basysKom

A deep dive into QML memory management internals

Frank Meerkötter

basysKom GmbH

07.10.2015

- Motivation/Intro
- QML memory management
- JavaScript memory management
- Tools
- Conclusion

About myself

- Qt developer since Qt3/Qtopia times, 10+ years experience
- Development Lead with basysKom GmbH in Darmstadt
- Strong focus on all things (Embedded) Linux
- Enthusiasm for systems programming

Why this talk?

- Memory management in QML is seen as (mostly) automatic
  - Convenient
  - Eliminates certain types of errors

- So why bother?
  - Intransparent
  - Less control
  - Demanding applications
  - Resource constrained devices

- Goal: get a conceptual understanding how this works

Scope

- Qt5.5 is used as reference
- Earlier versions are referenced when pointing out important changes
- Qt4/Qt5 <5.2 are not covered (anything before the V4 engine)

- A Linux platform is implicitly assumed
  – most insights can be applied to other platforms too

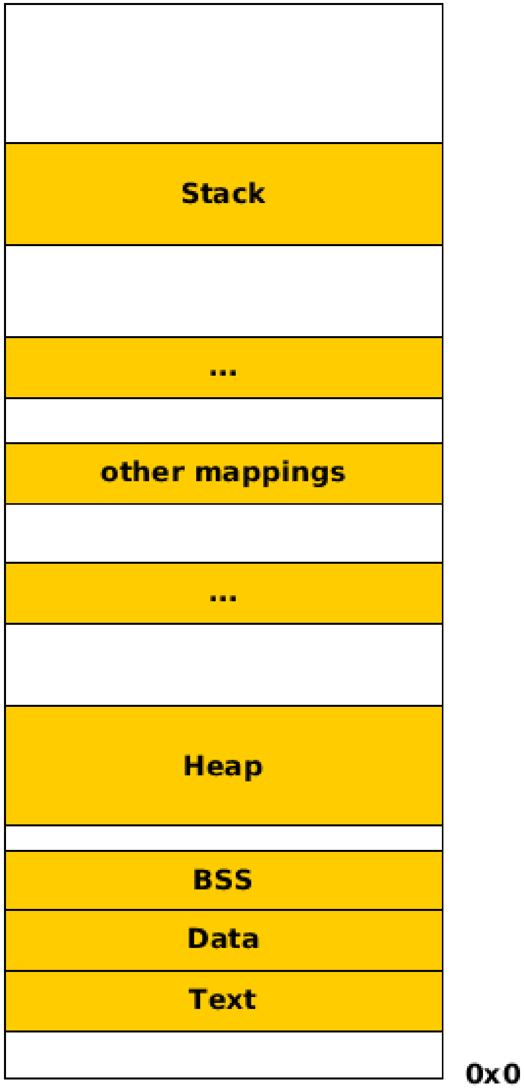
- This talk focuses on things related to memory management itself
  – Expect some glaring omissions and hand waving for other areas!

Before we get started...

Basics of memory management

- Virtual memory
  - Each process has its own address space
- Only certain segments are actually mapped
  - The dreaded segfault!
- Mappings can be created through the mmap() syscall

- Mappings have different roles
  - Text: program code
  - BSS/Data: (uninitialized) static variables
  - Stack(s)
  - Heap(s)
  - ...



The process heap

- Managed through a malloc implementation
  - Typically part of your libc
- Acquires memory from the OS either by
  - growing a special heap-mapping via sbrk()
  - creating additional mappings via mmap()
- Keeps memory in its own pool
- malloc()/new is served from this pool
- free()/delete gives back to this pool
- The malloc implementation can try to give memory back to the OS
  - Can't move around allocations of C/C++ programs
  - Might focus on performance

Memory management for QML & JS

- QML is a declarative language used to describe user interfaces
  - hierarchy and relationship of UI elements/objects
- JavaScript can be embedded to implement UI logic

- How are these two distinct parts handled by the engine?
- How does the memory management work for these two?

QML memory management

QML objects – the very basics

- QML object types are implemented in C++
  - Non-visual QML elements derive directly from QObject
  - Visual QML elements derive from QQuickItem (which is derived from QObject)
  - E.g. a „Rectangle {}“ is implemented by the C++ class QQuickRectangle

- The QML source describes how to assemble a tree of QObjects
- QML objects are allocated on the normal process heap
- Each object has a parent (leaving out the root)
  - the parent cannot be changed (from the QML side)
  - not to be confused with the visual parent

Methods to create QML objects

- Static:
  - QQuickView::setSource(QUrl("…"))
  - QQmlApplicationEngine::load(QUrl("…"))
  - …
- Dynamic:
  - Loader
  - Qt.createComponent()/component.createObject(parent)
- Typically a static "shell" is dynamically loading sub-components on demand

- All these methods create a tree of QML objects
- An object that gets destroyed will also (recursively) destroy its children
  - The same mechanism as in Qt
- No garbage collection involved (for the QML objects itself)!

