AN ENTITY-COMPONENT SYSTEM IN C++ AND QT - JAMES TURNER

QT3D'S ECS

JAMES TURNER

- C++ and OpenGL developer
- Consultant in Qt for ten years, currently at KDAB
- Maintainer of open-source FlightGear simulator
- Can also lift people above my head

QT3D

- 3D visualisation engine built using Qt & C++11
- Data-driven scenes and rendering architecture
- Integration of possible simulation domains
  - Rendering, physics, animation, input, audio, networking
  - Arbitrary user-supplied domains
  - Called ‘aspects’ in Qt3D

WHY QT3D

- Recurring need for modern visualisation & simulation framework at KDAB
  - Frustration with existing visualisation / scene toolkits
- Embedded, industrial and scientific users wary of incorporating a game engine
- Data-driven rendering architecture
- Comparatively light-weight

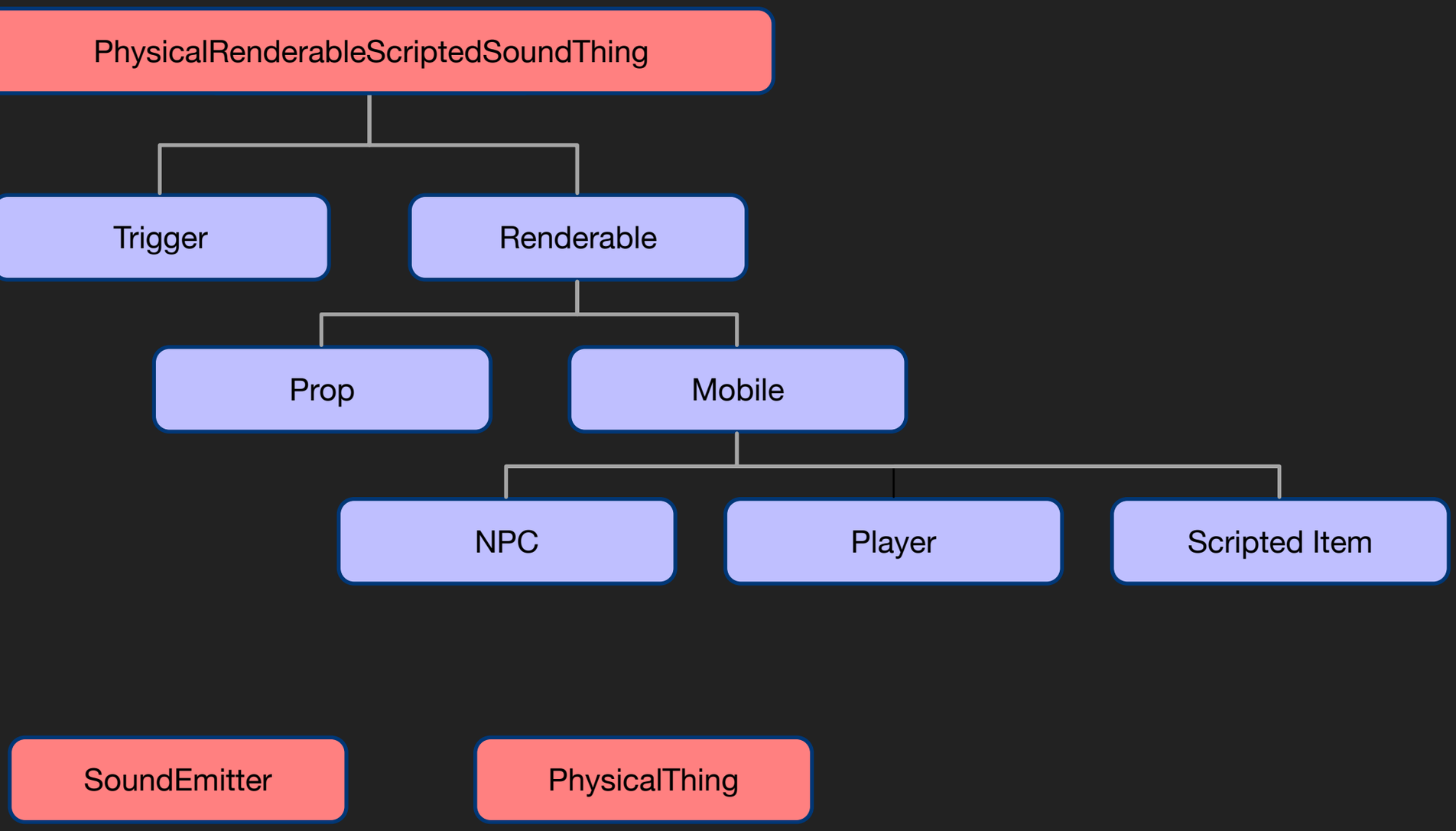
MOTIVATION

- Simulation architectures involve a collection of entities with behaviours & properties
  - Most entities use some subset of the behaviours and properties available
  - Different developers may want to extend, replace or reuse standard behaviours
  - Frameworks are only useful if you can reuse most of the architecture!
- Potential design patterns to deliver this in C++?

