Introduction to Qt 3D

ACCU 2017, 29 April 2017

Presented by Giuseppe D'Angelo <giuseppe.dangelo@kdab.com>

Overview of Qt 3D

- **The Story of Qt**
- Overview of Qt 3D
- Drawing with Qt 3D
- The Qt 3D Frame Graph
- The Future of Qt 3D

Meet Qt

- Started in 1994 by Trolltech
- In January 2008, Trolltech was acquired by Nokia.
- In October 2011, Qt was opened to the community through qt-project.org.
- In August 2012, Qt was acquired by Digia.
- In September 2014, Qt activities were transferred to The Qt Company.

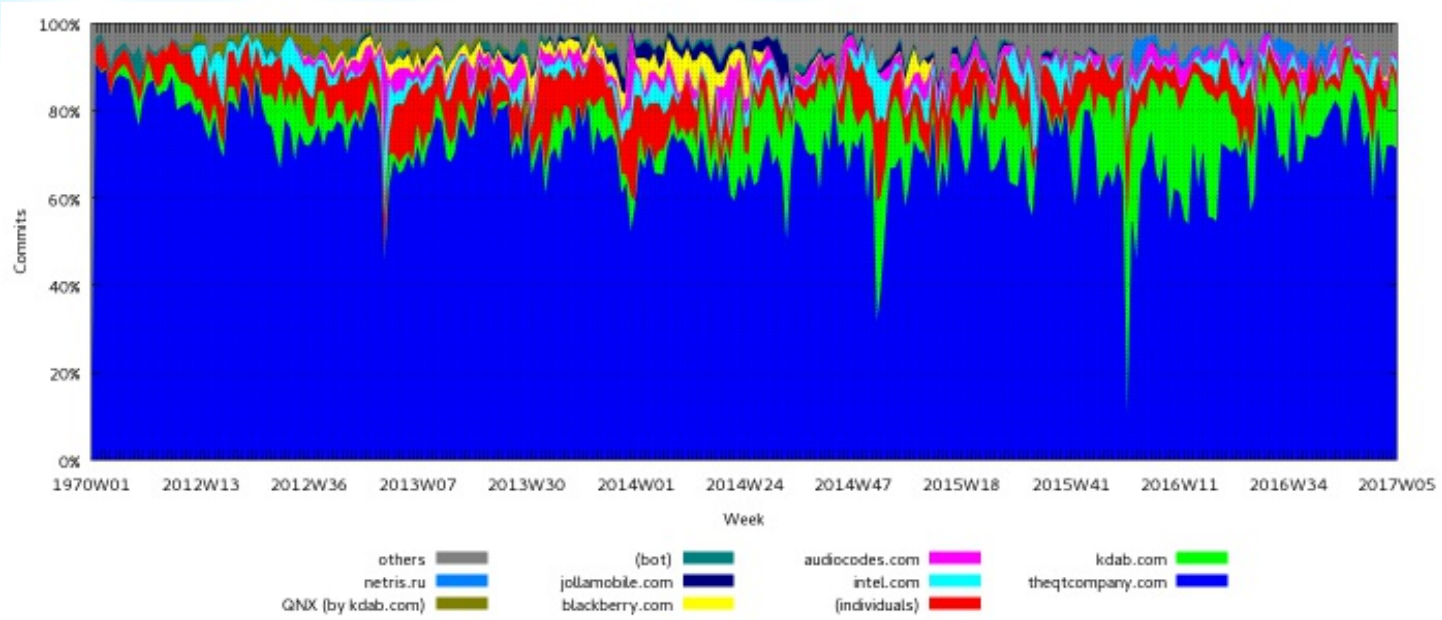
Why Qt?

- Write code once to target multiple platforms.
- Produce compact, high-performance applications.
- Focus on innovation, not infrastructure coding.
- Choose the license that fits you.
  - Commercial, LGPL or GPL
- Count on professional services, support and training.

15 years of customer success and community growth

Qt as a Community Project

- The Qt Company is not the only company developing Qt.
- KDAB is the second biggest contributor.
- Many other organizations and individuals contribute.
  - From bugfixes to entire modules



Contributions per employer, from the beginning to Q1 2017

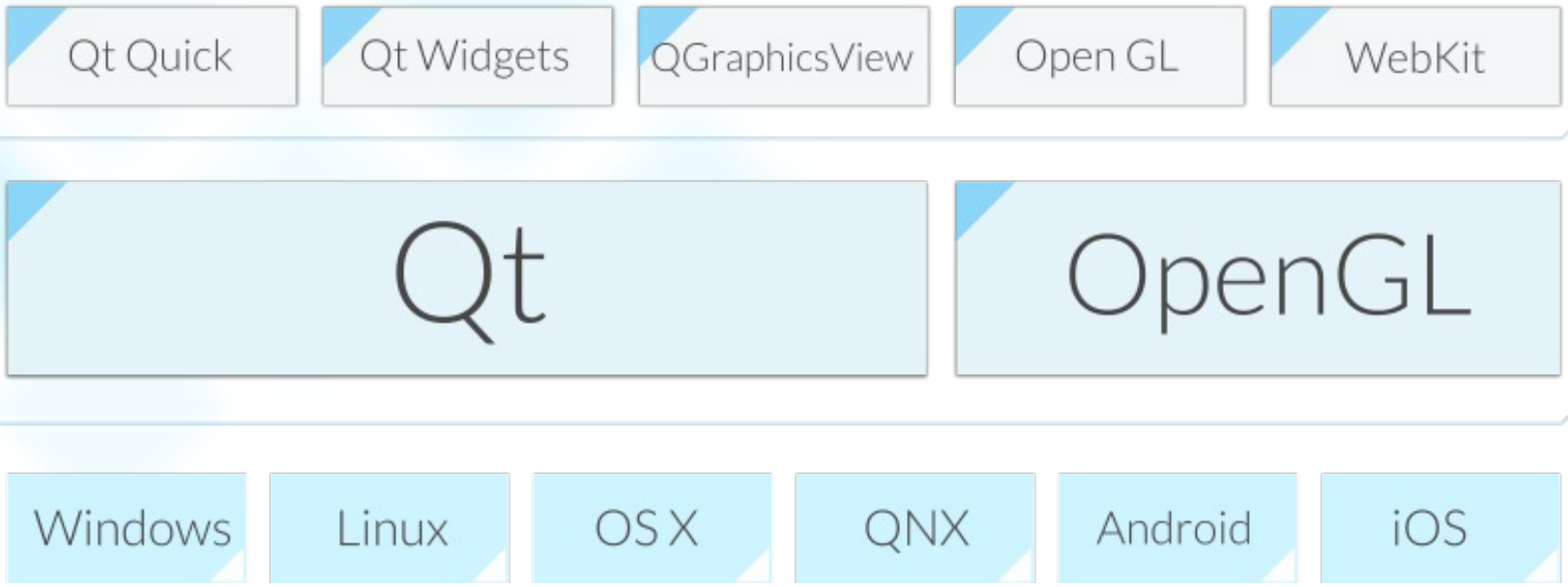
Qt is used Everywhere

From embedded devices to desktop applications

By companies from many industries

AMD, Google, Alcatel-Lucent, Canon, esa, VOLVO, BT, EPSON, DAIMLER, Schlumberger, HONDA, xerox, MITSUBISHI, vodafone, skype, PHILIPS, REUTERS, Mentor Graphics, SHARP, freescale semiconductor, ORACLE, ABB, SENNHEISER, SCANIA, PGS

Qt Architecture



GUI Technologies usable with Qt

- **QtWidgets**
  - Desktop integration
  - Mature layouts
  - C++ API
- **QtQuick**
  - OpenGL and scene graph based
  - Design oriented
  - Declarative language and JavaScript

What is Qt Quick?

A set of technologies including:

- Declarative markup language: QML
- Language runtime integrated with Qt
- Qt Creator IDE support for the QML language
- Graphical design tool
- C++ API for integration with Qt applications

Philosophy of Qt Quick

- Intuitive user interfaces
- Design-oriented
- Rapid prototyping and production
- Easy deployment

- The Story of Qt
- **Overview of Qt 3D**
  - Feature Set
  - Entity Component System? What's that?
  - Hello Donut
  - Qt 3D ECS Explained
- Drawing with Qt 3D
- The Qt 3D Frame Graph
- The Future of Qt 3D

- **Feature Set**
- Entity Component System? What's that?
- Hello Donut
- Qt 3D ECS Explained

What is Qt 3D?

- It is not about 3D!
- Multi-purpose, not just a game engine
- Soft real-time simulation engine
- Designed to be scalable
- Extensible and flexible

Simulation Engine

- The core is not inherently about 3D
- It can deal with several functional domains at once
  - AI, logic, audio, etc.
  - And of course it contains a 3D renderer too!
- All you need for a complex system simulation
  - Mechanical systems
  - Physics
  - ... and also games

Scalability

- Frontend / backend split
  - Frontend is lightweight and on the main thread
  - Backend executed in a secondary thread
    - Where the actual simulation runs
- Non-blocking frontend / backend communication
- Backend maximizes throughput via a thread pool

Extensibility and Flexibility

- Functional domains can be added by extending the runtime
  - ... only if there's not something fitting your needs already
- Provide both C++ and QML APIs
- Integrates well with the rest of Qt
  - Pulling your simulation data from a database anyone?
- Entity Component System is used to combine behavior in your own objects
  - No deep inheritance hierarchy

Why Qt 3D?

- Low level OpenGL code is tedious and error prone to write
- Deep integration with Qt and Qt Quick, not a black box
- Work on constrained resources
- Focus on innovation, not on plumbing

Use cases

- Automotive IVI
- Scientific, medical visualizations
- Machine status displays
- Interactive manuals
- Augmented reality (AR)
- ...

- Feature Set
- **Entity Component System? What's that?**
- Hello Donut
- Qt 3D ECS Explained

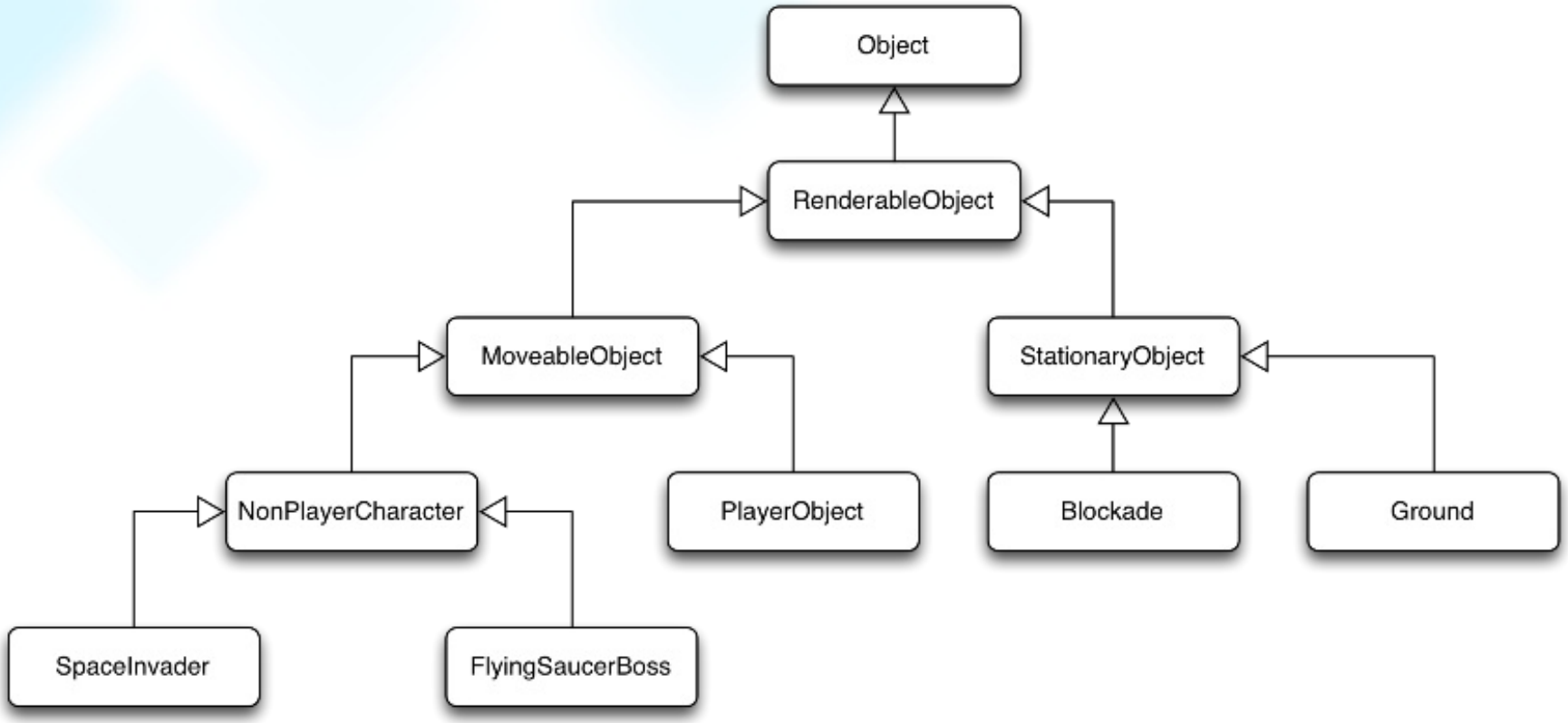
ECS: Definitions

- ECS is an architectural pattern
  - Popular in game engines
  - Favors composition over inheritance
- An entity is a general purpose object
- An entity gets its behavior by combining data
- Data comes from typed components