QML properties

- Rectangle { property int foo; property var bar }
- Properties defined in QML source need to
  - be stored somewhere
  - integrate with the rest of the metaobject system
- QQmlVMEMetaObject takes care of that
- typed properties (non-var) are stored on the process heap (QQmlVMEVariant objects)
- var properties are stored as QV4::Values in an QV4::Array which resides on the JS heap

- This will change with Qt5.6
  - QQmlVMEVariant weighs in at 8*sizeof(void*) + sizeof(int) => 36/72 bytes
  - Everything will be stored in a QV4::Value (8 bytes)

QML properties

- What happens to a property when its object is deleted?
  - The parts allocated on the process heap are directly deleted with the object
  - The parts stored on the JS-side are orphaned and left for garbage collection

- What happens to a QML object stored in a var property?
  - Still cleaned up via the QObject hierarchy, no GC

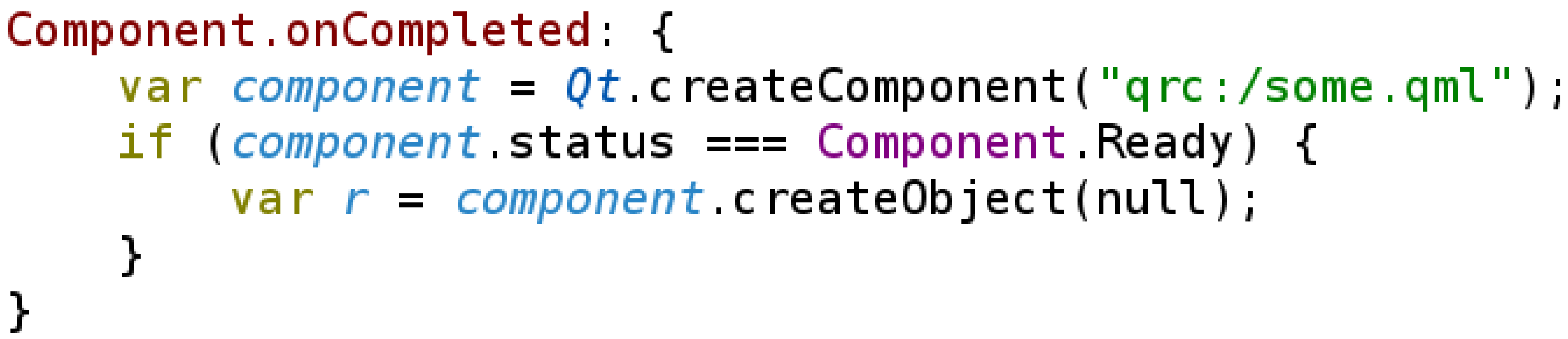
Is the GC ever collecting QObjects?

