3D Edition

February 2017 – Based on Qt 5.8

Contents

Objectives

› Know different options of using 3D graphics in Qt

› Learn 3D-related Qt modules

› To be able to start or continue serious 3D graphics development with Qt framework using C++ or Qt Quick

› Any questions at any point – please do not hesitate to ask!

OpenGL

Contents

OpenGL

- Abstract API for high performance 3D(2D) rendering with a programmable graphics processor
  - Platform independent
  - State machine, where each command is executed in a state

  - No window or user interaction support

- In desktops, OpenGL 3 or later is typically supported

- In embedded platforms, OpenGL ES 2 or 3 are typically supported

- OpenGL ES
  - A reduced version of OpenGL meant for embedded devices
  - Precision qualifiers in shaders (lowp, mediump, highp)

OpenGL Versions

- OpenGL 1.3
  - Fixed pipeline, everything defined with flags and states
  - Chip vendor specific extensions – some programmable features
    - For example, ARB_debug_output => added to OpenGL Core profile in 4.3
- OpenGL 2
  - Programmable pipeline
  - Vertex and fragment shaders introduced => many functions became obsolete
- OpenGL 3
  - Support to create primitives dynamically using geometry shader
- OpenGL 4
  - Support for tessellation

What Is Supported in My Platform?

- In Qt, you may explicitly set, which OpenGL functions you want to use
  - `QOpenGLFunctions` – Cross-platform API to OpenGL ES 2 functions
  - `QOpenGLFunctions_4_5_Core`

- Subclass from `QOpenGLFunctions` and initialize the functions

```
class GLWindow : public Qwindow, protected QOpenGLFunctions
// Create context and make it current
initializeOpenGLFunctions();
```

  - Provides common API for OpenGL 2 and OpenGL ES 2
  - Sub-class `QOpenGLExtraFunctions` provide the common API for OpenGL 3 and OpenGL ES 3

- Another option is just to request the functions

```
QOpenGLFunctions_3_3_Core* functions = 0;
functions = QOpenGLContext::currentContext()->versionFunctions<QOpenGLFunctions_3_3_Core>();
if (!functions) { // No chance
}
```

What Is Supported in My Platform?

- Easy to check availability of a feature
  - `QOpenGlFunction::hasOpenGLFeature(QOpenGLFunctions::BlendColor)`
  - `Shaders, Buffers, Framebuffers, Multitexture`

- `QQmlContext` may be used to check availability of an extensions in the library
  - `QQmlContext::hasExtension(const QByteArray &extension)`

OpenGL Libraries

- Window system-specific libraries
  - Allows using OpenGL in a window system –based window
  - GLX, WGL (Windows), CGL (OS X), EGL (cross-platform interface between OpenGL and native window system)

- Cross-platform libraries
  - GLFW - supports context, window, and input event management
  - OpenGLUT (Utility toolkit) – open-source, cross-platform interface between a window system and OpenGL
    - Provides more complicated primitives: cubes, primitives, Utah (Newell teapot)
  - GLEW (GL Extensions Wrangler) – streamlines dealing with OpenGL versions and their extensions

  - QtGui ☺
    - Window, context, input management and plenty of more

OpenGL and Window System

- Create a window system-specific window for OpenGL
- May support sharing resources between contexts

- Window and context creations are not part of OpenGL specification

- Context
  - State machine, storing all rendering related data
  - Color
  - Viewing and protection matrices
  - Drawing modes
  - Lighting
  - Materials
  - Anti-aliasing level

Qt Gui

- In Qt, there is always a window
  - No matter if widgets, OpenGL, Qt Quick, or web engine is used
- Possible to sub-class `QWindow`
  - Does not provide any render functions – must be implemented by the developer
  - Before rendering, an OpenGL context must be created and made current – API available in `QOpenGLcontext`
  - Window re-paint is requested with `QWindow::update()` in an event handler
    - For example, when an animation timer expires or any mouse, touch, key event changes the model
  - Custom event handler implemented for `QEvent::UpdateRequest` event, which calls the `render()` function
- More convenient to use `QOpenGLWidget` or `QOpenGLWindow`
  - No need to explicitly manage the context object
  - Provide common API: `initializeGL(),paintGL(),resizeGL()`

QOpenGLWidget vs. QOpenGLWindow

- Rather new classes, introduced in Qt 5.4
- Similar APIs in both classes
  - `initializeGL()` – initialize your resources
  - `resizeGL()` – set the viewport or projection, when window size changes
  - `paintGL()` – render
- Both render to FBO
- Both supports partial rendering with `QPainter`

QOpenGLWidget vs. QOpenGLWindow

- Function `update()` will request for a repaint – `paintGL()` function called
  - Avoid timers – use `frameSwapped()` signal instead
- OpenGL context created automatically and made current
- Both allow painting with `QPainter` and partial updates, made with `QPainter`
  - Easy way to have 2D UI controls in 3D scene
- `QOpenGLWindow` does not have dependencies on widgets => slightly faster
- Note that `QOpenGLWidget` is painted before other widgets (no matter of the stacking order)
  - You may use `setWindowFlags( Qt::WindowStaysOnTopHint)` to change that

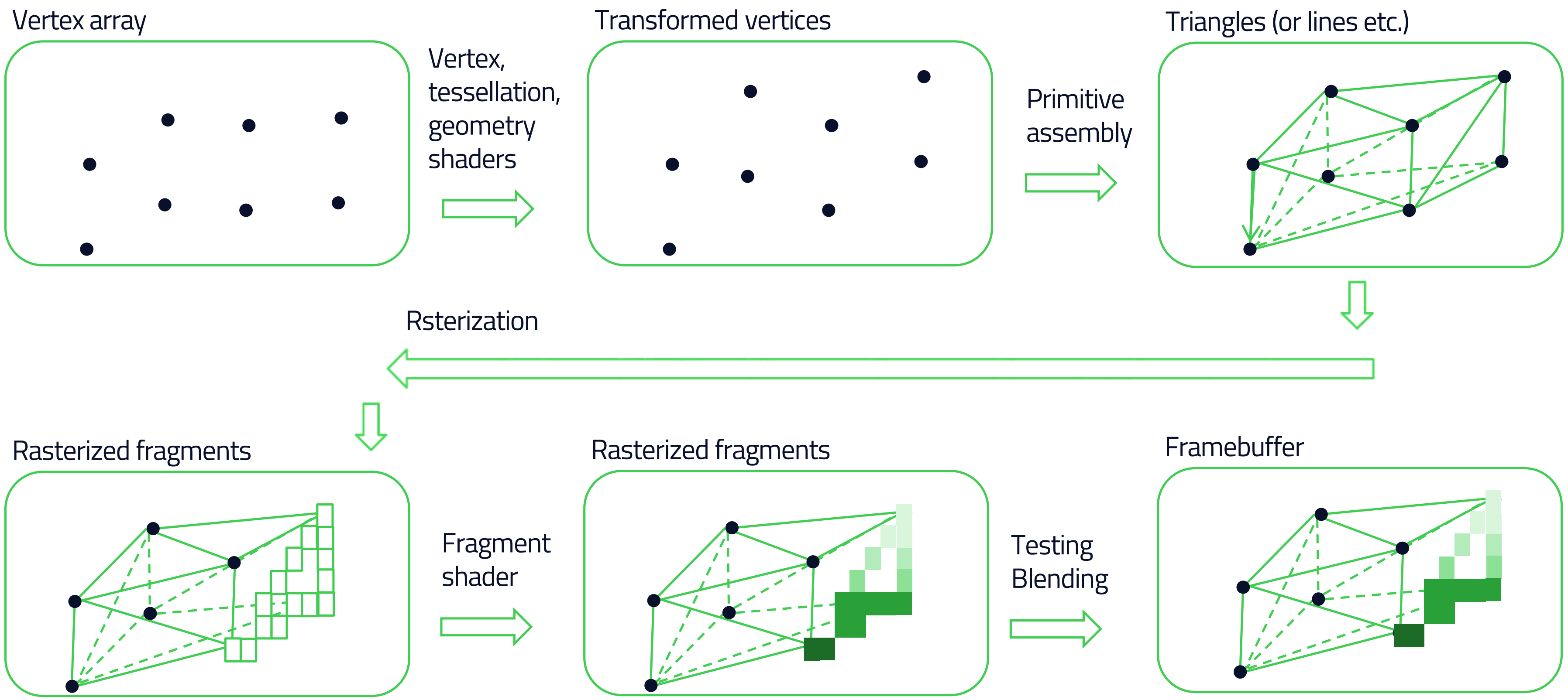
OpenGL Contexts – QOpenGLContext

- Managed by QOpenGLWidget and QOpenGLWindow
  - Access using the `context()` member
- Associates the surface with the context
  - Context may be shared by all windows
  - Context may be shared by multiple threads
    - Use `QObject::moveToThread()`) and then make context current before using gl functions
- Set the surface format with `QSurfaceFormat`
  - Render buffers: color, stencil, sample, alpha buffers
  - Swap behavior and interval

What Have We Achieved So Far?

```
class OpenGLWindow : public QOpenGLWindow
{
    Q_OBJECT
public:
    OpenGLWindow(QOpenGLWindow *parent = 0) {
        QSurfaceFormat format;
        format.setAlphaBufferSize(8);
        format.setMajorVersion(4);
        format.setMinorVersion(3);
        format.setSamples(8);
        format.setSwapInterval(10);
        setFormat(format);
    }
    ~OpenGLWindow();
protected:
    void initializeGL() Q_DECL_OVERRIDE { }
    void paintGL() Q_DECL_OVERRIDE { }
    void resizeGL(int, int) Q_DECL_OVERRIDE { }
};
```


OpenGL Rendering Pipeline



OpenGL Data Types

Data Type	C Type	Data Type	C Type
GLboolen	unsigned char	GLdouble	double
GLbyte	char	GLbitfield	unsigned int
GLchar	char	GLfloat	float
GLshort	short	GLclampx	int
GLushort	unsigned short	GLclampf GLclampd	float [0, 1] double [0, 1]
GLint	int	GLsizei	int
GLfixed	int	GLintptr	int
GLsizei	int	GLsizeptr	int
GLenum	unsigned int	GLvoid	void

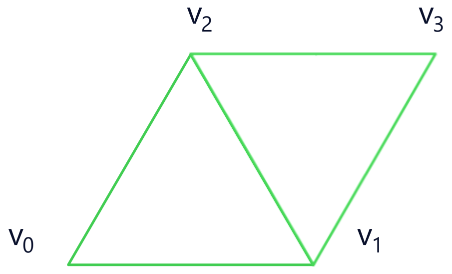
OpenGL ES does not support 64-bit data types

OpenGL Object Model

- Data is provided to OpenGL with objects
- Object names (handles) are opaque (managed internally by OpenGL) `GLuint` values
- Handles are created using `glGen*` functions
  - `glGenBuffers(GLsizei nofBufferObjectNames, GLuint *arrayOfObjectNames)`
  - `glGenVertexArrays() // Since OpenGL 3`
  - `glGenTextures()`
- Objects are used by binding their names to OpenGL target (`GL_ARRAY_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_TEXTURE_2D`) using `glBind*` functions
  - `glBindBuffer(GLenum target, GLunit bufferName)`
  - `glBindVertexArray(GLunit arrayName) // Since OpenGL 3`
  - `glBindTexture()`

Buffer Objects

- Vertex buffer object (VBO)
  - Contains vertices and vertex attributes
  - Location, texture coordinates, normals, colors
  - Rather easy to convert from *Wavefront .obj* format
- Index buffer object (`GL_ELEMENT_ARRAY_BUFFER`)
  - Array of indices (index = a pointer to a single vertex in the vertex array)
    - Easy to re-use the same vertices without defining each vertex several times
  - Defines, which vertices are fed into the pipeline
  - Defines how to construct the faces of a mesh (triangles) based on VBO objects
  - Some devices only support `GLushort/GLubyte` type indices
  - Vertex buffer without indexing: $\{ v_0, v_1, v_2, v_0, v_3, v_1 \}$
  - Vertex buffer with indexing: $\{ v_0, v_1, v_2, v_3 \}$
  - Index buffer: $\{ i_0, i_1, i_2, i_2, i_1, i_3 \}$



Vertex Array Object

- Encapsulates the state needed to specify per-vertex data to the OpenGL pipeline
- Remembers the state of buffer objects
- Easy and efficient way of switching between OpenGL buffer states
- Since OpenGL 3.0
  - On OpenGL ES 2, VAOs provided by the optional `GL_OES_vertex_array_object` extension

OpenGL Object Model – Allocation

- Targets will be used in OpenGL function calls as parameters
  - Target may be used, although not provided as a parameter in some functions

- Buffer allocation
  - ```
    glBufferData(GLenum target, GLsizeiptr buffer_size,
                 const GLvoid *buffer_data, GLenum usageHint);
    ```

- Usage hints
  - Bets place to storage the data, depending how often it will be accessed and changed
    - `GL_STATIC_*` Data store modified once and use infrequently
    - `GL_DYNAMIC_*` Data store modified once but used frequently
    - `GL_STREAM_*` Data store modified and used frequently
    - `GL_*_DRAW` Data store modified by the application
    ```

VBO and Element Array

```
static GLuint make_buffer(GLenum target, const void *buffer_data, GLsizei buffer_size) {
    GLuint buffer;
    glGenBuffers(1, &buffer);
    glBindBuffer(target, buffer);
    glBufferData(target, buffer_size, buffer_data, GL_STATIC_DRAW);
    return buffer;
}

// Simple rectangle
static const GLushort g_element_buffer_data[] = { 0, 1, 2, 3 };

static int make_resources(const char *vertex_shader_file)
{
    m_vboBufferHandle = make_buffer(GL_ARRAY_BUFFER, g_vertex_buffer_data,
sizeof(g_vertex_buffer_data));
    m_elementArrayHandle = make_buffer(GL_ELEMENT_ARRAY_BUFFER, g_element_buffer_data,
sizeof(g_element_buffer_data));
```


Buffer Objects in Qt

- Use `QOpenGLBuffer` to create and manage any kind of buffer objects
  - Uses shallow copy – no implicit sharing
  - If buffer copy is changed, original buffer data changes as well
- Qt also supports VAOs
  - Just create and bind them, whenever you use buffer VBOs

```
// Create one or more vertex buffer objects
m_modelVbo.create();
m_modelVbo.bind();
m_modelVbo.allocate(m_object.constData(), m_object.count() * sizeof(GLfloat));

// Optionally create a vertex array object - required after OpenGL 2
m_vao.create();
QOpenGLVertexArrayObject::Binder vaoBinder(&m_vao);
```

Rendering a Frame

- OpenGL rendering is state-based
  - OpenGL takes a snapshot of the state and adds it to the GPU's command queue
  - State can be changed and another command can be queued
  - `glSwapBuffers()` will wait until the commands have been executed and show the result in the framebuffer
- A state consists of
  - A shader program, consisting of shaders
  - Data parameters, provided to shaders
  - Render buffer settings
    - For example, clear, color, depth, stencil buffers

Setting the State

- Tell OpenGL, which vertex buffer (and other buffers) to use
  - `glBindBuffer(GL_ARRAY_BUFFER, vertexbufferName);`
  - `QOpenGLBuffer::bind(); // If Qt classes were used`

- Define the location and data format of the vertex attributes

```
glVertexAttribPointer(
   GLuint locationIndex, /* attribute location in a shader program */
   GLint size,            /* NOF components per attribute [1, 4] */
   GLenum type,          /* GL_BYTE, GL_FLOAT */
   Glboolen normalized,  /* GL_FALSE, GL_TRUE */
   Glsizei stride,       /* NOF bytes between attribute values */
   const GLvoid *pointer /* array buffer offset */
);
```

- The location index comes from the shader program – let's look at that later

Submitting the Rendering Job

- Enable values to be read from the vertex array
  - `glEnableVertexAttribArray(GLuint locationIndex);`

- Draw vertices

```
GLvoid glDrawArrays(
   GLenum mode,
   GLint first            /* Starting index in vertex array data store */
   GLsizei count)         /* NOF vertices to be drawn */
```

  - Mode
    - `GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN`

- After drawing make sure the array cannot be used anymore
  - `glDisableVertexAttribArray(GLuint locationIndex);`

Submitting the Rendering Job

- Several items may be called by repeating glDrawArrays()

- Since OpenGL 3, there has been another function to create several instances
  - `glDrawArraysInstanced(GLenum mode, Glint first, GLsizei count, Glsizei instanceCount)`
  - `glEnableVertexAttribArray(GLuint locationIndex);`

- Submitting a job with indeces