- Let's analyse a familiar example: Space Invaders

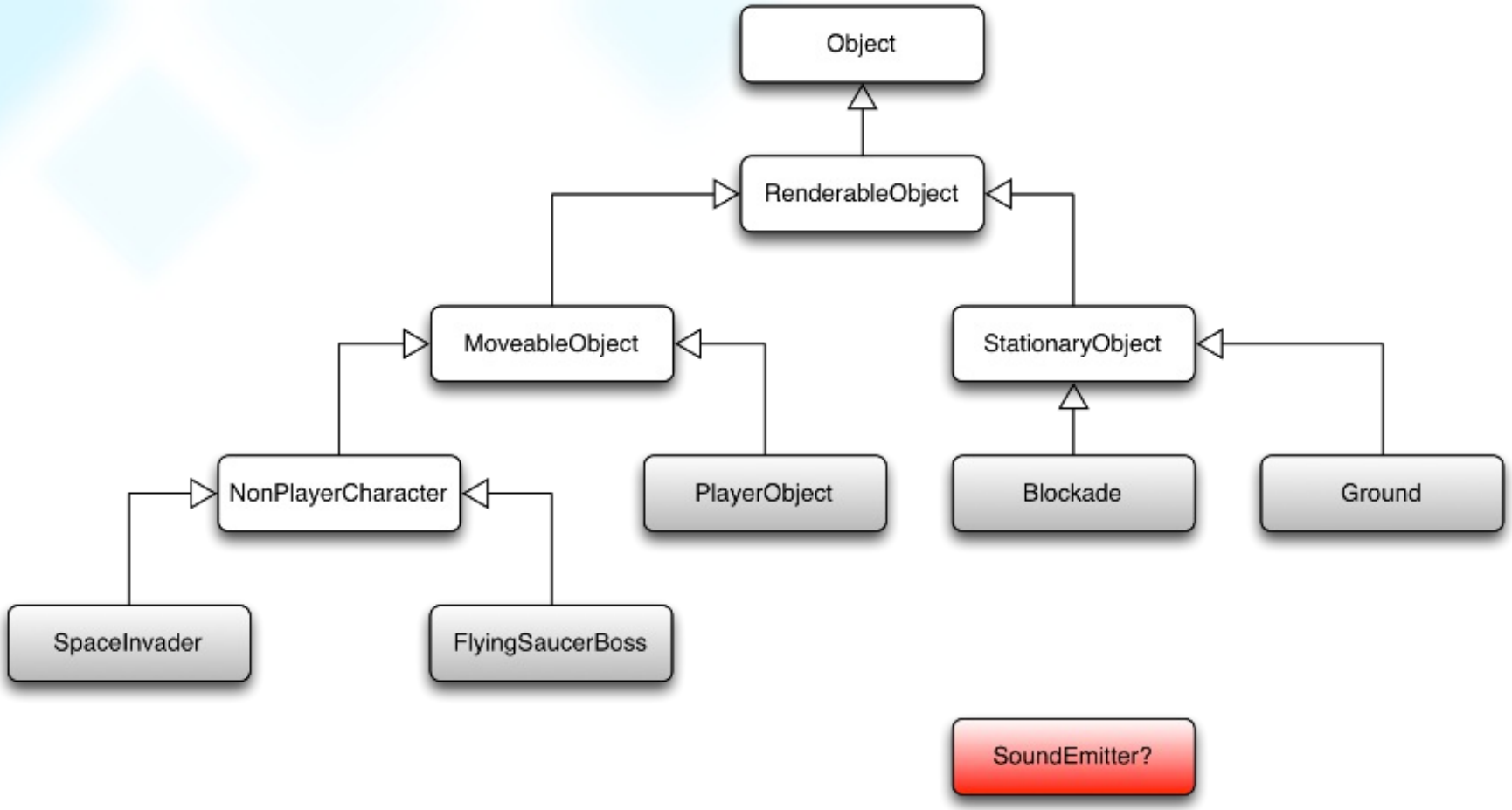
Composition vs Inheritance cont'd

- Typical inheritance hierarchy



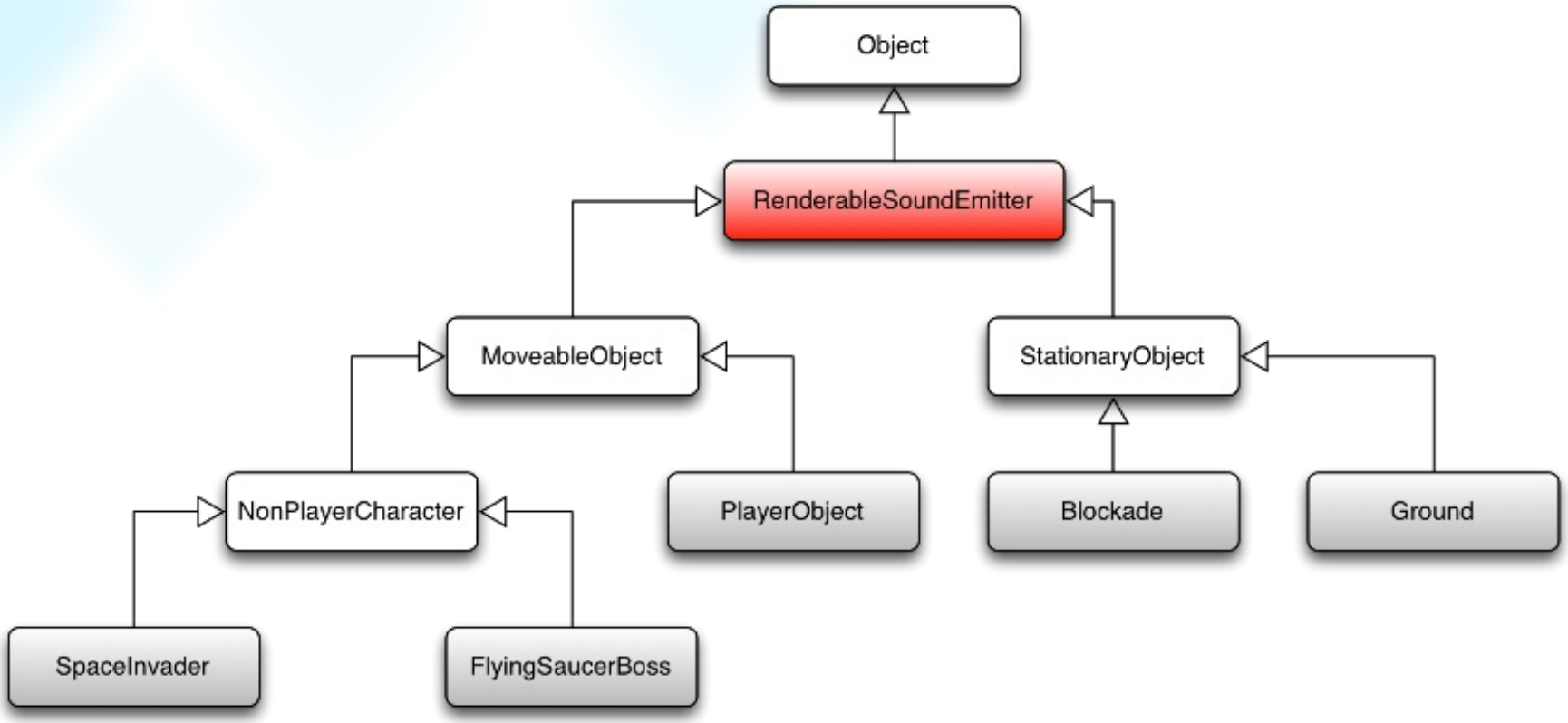
Composition vs Inheritance cont'd

- All fine until customer requires new feature:



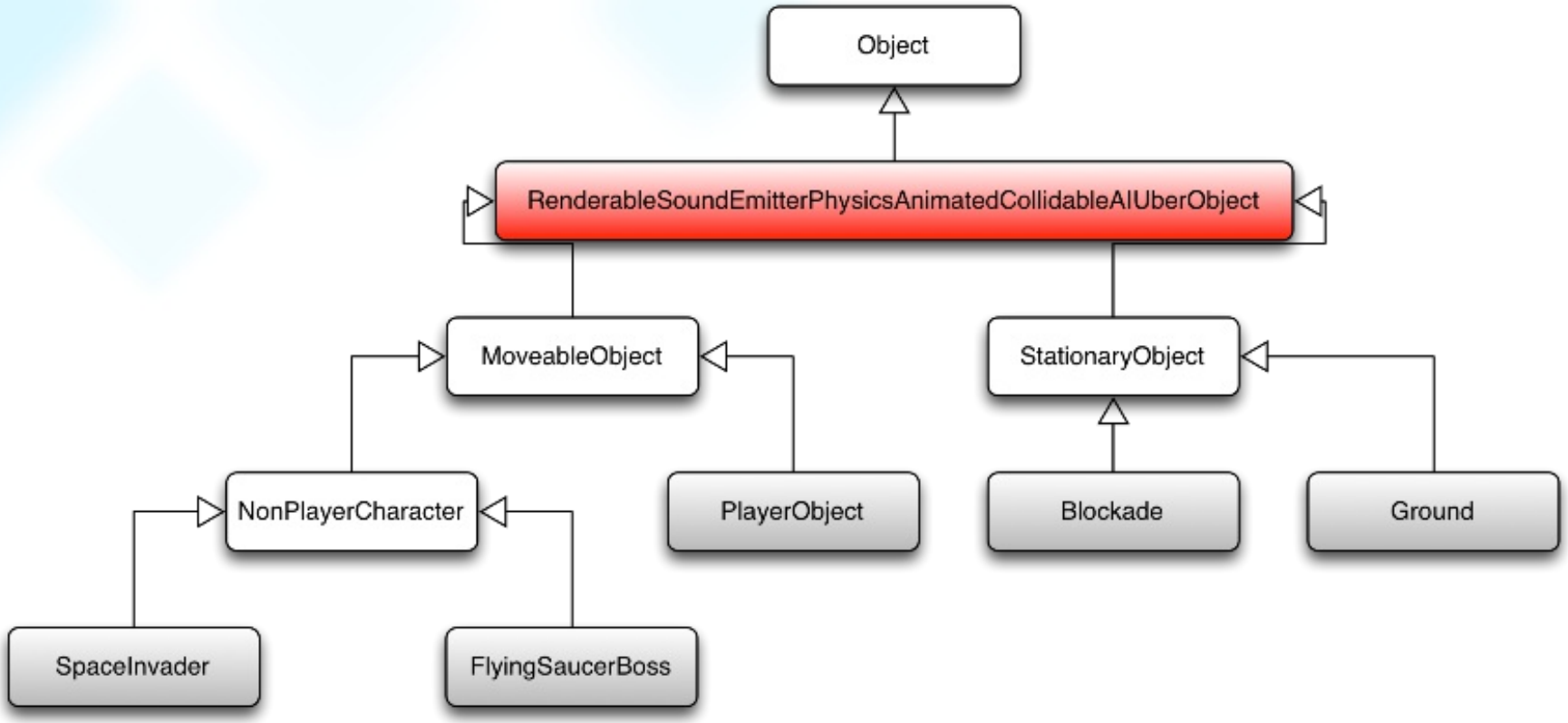
Composition vs Inheritance cont'd

- Typical solution: Add feature to base class



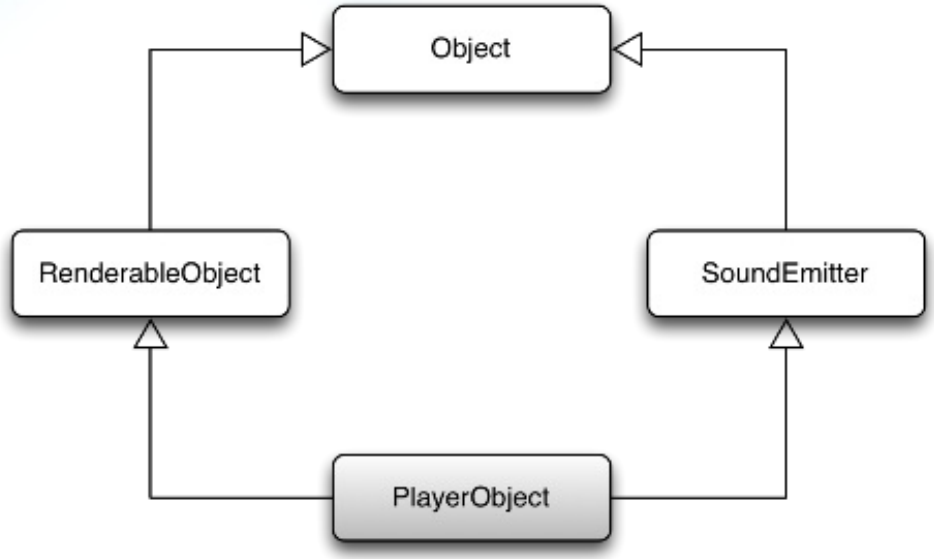
Composition vs Inheritance cont'd

- Doesn't scale:



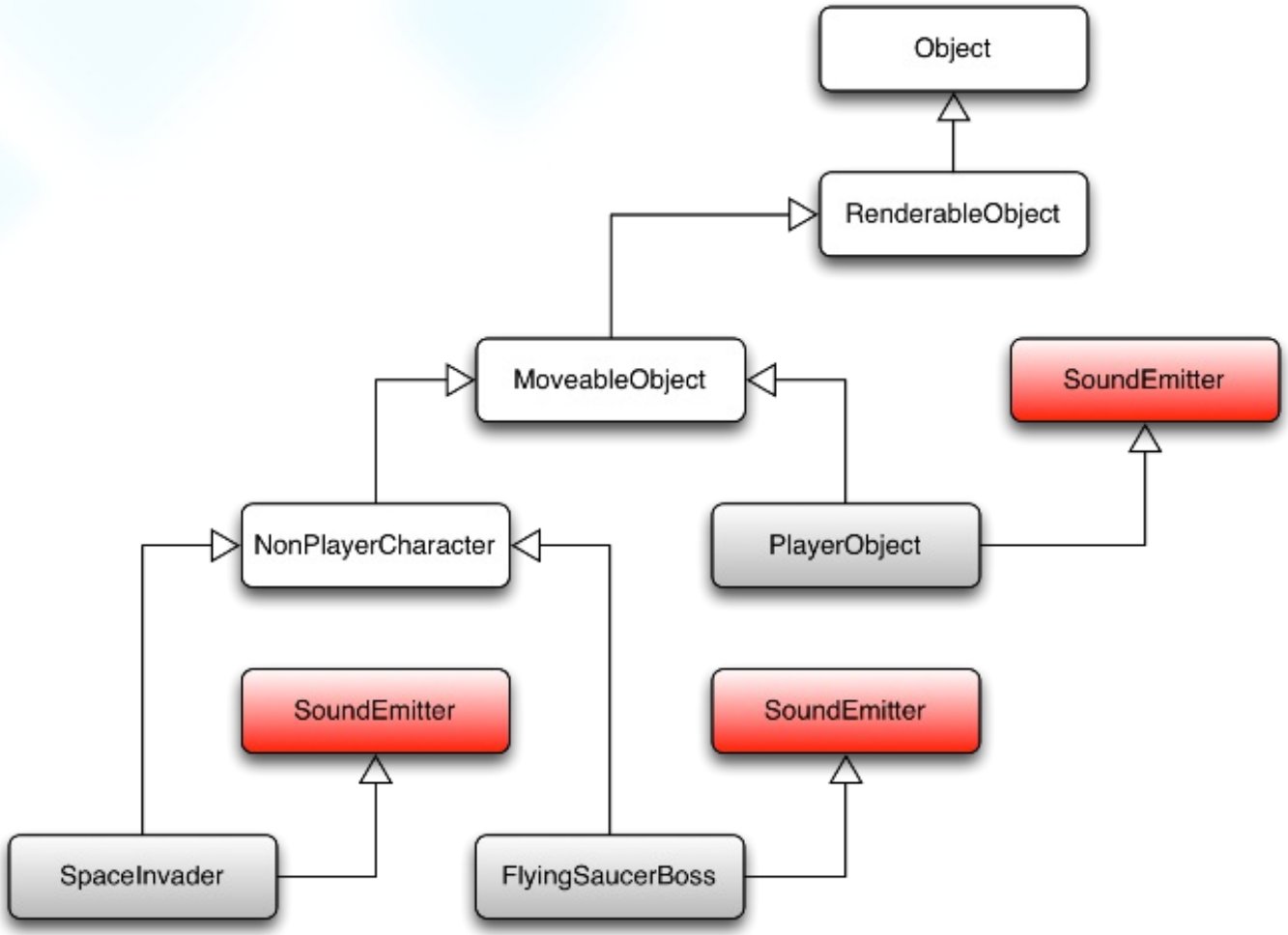
Composition vs Inheritance cont'd

- What about multiple inheritance?



