

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

Filip Wasił

January 11, 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

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                      | <b>3</b>  |
| 1.1      | Features                                 | 3         |
| 1.2      | Code structure                           | 4         |
| 1.3      | Getting started                          | 4         |
| 1.4      | Context creation                         | 5         |
| 1.5      | Rendering loop                           | 6         |
| <b>2</b> | <b>Digging into API</b>                  | <b>6</b>  |
| 2.1      | Entity                                   | 6         |
| 2.2      | Scene                                    | 7         |
| 2.3      | Camera                                   | 8         |
| 2.4      | Programs and Shaders                     | 8         |
| 2.5      | Store functions                          | 9         |
| 2.6      | Model                                    | 10        |
| 2.6.1    | Direct methods                           | 10        |
| 2.6.2    | Builders                                 | 12        |
| 2.6.3    | Effects                                  | 13        |
| 2.7      | Particles                                | 13        |
| 2.8      | Skybox                                   | 16        |
| 2.9      | Terrain                                  | 16        |
| 2.9.1    | Voxel terrain                            | 16        |
| 2.9.2    | Mesh terrain                             | 17        |
| 2.10     | Text                                     | 17        |
| 2.11     | Light                                    | 17        |
| 2.11.1   | Spot light                               | 17        |
| 2.11.2   | Directional light                        | 17        |
| 2.11.3   | Point light                              | 18        |
| 2.12     | Logging                                  | 18        |
| 2.13     | Event system                             | 19        |
| 2.13.1   | Focus functions and private callbacks    | 20        |
| 2.13.2   | Register, unregister and clear functions | 20        |
| 2.14     | Easing                                   | 21        |
| 2.15     | Physics                                  | 21        |
| 2.16     | Extras                                   | 22        |
| <b>3</b> | <b>Customization</b>                     | <b>24</b> |
| 3.1      | Custom events                            | 24        |
| 3.2      | Custom callbacks                         | 24        |
| 3.3      | Custom easing                            | 25        |
| <b>4</b> | <b>Examples</b>                          | <b>26</b> |
| <b>5</b> | <b>Licenses</b>                          | <b>28</b> |

# 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 well. 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:

---

```
1 #include <fillwave/Fillwave.h>
2
3 using namespace fillwave;
4
5 int main(int argc, char* argv[]) {
6     /* Create OpenGL/OpenGLES context */
7     Engine* fillwave = new Engine(argc, argv);
8     /* Create scene */
9     /* enter rendering loop */
10    delete engine;
11    /* Delete OpenGL/OpenGLES context*/
12    exit(EXIT_SUCCESS);
13 }
```

---

## 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.

---

```
1 void insertResizeScreen(GLuint width, GLuint height);
2 void insertInput(KeyboardEvent& e);
3 void insertInput(MouseButtonEvent& e);
4 void insertInput(ScrollEvent & e);
5 void insertInput(CharacterEvent& e);
6 void insertInput(CharacterModsEvent& e);
7 void insertInput(CursorEnterEvent& e);
8 void insertInput(CursorPositionEvent& e);
9 void insertInput(TouchEvent& e);
10 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.

---

```
1
2 void ContextGLFW1::keyboardCallback(GLFWwindow* window,
3                                     int key,
4                                     int scancode,
5                                     int action,
6                                     int mods) {
7     /* Create an event data and fill it */
8     fillwave::framework::KeyboardEventData data;
9     data.action = action;
10    data.key = key;
11    data.mode = mods;
12    data.scanCode = scancode;
13
14    /* Create an event */
15    fillwave::framework::KeyboardEvent event(data);
16
17    /* insert an event */
18    mGraphicsEngine->insertInput(event);
19 }
```

---

## 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:

---