Yes, if an object has

– QQmlEngine::JavaScriptOwnership

– no parent

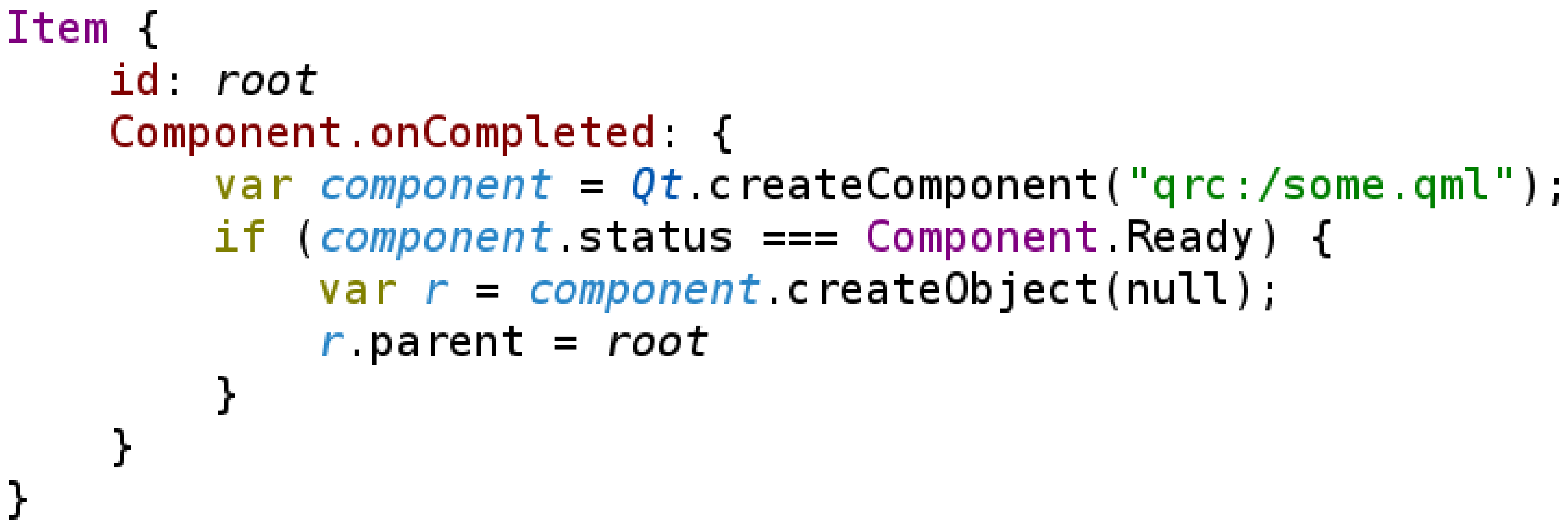
– no remaining JavaScript references



```
Component.onCompleted: {
    var component = Qt.createComponent("qrc:/some.qml");
    if (component.status === Component.Ready) {
        var r = component.createObject(null);
    }
}
```

Bonus question

- Will the GC ever collect a visible QObject?
- No, the visual parent will keep its visual children alive



```
Item {
    id: root
    Component.onCompleted: {
        var component = Qt.createComponent("qrc:/some.qml");
        if (component.status === Component.Ready) {
            var r = component.createObject(null);
            r.parent = root
        }
    }
}
```

Wrap up

- QML objects
  - are allocated from the process heap
  - deallocated via delete/deleteLater
- Children are cleaned up via the Qt object hierarchy

- QML allows you to control the life-time of objects
  - (typically) no garbage collection involved
- Make use of it!
  - Loader/dynamic object creation
  - Unload elements no longer needed
  - Make sure to call .destroy() on dynamically created components