```
GLvoid glDrawElements(
    GLenum mode,
    GLsizei count,
    GLenum type,                // GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT
    const GLvoid* offset) // Element array offset
```


Submitting the Rendering Job – Example

```
// Init function - after the shader program has been compiled
g_resources.attributes.position = glGetAttribLocation(g_resources.program, "position");

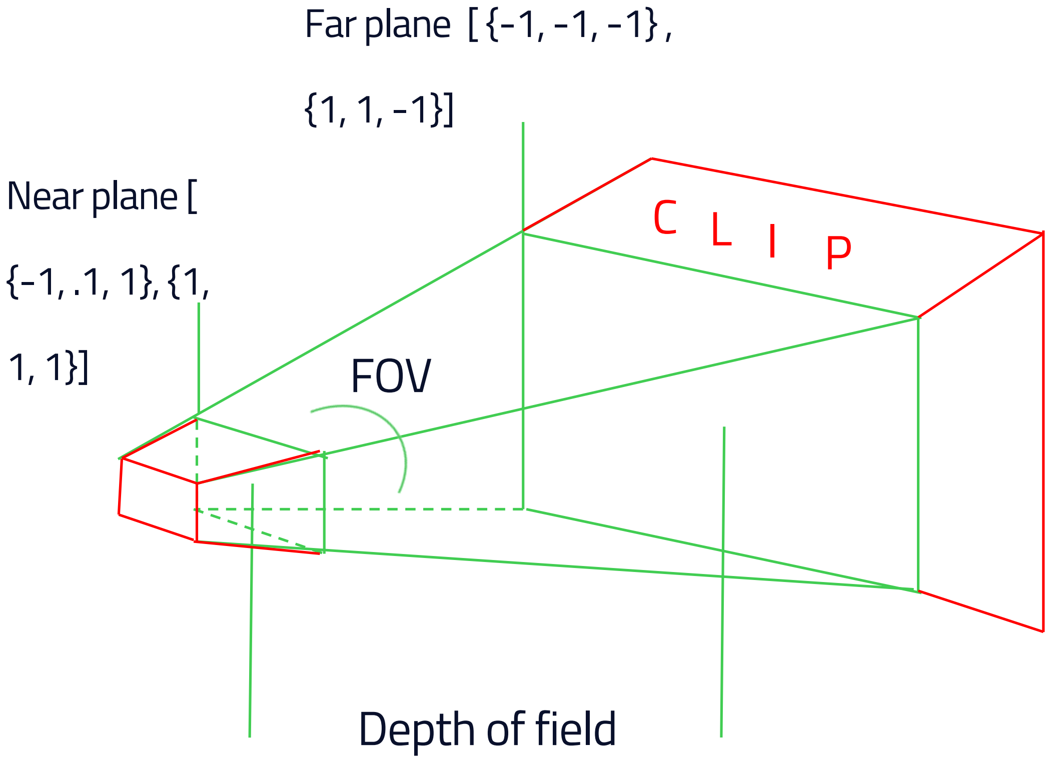
// Render function
static void renderFrame(void)
{
    glBindBuffer(GL_ARRAY_BUFFER, g_resources.vertex_buffer);
    glVertexAttribPointer(g_resources.attributes.position, 2, GL_FLOAT, GL_FALSE,
                          sizeof(GLfloat)*2, (void*)0);

    glEnableVertexAttribArray(g_resources.attributes.position);
    glDrawArrays(GL_TRIANGLES, 0, 4);
    glDisableVertexAttribArray(g_resources.attributes.position);

    // or using indeces - preferred
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_resources.element_buffer);
    glDrawElements(GL_TRIANGLE_STRIP, 4, GL_UNSIGNED_SHORT, (void*)0);
    glDisableVertexAttribArray(g_resources.attributes.position);
}
```

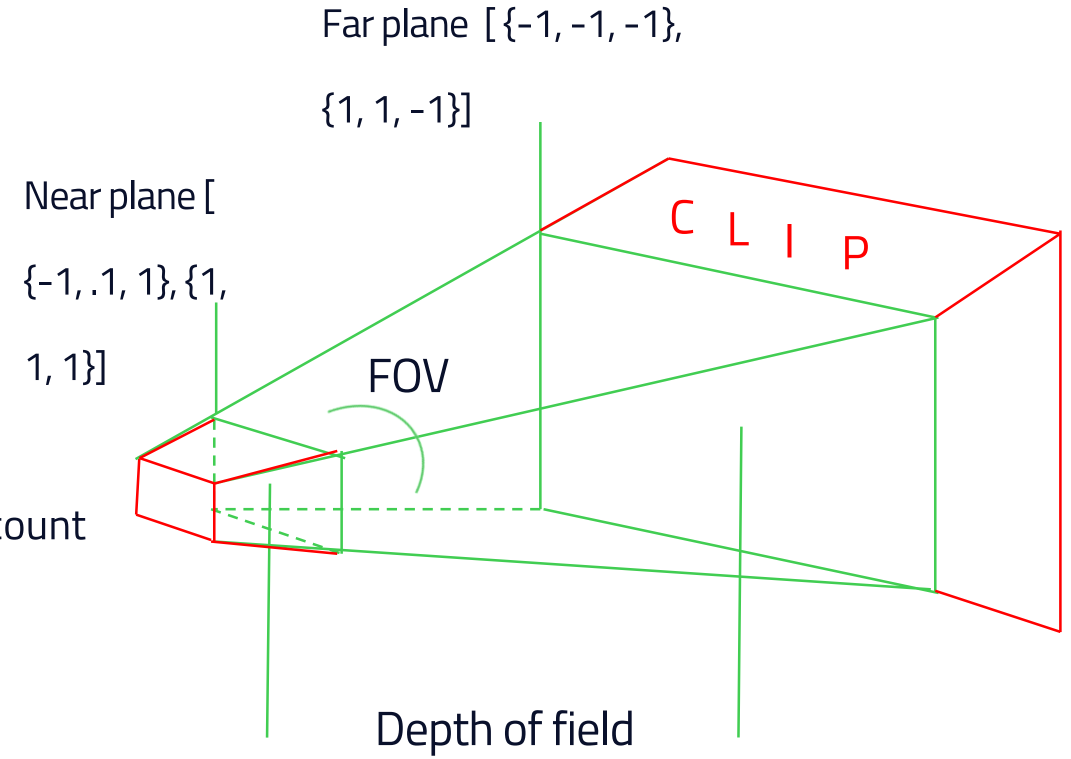
Model, View, Projection Spaces

- Model space
  - Your model untransformed coordinates
- World space
  - Transformed model coordinates
  - Transformations applied in shaders with transformation matrices
- We will use `QMatrix` classes for transformations
  - Reset – set matrix to identity matrix
  - Rotate, scale, translate
- Pay attention to the order of transformations
  - `translation * rotation * scale * vertex`



Model, View, Projection Spaces

- View space
  - Defines the location of a camera
  - By default at the center of the model
- Projection space
  - Space between and including near and far planes
  - Model coordinates are mapped to the projection space taken into account the distance
  - `QMatrix4x4::perspective(float verticalAngle, float aspectRation, float nearPlane, float farPlane)`
  - `QMatrix4x4::ortho(float left, …, float farPlane);`
- Finally, all transformations applied in the vertex shader
  - `projection * model-view * vertex`



Transformations

› More efficient to calculate matrices in your application once or rarely than repeat the calculations for every vertex in a shader

```
// All members are QMatrix4x4 types
m_proj.perspective(60.0f, GLfloat(width) / height, 0.01f, 100.0f);

m_world.setToIdentity();
m_world.translate(m_worldPos, 0, 0);

m_camera.setToIdentity();
m_camera.translate(0, 0, -1);

m_mvpMatrix = m_proj * m_camera * m_world;a;
```


Render Buffers

- By default rendering is made to a color buffer
  - Automatically created – id 0
  - Stores the final colored image generated by renderer

- Additional buffers may be enabled with capabilities
  - `glEnable(GLenum capability)`
  - Avoid enabling and disabling capabilities for performance reasons

- Depth render buffer
  - If enabled, after rasteriization each fragment's projected z value is compared to the z value stored in the depth buffer
  - Minimizes the cost of overdraws, if objects are rendered front-to-back
    - Rasterizer will still generate fragments, but fragment shader is not executed
  - `glEnable(GL_DEPTH_TEST):`
  - `glDepthFunc(GL_NEVER); // GL_LESS, GL_GREATER`

Render Buffers

- Back-face culling
  - Possible to discard back-facing primitives prior rasterization
  - By default primitives windowing counterclockwise are front-facing
  - Only the vertex shader is needed to run for these back-facing vertices
  - `glEnable(GL_CULL_FACE);`
  - `glCullFace(GL_BACK); // GL_FRONT, GL_FRONT_AND_BACK`

- Clipping
  - Discard fragments outside scissor rectangle
  - `glEnable(GL_SCISSOR_TEST):`
  - `glScissor(GLint x, GLint y, GLsizei width, GLsizei height);`

- At most one render buffer of each kind can be attached to the frame buffer
  - `glFramebufferRenderbuffer(GLenum target, GLenum attachment, GLenum renderbuffertarget, GLuint renderbufferid)`

Other Useful Capabilities

- `GL_BLEND` – Blend fragment color with the values in color buffers
- `GL_DEBUG_OUTPUT`
- `GL_DITHER` – Blend fragment color with the values in color buffers
- `GL_MULTISAMPLE` – Use multisampling for antialiasing

Rendering a Frame – Render Buffers

- Some of the render buffers must be reset for each frame
  - Otherwise the previous data values are combined with new ones

- For example, fill the framebuffer's color buffer
  - `GLvoid glClear(GLbitfield mask);`
  - `glClear(GL_COLOR_BUFFER_BIT);`
  - `GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT`

Texturing

- Texture coordinates (s, t, u, v) are mapped to the primitive vertices
  - OpenGL only supports power of two textures
  - Use them to optimize performance

- Texture itself is yet another object in video memory
  - `GLuint textureID;`
  - `glGenTextures(1, &textureID);`
  - `glBindTexture(GL_TEXTURE_2D, textureID);`

- Give the actual data

```
glTexImage2D(
GLenum target, GLint level,                // GL_TEXTURE_2D, GL_TEXTURE_1D_ARRAY
GLint internalFormat, GLsizei, witdh, GLsizei height, // GL_RGB8
GLint border, GLenum format,               // border must be 0, format e.g. GL_RGB
GLenum type, const GLvoid *data);          // GL_BYTE
```

Texture Sampling

- Sampling takes place in the fragment shader using u,v coordinates
  - Shaders sample a texture at one or more floating-point texture coordinate
- Sampling is controlled with texture parameters
  - `glTexParameteri(GLenum target, GLenum param, GLint value)`
  - `GL_TEXTURE_MIN_FILTER`
  - `GL_TEXTURE_MAG_FILTER`
- Sampling between the centers of texture points
  - GL_LINEAR / GL_NEAREST / `GL_LINEAR_MIPMAP_LINEAR`
- Sampling beyond the texture points
  - `GL_TEXTURE_WRAP_S / GL_TEXTURE_WRAP_T`

Texturing in OpenGL and Qt

```
QImage convImage = image.convertToFormat(QImage::Format_RGB888).mirrored();
width = convImage.width(); height = convImage.height();
void *pixels = convImage.bits();
GLuint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB8, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, pixels);

// Or simply
m_texture = new QOpenGLTexture(QImage(":/textures/lady.png"));
m_texture->setMinificationFilter(QOpenGLTexture::Nearest);
m_texture->setMagnificationFilter(QOpenGLTexture::Linear);
m_texture->setWrapMode(QOpenGLTexture::Repeat);
```

Frame Buffers

- Useful for implementing any kinds of graphics effects
  - Shadows
  - Car mirrors and other in-app cameras
  - Surface reflections

- Create and configure a frame buffer object, grouping
  - One or more color buffers (textures)
  - Zero or one depth buffer – if depth testing is needed

- Render to the created frame buffer object in the first render loop

- Render to the default frame buffer object in the second loop
  - Use shaders to access the color buffer data to draw fragments

Frame Buffers - Example

```
// Create a frame buffer object
Gluint frameBufferHandle = 0;
glGenFramebuffers(1, &frameBufferHandle);
glBindFramebuffer(GL_FRAMEBUFFER, frameBufferHandle);

// Create one or more color buffers
Gluint colorBufferHandle;
glGenTextures(1, &colorBufferHandle);
glBindTexture(GL_TEXTURE_2D, colorBufferHandle);
// Create a GL_DEPTH_COMPONENT instead of GL_RGB for the depth buffer
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB, width, height, 0,GL_RGB, GL_UNSIGNED_BYTE, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST); glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// Optionally, create a depth buffer
GLuint depthBufferHandle;
glGenRenderbuffers(1, &depthBufferHandle);
glBindRenderbuffer(GL_RENDERBUFFER, depthBufferHandle);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,
    depthBufferHandle);
```


Frame Buffers - Example

```
// Configure the frame buffer with one or more color buffers
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, colorBufferHandle, 0);
// Define an array of draw buffers
GLenum DrawBuffers[1] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, DrawBuffers);
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) // …

// Render to the color buffer
glBindFramebuffer(GL_FRAMEBUFFER, frameBufferHandle);
glViewport(0, 0, width, height);