```
1 void ContextGLFW1::render() {
2     while (!glfwWindowShouldClose(mWindow)) {
3         GLfloat timeSinceLastFrameInSec, now = glfwGetTime();
4
5         timeSinceLastFrameInSec = now - mTimeExpired;
6         mTimeExpired = now;
7         mGraphicsEngine->draw(timeSinceLastFrameInSec);
8
9         /* We were writing to back buffer - make it visible */
10        glfwSwapBuffers(mWindow);
11
12        /* evaluate GLFW input events */
13        glfwPollEvents();
14    }
15 }
```

---

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

---

```
1 void captureFramebufferToFile(const std::string& name);
2 void captureFramebufferToBuffer(GLubyte* buffer,
3                                GLint* sizeInBytes,
4                                GLuint format,
5                                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 emitters to it. You can move, rotate, and scale each of them.

---

```
1 pEntity entity_parent = buildEntity();
2 pEntity entity_child = buildEntity();
3 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:

---

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

---

Getting a transformation matrix is also possible:

---

```
1 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 executed during scene change.

---

```
1 /* Build scene */
2 pISceneOrtographic scene0 = buildSceneOrtographic();
3 pIScenePerspective sceneP = buildScenePerspective();
4
5 /* Build camera */
6 pCameraOrtographic c0 = std::make_shared<CameraOrtographic>();
7 pCameraPerspective cP = std::make_shared<CameraPerspective>();
8
9 /* Attach camera */
10 scene0->setCamera(c0);
11 sceneP->setCamera(cP);
12
13 /* Attach scene */
14 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:

---

```
1  /* Camera with perspective projection */
2  pCameraPerspective cameraP = std::make_shared<CameraPerspective>
3      (glm::vec3(0.0,0.0,6.0), /* position */
4       glm::quat(), /* rotation */
5       glm::vec3(0.0,1.0,0.0), /* head up direction */
6       glm::radians(90.0), /* field of view angle */
7       screenWidth/screenHeight, /* screen ratio */
8       0.1, /* projection near plane */
9       1000.0); /* projection far plane */
10
11
12
13 /* Camera with ortographic projection */
14 gCameraOrthographic camera0 = std::make_shared<CameraOrtographic>
15     (glm::vec3(0.0,0.0,6.0),
16      glm::quat(), /* rotation */
17      -10.0f, /* x left culling */
18      10.0f, /* x right culling */
19      10.0f, /* y up culling */
20      -10.0f, /* y down culling */
21      0.1f, /* z near culling */
22      1000.0f); /* z far culling */
```

---

## 2.4 Programs and Shaders

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

---

```
1  /* Create loader module */
2  loader::ProgramLoader loader(gEngine);
3
4  /* Default program */
5  pProgram default = loader.getDefault();
6
7  /* Default program with animations support */
8  pProgram animation = loader.getDefaultBones();
```

---



## 2.5 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.

---

```
1  /* Store the shaders providing source file path */
2  pShader storeShaderFragment(const std::string& path);
3  pShader storeShaderVertex(const std::string& path);
4  pShader storeShaderGeometry(const std::string& path);
5  pShader storeShaderTessellationControl(const std::string& path);
6  pShader storeShaderTessellationEvaluation(const std::string& p);
7
8  /* Store the shaders providing the source directly */
9  pShader storeShaderFragment(const std::string& name,
10                             std::string& source);
11 pShader storeShaderVertex(const std::string& name, std::string&
12                             source);
13 pShader storeShaderGeometry(const std::string& name,
14                             std::string& source);
15 pShader storeShaderTessellationControl(const std::string& name,
16                             std::string& source);
17 pShader storeShaderTessellationEvaluation(const std::string&
18                             name, std::string& source);
19
20 pProgram storeProgram(const std::string& , std::vector<pShader>);
21
22 pTexture storeTexture (const std::string&, const GLuint&);
23 pTexture2DRenderableDynamic
24     storeTextureDynamic (const std::string&
25                             fragmentShaderPath);
26 pTexture3D storeTexture3D(const std::string& path,
27                             const std::string& path,
28                             const std::string& path,
29                             const std::string& path,
30                             const std::string& path);
31
32 pLightSpot storeLightSpot(glm::vec3, glm::vec4, pEntity);
33 pLightPoint storeLightPoint(glm::vec3, glm::vec4, pEntity);
34 pLightDirectional storeLightDirectional(glm::vec4, glm::vec3);
35
36 pText storeText(std::string, std::string, GLfloat, GLfloat, GLfloat);
37
38 pCursor storeCursor(pProgram, pTexture, GLfloat);
```

---

## 2.6 Model

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

### 2.6.1 Direct methods

---