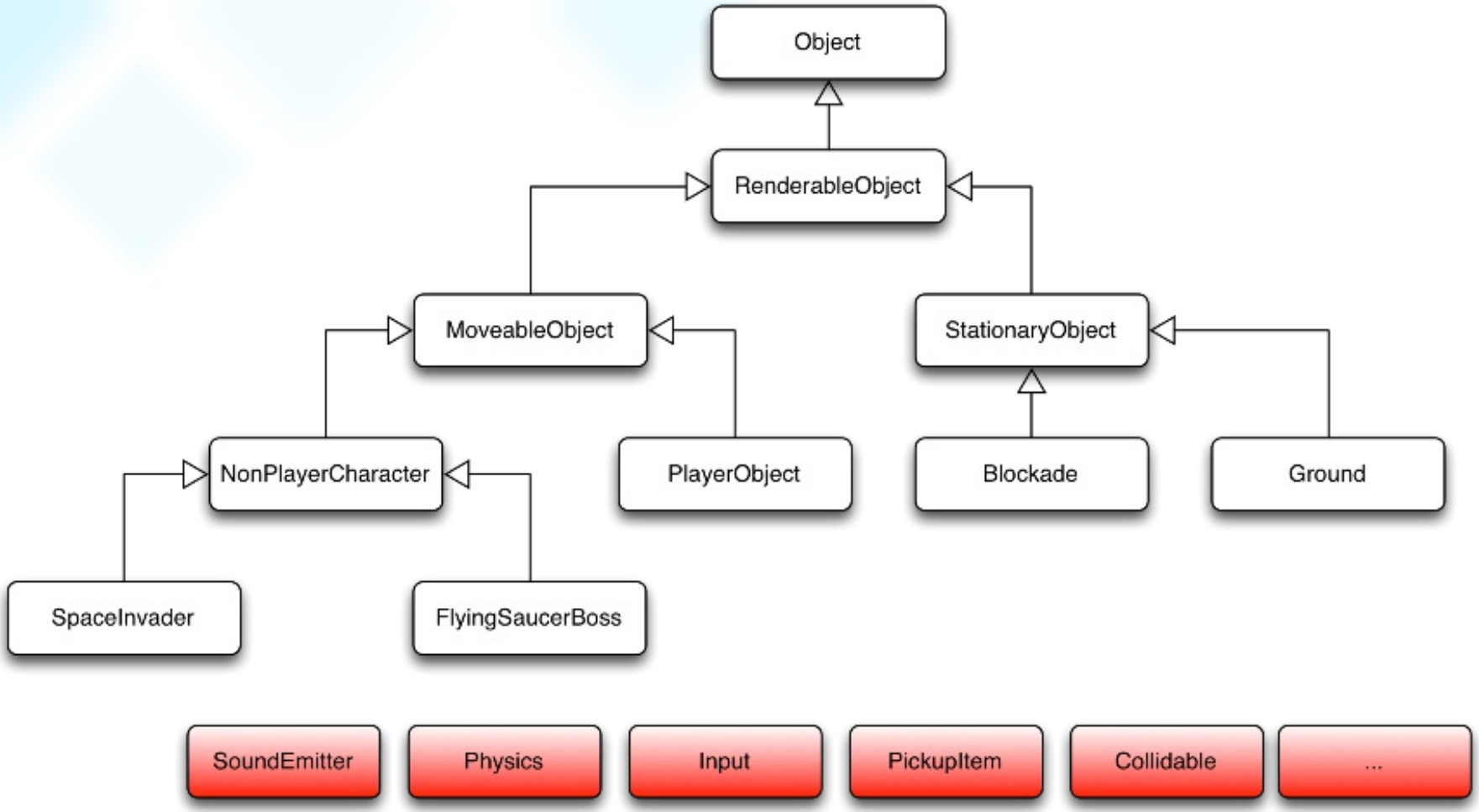
Composition vs Inheritance cont'd

- What about mix-in multiple inheritance?



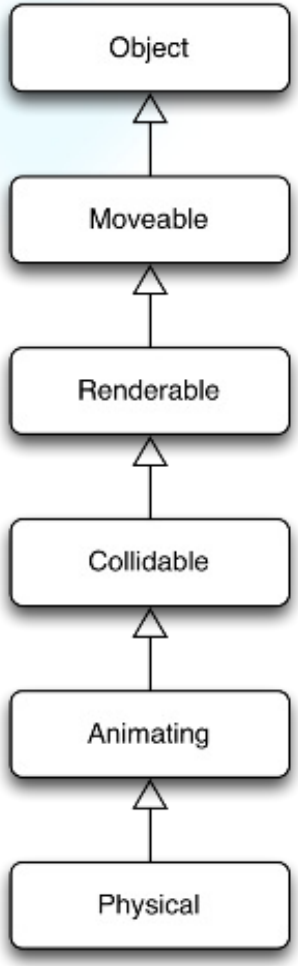
Composition vs Inheritance cont'd

- Does it scale?



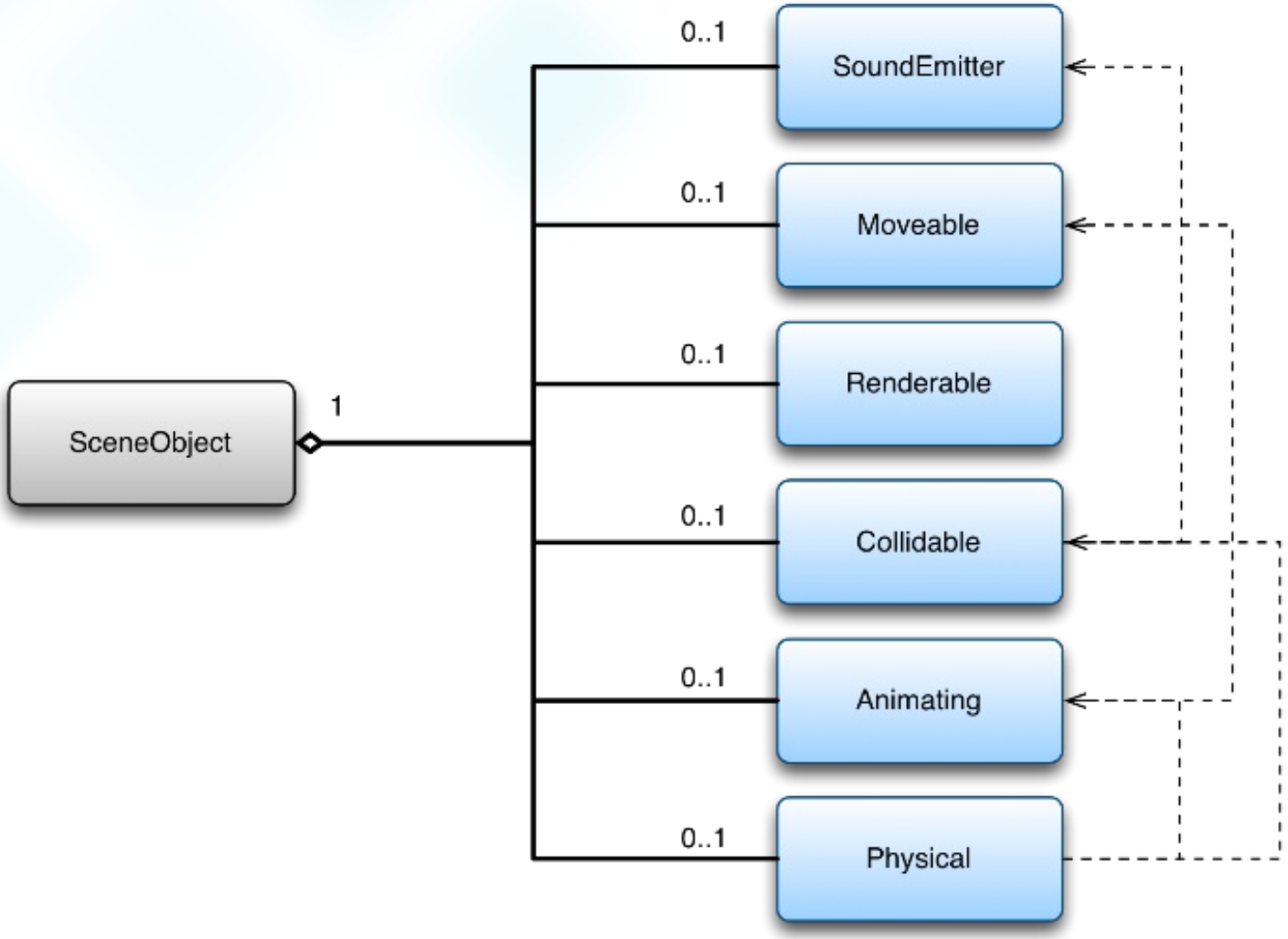
Composition vs Inheritance cont'd

- Is inheritance flexible enough?



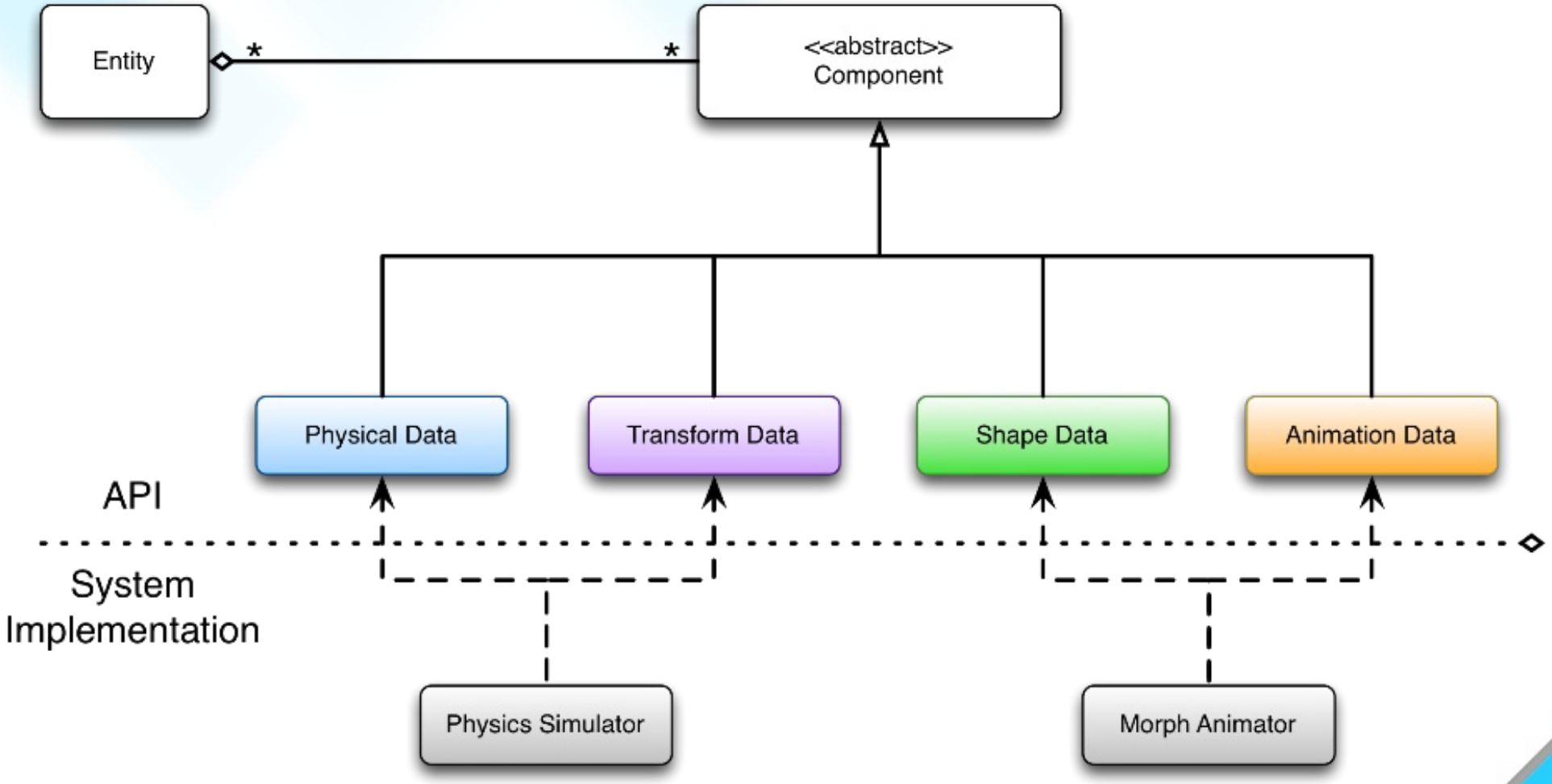
Composition then?

- Is traditional fixed composition the panacea?



