# OpenGL scene

Document version: v1.1

Dictionary:

* TBI – to be implemented
* not implemented – means the current status, usually planned for the future

Purpose of the project: create a 3D scene, create some toolset in C++ above simple C function calls.

*At the current state, even planned features are subject to change. They are written down for clarity.*

Planned features:

* Loading models – works but requires revision (there are several problems with loading different models and different file types)
* Simple “manual” models (code-described rather than model-file described) – works but requires its own class (plain code for now which doesn’t look good)
* Lighting system – works for simple models, requires an update for loadable models
* Camera – implemented
* Keyboard input (moving in any direction) – implemented
* Mouse input – implemented partially, mouse selection TBI
* Profiler\* – basic functionality
* GUI – not implemented
* Selectable and movable objects – not implemented
* World/map/scene class which saves all object positions – not implemented

\*Profiler. For example, it is used to count loading times.

Optional features:

* proper support for multiple model formats – not implemented
* collision – not implemented
* shadows – not implemented
* different graphic effects like depth of field, sharpness etc. – not implemented

## Dependencies

* GLFW (OpenGL API)
* glew (OpenGL API)
* glm (math)
* SOIL (texture loading)
* Assimp (model loading)

## Classes description

Those classes simply abstract OpenGL API:

1. VertexBuffer (VBO)
2. VertexBufferLayout
3. IndexBuffer (IBO)

Not required if glDrawArrays is used. Required for glDrawElements.

1. VertexArray (VAO)
2. Shader – parses shaders, compiles them

Modes:

* Vertex + fragment in one file (.glsl) – this mode requires *#shader vertex, #shader fragment* to be added before each shader (main mode for now)
* 1 per file (.frag, .vert)

1. Window – abstracts OpenGL window. Also responsible for callback functions (i.e. user input, camera movement for example)
2. Renderer – abstracts global drawing (and related) operations (like switching drawing mode to WireFrame)

Models loaded with assimp – they draw themselves. Simple models can’t draw themselves for now, so they are drawn by Renderer.

1. Model – abstracts model loading process (SOIL + Assimp)
2. Mesh – model mesh
3. Material – materials, textures too
4. Camera – responsible for camera view and movement
5. Light – responsible for lighting. Must have corresponding shaders to work
6. Settings – contains OpenGL and project settings

## Implementation notes

### Draw

All Draw operations perform corresponding shader.Bind() (glUseProgram internally).

This approach allows to stack all Draw operations in one place, instead of having Bind/set view-projection/transformations strict sequence. So, you can do all the preparations and then only call Draw methods of all objects.

Upside: more dynamic code structure

Downside: additional glUseProgram calls. Could be expensive.

*P.S. Is subject to change*

## Optimization notes

This section contains thoughts on optimization of 3D application. What I’ve found on the internet (mostly) or what I think myself.

1. Store multiple objects in one memory buffer whenever possible. Primarily when the objects are identical. Having multiple small buffers is much slower (TBI).