```
1  /*
2   * When the appropriate map paths are available
3   * together with your model asset file.
4   */
5
6  pModel model = buildModel(engine, program, "model.obj");
7
8  pModel model = std::make_shared<framework::Model>(engine,
9      program, "model.obj");
10
11 /*
12  * When the appropriate map paths are available in your
13  * file and you want to draw Your custom shape derived
14  * from framework::Shape<core::VertexBasic>
15  */
16
17 framework::Sphere sphere(1.0,10.0,10.0);
18
19 pModel model = buildModel(engine,
20     program,
21     sphere,
22     diffuseMap,
23     normalMap,
24     specularMap,
25     material);
26
27 pModel model = std::make_shared<framework::Model>(engine,
28     program,
29     sphere,
30     diffuseMap,
31     normalMap,
32     specularMap,
33     material);
34
35
36
37
```

```

38
39
40  /*
41  * When we want to explicitly provide texture paths
42  * but still use the model asset from file.
43  */
44
45  pModel model = buildModel(engine,
46                          program,
47                          "model.obj",
48                          "relativePathToDiffuseMap",
49                          "relativePathToNormalsMap",
50                          "relativePathToSpecularMap");
51
52  pModel model = std::make_shared<framework::Model>(engine,
53                          program,
54                          "model.obj",
55                          "relativePathToDiffuseMap",
56                          "relativePathToNormalsMap",
57                          "relativePathToSpecularMap");
58
59  /*
60  * When we want to use previously created texture
61  * and material objects.
62  */
63
64  pModel model = buildModel(engine,
65                          program,
66                          "model.obj",
67                          diffuseMapTexture,
68                          normalMapTexture,
69                          specularMapTexture,
70                          material);
71  pModel model = std::make_shared<Model>(engine,
72                          program,
73                          "model.obj",
74                          diffuseMapTexture,
75                          normalMapTexture,
76                          specularMapTexture,
77                          material);

```

---

## 2.6.2 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                                     pProgram program,
7                                     diffusePath,
8                                     normalPath,
9                                     specularPath);
10
11  pModel m = builder1.build();
12
13  /* Second method */
14  BuilderModelExternalMaps builder1(engine);
15
16  pModel 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,
31                               modelPath,
32                               program,
33                               diffuseMap,
34                               normalsMap,
35                               specularMap,
36                               material);
37  pModel m = builder2.build();
38
39  /* Second method */
40
```

```

41 BuilderModelManual builder2 (engine);
42
43 pModel m = builder2.setModelPath(modelPath).
44         setProgram(program).
45         setDiffuseMapTexture(diffuseMap).
46         setNormalMapTexture(normalsMap).
47         setSpecularMapTexture(specularMap).
48         setMaterial(material).
49         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".

```

1 void setActiveAnimation(GLint animationID)
2 GLint getAnimations();

```

---

### 2.6.3 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.