// And now we would need shaders to actually use the created color buffer
```

Frame Buffers in Qt

- Use `QOpenGLFramebufferObject`
  - Supports painting using `QPainter` and OpenGL functions
- By default, `GL_TEXTURE_2D` target created and bound to `GL_COLOR_ATTACHMENT0`
  - If supported by the OpenGL implementation, other color attachments may be added
  - `addColorAttachemnt(QSize, GLenum format) // Defualt internal format is RGBA8`
- Allows attaching depth and stencil buffers
  - `setAttachment(Depth, CombinedDepthStencil)`
- Get the texture location for the shader or to render the buffer(s) into an `QImage`
  - `QLuint texture()`
  - `QImage toImage(bool flipped, int colorAttachment)`

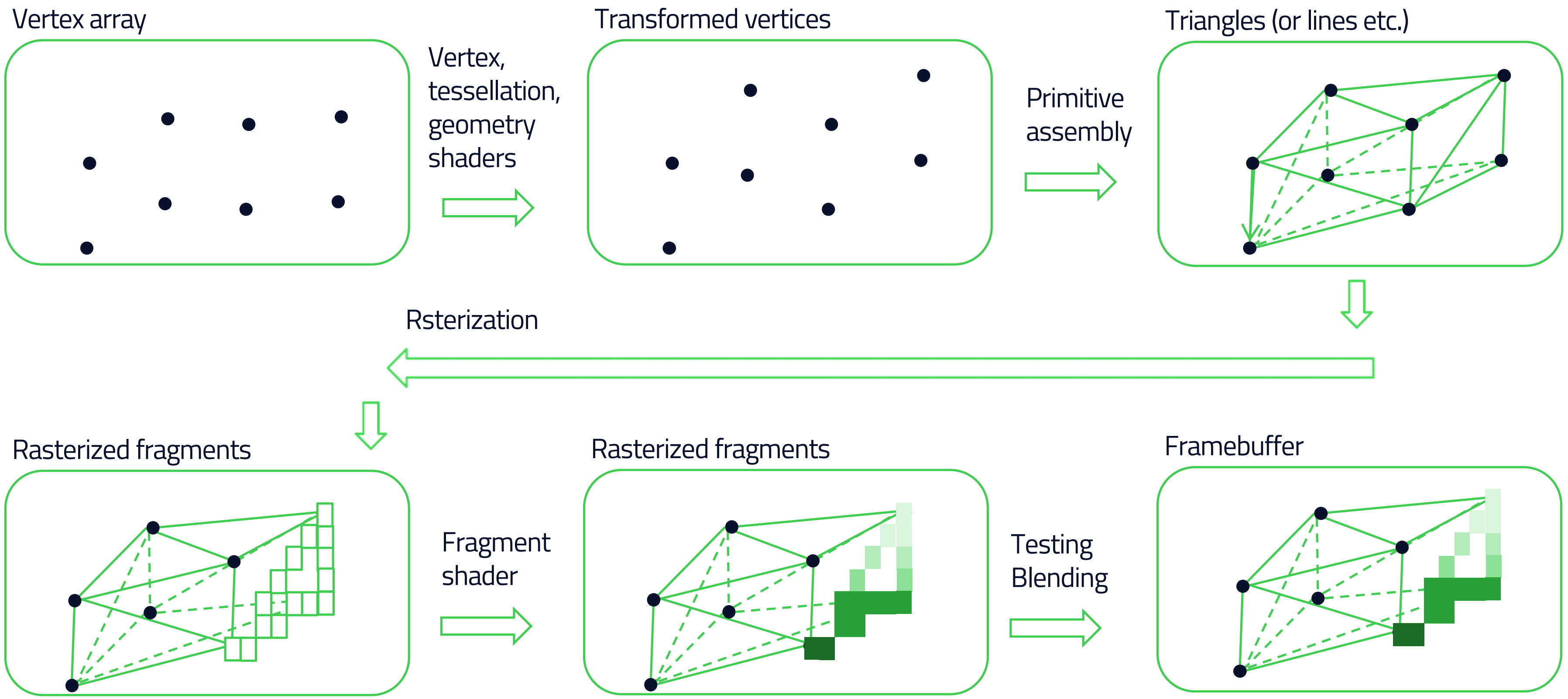
Summary

- Windows and event handling
  - `QOpenGLWidget, QOpenGLWindow`
- OpenGL content management
  - `QOpenGLContext, QSurfaceFormat, QOpenGLFunctions`
- Buffer and texture handling
  - `QOpenGLBuffer, QOpenGLVertexArrayObject, QOpenGLTexture, QOpenGLFrameBufferObject`
- QPainter-based painting
  - `QOpenGLWidget, QOpenGLPaintDevice, QOpenGLFrameBufferObject`
- Shaders
  - `QOpenGLShader, QOpenGLProgram`

Shaders

Contents

OpenGL Rendering Pipeline



OpenGL Rendering Pipeline – Shaders

- Vertex shader

- Tessellation shader
  - For curved shapes

- Geometry shader
  - Creates new vertices

- Primitive assembly (triangles)
  - Independent triangles
  - Triangle strip (last two vertices of each triangle are the first two vertices of the next)
  - Triangle fan (the first element is connected to every subsequent pair of elements)

- Rasterization
  - Clipping
  - Interpolates the varying values across the rasterized surface of each primitive (triangle)

- Fragment shader

- Testing and blending

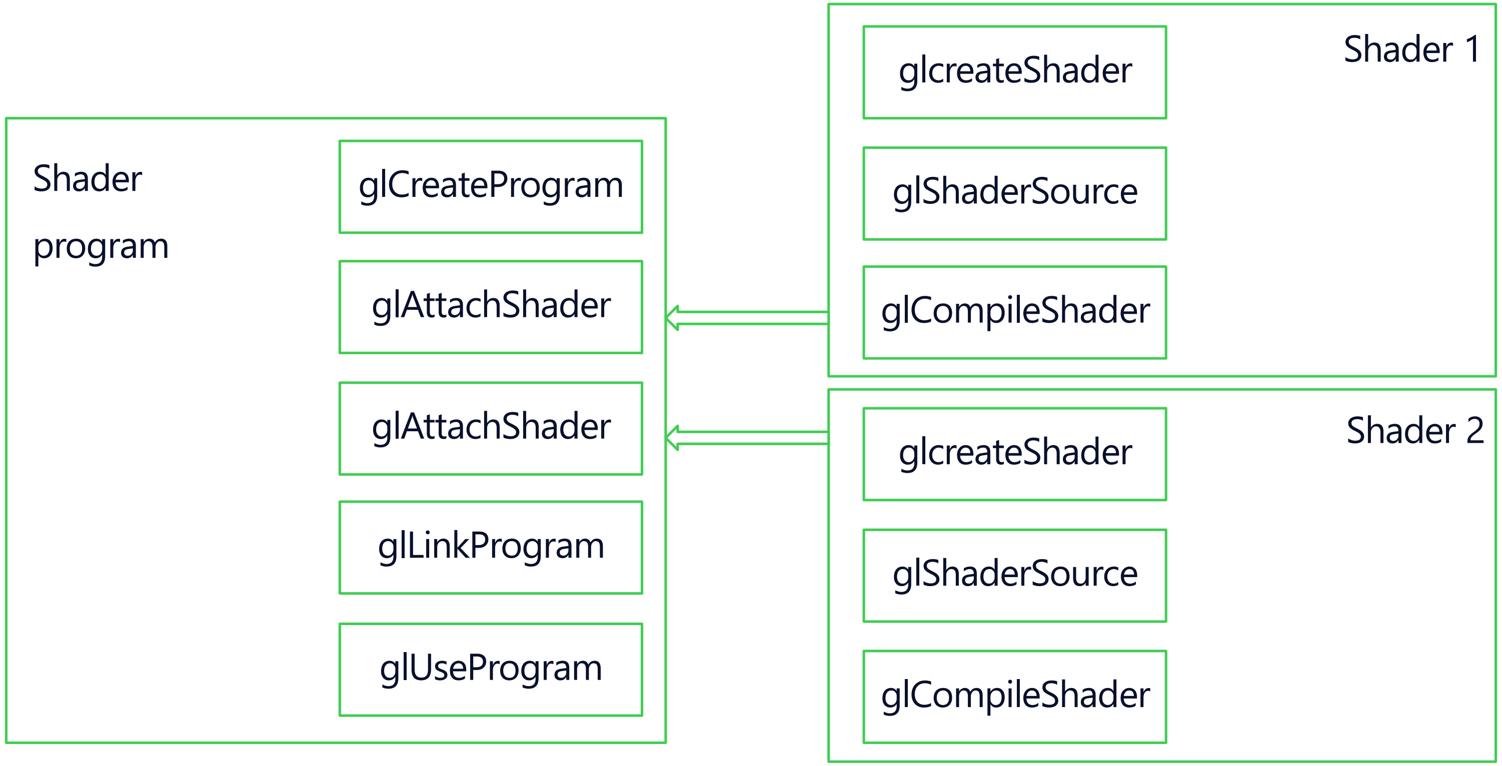
Shaders

- Programs you write for the programmable rendering pipeline

- OpenGL has a language called GLSL for writing shaders
  - Looks a bit like C
  - Supports typical language constructs like while, for, if, etc.
    - There are limitations though because of how GPUs work – more about this later

- The OpenGL implementation compiles and links the shaders in runtime from text to machine code the GPU executes
  - Some vendor specific extensions enable you to precompile and save some startup time

Creating Shader Programs



Creating a Shader

```
GLchar *source = array.data();
GLuint shader;
GLint shader_ok;

shader = glCreateShader(type);
glShaderSource(shader, 1, (const GLchar**)&source, &length); `
glCompileShader(shader);
glGetShaderiv(shader, GL_COMPILE_STATUS, &shader_ok);

if (!shader_ok) {
    qDebug() << "Failed to compile" << filename;
    GLint log_length;
    QByteArray log;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &log_length); `
    glGetShaderInfoLog(shader, log_length, NULL, log.data());
    glDeleteShader(shader);
    return 0;
}
```


Creating a Program

```
GLint program_ok;

GLuint program = glCreateProgram();
glAttachShader(program, vertex_shader);
glAttachShader(program, fragment_shader);
glLinkProgram(program);

glGetProgramiv(program, GL_LINK_STATUS, &program_ok);

if (!program_ok) {
    qDebug() << "Failed to link a shader program";
    GLint log_length;
    QByteArray log;
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &log_length);
    glGetProgramInfoLog(program, log_length, NULL, log.data());
    glDeleteProgram(program);
    return 0;
}
```


Creating Shader Programs in Qt

- `QOpenGLShader`
  - Check which shaders are supported – `hasOpenGLShaders()`
  - Compile from a file or bye array
  - Add to the shader program

- `QOpenGLShaderProgram`
  - Improves cross-compatibility
  - Add shader sources or binaries
  - Set attribute and unform locations and values
  - Set shader-specific values
  - Link
  - Use the program – `bind()` and `release()`

Creating Shader Programs in Qt

```
m_program = new QOpenGLShaderProgram;

m_program->addShaderFromSourceFile(QOpenGLShader::Vertex, ”vertexshader.glsl");
m_program->addShaderFromSourceFile(QOpenGLShader::Fragment, “fragmentshader.glsl"); m_program-
>bindAttributeLocation("vertex", 0);
m_program->bindAttributeLocation("normal", 1);

m_program->link();
m_program->bind();

m_projMatrixLoc = m_program->uniformLocation("projMatrix");
m_mvMatrixLoc = m_program->uniformLocation("mvMatrix");
```


GLSL Data Types

- Float vectors
  - vec2, vec3, vec4, ivec2
- Matrices
  - mat2, mat3, mat4
- Scalars
  - float, bool, int
- Textures samplers
  - samplerD, sampler2D, sampler3D, samplerCube, sampler2DShadow
- Precision attributes in OpenGL ES versions
  - Lowp (-2.0/2.0 or -256/256), mediump (-16384.0/16384.0 or -1024/1024), highp
  - Example: `uniform mediump vec4 color;`
- Arrays and structures

Data Access in GLSL

```
vec4 a = vec4(1.0,2.0,3.0,4.0);
float posX = a.x; // xyzw
float posY = a[1];
vec2 posXY = a.xy;
float depth = a.z;

vec4 b = vec4(1.0,2.0,3.0,4.0);
float blue = b.b; // rgba
float red = b[0];
```


Shader Variables – Since OpenGL 2.0

- Attributes - Keyword `attribute`
  - Used to declare input arguments to the vertex shader
  - Something that is specific to this vertex (vs. uniform which is specific to the batch of primitives getting drawn)

- Uniforms - keyword `uniform`
  - A value that is constant through out every vertex and every fragment
  - The value can be changed with OpenGL calls from the C/C++ side between draw calls
  - Example: you want to paint solid color triangles, all triangles aren't of the same color though

Shader Variables – Since OpenGL 2.0

- Varying - Keyword `varying`
  - Used to declare values to be interpolated and usable in the fragment shader
  - Declare the same name varying in both vertex and fragment shader
  - Assign a value to the varying variable in the vertex shader
  - Enjoy the interpolated values in your fragment shader!

- For instance, when performing lighting computation per fragment, we need to access the normal at the fragment
  - Normal is only available for vertex shader

  - `varying vec4 vertexColor;`

Passing Data to Shader Programs

- GLSL linker assign `GLint` location to uniforms and attributes
- Locations are used to assign value to them

- Uniforms
  - `glGetUniformLocation(programObject, vairableStringName)`

- Attributes
  - `glGetAttribLocation(programObject, vairableStringName)`

Passing Data to Shader Programs in Qt

› From VBO

```
m_shader->setAttributeBuffer("inputVertex", GL_FLOAT, 0, 3,
   8 * sizeof(float));
m_shader->enableAttributeArray("inputVertex");
```

› From vertex array

```
static GLfloat const vertices[] = { 60.0f, 10.0f, 0.0f, … };
int vertexLocation = m_shader.attributeLocation("vertex");
m_shader.enableAttributeArray(vertexLocation); m_shader.setAttributeArray(vertexLocation,
vertices, 3);
```

› Uniform variables

```
QMatrix4x4 mvpMatrix;
mvpMatrix.ortho(rect());
m_shader.setUniformValue(matrixLocation, mvpMatrix);
```

Shader Variables – Since OpenGL 3.0

- Assigning attribute and uniform locations is boring and slow
- New way is to use layout, which makes location queries obsolete

```
layout(location 0) in vec3 position;
layout(location 1) in vec2 textureCoordinates;

out vec4 vertexColor;
```

  - If you output a variable, the next shader stage must input the same variable, provided it is going to use

Vertex Shaders

- Gets run for each vertex
- Output a new set of attributes, referred to as `varying` or `out` value, fed to the rasterizer
- At a minimum calculates the projected position of the vertex on the screen
- Transformations, projections, deformations, etc. made here
  - Notice that with vertex shader you can manipulate the vertices in OpenGL buffers without changing them => no data traffic between CPU and GPU
- The input of the program is coordinates (state) in object space and their associated attributes (color, texture coordinates, etc.)
- The output of the program is coordinates in screen space and attributes to be interpolated

Vertex Shader Variables

- Input variables
  - `gl_VertexID` – either current index or vertex number, depending on the draw call
- Output variables
  - `gl_Position`
    - Usually transforming the vertex with the modelview and projection matrices
  - `gl_PointSize`
    - Defines how many fragments each 3D points uses – based on the vertex z-coordinate, for example
    - Useful to make particle effects, like fire
    - Must be enabled using `glEnable(GL_PROGRAM_POINT_SIZE);`

```
attribute highp vec4 vertex;
uniform highp mat4 mvpMatrix;
void main(void)
{
    gl_Position = mvpMatrix * vertex;
}
```


Fragment Shaders

- Fragment = pixel in the target buffer
  - Invoked once per rendered pixel

- Inputs to the fragment shader program are mainly the interpolated vertex attributes

- Output of the fragment shader program is a RGBA pixel color and depth values
  - If blending is enabled the returned result is blended with what used to be in that pixel of the rendering target
  - The shader can also discard the fragment which means that nothing is written to the rendering target
  - Texture mapping and lighting

Fragment Shader Variables

- Input variables from vertex, geometry or tessellation shader
- Output variables
  - `gl_FragColor` - the final color of the fragment
  - `gl_FragData` - array related to multiple drawable buffers (only 1 in OpenGL ES) when rendering to multiple targets
  - `gl_FragCoord` – window-space coordinates originating from the bottom left of the window
  - `boolen gl_FrontFacing` - tells whether a fragment is front or back facing
  - `gl_FragDepth` - we can set the depth value // disables all early depth testing => performance penalty

```
uniform lowp vec4 color;
void main(void)
{
    gl_FragColor = color;
}
```

Geometry Shader

- Allows creating new primitives
  - Since OpenGL 3.2
- Define input primitive values, received from the vertex shader and output values given to the next stage
  - `layout (points) in; // lines, triangles, depending what kind of a draw command is used`
  - `layout (line_strip, max_vertices = 2) out; // points, triangle_strip`
- Create one or more primitives using `EmitVertex()` and `EndPrimitive()`
  - One way to create curved shapes, for example