INHERITANCE GRAPHS

- Traditional OOP class hierarchy design
- Fix functionality at points in the hierarchy
- Common functionality rises higher and higher
- Multiple inheritance to aggregate functionality
  - But avoid diamond-inheritance graphs
- Use mix-in interfaces to improve somewhat

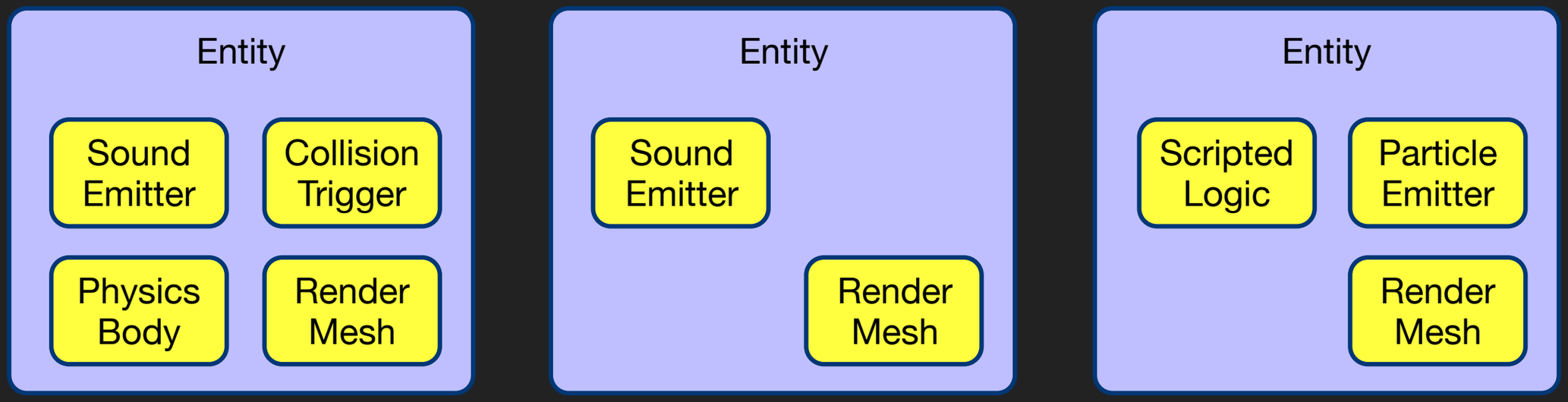
POSSIBLE CLASS HIERARCHY



BASIC ECS PRINCIPLES

- Composition replaces inheritance
- Entity is container for disparate components
  - Might be purely an ID, and C++ representation is generic
- Component is data (properties) describing some piece of functionality
  - Pertinent to one or several simulation systems (aspects)
- Entity is a collection of components
- Runtime behaviour of each entity derives entirely from its components

EXAMPLE LAYOUT



COMPONENT PRINCIPLES

- Fine-grained functionality
  - SoundEmitter, PhysicalMass, TriangleMesh, Material, ScriptedAnimation
- Component is data-only
  - Aspects contain code to process components
  - Different aspects can use the same component
- Components can be added and removed at run-time

ECS EVOLUTION AND USE

- Thief and Dungeon Keeper pioneered use of an ECS for games
- GDC presentations, Unity and ‘Game Design Patterns’ brought concept into public view
  - Most game engines now use an ECS, e.g. Lumberyard, XNA
- Outside the gaming world, things progress more slowly

QT3D ECS DESIGN GOALS

- All aspects are optional
- Easy to integrate existing logic as custom aspects
- Leverage existing Qt property system
  - Use existing QML declarative language for defining / binding properties
  - Mix-and-match C++ and QML defined entities
- Aggressive support for threading and dependencies between aspects
  - Manage non-blocking threading inside the aspects

