**gluxEngine Framework Manual**

**by Juraj Vanek, v0.9**

# Table of Contents

[Table of Contents 2](#_Toc308972861)

[1 Introduction 3](#_Toc308972862)

[2 Features 3](#_Toc308972863)

[2.1 Hardware&software requirements 4](#_Toc308972864)

[3 Basic usage 4](#_Toc308972865)

[3.1 Including into project 4](#_Toc308972866)

[3.2 Initialization 4](#_Toc308972867)

[3.3 Setting up camera 5](#_Toc308972868)

[3.4 Adding objects 5](#_Toc308972869)

[3.5 Adding materials and textures 7](#_Toc308972870)

[4 Advanced usage 8](#_Toc308972871)

[4.1 Texture formats and blending 8](#_Toc308972872)

[4.2 Writing custom shaders 8](#_Toc308972873)

[4.3 Advanced effects 8](#_Toc308972874)

# Introduction

This framework can be used as a basis of various applications requiring real-time rendering output. Right now it is only as source code, but it is possible to create some library from it. It is being developed for several years and new features are added as needed, so feel free to add/change anything, but let me know after change is made.

# Features

Basically, it is a framework with support of newest OpenGL 3/4 features. It has built-in object and material manager as well as shader generator for basic material types.

**What does it support now**

* OpenGL 3 and 4 core profiles, no fixed pipeline (only for fonts)
* Scene manager –scene settings (resolution, camera…), display modes (windowed, fullscreen), multisampling
* Two camera types: FPS camera and orbiting camera
* Unlimited light count, per-pixel or per-vertex lighting
* Built-in basic objects (cube, sphere, torus…and of course teapot)
* 3DS objects and whole scenes loading, basic object manipulation (move, rotate, scale)
* Shader generator for simple materials (with Phong lighting), support for multitexturing, bump mapping, displace mapping, env mapping, texture mixing modes
* Support for TGA textures with various filtering (linear, anisotropic, multisampled)
* Shadows through shadow maps, for spot and omnidirectional lights
* Custom shaders – for whole OpenGL 4 pipeline (vertex, tess control, tess eval, geometry, fragment)
* Multiple render targets, color, depth or normal, multisampled framebuffer
* Built-in HDR rendering pipeline with ambient occlusion
* Polygon mesh tessellation
* Geometry instancing
* Basic 2D font drawing on the screen
* Support for GPU computing through OpenCL (classes for setting up and running CL kernels)
* …

**What it does not support, but it is planned in the future:**

* Scene graph – right now there is no connection between objects
* Support for animations (keyframes, interpolation…)
* Stereo3D support
* Picking

**What libraries does it use:**

* OpenGL
* SDL – for window management and user control. I am using this and not WinAPI because I was trying to keep the framework portable on various platforms (Windows, Linux, Mac…)
* GLEW – for OpenGL extensions
* GLM – OpenGL math library
* Lib3DS for loading 3DS models
* OpenCL

## Hardware&software requirements

* **Graphics card**: with OpenGL 3.3 support (DirectX 10 generation) – *Radeon* 4xxx, 5xxx, 6xxx, 7xxx, *GeForce* 8xxx, 9xxx, 2xx, 4xx, 5xx, Intel HD 3xxx (not tested, better use discrete GFX from AMD or nVidia)
  + Recommended: GFX with OpenGL 4.2 support (DirectX 11 generation) – *Radeon* 5xxx, 6xxx, 7xxxx, *GeForce* 4xx, 5xx
* **CPU, RAM**: any decent processor with enough RAM to run your OS
* **OS**: Windows XP, Vista, 7, Linux (libraries not included!), MacOS (untested, but it should be possible to port it on this platform)
* **Software**:
  + Newest graphics drivers
  + MS Visual Studio 2008 (2010 recommended)
  + *Recommended*: OpenCL SDK (for AMD, it’s AMD APP, for nVidia it’s CUDA toolkit)

# Basic usage

Framework contains lists to all objects, materials and textures used in the scene. Objects are coupled with materials and one material can be assigned to any objects count (and vice-versa). Material shaders are generated dynamically based on material properties. In the following section, we will show how to set up basic scene as well as code listing used to set up the scene[[1]](#footnote-1).

## Including into project

1. Checkout newest version from SVN (<https://hpcg.purdue.edu/svn/gluxengine>)
2. Copy folder src/glux\_engine into your project source folder
3. Use src/main.cpp file as a framework for your application (it is needed to initialize application window and all necessary OpenGL settings)
4. Compile & run, libraries are included (set your project library path to \LIB folder)

## Initialization

Initialization of the scene is happening inside function bool InitScene(int resx, int resy) and steps are as follows:

1. Create instance of class TScene: s = new TScene();
2. Initialize scene. Initialization is done in two steps, at the beginning and at the end of InitScene function. First, init scene with:

bool PreInit(resx, resy, near\_p, far\_p, fovy, msaa)

*resx,resy* – screen resolution

*near\_p, far\_p* – near/far clipping planes

*fovy* – field of view angle

*msaa* – antialiasing level, 0 – 16 (32 on newest graphic cards)

1. Set up camera (section 3.3)
2. Add objects, materials, textures etc… All functions are throwing exceptions, so include this code in the try-catch environment (sections 3.4 and 3.5)
3. Couple objects with materials
4. Set post-processing effects (HDR, SSAO…) (section 4.3)
5. After everything is added into scene, call bool PostInit() as post-initialization. This call creates all material shaders, shadow textures, initialize buffers etc. Returns true or false.