```
void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0); EmitVertex();
    gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.0, 0.0, 0.0); EmitVertex();
    EndPrimitive();
}
```


Tessellation Shader

- Since OpenGL 4.0
- Tessellation control (TCS) – optional
  - How much tessellation used
- The tessellation primitive generator takes the input patch and subdivides it based on values computed by the TCS or provided as defaults – not programmable
- Tessellation evaluation shader (TES)
  - Computes vertex values for each generated vertex

Using Textures in GLSL

- Textures are usually used in fragment shaders, but some GPUs also support using them in vertex shaders too
- To use a texture declare a sampler and use texture2D…
  - `uniform sampler2D myTexture;`
  - `texture2D(sampler, someTextureCoordinate);`
- Sample the texture at specific coordinates (0..1,0..1) and do whatever you want with the value you go

Setting the State – Additional Objects

- Active the shader program
  - `glUseProgram(programObjectName)`
- Assign values to uniform variables
  - `glUniform1f(uniformObjectName, value)`
  - `glUniform[dim][type]`
- Assign textures to samplers
  - Set the active texture unit
    - `glActiveTexture(GL_TEXTUREX);`
  - Bind texture objects to texture unit
    - `glBindTexture(GL_TEXTURE_2D, textureObjectName)`
  - Assign a value to a texture uniform

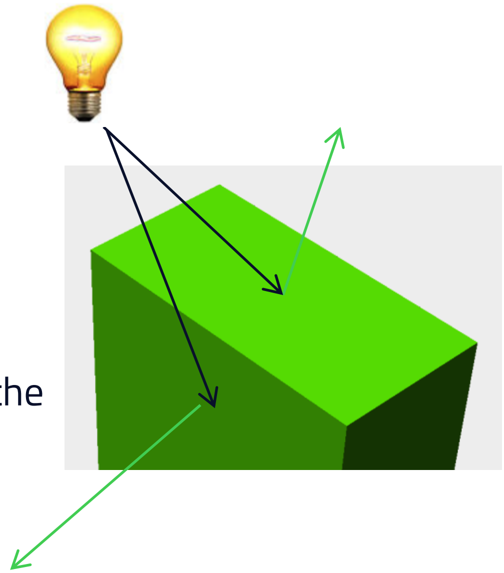
Shading

- Defines how bright or dark fragments are
- Diffuse and ambient reflection
  - Ambient reflection: reflection from the surfaces, which are not directly lit
  - Diffuse reflection: surface reflects light evenly in every direction of diffusely
- Specular reflection - shiny objects
  - The shine moves when looking in the reflection of light
- Shadows, mirror-like reflections, ambient occlusion etc. can be handled with Qt 3D

Shading – Diffuse Reflection

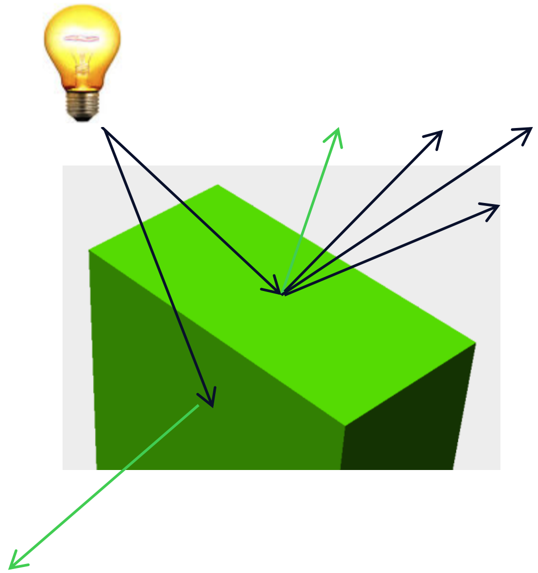
- Surface's shaded color may be calculated using Lambertian law
  - Surfaces reflect more light the more parallel to a light source they become

- To calculate the diffuse light reflection component, we need
  - The angle between the light vector and the surface normal
  - Use the angle to calculate the actual intensity
  - Instead of angle, we can use the *cos(angle)* as it is nicely got from the dot product of two vectors and is in the range [-1, 1]



- Get the light vector
  - `vec3 lightVector = normalize(lightPos - vertex);`
- Uses the dot product to calculate the cosine
  - `float diffuse = max(dot(normalize(vertNormal), lightVector), 0.0);`
- Calculate the shading value
  - `vec3 col = clamp(ambbientColor + diffuseColor * diffuse, 0.0, 1.0);`

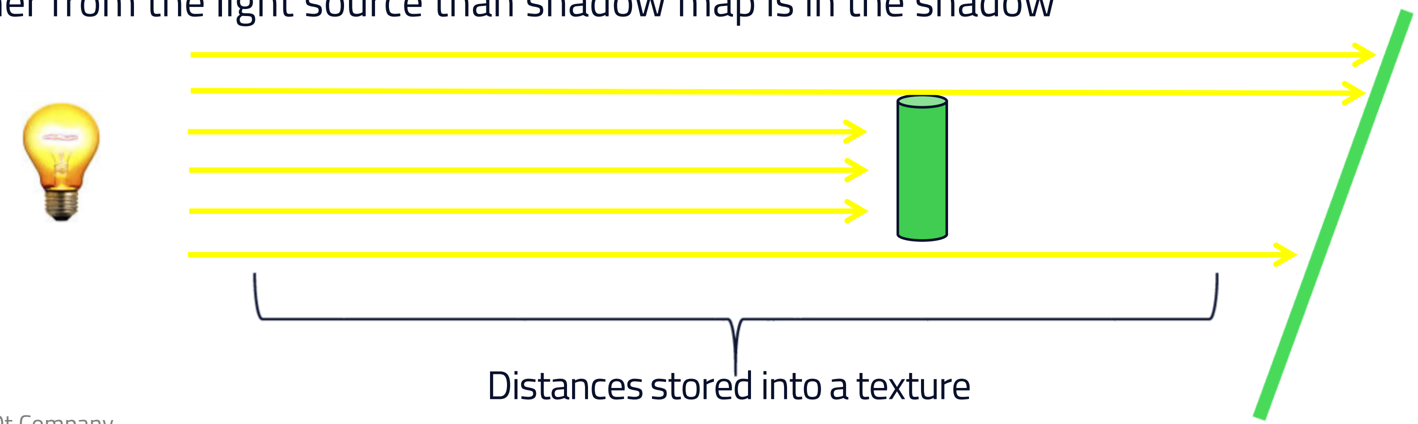
Shading – Specular Reflection



- Light s reflected mostly in the direction that is the reflection of the light on the surface
- Specular color component will be
  - 1, if we are looking into the reflection
  - < 1, if we are looking elsewhere
- Let’s calculate the reflection direction
  - `vec3 reflection = reflect(lightVector, vertexNormal);`
- Get the cosine between the camera vector and the reflection vector
  - `float cosAlpha = max(dot(cameraVec, reflection), 0.0);`
- Calculate the shading value
  - `vec3 col = clamp(ambbientColor + diffuseColor * diffuse + specularColor * pow(cosAlpha, specularLobe), 0.0, 1.0);`

Shadow Mapping – Render Pass Example

- Scene is rendered in two passes
- First it is rendered from the point of view of the light
  - Only the depth of each fragment is computed
  - Stored in the shadow map
- Second the scene is rendered as usual
  - With an extra test to see, if the fragment is in the shadow
  - Any fragment, which is further from the light source than shadow map is in the shadow



Shadow Map Texture

```
// Create a frame buffer object
Gluint depthBufferHandle = 0;
glGenFramebuffers(1, &depthBufferHandle );
glBindFramebuffer(GL_FRAMEBUFFER, depthBufferHandle );

// Create a depth texture
Gluint depthTexture;
glGenTextures(1, &depthTexture);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glTexImage2D(GL_TEXTURE_2D, 0,GL_DEPTH_COMPONENT16, width, height, 0, GL_DEPTH_COMPONENT, GL_FLOAT,
0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST); glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE); glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthTexture, 0);

// No draw to color buffer
glDrawBuffers(GL_NONE);
```

Shadow Mapping – Transformations

- Standard model-view projection matrix used
  - Model matrix may be anything
  - View rotates the world, so that in camera space the light direction is –Z
  - Projection is orthographic, containing the world frustrum
    - We assume the light source is so far, all light rays are parallel
- Shades are trivial
  - Vertex shader just multiplies each vertex with the projection, view, and model matrixes
  - Fragment shader fragmentdepth = gl_FragCoord.z;

```
QVector3D lightInvertedDirection(0.5f, 5, 5);
QMatrix4x4 viewMatrix(lightInvertedDirection.x(), ..., );
QMatrix4x4 projectionMatrix; projectionMatrix.ortho(-20, 20, -20, 20, -20, 20);
// left, right, top, bottom, near, far

glUniformMatrix4fv(depthMatrixID, 1, GL_FALSE, &depthMVP[0][0]);
```


Shadow Mapping – Usage

- Vertex shader is trivial, just multiply each vertex with protection, view, and model matrixes
  - We need the fragment position in the shadowmap space => need to multiply

- In the vertex shader
  - Compare the vertex position to the shadom map position
  - If the position is larger, add a shadow color component to the fragment

OpenGL in Qt

Contents

- Canvas 3D
- OpenGL and QtQuick

3D Canvas

- Based on signals and context like 2D canvas
- Before the first frame is rendered, `initializeGL()` emitted
  - Get the context (provides OpenGL API) - `canvas.getContext("3d", {depth:true, antialias:true});`
  - Set the rendering buffers
  - Create buffers
  - Create shaders
  - Load textures
- For each frame to be rendered, `paintGL()` emitted
  - Apply transformations
  - Draw elements

3D Canvas in QML

```
import QtQuick 2.4
import QtCanvas3D 1.0

import ”myOpenGLCode.js” as Code

Item {
    id: root; width: 640; height: 480; visible: true
    Canvas3D {
        id: canvas3d
        anchors.fill:parent
        onInitializeGL: {
            Code.initializeGL(canvas3d);
        }
        onPaintGL: {
            Code.paintGL(canvas3d);
        }
```

3D Canvas in JavaScript

```
var canvas3d;
var gl;

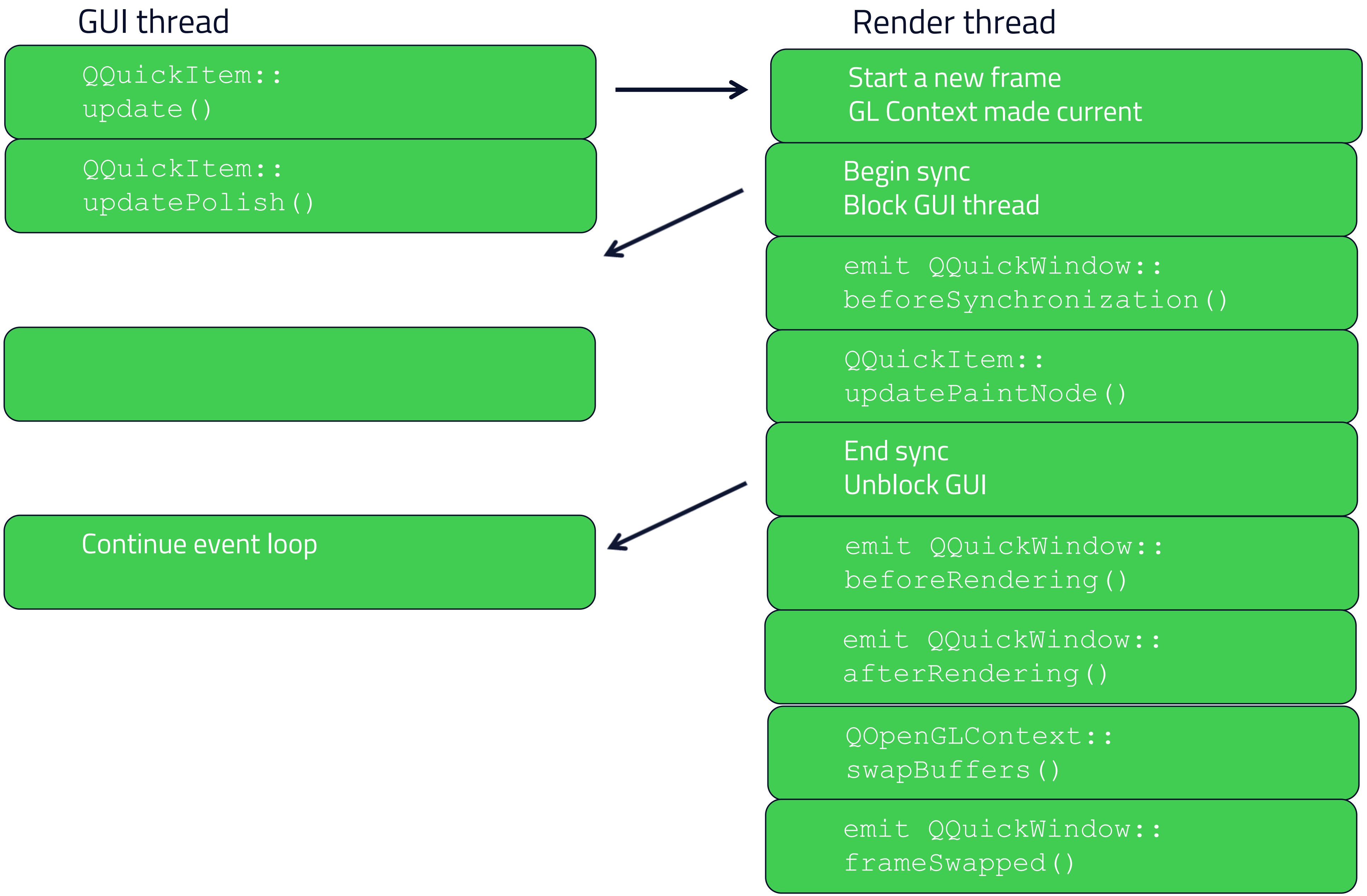
function initializeGL(canvas) {
    canvas3d = canvas;
    gl = canvas.getContext("canvas3d", {depth:true, antialias:true});
    gl.clearColor(0.98, 0.98, 0.98, 1.0);
    gl.viewport(0, 0, canvas.width, canvas.height);
    initShaders();
    initBuffers();
    loadTextures();
}
function paintGL(canvas) {
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.drawElements(gl.TRIANGLES, 36, gl.UNSIGNED_SHORT, 0);
}
```


OpenGL and QtQuick

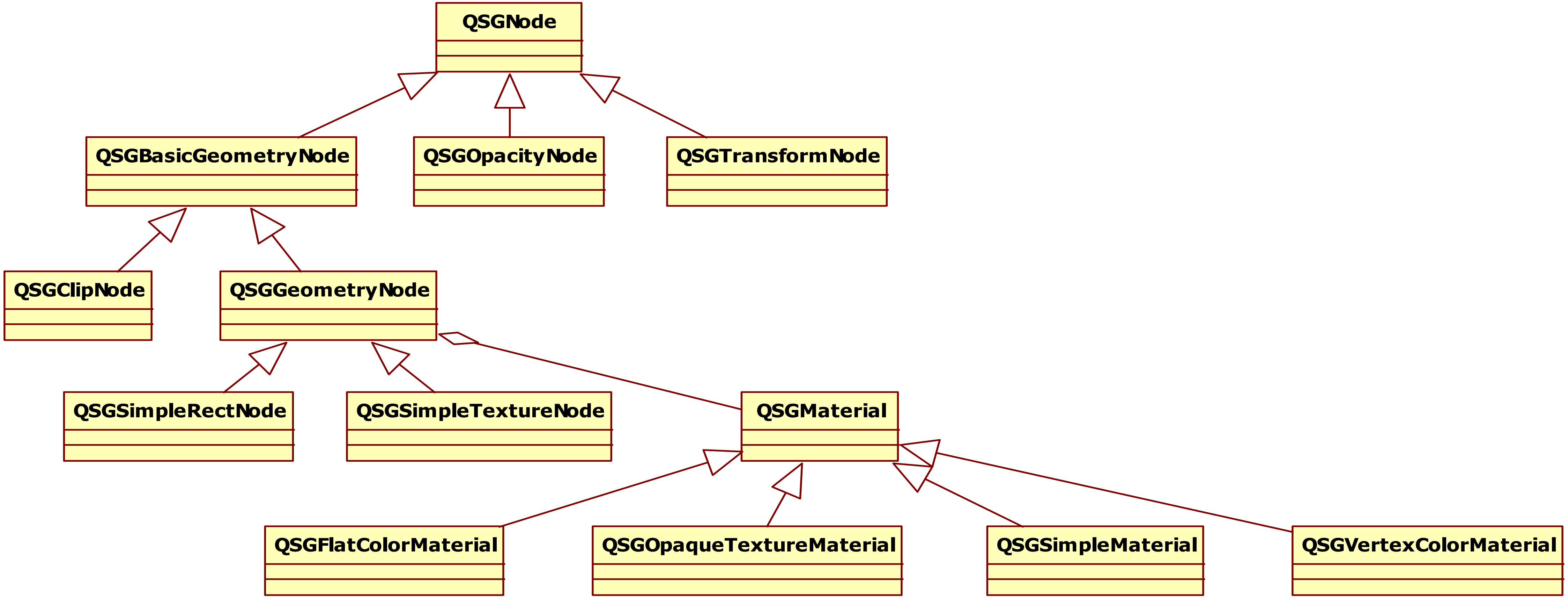
- QtQuick renderer allows you to use OpenGL
  - Before a frame is rendered (background)
  - For the QtQuick custom items
  - After rendering a frame (foreground)

- Possible to use `QQuickPaintedItem` as well with native painting

Demo: customItem



Scene Graph Nodes and Material Classes



Items and Scene Graph Nodes

- Derive from `QQuickItem`
- Implement `updatePaintNode(…)`
  - More efficient than `paint()`, which occurs in two phases – must be painted into the buffer (`QSGTextureNode`), rendered by the scene graph renderer
- Use `QSGNode::preprocess()`, if the node changes in every frame

Scene Graph Items

- Create and initialize a `QSGNode` subclass (e.g. `QSGGeometryNode`)
  - `QSGGeometry` to specify the mesh
  - `QSGMaterial` to encapsulate rendering state for a shader program
    - For one scene graph, there is one unique `QSGMaterialShader`, which encapsulates `QOpenGLShaderProgram` the scene graph uses to render that material
    - Each geometry node may have a unique `QSGMaterial`

- Similar to other QtQuick classes:
  - Export object from C++
  - Import and use in QML
  - Properties, signals/slots, `Q_INVOKABLE`

Custom Geometry Node

- Use `QSGGeometry` to define vertices (position, normals, and texture coordinates)
  - `QSGGeometry *geometry = new QSGGeometry(CustomAttributeSet, 3);`

- Define the drawing mode (triangles, triangle strips or points)