Entity Component System

- The Entity/Component data split gives flexibility to manage the API
- The System separation moves the behavior away from data avoiding dependencies between Components



Entity Component System Wrap-up

- Inheritance:
  - Relationships baked in at design time
  - Complex inheritance hierarchies: deep, wide, multiple inheritance
  - Features tend to migrate to base class
- Fixed Composition
  - Relationships still baked in at design time
  - Fixed maximum feature scope
  - Lots of functional domain details in the scene object
  - If functional domain objects contain both data and behavior they will have lots of inter-dependencies
- Entity Component System
  - Allows changes at runtime
  - Avoids inheritance limitations
  - Has additional costs:
    - More QObjects
    - Different to most OOP developer's experience
  - We don't have to bake in assumptions to Qt 3D that we can't later change when adding features.

- Feature Set
- Entity Component System? What's that?
- **Hello Donut**
- Qt 3D ECS Explained

Hello Donut (C++)

- Good practice having root `Qt3DCore::QEntity` to represent the scene
- One `Qt3DCore::QEntity` per "object" in the scene
- Objects given behavior by attaching `Qt3DCore::QComponent` subclasses
- For an `Qt3DCore::QEntity` to be drawn it needs:
  - A mesh geometry describing its shape
  - A material describing its surface appearance

Demo qt3d/ex-hellodonut

Hello Donut (QML)

- Good practice having root `Entity` to represent the scene
- One `Entity` per "object" in the scene
- Objects given behavior by attaching component subclasses
- For an `Entity` to be drawn it needs:
  - A mesh geometry describing its shape
  - A material describing its surface appearance

Demo qt3d/ex-hellodonut-qml

C++ API vs QML API

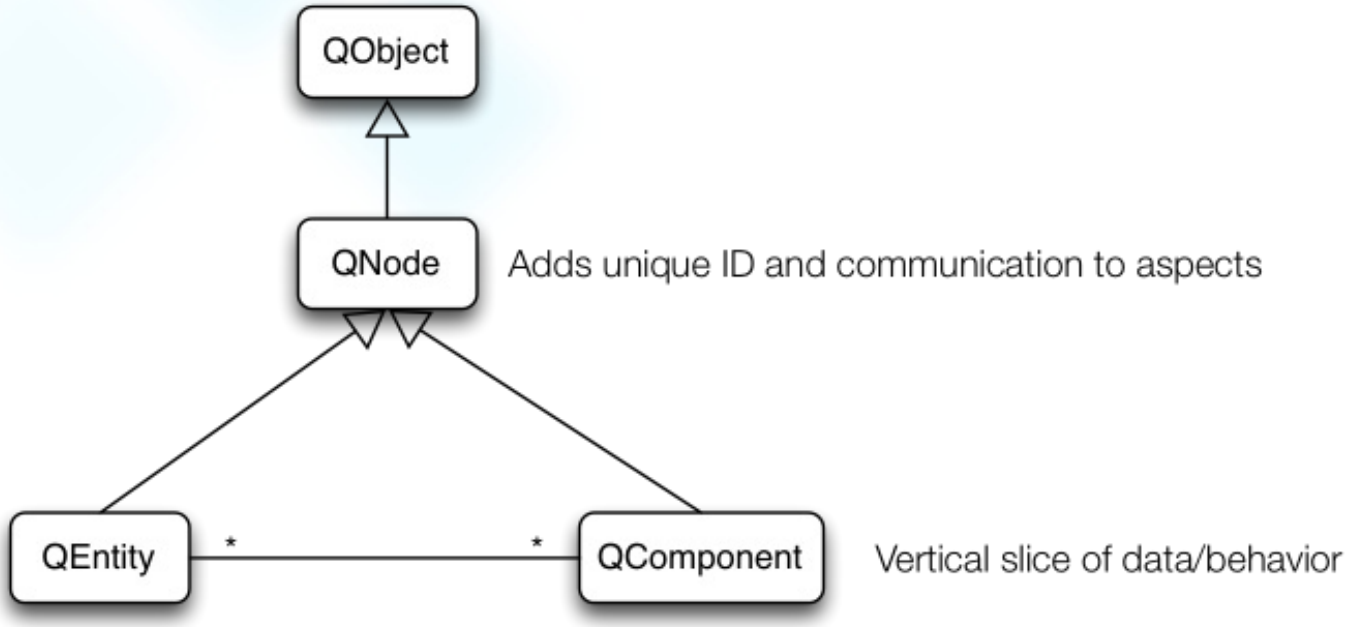
- QML API is a mirror of the C++ API
- C++ class names like the rest of Qt
- QML element names just don't have the Q in front
  - `Qt3DCore::QNode` vs `Node`
  - `Qt3DCore::QEntity` vs `Entity`
  - ...

- Feature Set
- Entity Component System? What's that?
- Hello Donut
- **Qt 3D ECS Explained**

Everything is a QNode

- `Qt3DCore::QNode` is the base type for everything
  - It inherits from `QObject` and all its features
  - Internally implements the frontend/backend communication
- `Qt3DCore::QEntity`
  - It inherits from `Qt3DCore::QNode`
  - It just aggregates `Qt3DCore::QComponents`
- `Qt3DCore::QComponent`
  - It inherits from `Qt3DCore::QNode`
  - Actual data is provided by its subclasses
    - `Qt3DCore::QTransform`
    - `Qt3DRender::QMesh`
    - `Qt3DRender::QMaterial`
    - ...

Everything is a QNode cont'd



Simulated object. Aggregates components

You Still Need a System

- The simulation is executed by `Qt3DCore::QAspectEngine`
- `Qt3DCore::QAbstractAspect` subclass instances are registered on the engine
  - Behavior comes from the aspects processing component data
  - Aspects control the functional domains manipulated by your simulation
- Qt 3D provides
  - `Qt3DRender::QRenderAspect`
  - `Qt3DInput::QInputAspect`
  - `Qt3DLogic::QLogicAspect`
- Note that aspects have no API of their own
  - It is all provided by `Qt3DCore::QComponent` subclasses

Engine and Application Tasks

- Engine Tasks
  - Create window and graphics context
  - Create and manage GPU buffers and textures
  - Create and manage shader programs
  - Create graphics pipeline and manage state
  - Kickoff the drawing and compute jobs on GPU!
  - Update AI, physics, application state, make coffee...
- Application Tasks
  - Provide per-vertex data
  - Provide texture image data
  - Provide shader program source code
  - Describe graphics state
  - Describe high-level rendering algorithm (see Frame Graph)

- The Story of Qt
- Overview of Qt 3D
- **Drawing with Qt 3D**
  - Introduction
  - Geometries
  - Transformations and Coordinate Systems
  - Materials
  - Texturing
  - Lights
- The Qt 3D Frame Graph
- The Future of Qt 3D

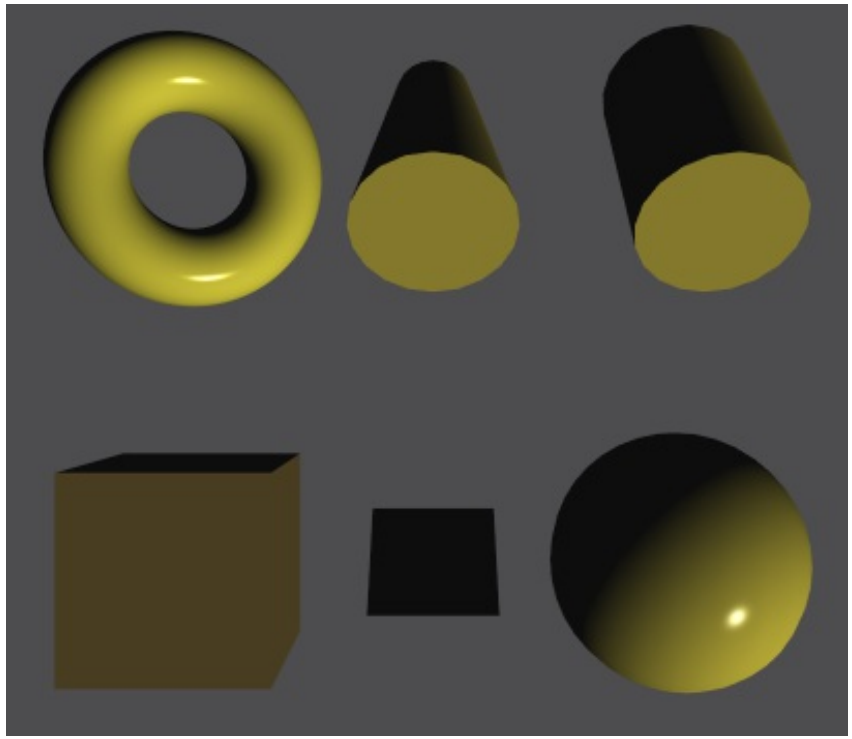
- **Introduction**
- Geometries
- Transformations and Coordinate Systems
- Materials
- Texturing
- Lights

The Scene Graph

- The scene graph provides the spatial representation of the simulation
- `Qt3DCore::QEntity`: what takes part in the simulation
  - `Qt3DRender::GeometryRenderer`: what's its shape
  - `Qt3DCore::QTransform`: where it is, what scale it is, what orientation it has
  - `Qt3DRender::Material`: how does it look like
- Hierarchical transforms are controlled by the parent/child relationship
  - Similar to `QWidget`, `QQuickItem`, etc.
- If the scene is rendered, we need a point of view on it
  - This is provided by `Qt3DRender::QCamera`

- Introduction
- **Geometries**
- Transformations and Coordinate Systems
- Materials
- Texturing
- Lights

- `Qt3DRender::QRenderAspect` draws `Qt3DCore::QEntitys` with a shape
- `Qt3DRender::QGeometryRenderer's` geometry property specifies the shape
- Qt 3D provides convenience subclasses of `Qt3DRender::QGeometryRenderer`:
  - `Qt3DExtras::QSphereMesh`
  - `Qt3DExtras::QCuboidMesh`
  - `Qt3DExtras::QPlaneMesh`
  - `Qt3DExtras::QTorusMesh`
  - `Qt3DExtras::QConeMesh`
  - `Qt3DExtras::QCylinderMesh`



Qt Demo examples/qt3d/basicshapes-cpp

More complex geometries

- `Qt3D.Extras` comes with simple common geometries
- `SphereMesh`, `PlaneMesh`, `TorusMesh`...
- But what about more complex shapes?
- What about those nice assets created by designers?

The Mesh Element

- `Qt3D.Render` provides a generic `Mesh` element
- It can load any supported mesh format
- Point it to a file using its source property
- If the file contains more than one mesh, you can select one using the meshName property

Demo qt3d/ex-mesh

Programmatically Generated Shapes

- `Mesh` assumes the data exists in a file
- What if I get my data from a database?
- What if my shape is the result of some algorithm executed at runtime?

Programmatically Generated Shapes

- We need a way to store mesh data in memory
- This is done using the `Geometry` element
- `Geometry` specifies geometry by means of:
  - `Buffer`s that contain the actual data
  - `Attribute`s that define the data format
- There are multiple strategies for managing data in `Buffer`s and `Attribute`s
- It is then rendered via a `GeometryRenderer`
  - `Mesh`, `TorusMesh` and so on are `GeometryRenderer`s using their own geometries
  - The primitiveType controls how the vertices are connected

- Move all the `GeometryRenderer` code on the C++ side
- Expose only a `ScribbleMesh` element

Demo qt3d/sol-geometry-step4

Summary

- Builtin simple 3D shapes from Qt3DExtras
- `Mesh` allows loading and rendering geometry from a file
- `GeometryRenderer`:
  - Is a component for drawing `Geometry`
- `Geometry` specifies geometry by means of `Buffer`s and `Attribute`

- Introduction
- Geometries
- **Transformations and Coordinate Systems**
- Materials
- Texturing
- Lights

Important Coordinate Systems

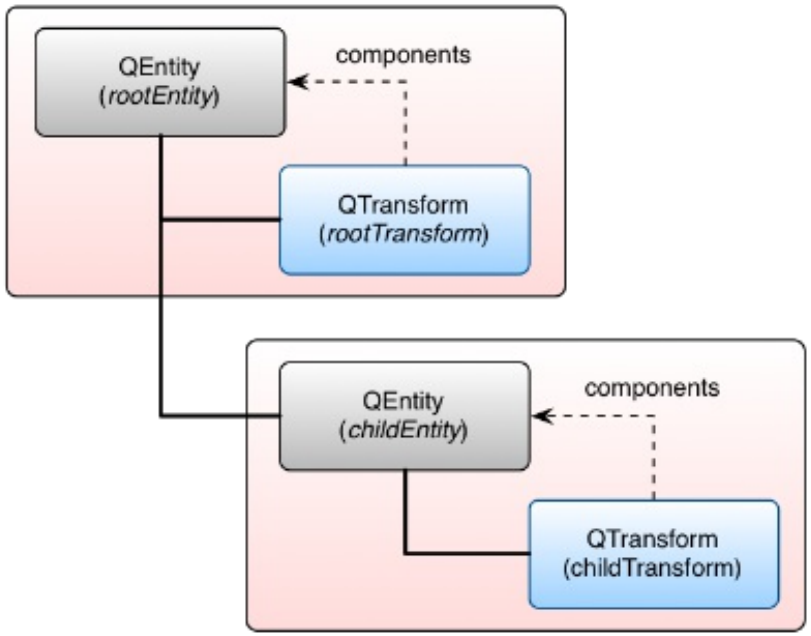
- **Model Space Local**
  - Coordinate system of individual object
- **World Space**
  - Application specific
- **Camera or Eye Space**
  - Eye position is origin, -z axis pointing away from us
- **Projection or Clip Space**
  - Variable sized cube, centered at origin
- **Normalised Device Coords**
  - Cube of edge 2, centered at origin [(-1, 1),(-1, 1),(-1, 1)]
- **Window Coords**
  - Pixel position in window