CORE CLASSES

- Qt defines `QObject`
  - Base class for Qt-level features such as properties, introspection and signals / slots
- Qt3D extends this to `QNode`
  - Base for ECS objects, manages threading and state updating
  - `QEntity` and `QComponent` subclasses
- Nodes have unique ID, parents & children
- Entities have components

C++ API

```
Qt3DRender::QMaterial *material =
        new Qt3DExtras::QPhongMaterial(rootEntity);

// Torus
Qt3DCore::QEntity *torusEntity = new Qt3DCore::QEntity(rootEntity);
Qt3DExtras::QTorusMesh *torusMesh = new Qt3DExtras::QTorusMesh;
torusMesh->setRadius(5);
torusMesh->setMinorRadius(1);
torusMesh->setRings(100);
torusMesh->setSlices(20);

Qt3DCore::QTransform *torusTransform = new Qt3DCore::QTransform;
torusTransform->setScale3D(QVector3D(1.5, 1, 0.5));
torusTransform->setRotation(QQuaternion::fromAxisAndAngle(
    QVector3D(1, 0, 0), 45.0f));

torusEntity->addComponent(torusMesh);
torusEntity->addComponent(torusTransform);
torusEntity->addComponent(material);
```

INSTANCES & PROTOTYPES

- ECS supports a prototype concept, not inheritance
  - Any entity with appropriate components can be considered conforming to some interface (drawable, physical simulated, etc)
- Creating new components is rare, defining new entities with particular components and data values is common
  - ‘Subclassing’ means using using one entity definition as a prototype and adding components or changing properties
  - Unlike C+ inheritance, this process can be applied to any instance at any time
- When many entities are copies (instances) of some prototype, component data can be shared via Copy-on-Write or explicit sharing

BENEFITS OF USING QOBJECT

- Standard ECS has data-only components
- Different solutions to intra-component communication
  - Expose additional state to pass between aspects
  - Add messaging concept to each entity
- Qt already has signals & slots
  - Configured at runtime
  - Excellent match for linking components together

MAKING IT THREADED

- Modern framework requires good use of multi-core CPUs
  - High-level APIs should not expose threading complexity
- Expose single-threaded public API to ECS users in C++, QML, etc
- Split components in two
  - Front-end classes are lightweight
  - Back-end classes must synchronise state with their front-end peers and each other
- Writing components is more complicated
  - Core system manages mirroring of tree structure and messaging

ASPECT INTERFACE

- Aspects loaded from plugin
- Supply collection of component classes
  - Front-end and backend implementation
    - API is largely through component properties
- Engine queries each aspect for runnable jobs
  - Notional simulation tick / frame
  - Jobs have execution dependencies
  - Frame is complete when all jobs have run

C++ API

- Design goal to support natural C++ and QML APIs
- C++ API is verbose but straightforward
  - Scope to improve this considerably if desired
- When using existing components, threading is handled automatically
  - Creating custom components needs awareness of asynchronous events between the front-end and backend.

FRONT-END / BACK-END

- Nodes have unique ID
- Front-end components are light-weight
  - Expose pleasant API to C++, QML
- Backend peer objects constructed automatically
- Changes to properties and structure generate events
  - Periodically dispatched to backend
  - Backend node hierarchy state is synchronised
  - Aspects notified about changes

MANAGING CHANGE

- Different kinds of change
  - Properties changing on components
  - Scene entity hierarchy changing
    - Add, destroy and re-parent
  - Adding or remove components dynamically
- Changes come from the front-end or backend
  - Different propagation behaviour
  - Ensure aspect is not processing jobs before processing changes

CODE...

RUNNING JOBS

- Job scheduling problem
- Originally we used ThreadWeaver
  - Worked very well!
  - Intel Thread Building Blocks also solves this nicely
  - Licence & dependency concerns prompted switch to an internal solution
- Considerable scope for further development
  - CPU intensive aspects
  - Compute (OpenCL) aspects

TOOLING

- ECS maps well to generic visual tooling
  - Non-coders can compose entities, set properties, publish and import libraries of entities
  - Qt signals/slots support introspection, so can also be connected up via tooling UI
- Tooling is considerable work
  - Being planned at the moment

ECS FTW

- ECS has delivered on its premise
  - Creating prototype aspects is straightforward
  - Excellent isolation of code & data between aspects
  - Least contentious design decision in Qt3D
- Combining the front/back-end split with the ECS increased complexity
  - For a narrower target application, potentially excessive
  - Hopefully future-proof, scalable and developer-friendly

DONE

- Thanks for listening!
- Try out Qt3D in Qt 5.7 or 5.8
- james@kdab.com
- Questions?