```
geometry->setDrawingMode(GL_TRIANGLES);
CustomVertex *verteces = static_cast<CustomVertex *>(geometry->vertexData())
vertices[0].set(…);
```


Custom Geometry Node

- Set the geometry to the geometry node
  - `QSGGeometryNode::setGeometry(geometry);`
- The geometry can be uploaded to the graphics memory
  - `setVertexDataPattern() // AlwaysUploadPattern, DynamicPattern, StaticPattern`
  - Use `markVertexDataDorty()` to specify, when the geometry should be uploaded, if other than `AlwaysUploadPattern` used

Custom Material

- Subclass `QSGSimpleMaterialShader`
  - Use a macro to add boilerplate code

```
class MyCoolMaterial : public QSGSimpleMaterialShader<StateStruct>
{
    QSG_DECLARE_SIMPLE_SHADER(MyCoolMaterial, StateStruct);
```

- Implement the shaders
- Implement the state updates

```
void updateState(const State *state, const State *)
{
    program()->setUniformValue(m_stateVariable, state->dataInMyStateStruct); `
}
```

- Set the material to the geometry node

```
QSGSimpleMaterial<StateStruct> *material = MyCoolMaterial::createMaterial();
QSGGeometryNode::setMaterial(material);
```

Shaders in Qt Quick

- Two QML elements: `ShaderEffectSource` and `ShaderEffect`
- `ShaderEffectSource` renders any item into a texture
  - `sourceItem` property holds the Item to be rendered
  - The item is drawn as it was a fully opaque root item, even the item is invisible
  - The texture may be used as a cache (`live: false`) to render complex items (rendered once to the texture, which can be animated)
  - Can be used as an opacity layer
  - Still performance (sometimes drops), video memory (always increases), and quality (anti-aliasing) issues to be considered
- `ShaderEffect` is a rectangle displaying the result of a shader program
  - The `fragmentShader` (`vertexShader`) property is a string with the fragment or vertex shader code
  - Note that `ShaderEffectSource` is an invisible element (not rendered itself) aimed at consumption in `ShaderEffect` instances

Vertex Shaders

- Four pre-defined input values provided to a vertex shader
  - `uniform mat4 qt_Matrix // Product of transformations from the root and projection matrix`
  - `uniform float qt_Opacity // Product of opacities from the root`
  - `attribute vec4 qt_Vertex`
  - `attribute vec2 qt_MultiTexCoord0 // from top-left (0,0) to bottom-right [1,1]`
- Any property that can be mapped to GLSL type is available as a uniform variable
  - `Bool`, `QColor` (`vec4`), `QPoint` (`vec2`), `QVector3D` (`vec3`), `QTransform` (`mat4`), `Image` and `ShaderEffectSource` (`sampler2D`)

Vertex Shaders

- For non-linear vertex transformations, use the `mesh` property (`GridMesh`)
  - Specifies the number of vertices of the `ShaderEffect` element
  - It must be large enough to resolve the transformation

- Use `log` property to see the latest warnings and compilation errors of your shader program

Wave Effect with a Vertex Shader

```
ShaderEffectSource {
    id: effectSource; ...
}
ShaderEffect { ...
    property variant source: effectSource
    property real pi: Math.PI
    property real offset: 0
    NumberAnimation on offset { ... }
    mesh: GridMesh { resolution: Qt.size(20, 1) }
    vertexShader: "
        uniform highp float offset; ...
        uniform highp mat4 qt_Matrix;
        attribute highp vec4 qt_Vertex;
        attribute highp vec2 qt_MultiTexCoord0;
        varying highp vec2 qt_TexCoord0;
        void main() {
            qt_TexCoord0 = qt_MultiTexCoord0;
            highp vec4 pos = qt_Vertex;
            pos.y = ...;
            gl_Position = qt_Matrix * pos;
        }"
}
```

Wave Effect with a Fragment Shader

```
ShaderEffect {
    width: 160; height: width
    property variant source: sourceImage // Image type
    property real frequency: 10
    property real amplitude: 0.08
    property real time: 0.0
    NumberAnimation on time {
        from: 0; to: Math.PI*2; duration: 1000;
        loops: Animation.Infinite }
    fragmentShader: "
        varying highp vec2 qt_TexCoord0;
        uniform sampler2D source;
        uniform lowp float qt_Opacity;
        uniform highp float frequency;
        uniform highp float amplitude;
        uniform highp float time;
        void main() {
            highp vec2 pulse = sin(time - frequency * qt_TexCoord0);
            highp vec2 coord = qt_TexCoord0 + amplitude * vec2(pulse.x, -pulse.x);
            gl_FragColor = texture2D(source, coord) * qt_Opacity;
            }"
```

Chaining Shaders

- `ShaderEffectSource` can have any Item as sourceItem
- Even a `ShaderEffect`!
- Allows to create complex effects by chaining shader programs

- A drop shadow is a combination of:
  - A blur operation
  - A darkening of the result of the blur
  - A composition of the original on top of the created shadow with an offset

Qt 3D

Contents

Qt 3D

- For general-purpose 2D and 3D (real-time) simulations in C++ and QML
  - Completely re-designed from v. 1, introduced already in Qt 4.7
- Configurable for any kind of 3D simulations
  - Qt Quick uses scene graph render optimizations (batches, rendering order, texture atlas)
    - Only vertex and fragment shaders supported
  - Qt 3D allows the developer to configure the rendering
    - Multiple render passes, multiple viewports
    - Any shader type supported (except compute shader)
- Optimized performance
  - Uses threads, running in multiple cores, for simulation tasks (jobs)
  - 3D assets optimizations

Qt Quick vs. Qt 3D Features

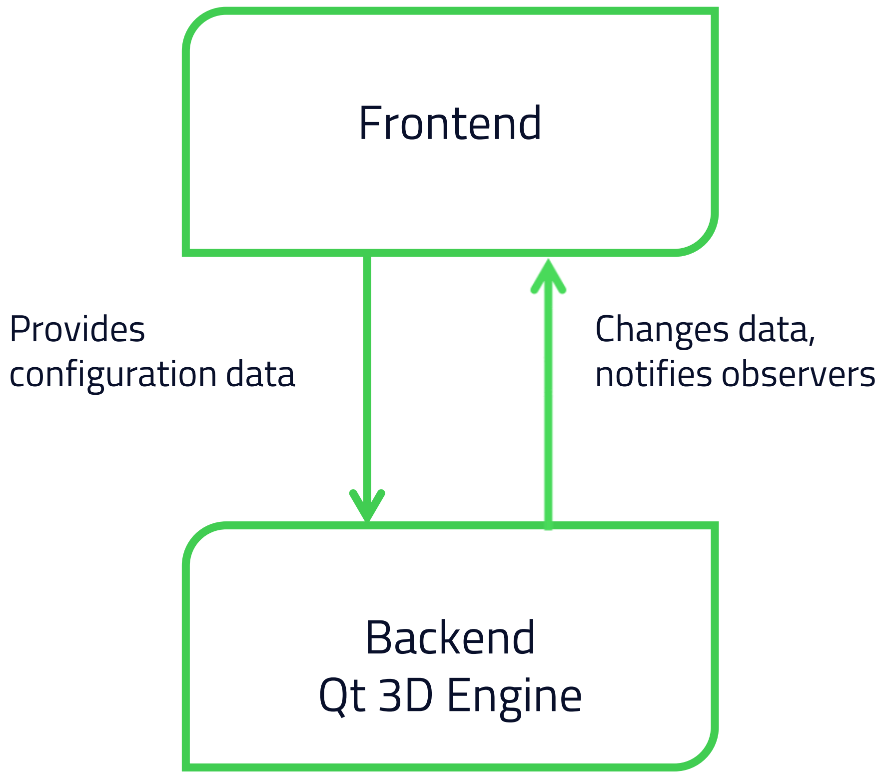
Feature	QML	Qt 3D	Remark
2D and 3D rendering	✓	✓	
Meshes	✓	✓	Meshes may be imported in Qt 3D
Materials	✓	✓	For Qt 3D, can be generated with `qgltf`
Shaders	✓	✓	Only vertex and fragments shaders in Qt Quick
Shadow mapping		✓	With multiple rendering passes
Ambient occlusion		✓	
High dynamic range		✓	
Deferred rendering	✓	✓	In QML, using FBOs
Multitexturing		✓	
Instanced rendering		✓	Not supported yet
Uniform buffer objects		✓	

Qt 3D Modules

- C++
  - Qt3DCore
  - Qt3DInput
  - Qt3DLogic – enables synchronizing frames with the Qt 3D backend
  - Qt3DRender
  - Qt3DExtras
- Qt Quick
  - Qt3D.Core
  - Qt3D.Input
  - Qt3D.Logic
  - Qt3D.Render

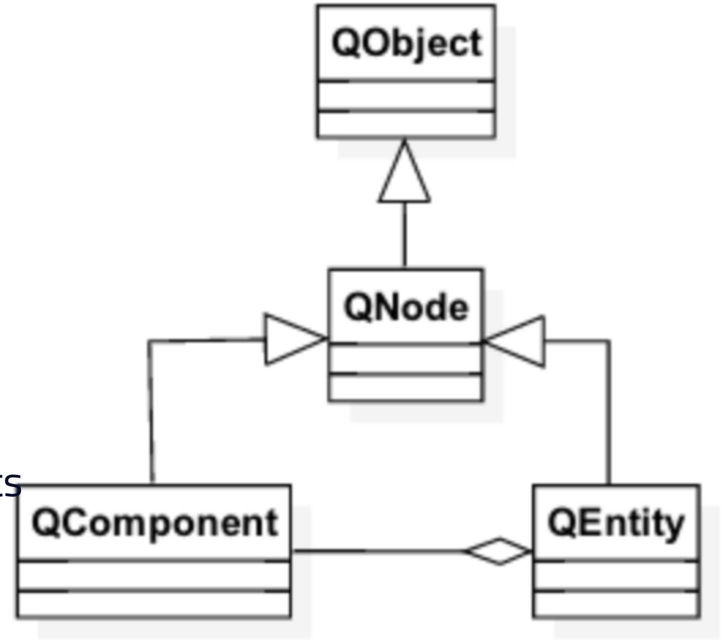
Architecture

- Frontend
  - Lightweight
  - `QObject` subclasses with properties, signals/slots
  - Non-blocking communication to the backend
- Backend
  - Running in a separate thread
  - Runs jobs for the frontend objects using the thread pool



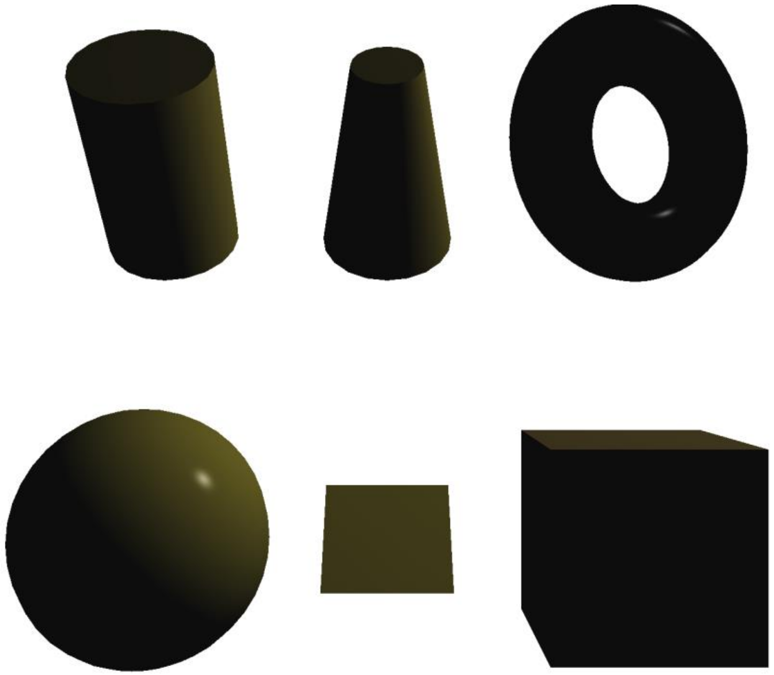
Architecture – Entity Component System (ECS)

- A Qt 3D scene consists of a hierarchy of nodes – no deep inheritance
  - Each node has a unique identifier,
  - Node property changes are notified to aspects (backend nodes) with NOTIFY signals
- An entity aggregates components in a scene
  - Components are basic frontend classes
  - An entity may be a 3D object, aggregating mesh, transform, and material components
- Frontend nodes are mapped to backend nodes by aspects
  - Define some slice of behavior, such as rendering, user input or audio
  - Aspects look for certain components in entities to gather the data, configuring the behavior
  - Frame or node property change trigger the backend to execute the aspect jobs



Creating a Scene

- Create an entity tree with a root entity
  - Add components with `QEntity::addComponent()`
- Mesh component - `QMesh`
  - Use `setSource(const QUrl &)` to load a mesh
  - Qt3DExtras provide a few pre-defined meshes: torus, cylinder, cone, cube, plane, sphere
- Material component - `QMaterial`
  - Use materials from Qt3DExtras or implement a custom material
- Transform component - `QTransform`
  - Use static functions to create a 4x4 matrix or quaternion



Creating a Scene

- Create and setup at least one camera entity - `QCamera`
  - Set camera position and projection

- Setup and configure each entity behavior
  - Create an aspect engine (`QAspectEngine`), handling the registered aspects
    - `QRenderAspect, QInputAspect, QLogicAspect, CustomAspect`
  - Configure the aspect behavior with components and related nodes and add components to your entities
    - `QRenderSettings, QInputSettings // examples of components`
    - `QForwardRenderer, QViewport, QAction, QAxis // examples of nodes`

- Finally, a window is needed - `QOpenGLWindow`, `QQuickWindow`
  - `Qt3DExtras::Qt3DWindow` creates a window with the default camera and render, input, and logic aspects
  - In Qt Quick apps, use `Qt3DExtras::Quick::Qt3DQuickWindow`

Good Practices

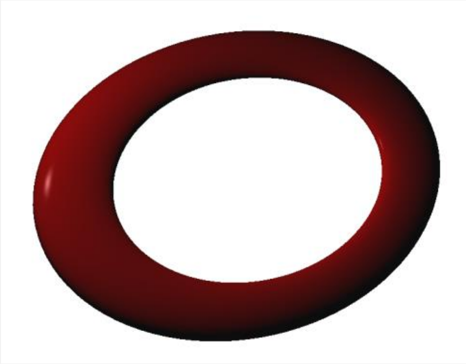
- Have a `Qt3DCore::QEntity` to represent the scene
- Have one `Qt3DCore::QEntity` per “object” in the scene
- Attach `Qt3DCore::QComponent` subclasses to objects to define the behaviour

Creating a Scene in C++

```
int main(int argc, char* argv[])
{
    QGuiApplication app(argc, argv);
    Qt3DExtras::Qt3DWindow view;
    Qt3DCore::QEntity scene;
    // Material, mesh, and transform properties omitted
    Qt3DRender::QMaterial *material = new Qt3DExtras::QPhongMaterial(&scene);
    Qt3DCore::QEntity *torusEntity = new Qt3DCore::QEntity(&scene);
    Qt3DExtras::QTorusMesh *torusMesh = new Qt3DExtras::QTorusMesh;
    Qt3DCore::QTransform *torusTransform = new Qt3DCore::QTransform;
    torusEntity->addComponent(torusMesh);
    torusEntity->addComponent(torusTransform);
    torusEntity->addComponent(material);

    Qt3DRender::QCamera *camera = view.camera(); // View crates a default camera
    camera->lens()->setPerspectiveProjection(45.0f, 16.0f/9.0f, 0.1f, 1000.0f);

    view.setRootEntity(&scene); view.show();
    return app.exec();
}
```



Creating a Scene in QML

```
QGuiApplication app(argc, argv);
Qt3DExtras::Quick::Qt3DQuickWindow view;
view.setSource(QUrl("qrc:/main.qml"));
view.show();
```

```
Entity {
    Camera {
        id: camera
        projectionType: CameraLens.PerspectiveProjection; fieldOfView: 45; aspectRatio: 16/9
        nearPlane: 0.1; farPlane: 1000.0; position: Qt.vector3d( 0.0, 0.0, -40.0 )
        upVector: Qt.vector3d(0.0, 1.0, 0.0); viewCenter: Qt.vector3d(0.0, 0.0, 0.0)
    }
    components: [
        RenderSettings { activeFrameGraph: ForwardRenderer { camera: camera } },
        InputSettings { } ]