```

1
2 pIEffect fog(new Fog());
3 pIEffect boost(new BoostColor(10.0));
4 pIEffect ccw(new ClockwiseDrawEffect());
5 pIEffect paint(new Painter());
6 pIEffect textureOnly(new TextureOnly());
7
8 model->addEffect(fog);

```

---

## 2.7 Particles

Particles system entry in Fillwave is in fact two (but powerfull) classes: **EmitterPointCPU** and **EmitterPointGPU**. The **EmitterPointGPU** particle emitter 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.

**EmitterPointCPU** emitter particles are precomputed on CPU. They are faster but the factors are less robust.

---

```
1  EmitterPointCPU::EmitterPointCPU(Engine* engine,
2                                  GLfloat emittingSurfaceRadius,
3                                  GLfloat robustness,
4                                  GLint  howMany,
5                                  glm::vec4 color,
6                                  glm::vec3 acceleration,
7                                  glm::vec3 velocity,
8                                  glm::vec3 distance,
9                                  pTexture texture,
10                                 GLfloat lifetimeInSec,
11                                 GLfloat pointSize,
12                                 GLboolean dephtest,
13                                 GLfloat alphaCutOff)
14
15  pEmitterPointGPU::EmitterPointGPU(Engine* engine,
16                                   GLfloat emittingSourceRate,
17                                   GLuint howMany,
18                                   glm::vec4 color,
19                                   glm::vec3 acceleration,
20                                   glm::vec3 startVelocity,
21                                   glm::vec3 robustnessVelocity,
22                                   glm::vec3 startPosition,
23                                   glm::vec3 robustnessPosition,
24                                   GLfloat startSize,
25                                   GLfloat lifetime,
26                                   pTexture texture,
27                                   GLenum blendingSource,
28                                   GLenum blendingDestination,
29                                   GLboolean dephtest,
30                                   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);
```

---

**EmitterPointCPU** emits particles using a round surface source. You can set the radius of this surface (**emittingSurfaceRadius**), and emitting **robustness**. **Robustness = 0** will make the particles flow perpendicular to the emitting 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

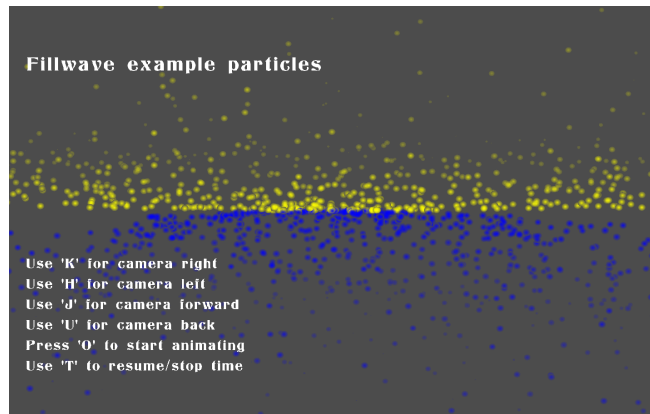


Figure 1: Particles with depth test active

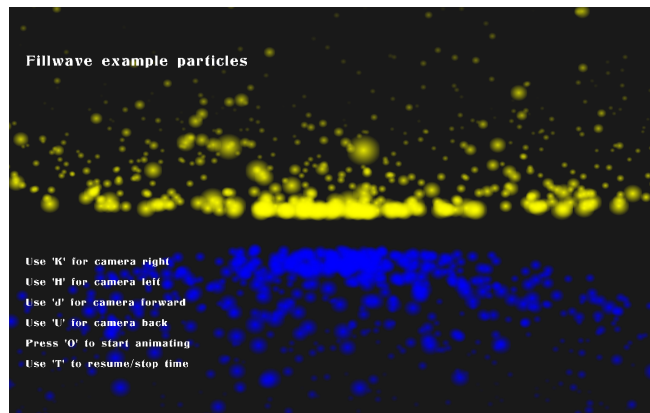


Figure 2: Particles without depth test active

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 provides additional feature to discard all pixels with alpha value less than alphaCutOff.

## 2.8 Skybox

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

---

```
1
2 pTexture3D texture=pTexture3D(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 pSkybox skybox = buildSkybox(engine,
9                               texture);
10 scene->setSkybox(skybox);
```

---

## 2.9 Terrain

Terrain in Fillwave can be generated using a voxel or quad chunks. These two methods provides complete mechanism for terrain generation.

---