Qt3DCore::QTransform

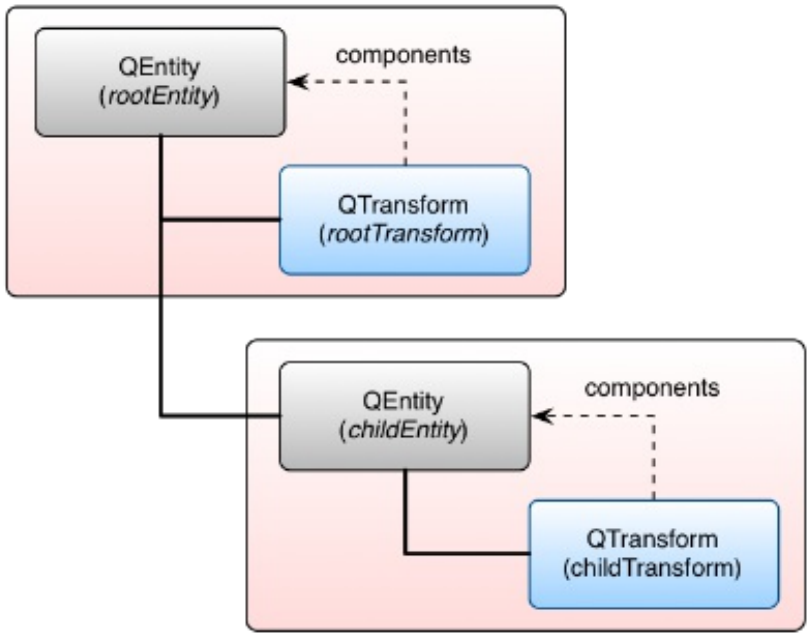
- Objects in the scene generally needs to be transformed
- Inherits from `Qt3DCore::QComponent`
- Represents an affine transformation
- Three ways of using it:
  - Through properties: scale3D, rotation, translation
  - Through helper functions: `rotateAround()`
  - Through the matrix property

Transforms cont'd

```
1  #include <Qt3DCore/QEntity>
2  #include <Qt3DCore/QTransform>
3  ...
4
5  auto rootEntity = QSharedPointer<Qt3DCore::QEntity>::create();
6  auto rootTransform = new Qt3DCore::QTransform(rootEntity);
7  rootTransform->setScale3D(QVector3D(1.0f, 2.0f, 1.5f));
8  rootTransform->setTranslation(QVector3D(0.0f, 0.0f, -1.0f));
9  rootEntity->addComponent(rootTransform);
10
11 auto childEntity = new Qt3DCore::QEntity(rootEntity.data());
12 auto childTransform = new Qt3DCore::QTransform;
13 childTransform->setTranslation(QVector3D(0.0f, 1.0f, 0.0f));
14 childEntity->addComponent(childTransform); // Takes ownership
```



```
1  import Qt3D.Core 2.0
2
3  Entity {
4      components: [
5          Transform {
6              scale3D: Qt.vector3d(1, 2, 1.5)
7              translation: Qt.vector3d(0, 0, -1)
8          }
9      ]
10
11     Entity {
12         components: [
13             Transform { translation: Qt.vector3d(0, 1, 0) }
14         ]
15     }
16 }
```



Controlling Matrices in Qt 3D

- The model to world matrices are controlled by the `Transform` components on `Entity`s
- How do we control the view and projection matrices?
- `Camera` to the rescue:
  - View matrix controlled by: position, upVector and viewCenter properties
  - Projection matrix controlled by the attached `CameraLens` component

Projection Matrix

- Let the `CameraLens` component worry about the maths
- Type of projection determined by the projectionType property
- Perpective projection controlled by:
  - fieldOfView - this is the *vertical* field of view
  - aspectRatio
  - nearPlane, farPlane
- Orthographic projection controlled by:
  - left, right
  - bottom, top
  - nearPlane, farPlane

How does this look in practice?

```
 1 #version 150
 2
 3 in vec3 vertexPosition;
 4 in vec3 vertexNormal;
 5
 6 out vec3 worldPosition;
 7 out vec3 worldNormal;
 8
 9 uniform mat4 modelMatrix;
10 uniform mat3 modelNormalMatrix;
11 uniform mat4 mvp;
12
13 void main()
14 {
15     worldPosition = vec3(modelMatrix * vec4(vertexPosition, 1.0));
16     worldNormal = normalize(modelNormalMatrix * vertexNormal);
17     gl_Position = mvp * vec4(vertexPosition, 1.0);
18 }
```

Transformations Lab

- Create a model solar system
- Use provided scene of `OrbitingBody`s
- Each planet has some properties already configured
- Complete the `OrbitingBody` to apply transformations for:
  - Planet size
  - Orbital radius
  - Orbital phase (position in orbit)
  - Orbital inclination
- Add moons to some planets
- Make the camera zoom in and track a planet when you click on it

Demo qt3d/sol-solar-system

Summary

- There are several coordinate systems to be aware of
- You can transform between coordinate systems using matrices
- Qt 3D automatically provides common transformation matrices as GLSL shader uniforms
- The view matrix is controlled by the `Camera` entity
- The projection matrix is controlled by the `CameraLens` component

- Introduction
- Geometries
- Transformations and Coordinate Systems
- **Materials**
- Texturing
- Lights

- If a `Qt3DCore::QEntity` only has a shape it won't be visible
- The `Qt3DRender::QMaterial` component provides a surface appearance
- Qt 3D provides convenience subclasses of `Qt3DRender::QMaterial`:
  - `Qt3DExtras::QPhongMaterial`
  - `Qt3DExtras::QPhongAlphaMaterial`
  - `Qt3DExtras::QDiffuseMapMaterial`
  - `Qt3DExtras::QDiffuseSpecularMapMaterial`
  - `Qt3DExtras::QGoochMaterial`
  - ...

Demo qt3d/ex-hellodonut-qml-uber

Qt Demo examples/qt3d/materials-cpp

Qt Demo examples/qt3d/materials

Custom Material example

```
1  import Qt3D.Render 2.0
2  ...
3
4  Material {
5      effect: Effect {
6          techniques: [
7              Technique {
8                  filterKeys: FilterKey { name: "renderingStyle"; value: "forward" }
9
10                 graphicsApiFilter {
11                     api: GraphicsApiFilter.OpenGL
12                     majorVersion: 3
13                     minorVersion: 2
14                     profile: GraphicsApiFilter.CoreProfile
15                 }
16
17                 renderPasses: RenderPass {
18                     shaderProgram: ShaderProgram {
19                         vertexShaderCode: loadSource("qrc:/customshader.vert")
20                         fragmentShaderCode: loadSource("qrc:/customshader.frag")
21                     }
22                 }
23             }
24         ]
25     }
26 }
```

Demo qt3d/ex-glsl

Varying Degrees of Change

- Shaders can have constant variables:

```
const float pi = 3.14159;
const vec2 resolution = vec2( 1024.0, 768.0 );
```

- Geometry can provide per-vertex attributes:
  - Position
  - Normal vectors
  - Texture coordinates
  - Colors
  - Temperature
  - Density
  - Fluffiness...

What about in between these extremes?

Shader Uniform Variables

Shader Uniform Variables:

- Middle ground between per-vertex and constant
- Constant for a particular `GeometryRenderer`
- Declared with `uniform` keyword in the shader code
- Common to the entire shader program (must be consistent)
- Use as any other constant in GLSL

```
uniform vec4 lightPosition;
```

- Provided by `Parameter` elements
- Set on the parameters property of:
  - `RenderPass`
  - `Technique`
  - `Effect`
  - `Material`
- The effective value set for the uniform is cascaded

Shader Uniform Variables cont'd

```
1  import Qt3D.Render 2.0
2  ...
3
4  Material {
5      parameters: Parameter { name: "colorTint"; value: "yellow" }
6
7      effect: Effect {
8          parameters: Parameter { name: "colorTint"; value: "green" }
9
10         techniques: [
11             Technique {
12                 parameters: Parameter { name: "colorTint"; value: "blue" }
13
14                 filterKeys: FilterKey { name: "renderingStyle"; value: "forward" }
15
16                 graphicsApiFilter { ... }
17
18                 renderPasses: RenderPass {
19                     parameters: Parameter { name: "colorTint"; value: "red" }
20
21                     shaderProgram: ShaderProgram {
22                         vertexShaderCode: loadSource("qrc:/tintingshader.vert")
23                         fragmentShaderCode: loadSource("qrc:/tintingshader.frag")
24                     }
25                 }
26             }
27         ]
28     }
29 }
```

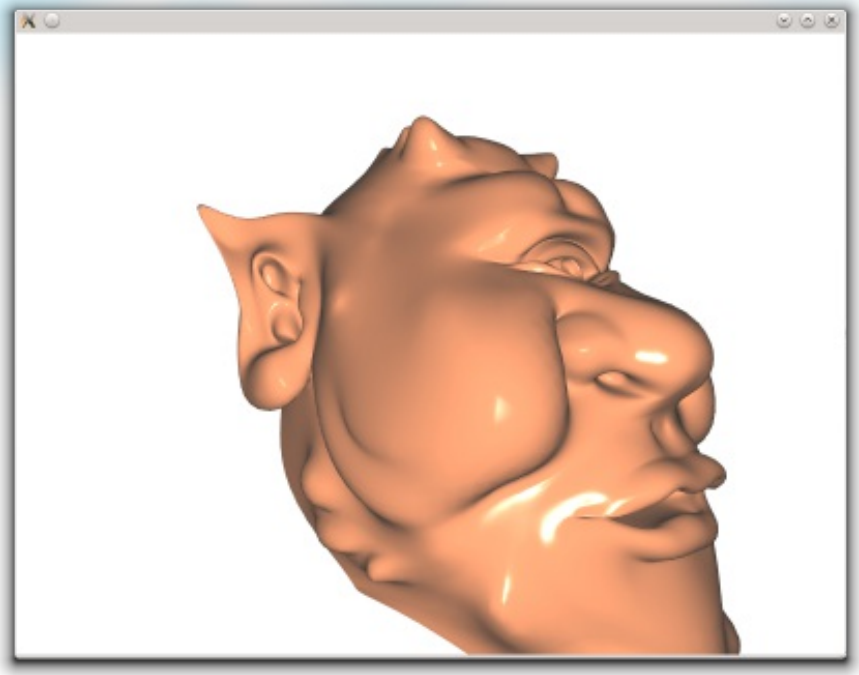
Example: Custom Material with Parameters

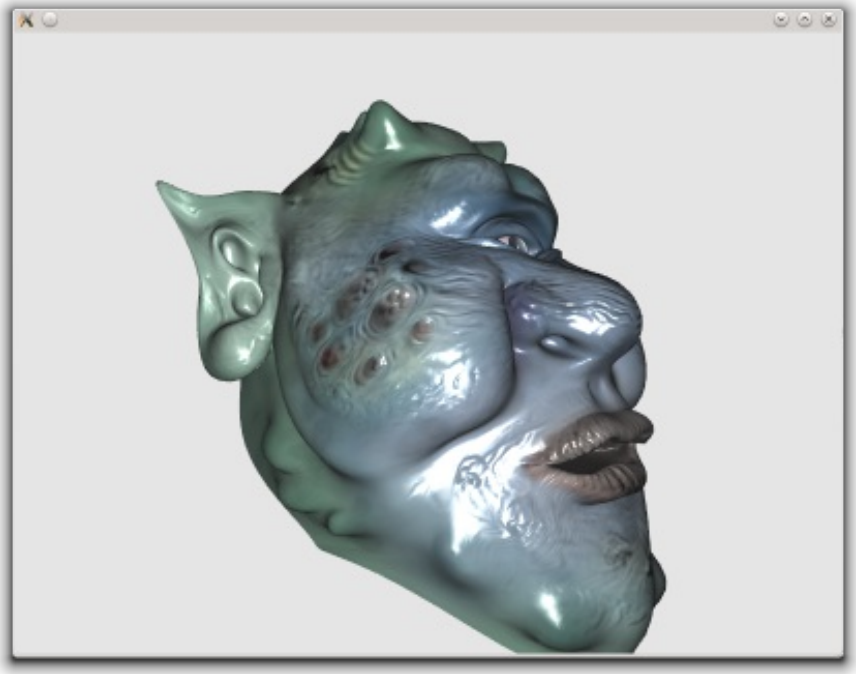
```
1  import Qt3D.Render 2.0
2  import QtQuick 2.0
3  ...
4
5  Material {
6      parameters: Parameter {
7          name: "darkness";
8          value: 0
9
10         SequentialAnimation on value {
11             loops: Animation.Infinite
12
13             NumberAnimation {
14                 from: 0; to: 1
15                 duration: 1000
16             }
17
18             NumberAnimation {
19                 from: 1; to: 0
20                 duration: 1000
21             }
22         }
23     }
24
25     effect: Effect { ... }
26 }
```

Demo qt3d/ex-glsl-animated

- Introduction
- Geometries
- Transformations and Coordinate Systems
- Materials
- **Texturing**
- Lights

The devil is in the details!





What is Texturing?

- Encoded information (color, normal, height, ...)
- Made available to the graphics API and GPU
- Indexed by per-vertex attribute - *texture coordinates*
- Shaders perform texture lookups
- Use resulting data in calculations

Demo qt3d/ex-simpletexture

Accessing Textures

- Several texture types
  - 1D - Indexed color maps, gradients, complicated functions
  - 2D - Color, specular (gloss) maps, normal (bump) maps, height maps
  - 3D - Volumetric techniques, density functions, terrain generation
  - Cube Maps - Environment mapping, reflection, refraction
  - ...
- Many filtering options
  - Nearest
  - Linear
  - Mipmaps
- Hardware supports multiple *Texture Units*

Textures

- Some materials can vary data across the surface
- This is handled by using a texture
  - A texture consists of one or more images
- Subclasses of `Qt3DRender::QAbstractTexture` provide different types of texture:
  - 1D - useful for lookup functions such as gradients
  - 2D - most common type, used for general image data
  - 3D - useful for volumetric data
  - Arrays of 2D - used when optimizing (see later)
  - ...
- `Qt3DRender::QTextureLoader` can load all types of texture from supported file types

Accessing Textures cont'd

```
1  import Qt3D.Render 2.0
2  ...
3
4  Material {
5      parameters: [
6          ...,
7          Parameter {
8              name: "baseTexture"
9              value: Texture2D {
10                 minificationFilter: Texture.Linear
11                 magnificationFilter: Texture.Linear
12                 wrapMode {
13                     x: WrapMode.Repeat
14                     y: WrapMode.Repeat
15                 }
16                 generateMipMaps: true
17                 maximumAnisotropy: 16.0
18
19                 TextureImage {
20                     source: "bricks.png"
21                 }
22             }
23         },
24         ...
25     ]
26     ...
27 }
```

Textures in Shaders

- Textures accessed in shaders via *sampler* variables
- Opaque type used to access texture unit hardware
- Declared as uniform variable

```
uniform sampler2D diffuseTexture;
```

- Uniform is associated with texture unit thanks to `Parameter`

```
name: "baseTexture"
```

- Texture coordinates passed in as per-vertex attribute (or calculated)

```
in vec2 texCoords;
```

- Lookup value with `texture( sampler, texCoords )` function

```
vec4 color = texture( baseTexture, texCoords );
```

- Use value

```
fragColor = color;
```

Textures in Shaders cont'd.