    Entity {
        id: torusEntity
        components: [ torusMesh, material, torusTransform ]
    }
    PhongMaterial { id: material; diffuse: Qt.rgba(1.0, 0, 0, 0) }
    TorusMesh { id: torusMesh; radius: 5; minorRadius: 1; rings: 100; slices: 20 }
    Transform { id: torusTransform; scale3D: Qt.vector3d(1.5, 1, 0.5);
        rotation: fromAxisAndAngle(Qt.vector3d(1, 0, 0), 45) }
}
```


Mesh Loading

- Mesh is loaded by scene parser plugin
  - Assimp plugin uses Open Asset Library to load different asset formats
  - glTF plugin loads GL Transmission Format files
- Assets can be converted and optimized into glTF using `qgltf` tool
  - Can generate a compressed, binary JSON file, describing the scene
  - Can generate tangent vectors and scale vertices during build time
  - Can generate techniques and GLSL 1.00 and 1.50 shaders (materials)
  - Can compress texture assets (PNG only)

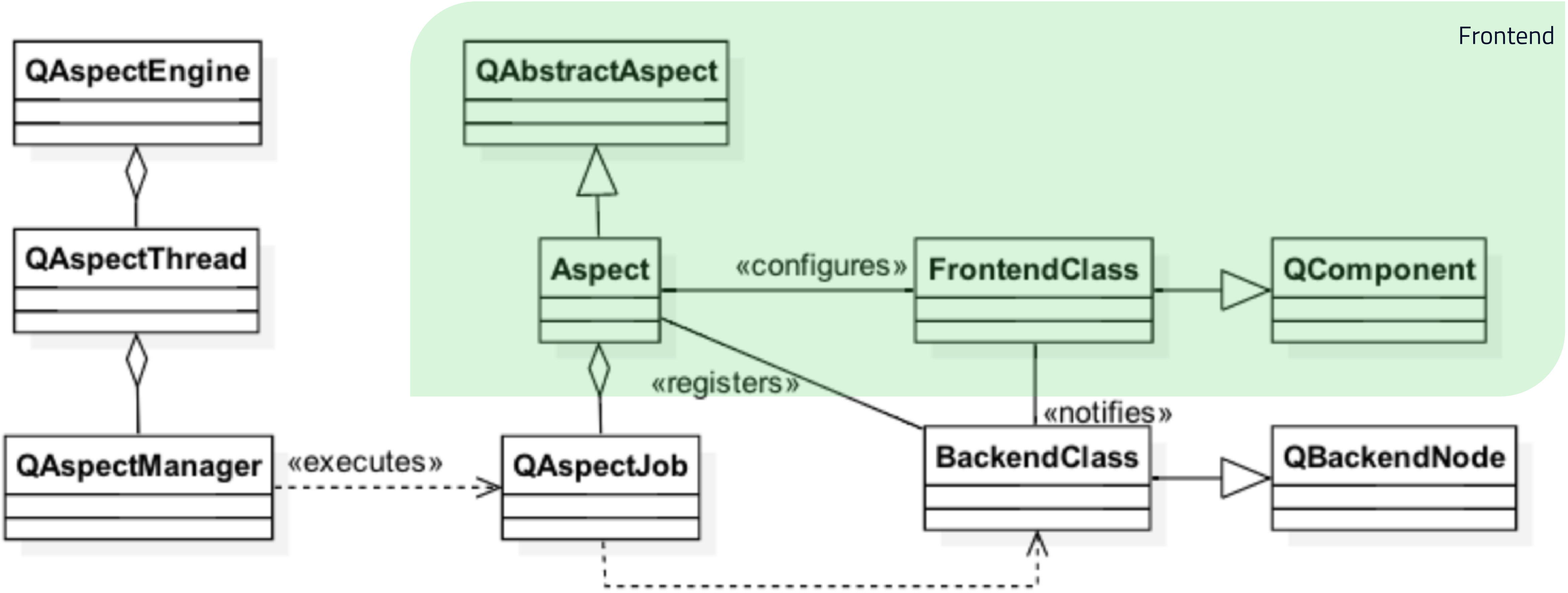
```
# .pro file
QT3D_MODELS += assets/3d/ferrari.obj assets/3d/lamborgini.obj
# -b use binary JSON file, -g generate OpenGL 3.2+ core profile shaders too
QGLTF_PARAMS = -b -g
load(qgltf)
```


Making the Scene Alive – Aspects

- Aspects (`QAbstractAspect`) define a vertical slide of behavior
  - Developers configure the behavior in aspect-specific components (frontend nodes), added to entities
  - The aspect maps frontend components to backend nodes
    - Changes in frontend nodes are notified to backend nodes
    - Backend nodes may notify frontend nodes as well
  - Behaviour comes from aspects processing component data

- Aspects are registered with `QAspectEngine`, who will become the owner of aspects
  - Registered aspects are accessed from `QAspectManager`, which is running the simulation event loop in the aspect thread
  - For each frame, the manager asks aspects for jobs to be executed by the thread pool threads

Qt 3D Backend Engine



QLogicAspect

- Handles frame synchronization jobs

- Only one frontend component – `QFrameAction`
  - Emits a signal just before the current buffer is swapped
  - Useful for animations synchronized with the Qt 3D engine backend
    - E.g. check input and transform an entity or camera in the slot

```
void AnEntity::init()
{
    m_frameAction(new Qt3DLogic::QFrameAction());
    // float argument gives the elapsed time since the last frame in ms
    QObject::connect(m_frameAction, SIGNAL(triggered(float)), this, SLOT(onTriggered(float)));
    addComponent(m_frameAction);
}
```


`QInputAspect`

- Two essential configuration components
  - `QInputSettings` define the event source
    - `Qt3DExtras::Qt3DWindow` sets the window itself as an event source
  - `QLogicalDevice` defines, which actions (`QAction`) an entity uses
    - The input aspect provides jobs to handle actions for the backend engine
    - Jobs notify frontend components about input events

- `QAction` links together action inputs, which trigger the same event
  - Action inputs store physical devices and buttons, triggering an event

- In addition to actions, input may affect the axis (`QAxis`)
  - Like actions, `QAxis` stores axis inputs, which trigger an event

Basic Input

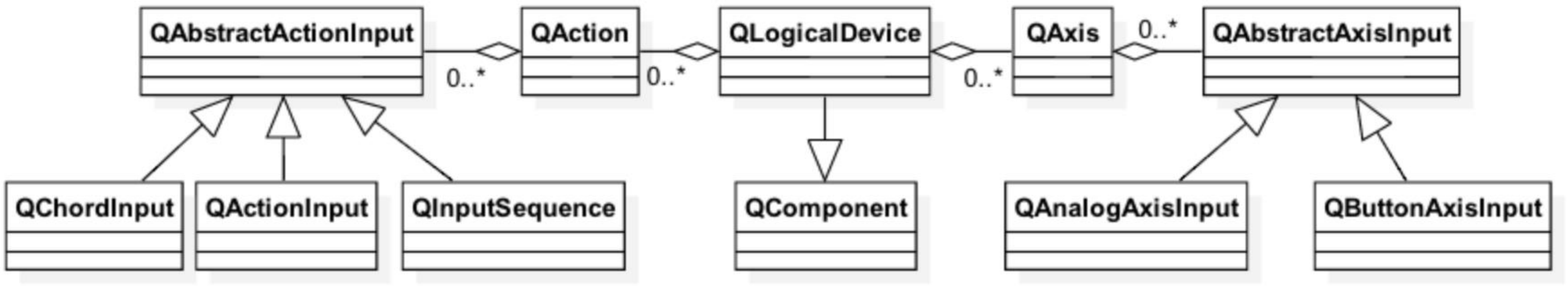
- Physical devices (`QMouseDevice, QKeyboardDevice, QGamepadInput`) dispatch events to input handlers
- Input handlers (`QKeyboardHandler, QMouseHandler`) provide event notifications when attached to physical devices
  - `clicked(), entered(), pressed(), released()`
- `Qt3DRender::ObjectPicker` component provides high level picking
  - Ray-cast-based picking

Advanced Input

› Provided by logical devices `QLogicalDevice`

› Maps physical devices

› Supports collections of actions and analog axis values



Using Input Aspect Front End

```
CameraController::CameraController(Qt3DCore::QNode *parent) :
    Qt3DCore::QEntity(parent),
    m_logicalDevice(new Qt3DInput::QLogicalDevice(),
    m_mouseDevice(new Qt3DInput::QMouseDevice()),
    m_leftMouseButtonAction(new Qt3DInput::QAction()),
    m_rxAxis(new Qt3DInput::QAxis()),
    m_mouseRyInput(new Qt3DInput::QAnalogAxisInput()) // + other actions, axes, and inputs
{
    // Left Mouse Button Action
    m_leftMouseButtonInput->setButtons(QVector<int>() << Qt::LeftButton);
    m_leftMouseButtonInput->setSourceDevice(m_mouseDevice);
    m_leftMouseButtonAction->addInput(m_leftMouseButtonInput);
    // Mouse X axis
    m_mouseRxInput->setAxis(Qt3DInput::QMouseDevice::X);
    m_mouseRxInput->setSourceDevice(m_mouseDevice); m_rxAxis->addInput(m_mouseRxInput);
    // Logical device
    m_logicalDevice->addAction(m_leftMouseButtonAction);
    m_logicalDevice->addAxis(m_rxAxis);
    addComponent(m_logicalDevice);
    // + other actions, axes, and inputs
}
```


Using Input Aspect Front End

```
void CameraController::frameActionTriggered(float dt)
{
    // Mouse input
    if (m_leftMouseButtonAction->isActive()) {
        if (m_rightMouseButtonAction->isActive()) {
            //Do something with the input value
            m_camera->translate(QVector3D(0, 0, m_ryAxis->value()),
                                               m_camera->DontTranslateViewCenter);
        }
    }
}
```


Custom Aspects

- Subclass at least `QAbstractAspect` and `QAspectJob`
  - Additionally, any number of backend classes, executed by the aspect engine and corresponding frontend classes with configurable properties

- `QAbstractAspect`
  - `onRegistered()` low the aspect to do some work now that it is registered
  - `onEngineStartup()`
  - Behavior consists of a set of jobs and commands, created in the backend nodes
    - `virtual QVector<QAspectJobPtr> jobsToExecute(qint64 time);`
    - `virtual QVariant executeCommand(const QStringList &args);`
- `QAspectJob`
  - Has pure virtual `run()` function
  - Jobs may have dependencies => queued by the same thread

Frontend

- Rather basic `QObject` subclass
  - Typically derived from `Node` or `Component`
  - Add properties for providing data to backend nodes
  - Add signals for backend communication and for notifying entities

```
class FrontendClass: public Qt3DCore::QComponent
{
    Q_OBJECT
    Q_PROPERTY(bool focus READ focus WRITE setFocus NOTIFY focusChanged)
public:
    explicit FrontendClass(QNode *parent = nullptr);
    ~FrontendClass();
Q_SIGNALS:
    void focusChanged(bool focus);
```

Aspect

- Map frontend and backend nodes
- Nodes are created, if needed
- Provide jobs
  - Jobs are executed for each frame

```
CustomAspect::CustomAspect(QObject *parent)
: QAbstractAspect(parent)
{
    registerBackendType<FrontendClass>(QBackendNodeMapperPtr(
        new BackendNodeFunctor(this, m_handler.data())));
}

QVector<QAspectJobPtr> CustomAspect::jobsToExecute(qint64 time)
{
    QVector<QAspectJobPtr> jobs;
    jobs.append(m_handler->jobs());
}
```


Backend Nodes

- Derive from `QBackendNode`
  - Use `sceneChangeEvent()` to notify property changes in the frontend class
  - `initializeFromPeer()` is used for further initialization after the backend has been created
  - Nodes or node ids often managed by a custom class

```
class BackendNode : public Qt3DCore::QBackendNode
{
public:
    BackendNode();
    void sceneChangeEvent(const Qt3DCore::QSceneChangePtr &e) Q_DECL_OVERRIDE;
protected:
    void requestFocus();
private:
    void initializeFromPeer(const Qt3DCore::QNodeCreatedChangeBasePtr &change) Q_DECL_FINAL;
    Handler *m_Handler;
    bool m_focus;
};
```

Backend Node Mapper

- Creates and maps backend nodes

```
Qt3DCore::QBackendNode *BackendFunctor::create(const Qt3DCore::QNodeCreatedChangeBasePtr &change)
const
{
    BackendNode *node = m_handler->resourceManager()->getOrCreateResource(change->subjectId());
    node->setCustomAspect(m_customAspect);
    node->setHandler(m_handler);
    return node;
}
Qt3DCore::QBackendNode *BackendFunctor::get(Qt3DCore::QNodeId id) const
{
    return m_handler->resourceManager ()->lookupResource(id);
}
void QBackendNode ::destroy(Qt3DCore::QNodeId id) const
{
    m_handler->removeBackendNode(m_handler->resourceManager()->lookupHandle(id));
    m_handler->resourceManager ()->releaseResource(id);
}
```


Jobs

› Implement the required behavior
› Notify frontend by using properties or signals

```
void Handler::keyEvent(const QKeyEventPtr &event)
{
    auto e = Qt3DCore::QPropertyUpdatedChangePtr::create(peerId());
    e->setDeliveryFlags(Qt3DCore::QSceneChange::DeliverToAll);
    e->setPropertyName("event");
    e->setValue(QVariant::fromValue(event));
    notifyObservers(e);
}

void AnotherHandler::triggerSignal()
{
   // Send an event to an object in GUI thread. That object asks frontend class to emit signals
   qApp->postEvent(m_notifier, new CustomEvent(m_dt));
}
```

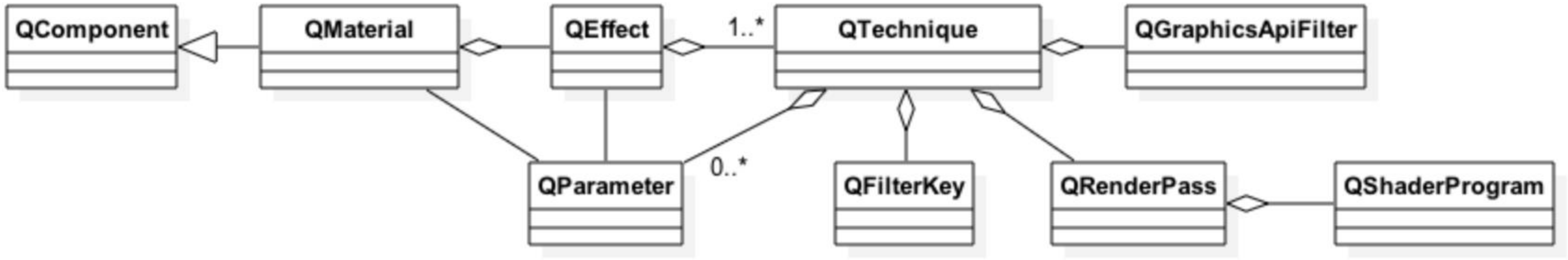

Materials and Effects

- Materials specify the rendering of an entity
  - `QPhong(Alpha)Material`, `QPerVertexColorMaterial`, `QDiffuseMapMaterial`, `QNormalDiffuseMapMaterial`
  - Effect defines with a technique, how the material is rendered
    - Easy to change the effect without touching any shader code

- Material properties are defined with `<name, value>` pairs – `QParameter`
  - Textures, lighting, colors
  - Parameters are exposed to shaders as uniforms
    - In a material class: `QParameter(QStringLiteral("aColor"), QColor::fromRgbF(0.01f, 0.01f, 0.01f, 1.0f))`
    - In a shader: `uniform vec4 aColor;`

Material Techniques

- Material is rendered using one of possibly several rendering techniques
  - `QGraphicsApiFilter` defines the required API for the technique
    - OpenGL version, profile, platform
  - `QFilterKey(s)` define, when the renderer should use the defined rendering configuration
  - `QRenderPass` defines the shader program and render state
    - `QRenderState` allows modifying global render state (blend, clip, stencil, color mask properties)



Custom Material

