::string fontName = "",
                                 GLfloat xPosition = -0.95,
                                 GLfloat yPosition = 0.95,
                                 GLfloat size = 100.0);
```

empty or not valid font name will disable the FPS counter.
To set/reset reset file logging use:

```
void Engine::configureFileLogging(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);
```

3 Customization

3.1 Events

```

namespace fillwave {
namespace actions {
struct NewEventData {
   int data;
   const eEventType type = eEventType::custom0; /* event ID */
};

class NewEvent: public Event<NewEventData> {
public:
   NewEvent();
   virtual ~NewEvent();
};
} /* actions */
} /* fillwave */
```

3.2 Callbacks

```
namespace fillwave {
namespace actions {

class NewEngineCallback: public EngineCallback {
private:
   float mMaximimData;
   void sayHello() {FLOG_USER("Hello event");};
public:
   NewEngineCallback(eEventType eventType, float data);
   NewEngineCallback(float data);

   virtual ~NewActionCallback();
   void perform (Engine* engine, EventType* event);
   };

} /* actions */
} /* fillwave */
```

3.3 Easing

TimedMoveCallbackCustom.h

```
namespace fillwave {
namespace actions {

class TimedMoveCallbackCustom: public
    fillwave::framework::TimedMoveCallback {
public:
   TimedMoveCallbackCustom(pEntity entity,
                          glm::vec3 endPosition,
                          GLfloat lifeTime);
   virtual ~TimedMoveCallbackCustom();
   GLfloat easeCustom(GLfloat progress);
};
} /* actions */
} /* fillwave */
```

TimedMoveCallbackCustom.cpp

```
namespace fillwave {
namespace actions {

TimedMoveCallbackCustom::TimedMoveCallbackCustom(pEntity entity,
                                                  glm::vec3 endPosition,
                                                  GLfloat
                                                      lifeTime):TimedMoveCallback(entity,
                                                      endPosition,lifeTime,
                                                      eEasing::Custom) {

}

TimedMoveCallbackCustom::~TimedMoveCallbackCustom() {

}

GLfloat TimedMoveCallbackCustom::easeCustom(GLfloat progress) {
   /* You custom easing function goes here. For example: */
   return QuinticEaseIn(progress)*QuinticEaseIn(progress);
}

} /* actions */
} /* fillwave */
```

3.4 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;
};

} /* namespace framework */
} /* namespace fillwave */
```

4 Examples

Basic example You can find below:

```

   /* Camera */
```

```
    pCameraPerspective camera = pCameraPerspective ( new
        CameraPerspective(glm::vec3(0.0,0.0,16.0),
                                                    glm::quat(),
                                                    glm::radians(90.0),
                                                    1.0,
                                                    0.1,
                                                    1000.0));
    /* Programs */
    ProgramLoader loader(ContextGLFW::mGraphicsEngine);
    pProgram program = loader.getDefault();

    /* Models */
    pModel sphere = buildModel(ContextGLFW::mGraphicsEngine,
        program, "meshes/sphere.obj", "255_255_255.color");

    scene->attach(sphere);
    scene->setCamera(camera);
    ContextGLFW::mGraphicsEngine->setCurrentScene(scene);

    mContext.render();
    delete ContextGLFW::mGraphicsEngine;
    exit(EXIT_SUCCESS);
}
```

Basic example You can find below:

Main example repository

Example: Text
Example: Animation
Example: Timed callbacks with custom easing
Example: Picking
Example: Dynamic texture
Example: Effects
Example: Specular and normal maps
Example: Skybox
Example: Lights
Example: Particles
Example: Quad Terrain
Example: Voxel terrain
Example: Custom shader shape
Example: Postprocessing
Example: Ortographic projection
Example: Effects

Waveracer game draft

Example: Android activity
Example: Android JNI library
Example: Android pure native project

5 Licenses

```
/**************************************************************************
 * Fillwave C++11 graphics Engine
 * Copyright (C) 2015 Filip Wasil
 *  All Rights Reserved.
 * This library is available free of charge for any commercial
 * or non-commercial use. However, You are obligated to put
 * a clearly visible information in Your license agreement
 * that Your Software uses Fillwave library. Fillwave uses
 * few external libraries and their licenses are written below.
 * If You are interested in extra support, extra features
 * or cooperation I look forward to hearing from You.
 *
 *    Filip Wasil           fillwave@gmail.com
 */

 /* OpenGL
http://www.opengl.org/

* AssImp library
Open Asset Import Library (assimp)

Copyright (c) 2006-2012, assimp team
All rights reserved.

Redistribution and use of this software in source and binary forms,
with or without modification, are permitted provided that the
following conditions are met:

```

```
* Redistributions of source code must retain the above
  copyright notice, this list of conditions and the
  following disclaimer.

* Redistributions in binary form must reproduce the above
  copyright notice, this list of conditions and the
  following disclaimer in the documentation and/or other
  materials provided with the distribution.

* Neither the name of the assimp team, nor the names of its
  contributors may be used to endorse or promote products
  derived from this software without specific prior
  written permission of the assimp team.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
```

```
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



*****************************************************************************

AN EXCEPTION applies to all files in the ./test/models-nonbsd folder.
These are 3d models for testing purposes, from various free sources
on the internet. They are - unless otherwise stated - copyright of
their respective creators, which may impose additional requirements
on the use of their work. For any of these models, see
<model-name>.source.txt for more legal information. Contact us if you
are a copyright holder and believe that we credited you inproperly or
if you don't want your files to appear in the repository.


*****************************************************************************

Poly2Tri Copyright (c) 2009-2010, Poly2Tri Contributors
http://code.google.com/p/poly2tri/

All rights reserved.
Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.
```

```
* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.
* Neither the name of Poly2Tri nor the names of its contributors may be
  used to endorse or promote products derived from this software without specific
  prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
* FreeType 2 library (FTL licence)
http://www.freetype.org/

  This license grants a worldwide, royalty-free, perpetual and
```

```
irrevocable right and license to use, execute, perform, compile,
display, copy, create derivative works of, distribute and
sublicense the FreeType Project (in both source and object code
forms) and derivative works thereof for any purpose; and to
authorize others to exercise some or all of the rights granted
herein, subject to the following conditions:

  o Redistribution of source code must retain this license file
    (‘FTL.TXT’) unaltered; any additions, deletions or changes to
    the original files must be clearly indicated in accompanying
    documentation. The copyright notices of the unaltered,
    original files must be preserved in all copies of source
    files.

  o Redistribution in binary form must provide a disclaimer that
    states that the software is based in part of the work of the
    FreeType Team, in the distribution documentation. We also
    encourage you to put an URL to the FreeType web page in your
    documentation, though this isn’t mandatory.

These conditions apply to any software derived from or based on
the FreeType Project, not just the unmodified files. If you use
```

```
  our work, you must acknowledge us. However, no fee need be paid
  to us.

* GLEW library
http://glew.sourceforge.net/

GLEW is originally derived from the EXTGL project by Lev Povalahev. The source
code is licensed under the Modified BSD License, the Mesa 3-D License
(MIT License), and the Khronos License (MIT License). The automatic code
generation scripts are released under the GNU GPL.

* Lanyon
Released under MIT License
Copyright (c) 2014 Mark Otto.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom
the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice
shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
```

```
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

* fontGenerator
/****************************************************************************
| OpenGL 4 Example Code.                                                     |
| Accompanies written series "Anton's OpenGL 4 Tutorials"           |
| Email: anton at antongerdelan dot net                             |
| First version 5 Feb 2014                                                  |
| Copyright Dr Anton Gerdelan, Trinity College Dublin, Ireland.     |
| See individual libraries and assets for respective legal notices  |

* Sean Barrett's public domain stb_image and stb_image_write libraries
http://nothings.org/

/* stbiw-0.92 - public domain - http://nothings.org/stb/stb_image_write.h
   writes out PNG/BMP/TGA images to C stdio - Sean Barrett 2010
                        no warranty implied; use at your own risk


Before including,
```

```
    #define STB_IMAGE_WRITE_IMPLEMENTATION

in the file that you want to have the implementation.


ABOUT:

   This header file is a library for writing images to C stdio. It could be
   adapted to write to memory or a general streaming interface; let me know.

   The PNG output is not optimal; it is 20-50% larger than the file
   written by a decent optimizing implementation. This library is designed
   for source code compactness and simplicitly, not optimal image file size
   or run-time performance.

*/
```