```
1 pShader fs = engine->storeShaderFragment("default.frag");
2 pShader vs = engine->storeShaderVertex("default.vert");
3 pProgram program = buildProgram(fs + vs);
4 pTerrain terrain = buildTerrainVoxel(engine,
5                                     program,
6                                     "textures/test.png",
7                                     new terrain::MountainConstructor(),
8                                     5);
9 gScene->attach(terrain);
```

---

### 2.9.1 Voxel terrain

To create a voxel terrain You should implement a derivative **VoxelConstructor** class, and implement a **calculateActive** method. The method will take coordinates of a Voxel in VoxelChunk and decide if it should be Active or not. Active Voxels will be drawn. More information can be found in **example.terrain\_voxel** and **example.terrain\_quad** examples.



## 2.9.2 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.

## 2.10 Text

To create a 2D on screen text using ttf fonts You can use the **storeText** function.

```
1 pText text = engine->storeText( "Hello Fillwave", /* content */  
2                                "FreeMono", /* font to use */  
3                                -0.95, /*left bottom y start (-1,1)*/  
4                                -0.80, /*left bottom x start (-1,1)*/  
5                                100.0, /* text size */  
6                                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.

## 2.11 Light

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

### 2.11.1 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.

### 2.11.2 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).

### 2.11.3 Point light

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

## 2.12 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:

---

```
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)
```

---

## 2.13 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:

---

```
1 TimedCallback(GLfloat timeToFinish,
2               EasingFunction easing = eEasing::None);
3 TimedScaleCallback(pEntity entity,
4                   glm::vec3 normalizedScaleVec,
5                   GLfloat lifetime,
6                   EasingFunction easing);
7 TimedRotateCallback(pEntity entity,
8                    glm::vec3 axis,
9                    GLfloat angle,
10                   GLfloat lifeTime,
11                   EasingFunction easing);
12 TimedMoveCallback(pEntity entity,
13                  glm::vec3 endPosition,
14                  GLfloat lifeTime,
15                  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**.

### 2.13.1 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.

---

```
1
2 void setFocus(eEventType eventType, pEntity entity);
```

---

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

---

```
1 void Entity::attachHierarchyCallback(Callback* c);
2 void Entity::attachPrivateCallback(Callback* c);
3 void Entity::detachHierarchyCallback(Callback* c);
4 void Entity::detachPrivateCallback(Callback* c);
```

---

### 2.13.2 Register, unregister and clear functions

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

---

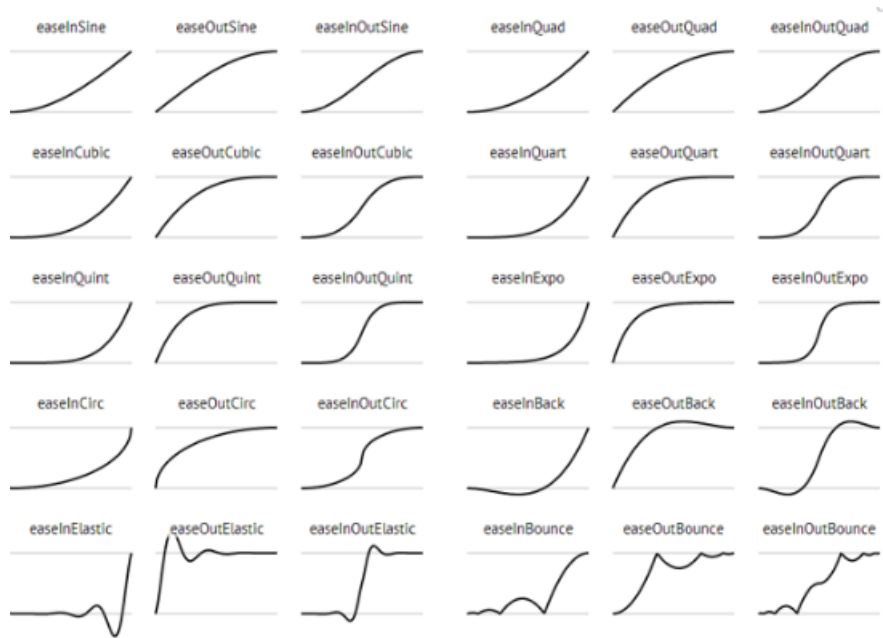
```
1 void registerCallback(Callback* c);
2 void unregisterCharacterModsCallback(Callback* c);
```

---

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

## 2.14 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**:



## 2.15 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 explicitly.

---