```
CustomMaterial::CustomMaterial(QNode *parent)
    : QMaterial(parent), m_materialGL3Shader(new QShaderProgram(this)), // allocate other members
{
    m_myParameter = new QParameter(QStringLiteral("parameterName"), 42);
    // Notify parameter changes with parameter-specific signals: void parameterNameChanged(int)
    connect(m_myParameter, &Qt3DRender::QParameter::valueChanged, this, … );

    m_materialGL3Shader ->setVertexShaderCode(QShaderProgram::loadSource(QUrl( … ));
    // Similarly shader program for OpenGLES etc.
    // Define the OpenGL version and profile required
    m_materialGL3Technique->graphicsApiFilter()->setApi(QGraphicsApiFilter::OpenGL);
    m_materialGL3RenderPass->setShaderProgram(m_materialGL3Shader);
    m_materialGL3Technique->addRenderPass(m_materialGL3RenderPass);

    m_filterKey->setName(QStringLiteral("renderingStyle"));
    m_filterKey->setValue(QStringLiteral("forward"));
    m_materialGL3Technique->addFilterKey(m_filterKey);

    m_customEffect->addTechnique(m_materialGL3Technique);
    m_customEffect->addParameter(m_myParameter);
    setEffect(m_customEffect);
}
```

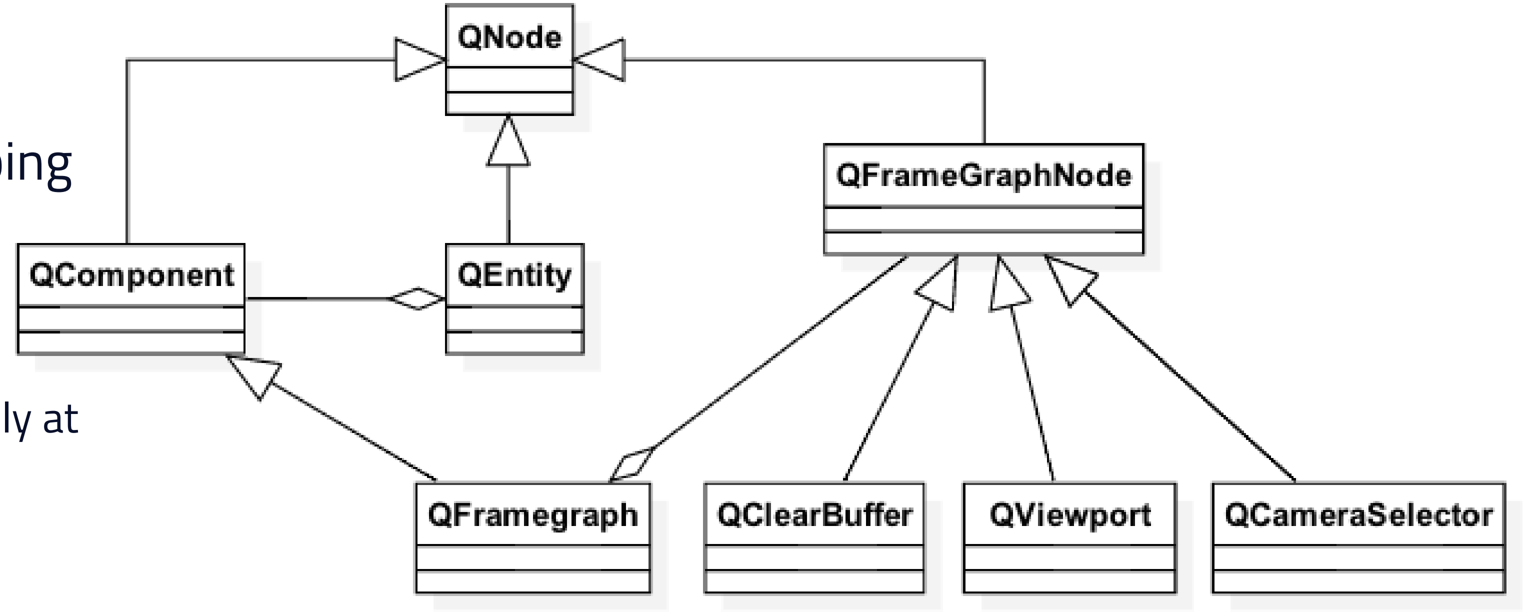
Textures and Lights

- Subclasses of `Qt3DRender::QAbstractTexture` provide various texture types
  - 1D – useful for lookup functions
  - 2D – most commonly used
  - 3D – typically volumetric data
  - Arrays of 2D…
- `Qt3DRender::QTextureLoader` supports all texture types loaded from DDS files

- The built-in materials work with lights - `Qt3DRender::QAbstractLight` component
  - Point light
  - Directional light
  - Spotlight
- Lights do not get rendered, we only see their effect on other entities

Rendering – Render Aspect

- Draws entities with a shape and material
- Uses a frame graph node for describing the rendering algorithm
  - Render settings component defines the active frame graph
  - Data-driven – can be changed dynamically at run time
- Several node types for selecting the camera, render target, layers of entities, viewport dimensions, render state, material etc.



Frame Graph

- Often rendering a single frame requires states and data sets
  - Each path to a render graph leaf defines a state (`RenderView`)
    - Renderer uses the depth-first search to collect state information from the graph nodes
  - Renderer collects all entities to be rendered into a set of `RenderCommands` and associates a `RenderView` with them
  - `RederView` and `RenderCommands` are passed over for submission to OpenGL

- Forward rendering, deferred rendering, reflections, shadows, multiple viewports etc.

- The actual shaders selected using `RenderPassFilter` and `Annotations`
  - Select only Entities in the scene that have a Material and Technique matching the annotations in the `RenderPassFilter`

Single Render Pass

```
Entity {
    id: sceneRoot
    Camera {
        id: camera
        projectionType: CameraLens.PerspectiveProjection
        fieldOfView: 45
        nearPlane : 0.1
        farPlane : 1000.0
        position: Qt.vector3d(0.0, 0.0, 40.0)
        upVector: Qt.vector3d( 0.0, 1.0, 0.0 )
        viewCenter: Qt.vector3d( 0.0, 0.0, 0.0 )
    }

    components: [
        RenderSettings {
            activeFrameGraph: ForwardRenderer {
                camera: camera
                clearColor: "transparent"
            }
        },
       InputSettings { }
]
```

Multiple Render Passes

```
RenderSettings {
    id: root
    readonly property Texture2D shadowTexture: depthTexture
    activeFrameGraph: Viewport {
        normalizedRect: Qt.rect(0.0, 0.0, 1.0, 1.0)
        RenderSurfaceSelector {
            RenderPassFilter {
                matchAny: [ FilterKey { name: "pass"; value: "shadowmap" } ]
                RenderTargetSelector {
                    target: RenderTarget {
                        attachments: [
                            RenderTargetOutput { objectName: "depth"
                                attachmentPoint: RenderTargetOutput.Depth
                                texture: Texture2D {} } ] }
                    ClearBuffers { buffers: ClearBuffers.DepthBuffer
                        CameraSelector { id: lightCameraSelector } } } }

            RenderPassFilter { matchAny: [ FilterKey { name: "pass"; value: "forward" } ]
                ClearBuffers { clearColor: Qt.rgba(0.0, 0.4, 0.7, 1.0)
                    buffers: ClearBuffers.ColorDepthBuffer
                CameraSelector { id: viewCameraSelector } } } } } }
```

Performance Considerations

- Startup time
  - Postpone memory allocations – use lazy loading
  - Use compressed texture formats
  - Use qltf for 3D assets
  - Shared backend nodes
- GUI thread
  - Do not overload GUI thread with notify events
- Rendering
  - Minimize the number of `RenderViews`
    - Put state that remains constant longest close to the root node
- Qt 3D profiler tool lab exists in code.qt.io

Summary

- Qt3D provides a general-purpose framework for creating near real-time 2D/3D simulations
- Compared to QML scene graph, which minimizes the number of state changes using batches, Qt3D uses a frame graph to make it easy to manage a set of states in a frame
- Developer provides an ECS tree, defining what to render (3D world)
- Qt3D engine uses aspects, which are component sets aggregating entities, to define the behavior of 3D world entities
- Render uses a frame graph to easily allow developers to change, how the world entities are rendered

Data Visualization

Contents

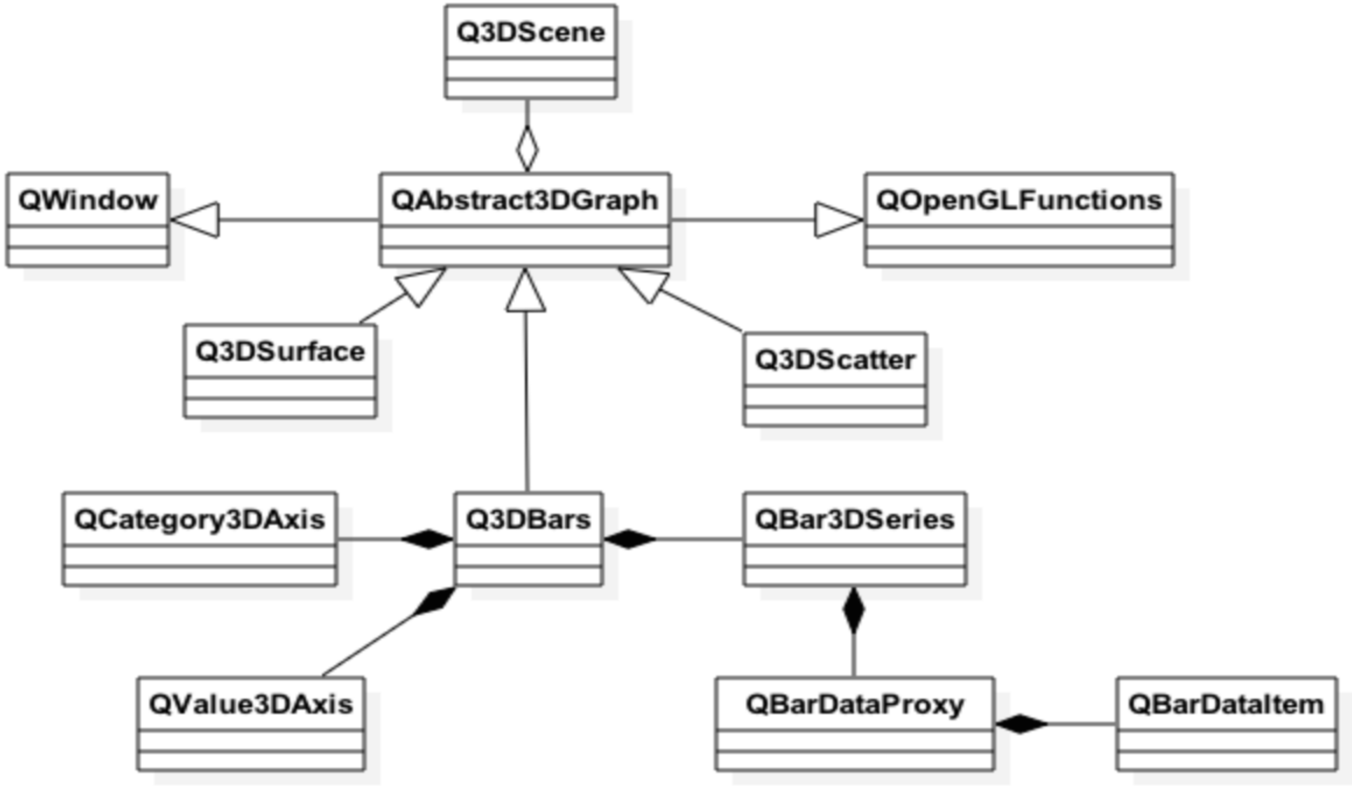
- Types
- Item Model Use
- Rendering
- Customization
- Performance

Data Visualization

- Module, allowing data to be visualized as 3D bars 3D scatter, and 3D surface
- Similar interactions to Qt Charts: rotate, zoom, data highlight
- OpenGL-based rendering
- Customizable: themes, input handling, items, labels
- 2D slice views of the 3D data
- Item model support
- Perspective and orthographic projections
- Volumetric custom items

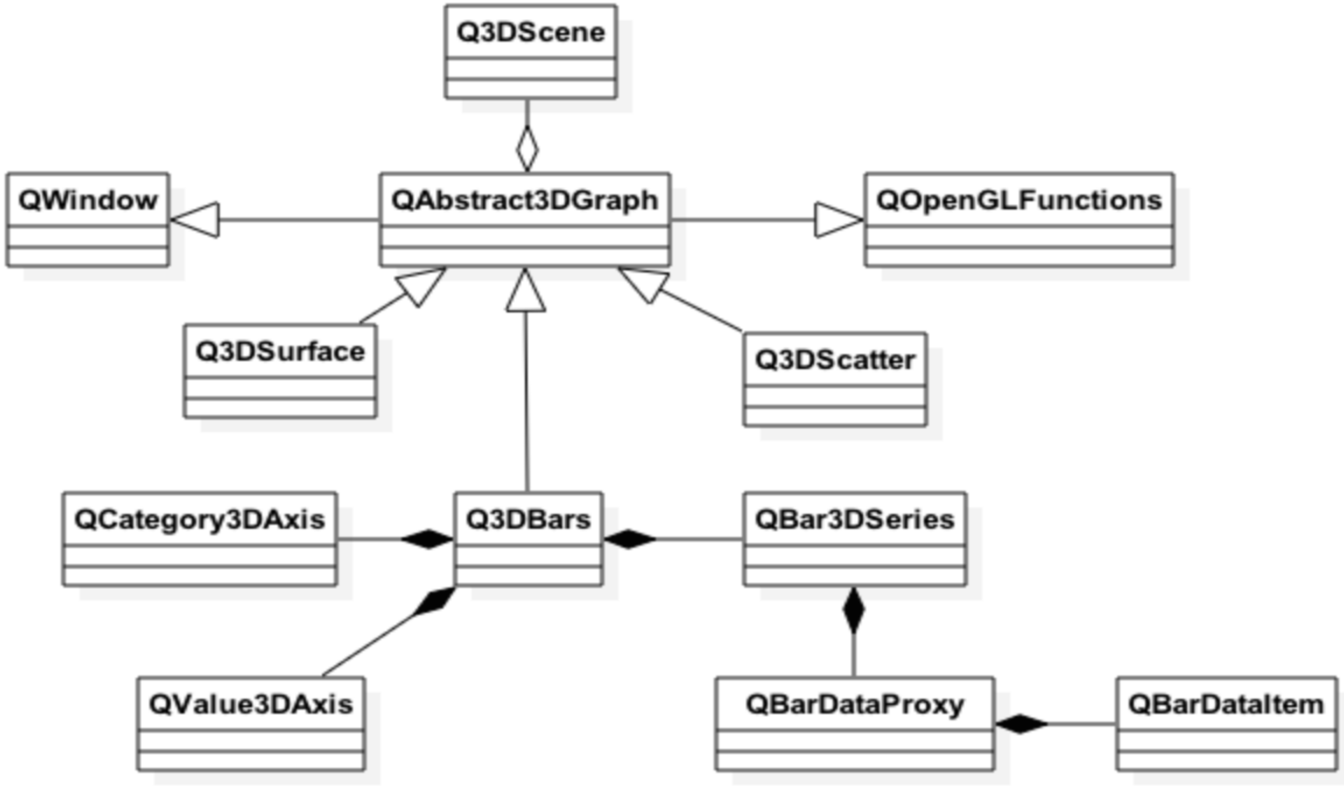
Architecture

- 3D data is renderer in one of `QWindow` sub-cl
  - Defines a render loop for the visualization type
  - Q3DBars (Bars3D), Q3DSurface (Surface3D), Q3DSca
  - By default a frameless window
  - By default anti-aliasing enabled
- Data items are handled with visualization type
  - `BarDataProxy`, `SurfaceDataProxy`, `ScatterI`
  - Uncreatable types, but provide properties for creatab



Architecture

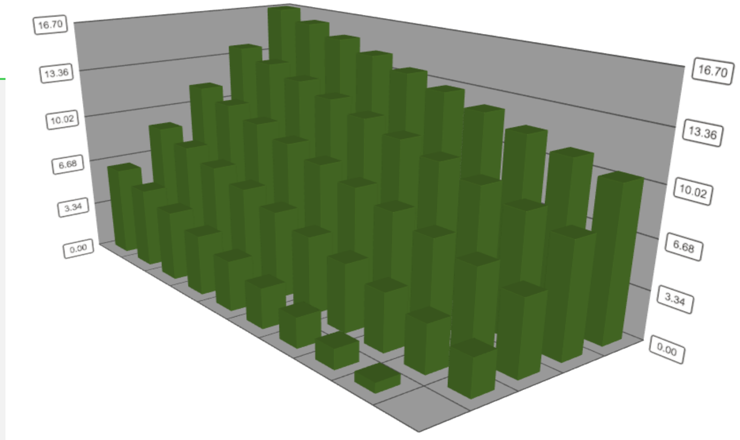
- Data items are provided to windows using data series
  - Defines how data items in the proxy should be rendered (`mesh`, `baseColor`, `itemLabel`, `itemLabelFormat`)
  - `Bar3DSeries`, `Surface3DSeries`, `Scatter3DSeries`
- Series may own only a single proxy at a time
- Windows may have several series though
- Switching series is typically more efficient than switching proxies



Data Visualization Hello World

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Q3DBars graph;
    QBarDataProxy *newProxy = new QBarDataProxy;
    QBarDataArray *dataArray = new QBarDataArray;
    dataArray->reserve(10);
    for (int i = 0; i < 10; i++) {
        QBarDataRow *dataRow = new QBarDataRow(5);
        for (int j = 0; j < 5; j++)
            (*dataRow)[j].setValue(0.7 * i + 2.6 * j);
        dataArray->append(dataRow);
    }
    newProxy->resetArray(dataArray);
    QBar3DSeries *series = new QBar3DSeries(newProxy);
    graph.addSeries(series);

    graph.show();
    return a.exec();
}
```