- Make lots of data available to shaders
- No "right" way of using textures
- Simple through to complex
  - Just use texture value as fragment color
  - Model atmospheric scattering and extinction
- Lots of available techniques
- Embellish and experiment

Use your imagination!

Texturing the Earth

- Render the Earth using multiple textures
- Using multiple textures at the same time

Demo qt3d/sol-earth

Summary

- Qt 3D uses GLSL shader programs
- Shaders execute on the GPU and can process large amounts of data
- `Material`, `Effect`, `Technique`, `RenderPass`, `ShaderProgram` allow for custom materials
- GLSL uniforms set via `Parameter`s
  - Parameters are cascaded to allow reusing `Material`, `Effect`, `Technique`, `RenderPass` elements
- Qt 3D lighting system is an application of uniform variables and `Parameter`s

- Introduction
- Geometries
- Transformations and Coordinate Systems
- Materials
- Texturing
- **Lights**

- Even with shapes and materials we would see nothing
- We need some lights
  - ... luckily Qt 3D sets a default one for us if none is provided
- In general we want some control of the scene lighting
- Light components are provided by `Qt3DRender::QAbstractLight` and its subclasses
- Lights don't appear in the scene, we only see their effects on other entities
  - `DirectionalLight`
  - `PointLight`
  - `SpotLight`

Demo qt3d/ex-lights-qml

Summary

- `Entity`s containing `Transform`s provide a scene graph
- `Qt3D.Extras` module provides some common building blocks:
  - Basic geometric primitives
  - Phong-like materials
- `Qt3D.Render` module provides some common light types

- The Story of Qt
- Overview of Qt 3D
- Drawing with Qt 3D
- **The Qt 3D Frame Graph**
  - Viewports and Layers
  - Image-Based Techniques
- The Future of Qt 3D

What is the Frame Graph For?

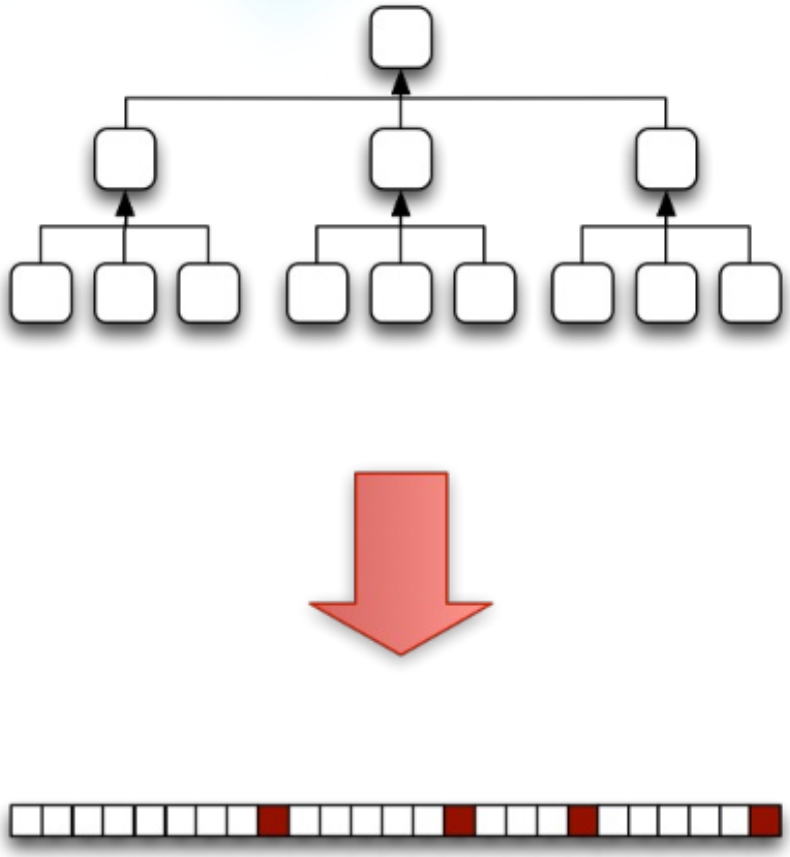
- With what we have seen so far, we can:
  - Draw geometry loaded from disk or generated dynamically
  - Use custom materials with shaders to change surface appearance
  - Make use of textures to increase surface details
- What about shadows?
- What about transparency?
- What about post processing effects?
- All these and others require control over *how* we render the Scene Graph
- The Frame Graph describes the rendering algorithm

Render Settings

- `RenderSettings` is a `Component` allowing to control the render aspect
- Only one instance is allowed
- It is generally set on the root `Entity` of the scene
- Its activeFramegraph property is the root of the Frame Graph
  - Can be a pre-made Frame Graph like `ForwardRenderer`
  - Or your own, generally starting with `RenderSurfaceSelector`
- It also allows to control picking via the pickingSettings grouped property
  - By default it uses bounding sphere volume picking (PickingSettings.BoundingVolumePicking)
  - Some scenes require the more expensive triangle picking (PickingSettings.TrianglePicking)

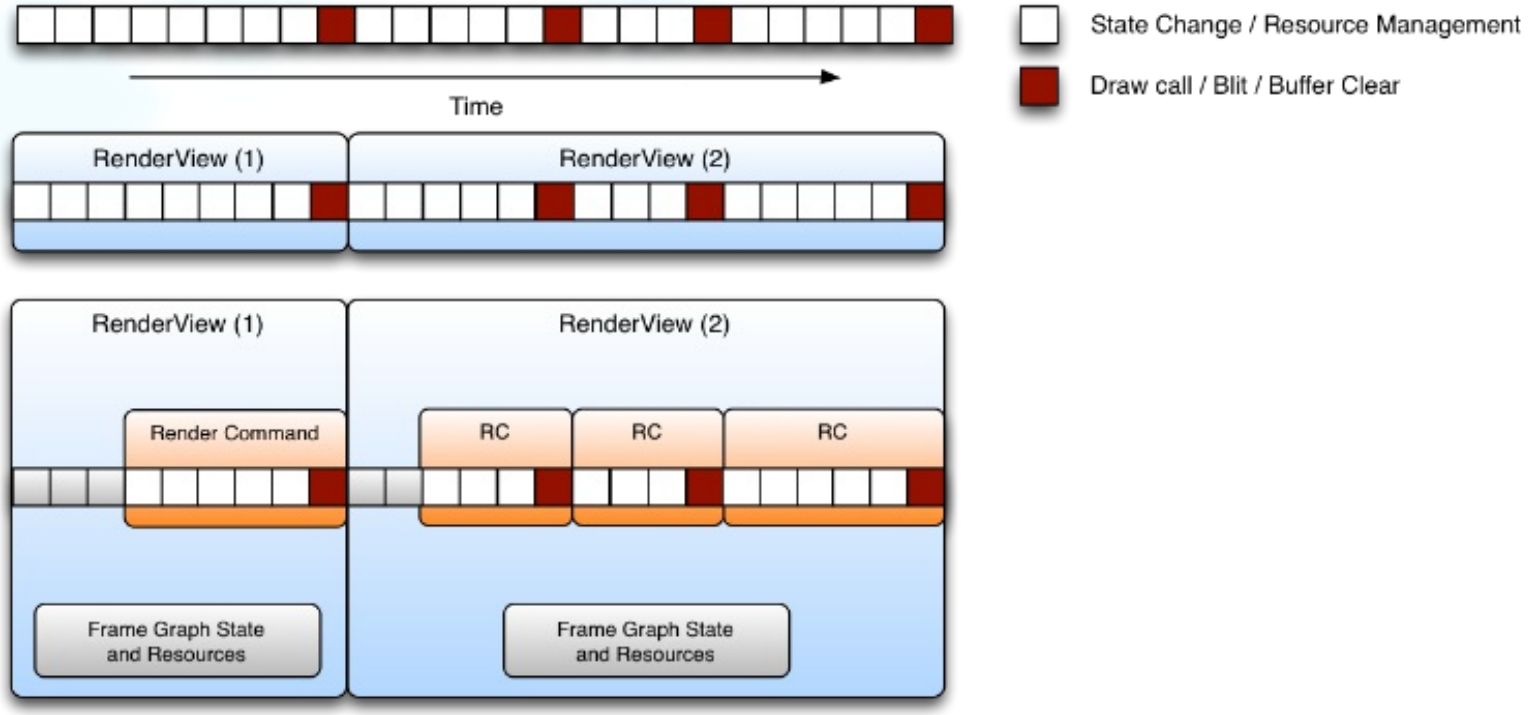
This module is focusing on writing Frame Graphs for different uses

- The nodes of the Frame Graph form a tree
- The entities of the Scene Graph form a tree
- The Frame Graph and Scene Graph are linearized into render commands



Render Views

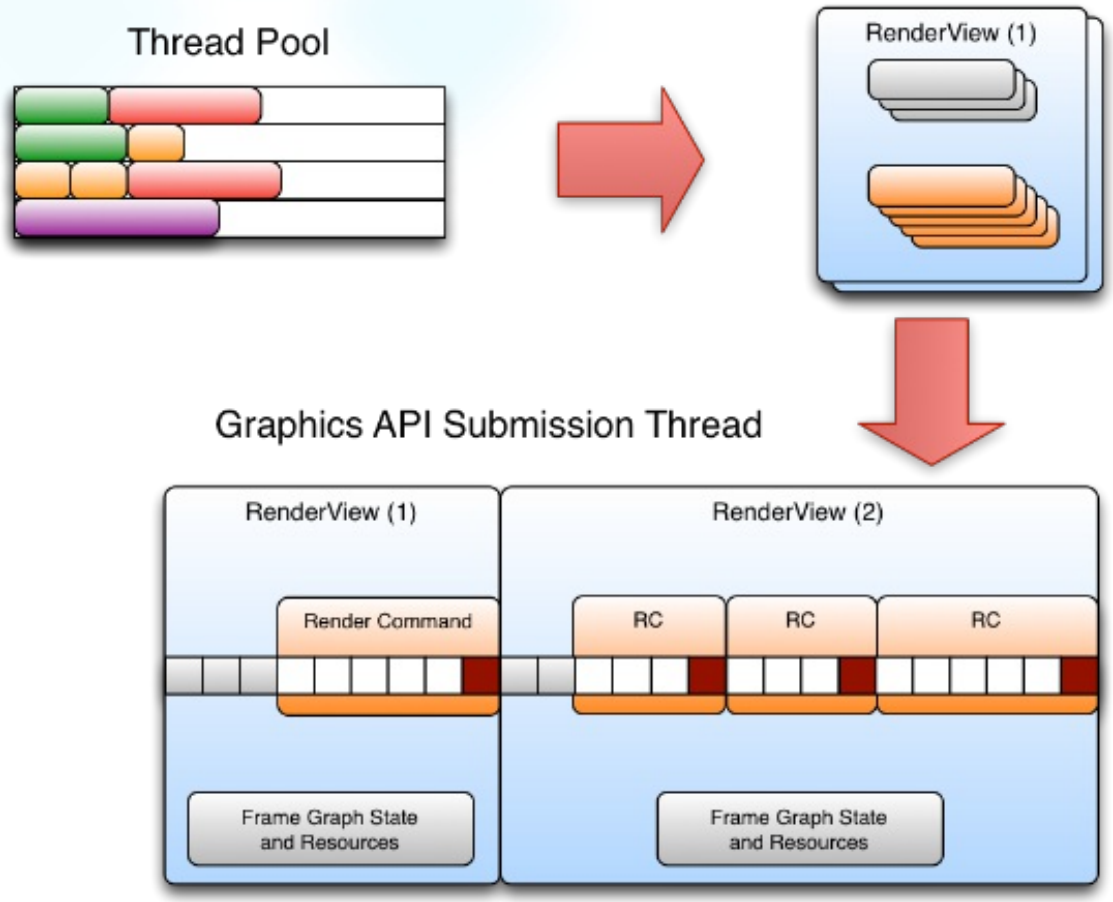
- The Frame Graph is traversed in a depth first manner to look for leaf nodes
- The Scene Graph is rendered for leaf nodes only
- Each leaf node generates a RenderView in the backend



#NOTES

Commands Submission

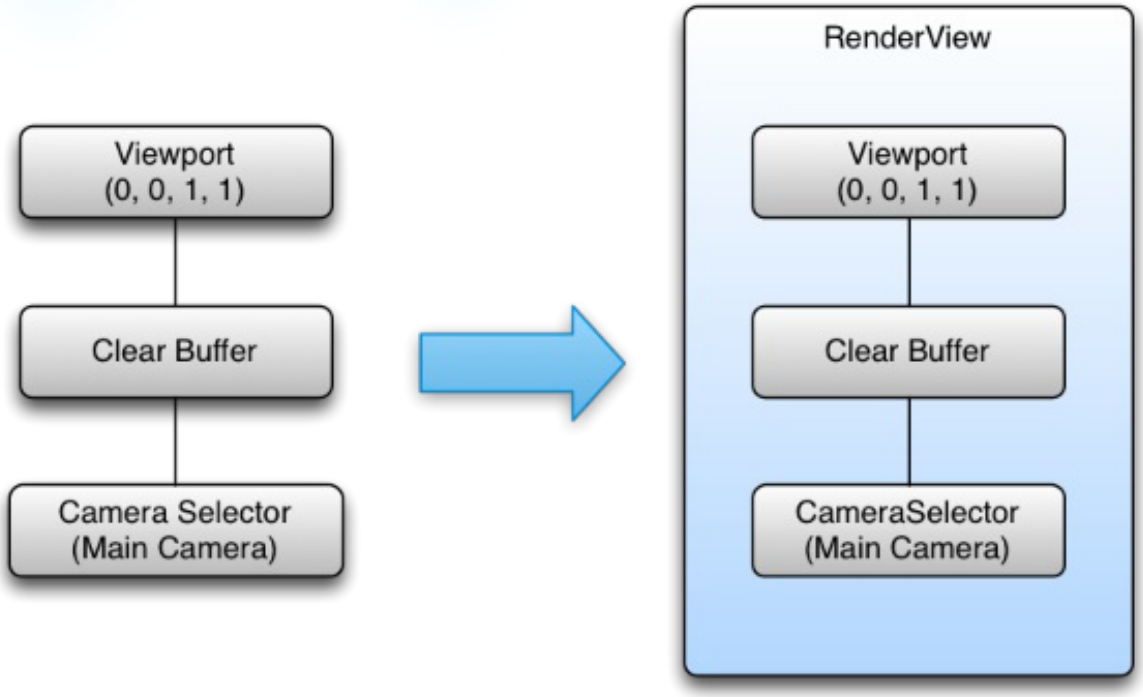
- The linearization of the Frame Graph is multi-threaded
- Submission of the commands is then done by a specific thread