```
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:

---

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

---

## 2.16 Extras

To change the background color use:

---

```
1 void Engine::configureBackgroundColor (glm::vec3 color);
```

---

To apply the time factor to in Fillwave engine use:

---

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

---

To get the current executable directory use:

---

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

---

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

---

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

---

empty or not valid font name will disable the FPS counter.

To set/reset reset file logging use:

---

```
1 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:

---

```

1  /* [R]_[G]_[B].color - for color texture */
2  /* [R]_[G]_[B].checkboard - for color checkboard texture */
3  /* "" - Black texture */
4
5  pModel model = buildModel(engine,
6      programDefault,
7      "model.obj",
8      "255_0_0.checkboard", /* Red checkboard diffuse texture */
9      "", /* black normal map */
10     "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.

---

```

1  enum class eDebuggerState {
2      lightsSpot,
3      lightsSpotColor,
4      lightsSpotDepth,
5      lightsPoint,
6      lightsPointDepth,
7      lightsPointColor,
8      pickingMap,
9      off,
10     toggleState
11 };
12
13 void Engine::configureDebugger(eDebuggerState state);

```

---

## 3 Customization

### 3.1 Custom events

---

```
1
2 namespace fillwave {
3 namespace actions {
4 struct NewEventData {
5     int data;
6     const eEventType type = eEventType::custom0; /* event ID */
7 };
8
9 class NewEvent: public Event<NewEventData> {
10 public:
11     NewEvent();
12     virtual ~NewEvent();
13 };
14 } /* actions */
15 } /* fillwave */
```

---

### 3.2 Custom callbacks

---

```
1 namespace fillwave {
2 namespace actions {
3
4 class NewEngineCallback: public EngineCallback {
5 private:
6     float mMaximimData;
7     void sayHello() {FLOG_USER("Hello event");};
8 public:
9     NewEngineCallback(eEventType eventType, float data);
10    NewEngineCallback(float data);
11
12    virtual ~NewActionCallback();
13    void perform (Engine* engine, EventType* event);
14 };
15
16 } /* actions */
17 } /* fillwave */
```

---



### 3.3 Custom easing

#### TimedMoveCallbackCustom.h

---

```
1 namespace fillwave {
2 namespace actions {
3
4 class TimedMoveCallbackCustom: public
    fillwave::framework::TimedMoveCallback {
5 public:
6     TimedMoveCallbackCustom(pEntity entity,
7                             glm::vec3 endPosition,
8                             GLfloat lifeTime);
9     virtual ~TimedMoveCallbackCustom();
10    GLfloat easeCustom(GLfloat progress);
11 };
12 } /* actions */
13 } /* fillwave */
```

---

#### TimedMoveCallbackCustom.cpp

---

```
1 namespace fillwave {
2 namespace actions {
3
4 TimedMoveCallbackCustom::TimedMoveCallbackCustom(pEntity entity,
5                                                    glm::vec3 endPosition,
6                                                    GLfloat
7                                                    lifeTime):TimedMoveCallback(entity,
8                                                    endPosition,lifeTime,
9                                                    eEasing::Custom) {
10 }
11
12 TimedMoveCallbackCustom::~TimedMoveCallbackCustom() {
13 }
14
15 GLfloat TimedMoveCallbackCustom::easeCustom(GLfloat progress) {
16     /* You custom easing function goes here. For example: */
17     return QuinticEaseIn(progress)*QuinticEaseIn(progress);
18 }
19
20 } /* actions */
21 } /* fillwave */
```