## Setting up camera

There are two types of camera: **orbit camera**, which can be controlled by mouse only by rotating around the beginning of scene axis system, and **FPS camera**, which uses keys W-S-A-D and mouse to move and look with camera. At default, camera is always at the position [0,0,0] and no rotation. We can move, rotate and change look and type of the camera through functions:

void MoveCamera(GLfloat wx, GLfloat wy, GLfloat wz)

void MoveCameraAbs(GLfloat wx, GLfloat wy, GLfloat wz)

void LookCameraAt(GLfloat wx, GLfloatwy, GLfloat wz)

*wx, wy, wz* – new position (increment) of the camera (or looking point)

void RotateCamera(GLfloat angle, GLint axis)

void RotateCameraAbs(GLfloat angle, GLint axis)

*angle* – new rotation angle (or increment) of the camera

*axis* – rotational axis (can be A\_X, A\_Y, A\_Z)

void SetCamType(int type)

*type* – ORBIT or FPS

If we want some custom (or more complex) camera transformations, it is possible to switch default camera off, but the we need to manually update uniform buffers with camera transformation matrix in all shaders.

## Adding objects

Objects can be added using built-in primitives (cube, plane, sphere, torus) or loaded from 3DS file. Very important is object name, since it is used as an index to access the object for further modifications (e.g moving).

To create object from primitive, call:

void AddObject(const char \*name, int primitive, GLfloat size, GLfloat height,

GLint sliceX, GLint sliceY)

*name* – object name (must be unique within the scene)

*primitive* – primitive type. Can be: CUBE, PLANE, STRIP\_PLANE, SPHERE or TORUS

*size* – primitive size, based on the type:

* CUBE: length of edge of the cube
* PLANE: width of plane
* SPHERE: sphere radius
* TORUS: outer radius

*height* – primitive height, based on the type:

* CUBE: no effect
* PLANE: height of plane
* SPHERE: no effect
* TORUS: inner radius

*sliceX, sliceY* – mesh subdivision in X or Y direction. Has no effect on CUBE

To add object from external 3DS file, call

void AddObject(const char \*name, const char\* file);

*name* – object name (must be unique within the scene

*file* – path to 3DS file

After object is loaded, it is also added into object cache so if there will be request for loading the same objects, only reference from the cache will be used (no data duplication occurs to save memory).

To position objects in the scene, there are two possible approaches: either change position absolutely or by some increment (like in camera control in section 3.3). The same is done with object rotation and scale:

void MoveObj(const char\* name, GLfloat wx, GLfloat wy, GLfloat wz)

void MoveObjAbs(const char\* name, GLfloat wx, GLfloat wy, GLfloat wz)

void RotateObj(const char\* name, GLfloat angle, GLint axis)

void RotateObjAbs(const char\* name, GLfloat angle, GLint axis)

void ResizeObj(const char\* name, GLfloat sx, GLfloat sy, GLfloat sz)

*name* – object name (must be unique within the scene

*wx, wy, wz –* new position (or new increment to old position)

*sx, sy, sz* – scaling values for all axis (like glScalef)

*axis* – rotational axis in 3D (symbols: A\_X, A\_Y, A\_Z)

## Adding materials and textures

Similar to objects, materials are also indexed by their names, so they must be unique. Based on material parameters, in post-initialization phase, material shader is generated by the framework. Optionally, it is possible to use custom shaders (more about writing custom shaders and how to connect them in framework in section 4.2).

At first, we need to create material:

void AddMaterial(const char\* name, glm::vec3 amb, glm::vec3 diff, glm::vec3 spec,

GLfloat shin, GLfloat reflect, GLfloat transp, GLint lm)

*name* - material name (must be unique)

*amb* - ambient color (as RGB vector)

*diff* - diffuse color (as RGB vector)

*spec* - specular color (as RGB vector)

*shin* – material shininess (0.0 - most shiny, 128.0 - least shiny)

*reflect* – reflection factor (0.0 – no reflections, 1.0 – perfect mirror)

*transp* - material transparency (0.0 – opaque, 1.0 – fully transparent)

*lm* - light model (can be PHONG,GOURAUD,NONE)

After this creation, we can add textures to material. We can add as many textures as we want, only limit is hardware texture unit count in OpenGL implementation (typically 64). We can stack textures and blend them together in different modes. To add texture, call

void AddTexture(const char \*name, const char \*file, GLint textype, GLint texmode,

GLfloat intensity, GLfloat tileX, GLfloat tileY,

bool mipmap, bool aniso);

*file* - external texture file (.tga)

*textype* - texture type, can be BASE, ENV, BUMP, PARALLAX, DISPLACE, CUBEMAP, CUBEMAP\_ENV, ALPHA, SHADOW, SHADOW\_OMNI, RENDER\_TEXTURE, or RENDER\_TEXTURE\_MULTISAMPLE (see section 4.1 for more info)

*texmode* - texture blending mode, can be ADD, MODULATE, DECAL, BLEND, REPLACE (see section XX for more info)

*intensity* – blending intensity (0.0 - 1.0)

*tileX, tileY* - horizontal/vertical texture tiles

*mipmap* - should we generate mipmaps for texture?

*aniso* - should we use anisotropic filtering?

*cache* - whether texture should be cached (in some cases, caching is not desirable)

If texture was already loaded, we will use stored texture from texture cache (similar to object cache, see in section 3.4).

After all materials and objects are created, it is time to couple them together:

void SetMaterial(const char\* obj\_name, const char \*mat\_name);

*obj\_name* – object name as stored in the object list

*mat\_name* – material name

If there is no object/material with that name, nothing will happen and framework will show a warning.

# Advanced usage

## Texture formats and blending

## Writing custom shaders

## Advanced effects

1. Note that framework is still in development so function names and parameters can be different. Refer to the newest version of Doxygen-generated class and function reference [↑](#footnote-ref-1)