Data Handling in C++ Proxies

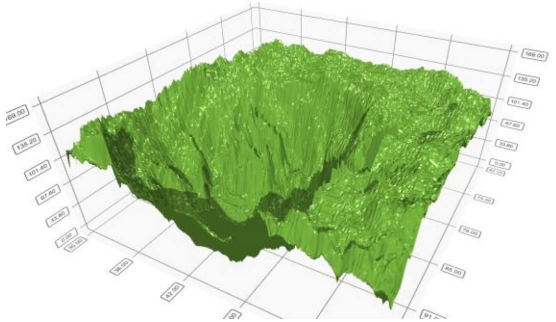
- Proxies handle adding, inserting, changing, and removing data items
  - Proxies use signals to notify the graph about data changes
  - If data items are manipulated directly without using proxy member functions, developer should take care of emitting the signals
- Bar data proxy
  - Owns `QBarDataItem` objects, providing two floats: value and rotation
  - Data may be added row by row using `QBarDataRow (QVector <QBarDataItem>)`
  - Several rows may be added using `QBarDataArray` container (`QList<QBarDataRow *>`)
  - Possible to store row and column labels as well
- In a similar way
  - Surface data items, providing a 3D position, may be added to the surface proxy
  - Scatter data items, providing a 3D position and rotation, may be added to the scatter proxy (`QScatterDataRow` does not exist though)

Data Mapping from Item Models

- Provided by item model data proxies, which are proxy sub-types
  - `ItemModelBarDataProxy`, `ItemModelSurfaceDataProxy`, HeightMapSurfaceDataProxy, `ItemModelScatterDataProxy`
- Set a model used by the proxy (QML model or `QAbstractItemModel`)
- Define how model items are mapped to data visualization item fields (rows, columns, value)

1. Use model categories (`useModelCategories = true`)
   - In C++, `QAbstractItemModel` rows and columns mapped to C++ `Q3D<visualization type>` rows and columns
   - In QML, custom categories ignored
2. Define, which item model roles are mapped to graph rows, columns, and data
   - `rowRole`, `columnRole`, `valueRole`
3. Define an explicit list of categories for rows and columns
   - Allows defining which rows and columns are included and in which order

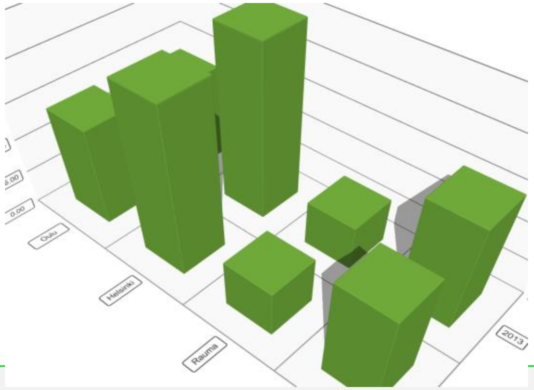
Data Visualization Hello World in QML



- Using `HeightMapSurfaceDataProxy`
- Y value read from the height map file
  - Preferred format `QImage::Format_RGB32` in grayscale (height value from red component)
- X and Z range set manually (or with default values)

```
Surface3D {
    id: surfacePlot
    width: surfaceView.width; height: surfaceView.height
    Surface3DSeries {
        id: heightSeries
        drawMode: Surface3DSeries.DrawSurface
        HeightMapSurfaceDataProxy {
            heightMapFile: ":/heightmaps/image”
            minZValue: 30; maxZValue: 60; minXValue: 67; maxXValue: 97
        }
    }
}
```


Data Visualization Hello World 2 in QML



- Same role values can be handled using `multiMatchBehavior`
  - First or last value used, average or cumulative value

```
Bars3D {
    width: parent.width height: parent.height
    Bar3DSeries {
        itemLabelFormat: "@colLabel, @rowLabel: @valueLabel”
        ItemModelBarDataProxy {
            itemModel: dataModel; autoColumnCategories: false
            rowRole: "year”; columnRole: "city”; valueRole: "expenses"
            rowCategories: ["2010", "2011", "2012", "2013"]
            columnCategories: ["Oulu", "Helsinki", "Rauma", "Tampere"]

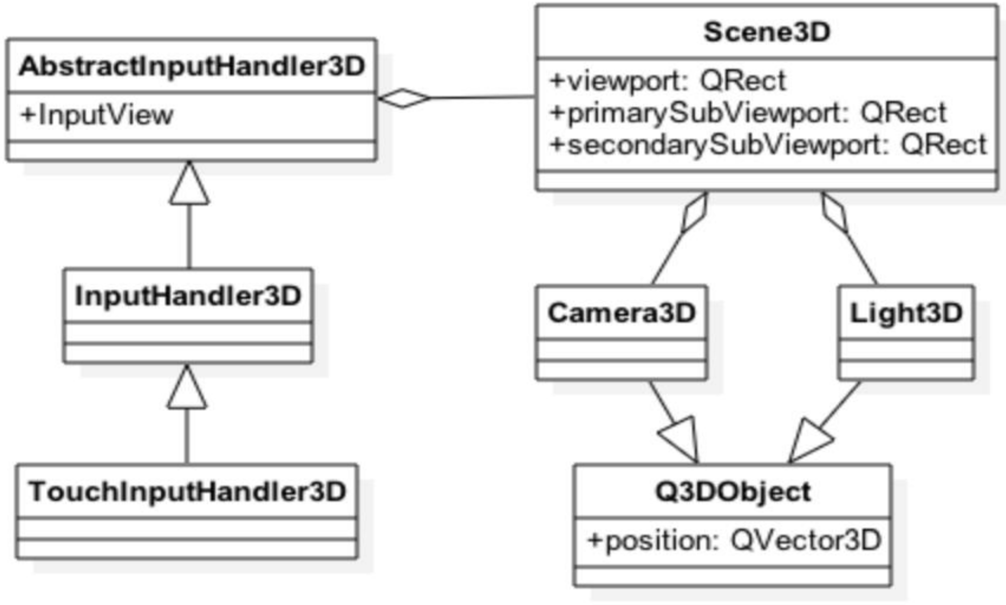
ListModel {
    id: dataModel
    ListElement{ year: "2012"; city: "Oulu"; expenses: "4200"; }
    ListElement{ year: "2012"; city: "Rauma"; expenses: "2100"; }
    ListElement{ year: "2012"; city: "Helsinki"; expenses: "7040"; }
    // Clipped
}
```


Sharing Roles with Several Categories

- Possible to use regular expressions search patterns and replace rules to format the value for each role before it is used in the category
- For example, we want to share timestamp role (date) with two categories: year and month

```
ListElement{ timestamp: "2006-01"; expenses: "-4"; income: "5" }
ItemModelBarDataProxy {
    itemModel: graphData.model
    rowRole: "timestamp”; columnRole: "timestamp”; valueRole: "expenses”
    rowRolePattern: /^(\d\d\d\d).*$/
    columnRolePattern: /^.*-(\d\d)$/
    valueRolePattern: /-/
    rowRoleReplace: "\\1"
    columnRoleReplace: "\\1"
    multiMatchBehavior: ItemModelBarDataProxy.MMBCumulative
}
```

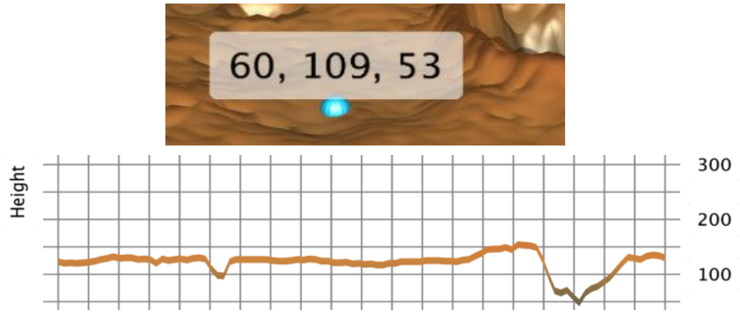
Data Interactions



- Implemented in input handlers
  - Translate events to scene camera and light movements
  - And to slicing and selection events in the scene
- Scene3D
  - Defines a single active camera and light source
  - Keeps track of the viewport in which visualization rendering is done
- Camera3D represents an orbit around a centerpoint, where data items are rendered
  - Properties: `zoomLevel(100.0)` in percentage, `minZoomLevel(10.0)`, `maxZoomLevel(500.0)`, `target(0.0, 0.0, 0.0)`, `xRotation` and other rotations around the target point
- Light 3D represents a monochrome light source

Selections

- All graphs support selection of a single data item
- Bar and surface graphs support also slice selection mode
  - Enumeration in `AbstractGraph3D`
  - Selected row or column drawn in a separate viewport as a 2D graph
- Signals may be used to customize the selection behavior
  - `Bar3DSeries.onSelectedBarChanged()`, `Scater3DSeries.onSelectedItemChanged()`, `Surface3DSeries.onSelectedPointChanged()`
- Mode: `SelectionItem`
- Mode: `SelectionSlice|SelectionRow`

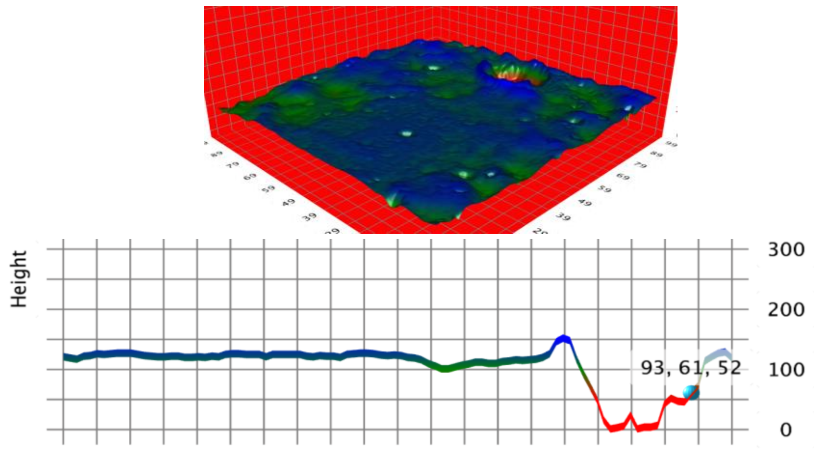


Customization - Theming

- Set a built-in theme or a custom theme to `theme` (`Theme3D`) property of your graph
  - Affects the whole graph
  - Built-in themes: `ThemeQt`, `ThemePrimaryColors`, `ThemeDigia`, `ThemeStoneMoss`…

- Some `Theme3D` properties: `baseColors`, `baseGradients`, `colorStyle`(uniform, object gradient, range gradient), `font`, `gridEnabled`, `lightColor`, `singleHighlightColor`

```
Surface3D {
    theme: Theme3D {
        type: Theme3D.ThemePrimaryColors
        backgroundColor: "red"
        font.family: "STCaiyun"; font.pointSize: 35
        colorStyle: Theme3D.ColorStyleRangeGradient
        baseGradients: [surfaceGradient] }
```



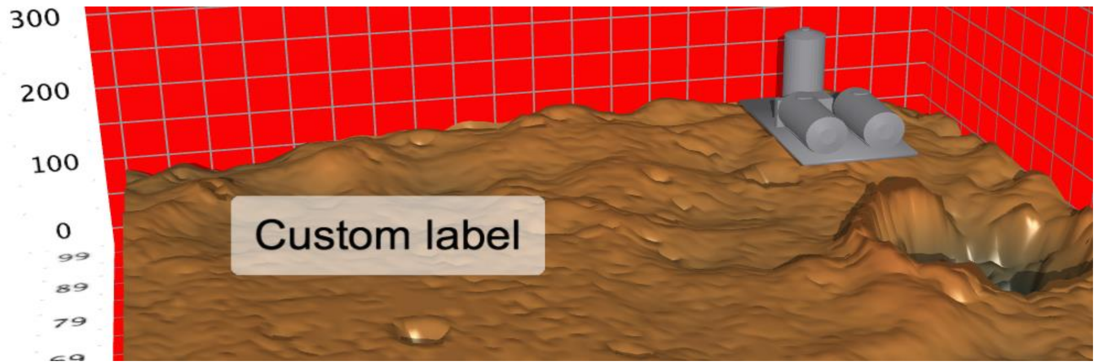
Customization – Items

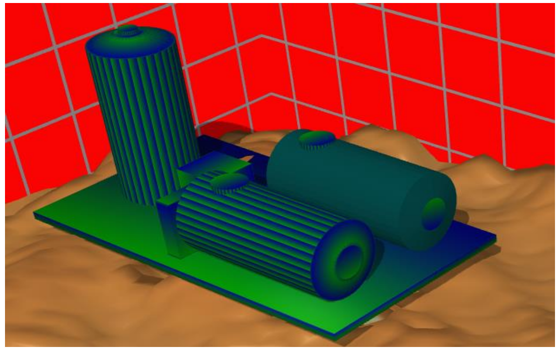
- Plenty of pre-defined item mesh types
  - Rectangular bar, cube, triangular pyramid, four-sided pyramid, 2D point, cone, cylinder, sphere, arrow pointing upwards

- Custom mesh may be defined as well
  - `Bar3DSeries.mesh: Bar3DSeries.MeshUserDefined`
  - `Bar3DSeries.userDefinedMesh(“qrc:/meshes/coolMesh.obg”)`
    - Mesh needs to include vertices, normals, and UVs (texture coordinates) and needs to be in triangles

- Colors, gradients, and highlight colors can be defined as in theming

Customization – Items





- It is possible to add custom items to the graph
  - With a mesh and optional texture file
  - Without texture file, solid gray color is used

```
Surface3D {
    customItemList: [
        Custom3DLabel {
            text: qsTr("Custom label")
            facingCamera: true
            scaling: Qt.vector3d(1.0, 1.0, 1.0)
            position: Qt.vector3d(20.0, 400.0, 2.0) },
        Custom3DItem {
            position: Qt.vector3d(80.0, 150.0, 80.0)
            meshFile: ":/refinery.obj”
        }
    ]
}
```


Customization – Other Properties

- Shadows
  - `AbstractGraph3D.shadowQuality`: disabled, low, medium, high
- Reflections
  - Affects only `Bars3D` containing only positive or negative values (not both)
  - `Bars3D.reflection: true; Bars3D.reflectivity: 0.9`
- Locale
  - Affects number formats
- Margin
  - Between the graph itself and possible labels
  - Defined as a fraction of Y axis range
- FPS
  - Shows `currentFps`, if turned on

Performance

- Try to re-use existing data sets (possibly with new values) between frames to minimize the number of memory allocations
- Commit data changes in batches (e.g. `addRows()`) in data proxies, if those are used

- Adding data to bars or surface data proxies does not have a big effect on rendering performance
  - Adding data similarly to the scatter data proxy requires all data points to be checked for visibility
  - It is not recommended to continuously add data items to the scatter data proxy
- The best performance is achieved by rendering a graph or graphs to the window background
  - `AbstractGraph3D.renderingMode`
  - Background rendering disables some item behavior like Z-order

- Rendering optimization
  - `AbstractGraph3D.renderingOptimization`
  - Default
  - Static: optimal for large non-changing data sets on Scatter graphs

Limitations

- Re-parenting a graph to an item in another window not supported
- Only OpenGL ES emulation available for SW renderer
- Items outside straight rows or columns on the surface graphs may get clipped incorrectly
- Surfaces with a lot of visible vertices may not render, as the per-draw count is exceeded
- Some limitations in OpenGL ES (and Angle builds)
  - Shadows not supported
  - Anti-aliasing does not work
  - Custom3DVolume not supported

Thank you

NN@qt.io