---

## 4 Examples

Basic example You can find below:

---

```
1
2  /* Camera */
3  pCameraPerspective camera = pCameraPerspective ( new
4      CameraPerspective(glm::vec3(0.0,0.0,16.0),
5                          glm::quat(),
6                          glm::radians(90.0),
7                          1.0,
8                          0.1,
9                          1000.0));
10
11 /* Programs */
12 ProgramLoader loader(ContextGLFW::mGraphicsEngine);
13 pProgram program = loader.getDefault();
14
15 /* Models */
16 pModel sphere = buildModel(ContextGLFW::mGraphicsEngine,
17                             program, "meshes/sphere.obj", "255_255_255.color");
18
19 scene->attach(sphere);
20 scene->setCamera(camera);
21 ContextGLFW::mGraphicsEngine->setCurrentScene(scene);
22
23 mContext.render();
24 delete ContextGLFW::mGraphicsEngine;
25 exit(EXIT_SUCCESS);
26 }
```

---

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

```
1  /*****
2  * Fillwave C++11 graphics Engine
3  * Copyright (C) 2015 Filip Wasil
4  * All Rights Reserved.
5  * This library is available free of charge for any
6  * commercial
7  * or non-commercial use. However, You are obligated
8  * to put
9  * a clearly visible information in Your license
10 * agreement
11 * that Your Software uses Fillwave library. Fillwave
12 * uses
13 * few external libraries and their licenses are
14 * written below.
15 * If You are interested in extra support, extra
16 * features
17 * or cooperation I look forward to hearing from You.
18 *
19 *      Filip Wasil      fillwave@gmail.com
20 */
21
22 /* OpenGL
23 http://www.opengl.org/
24
25 * AssImp library
26 Open Asset Import Library (assimp)
27
28 Copyright (c) 2006-2012, assimp team
29 All rights reserved.
30
31 Redistribution and use of this software in source and
32 binary forms,
33 with or without modification, are permitted provided
34 that the
35 following conditions are met:
```

29 \* Redistributions of source code must retain the above  
30 copyright notice, this list of conditions and the  
31 following disclaimer.  
32  
33 \* Redistributions in binary form must reproduce the  
34 above  
35 copyright notice, this list of conditions and the  
36 following disclaimer in the documentation and/or  
37 other  
38 materials provided with the distribution.  
39  
40 \* Neither the name of the assimp team, nor the names  
41 of its  
42 contributors may be used to endorse or promote  
43 products  
44 derived from this software without specific prior  
45 written permission of the assimp team.  
46  
47 THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS  
48 AND CONTRIBUTORS  
49 "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,  
50 INCLUDING, BUT NOT  
51 LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY  
52 AND FITNESS FOR  
53 A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT  
54 SHALL THE COPYRIGHT  
55 OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,  
56 INDIRECT, INCIDENTAL,  
57 SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES  
58 (INCLUDING, BUT NOT  
59 LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR  
60 SERVICES; LOSS OF USE,  
61 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER  
62 CAUSED AND ON ANY  
63 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT  
64 LIABILITY, OR TORT  
65 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY  
66 WAY OUT OF THE USE

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

54

55

56

57

\*\*\*\*\*

58

59 AN EXCEPTION applies to all files in the  
60 ./test/models-nonbsd folder.

61 These are 3d models for testing purposes, from  
62 various free sources

63 on the internet. They are - unless otherwise stated -  
64 copyright of

65 their respective creators, which may impose  
66 additional requirements

67 on the use of their work. For any of these models, see  
68 <model-name>.source.txt for more legal information.

69 Contact us if you  
70 are a copyright holder and believe that we credited  
71 you improperly or  
72 if you don't want your files to appear in the  
73 repository.