#NOTES

The Simplest Frame Graph

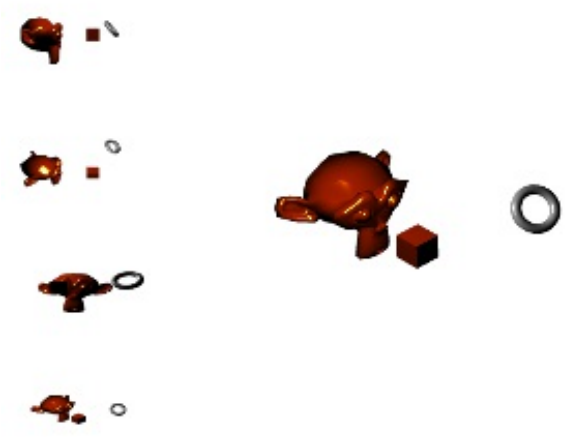
- It is important to structure your Frame Graph properly for performance reasons
- Might lead to deep and narrow trees
  - Simplest case being a one pass forward renderer



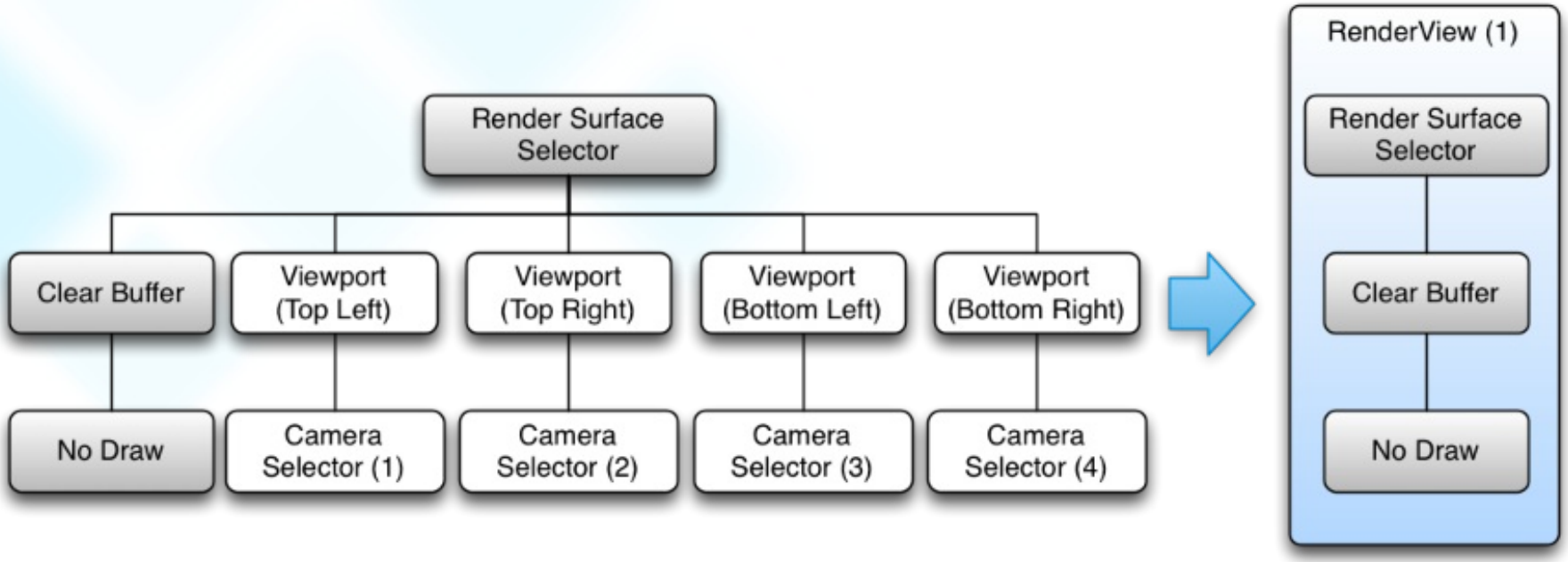
- **Viewports and Layers**
- Image-Based Techniques

Several Points of View on a Scene

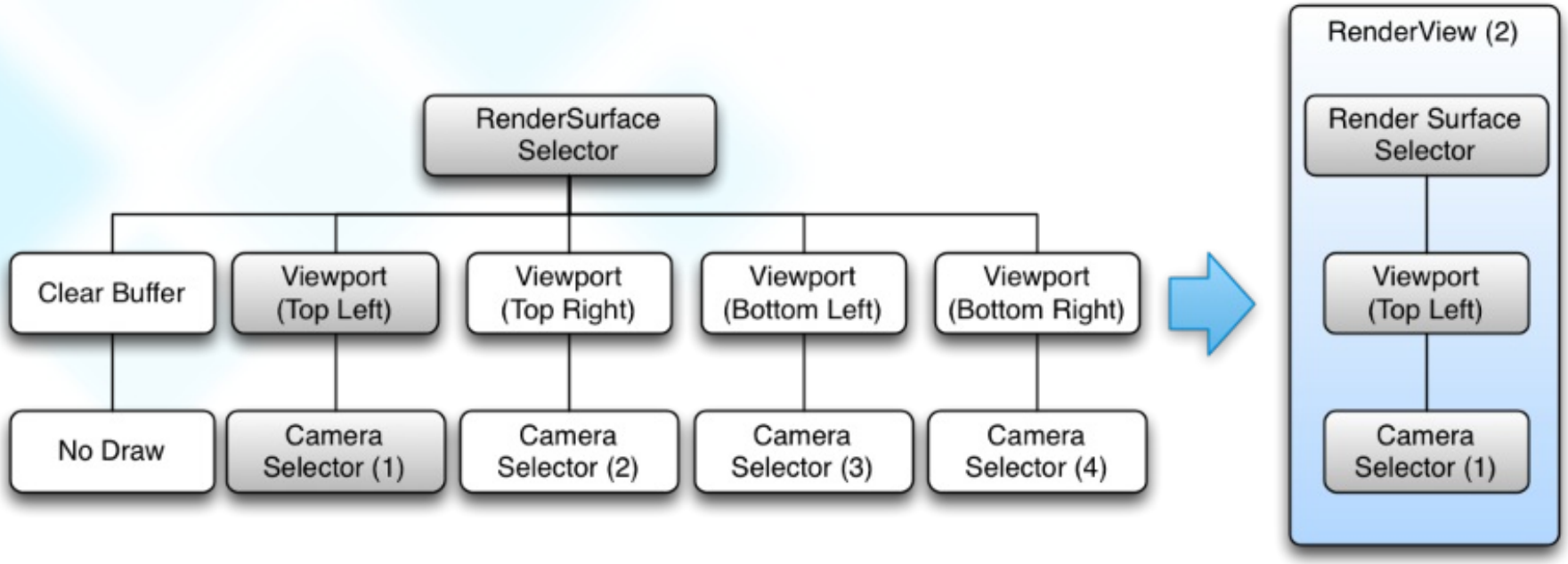
- `Camera` describes a point of view on a scene
- `Viewport` allows to split the render surface in several areas
  - They can be nested for further splitting
- `CameraSelector` allows to select a camera to render in a `Viewport`
- `ClearBuffers` decribes which buffers are cleared during the rendering
  - Generally necessary to get anything on screen
  - Also an easy way to control background color
- To avoid a branch to trigger a rendering give it a `NoDraw` element as leaf



Several Points of View on a Scene cont'd



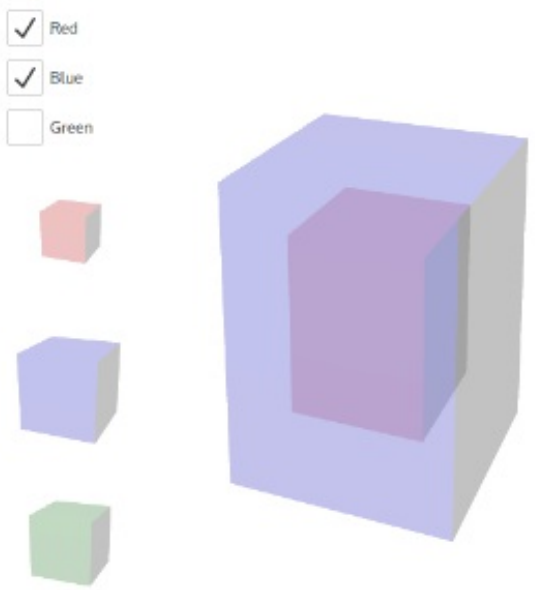
Several Points of View on a Scene cont'd



Demo qt3d/ex-viewports

Showing Different Scenes in Viewports

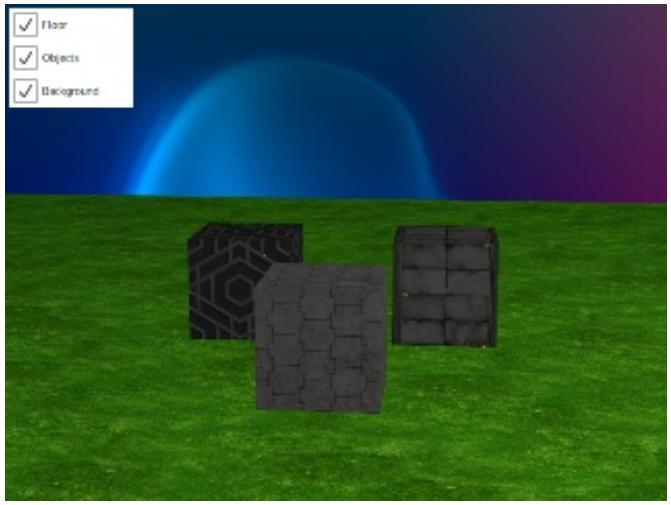
- Our `Viewport`s all display the same scene...
- But they can display different subsets of the scene using layers
- Attach each entity to a `Layer`
- Have each `Viewport` display a subset of the entities using `LayerFilter`



Demo qt3d/ex-viewports-and-layers

Composing Frames with Layers

- `Layer`s and `LayerFilter`s can also be used on their own
- They allow controlling how the final frame is composed
- Useful for:
  - Some post-processing effect
  - Tuning performances (e.g. in case of expensive fragment shader)
  - Showing optional debug displays
  - Controlling ordering (e.g. opaque entities before transparent entities for alpha blending)



Demo qt3d/ex-composing-layers

#NOTES

- Viewports and Layers
- **Image-Based Techniques**
  - Rendering to a Texture
  - Post-Processing Effects

- **Rendering to a Texture**
- Post-Processing Effects

Multiple passes

- So far, we have followed a standard single-pass pattern
  - Provide a `RenderSurfaceSelector`
  - Clear the buffers with `ClearBuffers`
  - Trigger the rendering with `CameraSelector`
    - Possibly combined with some other nodes seen previously

There are many techniques we can achieve by expanding on the basic pattern above - stereo rendering, more realistic lighting and shadowing, post-processing and more.

#NOTES

Rendering twice

- Simplest multi-pass renders (some of) the scene more than once
  - Add more than one `RenderPass` to some of your `Materials`
  - Provide a `RenderSurfaceSelector`
  - Clear the buffers with `ClearBuffers`
  - Trigger the rendering with `CameraSelector`
  - Provide a `RenderPassFilter` to activate a different set of shaders
  - Trigger the rendering with `CameraSelector` *again*
- An example would be to highlight certain objects in a scene - in a second pass, draw with a translucent texture, possibly with an adjusted scale.
- Will be covered in details later!

#NOTES

Rendering targets

- Hardware renders to blocks of memory with a pixel (surface) format
  - `QSurfaceFormat` specifies color depth, stereo rendering, depth, stencil, samples
  - When window is initialised, buffers matching the format are created (allocated)
- Render to custom buffer, instead of the back buffer
- `RenderTargetSelector` allows to render to *framebuffer objects* or *FBOs*

#NOTES

- Framebuffers are rendering targets
- Create them using `RenderTarget`
- It has one or more attachments of type `RenderTargetOutput`
- The memory of each output allocated in a texture