67

68

69

\*\*\*\*\*

70

71 Poly2Tri Copyright (c) 2009-2010, Poly2Tri  
72 Contributors

73 <http://code.google.com/p/poly2tri/>

73

74 All rights reserved.

75 Redistribution and use in source and binary forms,  
76 with or without modification,

77 are permitted provided that the following conditions  
78 are met:

77

78 \* Redistributions of source code must retain the  
79 above copyright notice,

79 this list of conditions and the following disclaimer.

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

83 \* Neither the name of Poly2Tri nor the names of its  
contributors may be  
84 used to endorse or promote products derived from  
this software without specific  
85 prior written permission.

86

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

98 \* FreeType 2 library (FTL licence)  
99 <http://www.freetype.org/>

100

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

102     irrevocable right and license to use, execute,  
103     perform, compile,  
104     display, copy, create derivative works of,  
105     distribute and  
106     sublicense the FreeType Project (in both source and  
107     object code  
108     forms) and derivative works thereof for any  
109     purpose; and to  
110     authorize others to exercise some or all of the  
111     rights granted  
112     herein, subject to the following conditions:  
113  
114     o Redistribution of source code must retain this  
115     license file  
116     ('FTL.TXT') unaltered; any additions, deletions  
117     or changes to  
118     the original files must be clearly indicated in  
119     accompanying  
120     documentation. The copyright notices of the  
121     unaltered,  
122     original files must be preserved in all copies  
123     of source  
124     files.  
125  
126     o Redistribution in binary form must provide a  
127     disclaimer that  
128     states that the software is based in part of the  
129     work of the  
130     FreeType Team, in the distribution  
131     documentation. We also  
132     encourage you to put an URL to the FreeType web  
133     page in your  
134     documentation, though this isn't mandatory.  
135  
136     These conditions apply to any software derived from  
137     or based on  
138     the FreeType Project, not just the unmodified files.  
139     If you use



124     our work, you must acknowledge us. However, no fee  
125         need be paid  
126     to us.

127     \* GLEW library  
128     <http://glew.sourceforge.net/>

129  
130     GLEW is originally derived from the EXTGL project by  
131         Lev Povalahev. The source  
132     code is licensed under the Modified BSD License, the  
133         Mesa 3-D License  
134     (MIT License), and the Khronos License (MIT License).  
135     The automatic code  
136     generation scripts are released under the GNU GPL.

137     \* Lanyon  
138     Released under MIT License  
139     Copyright (c) 2014 Mark Otto.

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

152  
153     The above copyright notice and this permission notice  
154     shall be included in all copies or substantial  
155     portions of the Software.

156  
157     THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF  
158     ANY KIND,

```

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

```

```

155
156 * fontGenerator
157 /*****
158 | OpenGL 4 Example Code.
159 |
160 | Accompanies written series "Anton's OpenGL 4
161 |   Tutorials"
162 |   Email: anton at antongerdelan dot net
163 |   First version 5 Feb 2014
164 |
165 | Copyright Dr Anton Gerdelan, Trinity College
166 |   Dublin, Ireland.
167 | See individual libraries and assets for respective
168 |   legal notices
169 |
170 * Sean Barrett's public domain stb_image and
171   stb_image_write libraries
172 http://nothings.org/
173
174 /* stbiw-0.92 - public domain -
175    http://nothings.org/stb/stb_image_write.h
176    writes out PNG/BMP/TGA images to C stdio - Sean
177    Barrett 2010
178
179    no warranty implied; use at
180    your own risk

```

```

172 Before including,
173

```

```
174
175     #define STB_IMAGE_WRITE_IMPLEMENTATION
176
177 in the file that you want to have the implementation.
178
179
180 ABOUT:
181
182     This header file is a library for writing images to
183     C stdio. It could be
184     adapted to write to memory or a general streaming
185     interface; let me know.
186
187     The PNG output is not optimal; it is 20-50% larger
188     than the file
189     written by a decent optimizing implementation. This
190     library is designed
191     for source code compactness and simplicity, not
192     optimal image file size
193     or run-time performance.
194
195 */
```

---