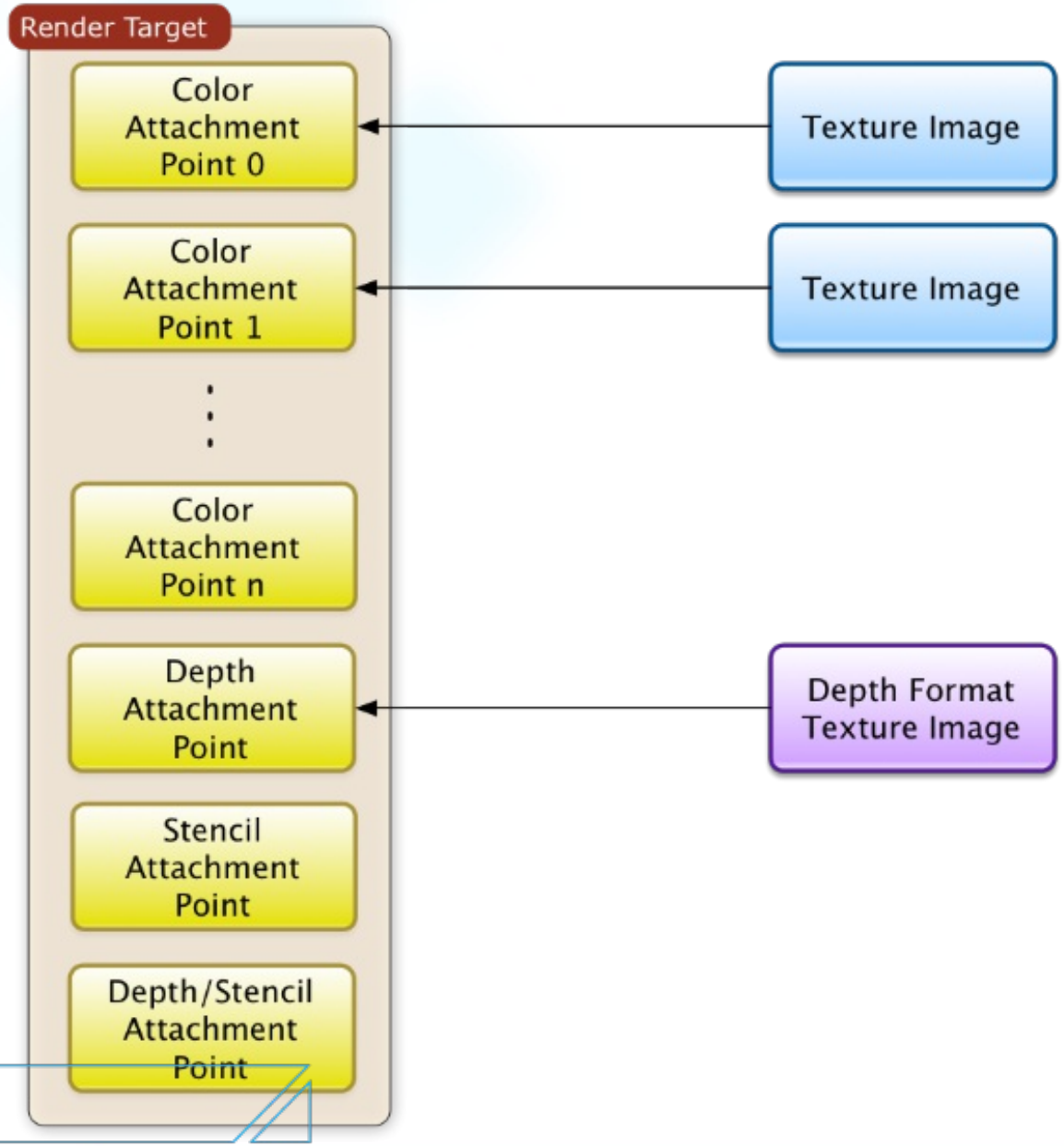
#NOTES

Attaching textures

- A texture is *not* a block of memory (an image), but a collection of them
  - *layer* of a 2D texture is mip-map level, or cube-map face
- Attach a texture image, to a framebuffer attachment point
  - `RenderTargetOutput` has an attachmentPoint property
  - Storage format must be compatible - no compressed images

#NOTES

Render Targets

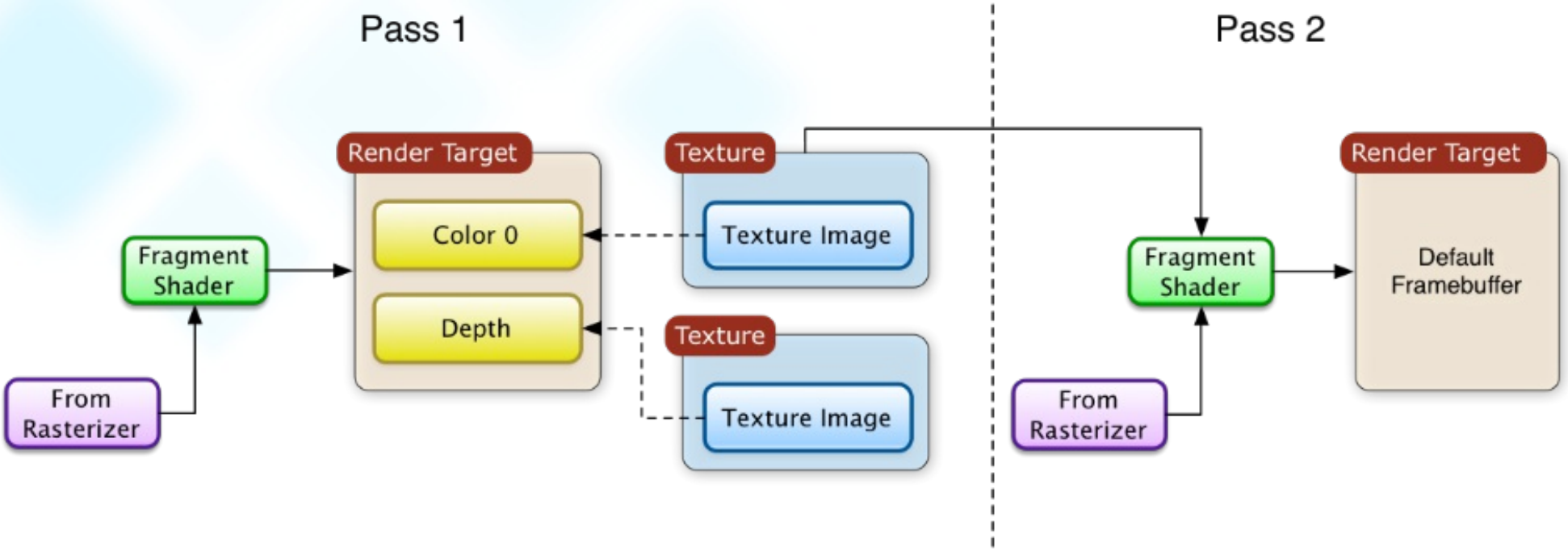


#NOTES

```
1  import Qt3D.Render 2.0
2  ...
3  RenderTarget {
4      attachments : [
5          RenderTargetOutput {
6              attachmentPoint: RenderTargetOutput.Color0
7              texture: Texture2D {
8                  width: 1024
9                  height: 1024
10                 format: Texture.RGBA8_UNorm
11             }
12         },
13         RenderTargetOutput {
14             attachmentPoint: RenderTargetOutput.Color1
15             texture: Texture2D {
16                 width: 1024
17                 height: 1024
18                 format: Texture.RGB16F
19             }
20         },
21         RenderTargetOutput {
22             attachmentPoint : RenderTargetOutput.Depth
23             texture : Texture2D {
24                 width: 1024
25                 height: 1024
26                 format: Texture.DepthFormat
27             }
28         }
29     ]
30 }
```

#NOTES

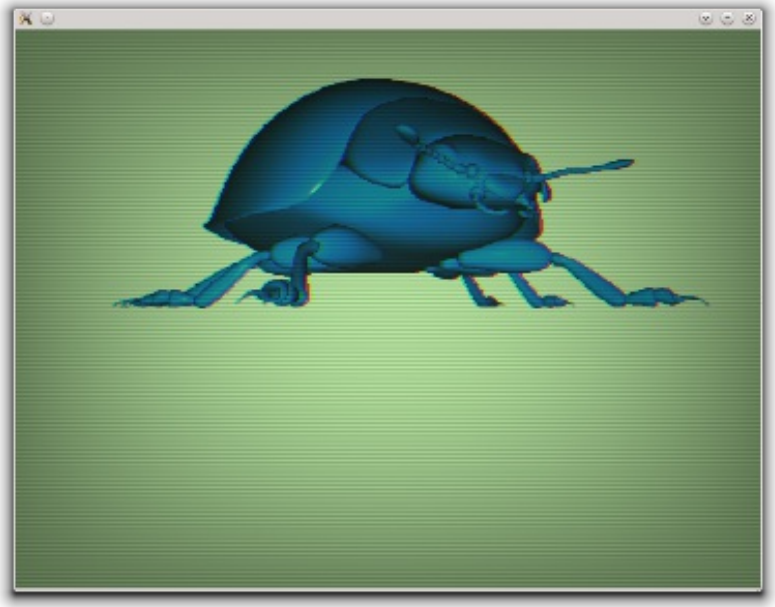
Two-pass Rendering



- Rendering to a Texture
- **Post-Processing Effects**

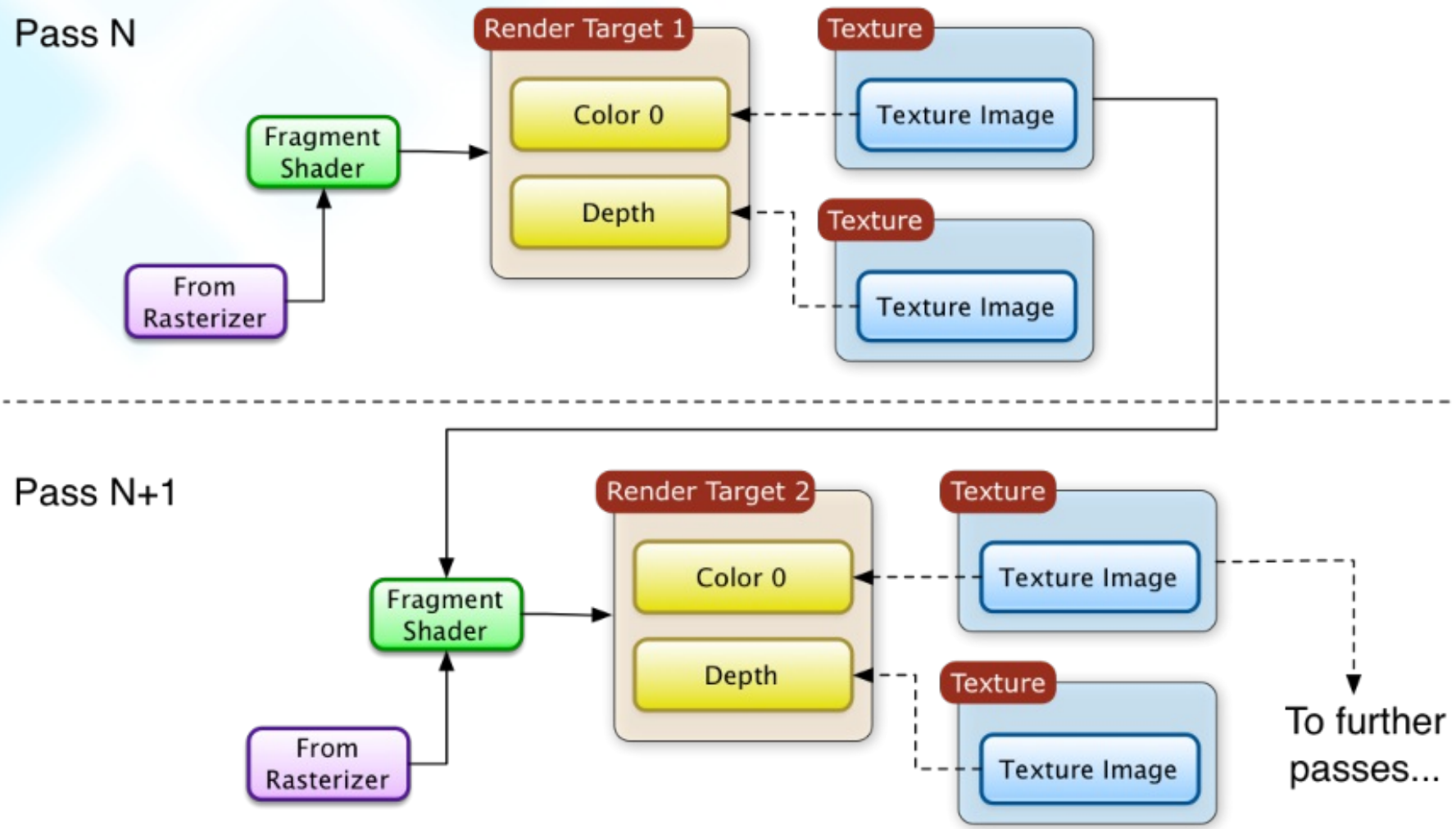
Post Processing Effects

- Uses 2 or more rendering passes
- Render to texture
- Render using texture
- Modifies original
  - Simulate poor zoom
  - Adjust levels/contrast
  - Color tint
  - Interference lines
  - Vignette
  - Flickering



Demo qt3d/ex-multiple-effects

Demo qt3d/sol-selection-overlay


Pass N
Render Target 1
Color 0
Depth
Fragment Shader
From Rasterizer
Texture
Texture Image
Texture
Texture Image
Pass N+1
Render Target 2
Color 0
Depth
Fragment Shader
From Rasterizer
Texture
Texture Image
Texture
Texture Image
To further passes...

#NOTES

- The Story of Qt
- Overview of Qt 3D
- Drawing with Qt 3D
- The Qt 3D Frame Graph
- **The Future of Qt 3D**
  - Beyond the Tip of the Iceberg
  - The Future of Qt 3D

- **Beyond the Tip of the Iceberg**
- The Future of Qt 3D

And more...

- Texture mipmaps
- Cube Maps
- Portability of your code accross several OpenGL versions
- Complete control over the rendering algorithm
- Loading complete objects or scenes from files (3ds, collada, qml...)
- Post-processing effects (single or multi-pass)
- Instanced rendering
- etc.

- Beyond the Tip of the Iceberg
- **The Future of Qt 3D**

What does the future hold for Qt 3D?

- Qt 3D Core
  - Efficiency improvemments
  - Backend threadpool and job handling improvements - jobs spawning jobs
- Qt 3D Render
  - Use Qt Quick or QPainter to render into a texture (5.9)
  - Embed Qt Quick into Qt 3D including input handling (5.9)
  - Level of Detail (LOD) support for meshes (5.9)
  - Text support - 2D and 3D (5.9)
  - Additional materials such as PBR materials (5.9)
  - Generating and filling buffers out of QAbstractItemModels
  - Billboards - camera facing entities
  - Particle systems
  - VR support

What does the future hold for Qt 3D?

- Qt 3D Input
  - Axis inputs that apply cumulative axis values as position, velocity or acceleration
  - Additional input device support
    - 3D mouse controllers, game controllers
  - Enumerated inputs such as 8-way buttons, hat switches or dials

What does the future hold for Qt 3D?

- New aspects:
  - Collision Detection Aspect
    - Allows to detect when entities collide or enter/exit volumes in space
  - Animation Aspect
    - Keyframe animation (5.9 TP)
    - Skeletal animation
    - Morph target animation
    - Removes animation workload from main thread
  - Physics Aspect
    - Rigid body and soft body physics simulation
  - AI Aspect, 3D Positional Audio Aspect ...
- Tooling:
  - Design time tooling - scene editor
  - Qt 3D Studio
  - Build time tooling - asset conditioners for meshes, textures etc.

What does the future hold for Qt 3D?

- Qt 3D and the rest of Qt:
  - DataVis, Mapping, etc. are likely to be based on Qt 3D
  - Work on unifying rendering toolset
    - Single renderer for Qt
    - Vulkan, Direct3D 12, Metal backends

Questions?

Thank you

Giuseppe D'Angelo

giuseppe.dangelo@kdab.com