JavaScript memory management

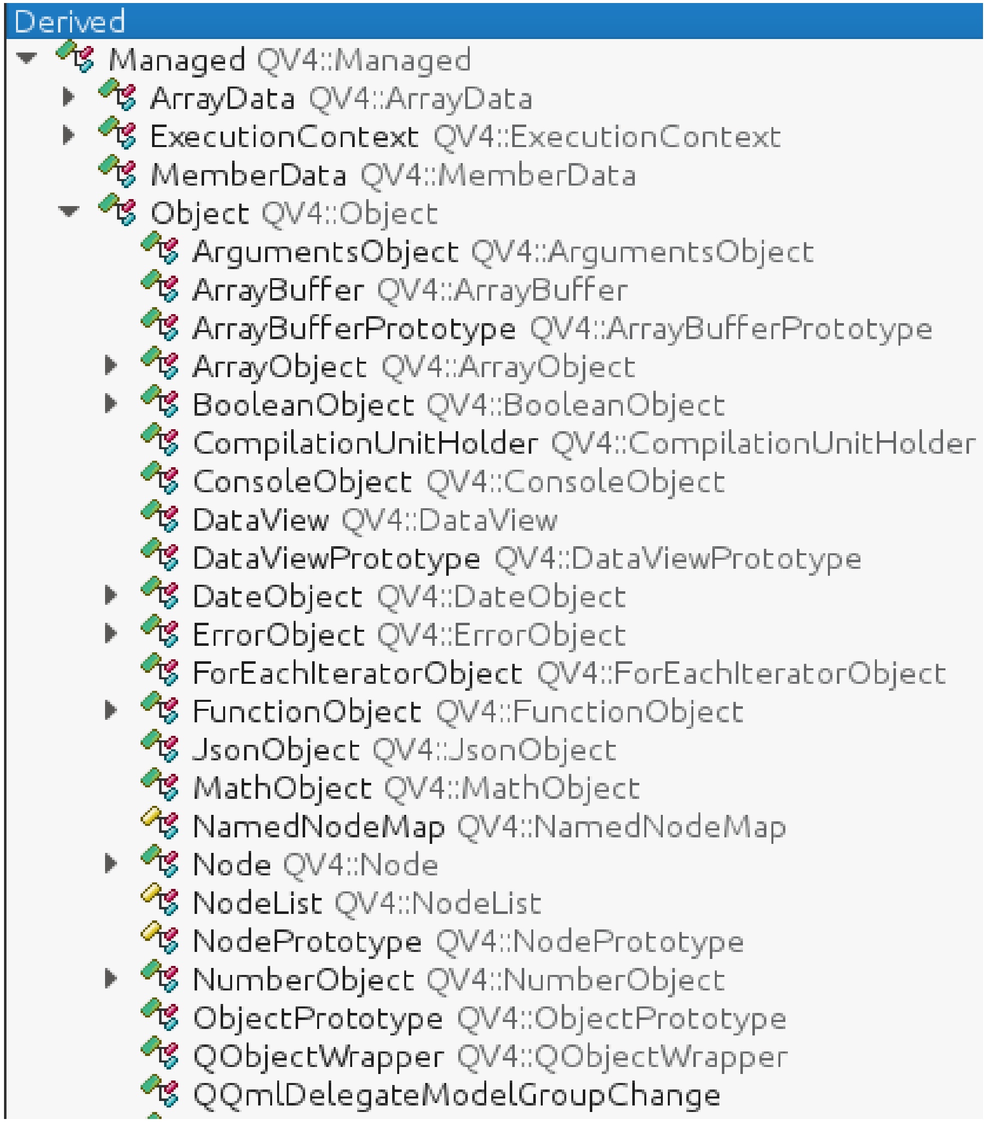
JavaScript

- JavaScript in QML can by used in
  - property bindings
  - signal handlers
  - custom methods
  - standalone

- To support this the QML engine implements a JS host environment
  - The V4 engine since Qt5.2
- The code for the various JavaScript types is written in C++
- Instances are allocated from a separate garbage collected JS heap

JavaScript types

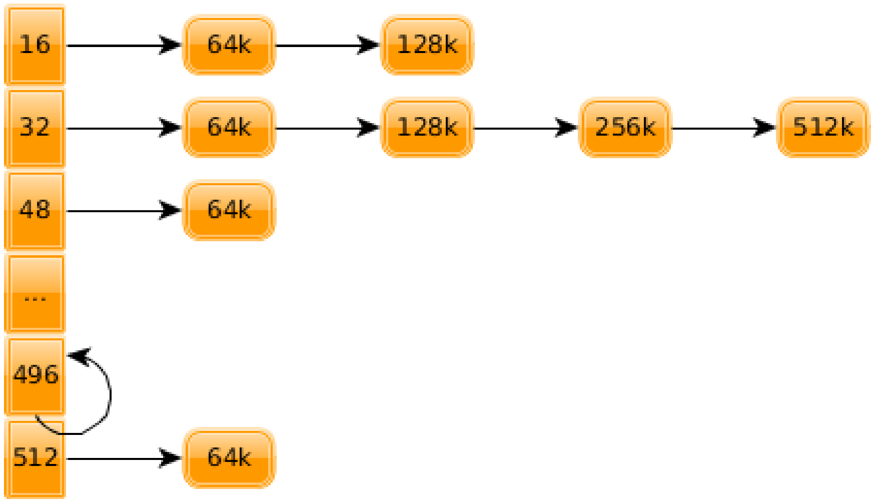
- A JavaScript type can be something visible in the host environment
  - Object, Array, Date, RegEx
- Or it can be something internal
  - plumbing of the JS host environment
    - QV4::MemberData
    - QV4::ExecutionContext
    - ...
  - QML/JS integration
    - QV4::QQmlBindingWrapper
    - QV4::QObjectWrapper
    - ...



```
Derived
▼ Managed QV4::Managed
    ▶ ArrayData QV4::ArrayData
    ▶ ExecutionContext QV4::ExecutionContext
      MemberData QV4::MemberData
    ▼ Object QV4::Object
          ArgumentsObject QV4::ArgumentsObject
          ArrayBuffer QV4::ArrayBuffer
          ArrayBufferPrototype QV4::ArrayBufferPrototype
        ▶ ArrayObject QV4::ArrayObject
        ▶ BooleanObject QV4::BooleanObject
          CompilationUnitHolder QV4::CompilationUnitHolder
          ConsoleObject QV4::ConsoleObject
          DataView QV4::DataView
          DataViewPrototype QV4::DataViewPrototype
        ▶ DateObject QV4::DateObject
        ▶ ErrorObject QV4::ErrorObject
          ForEachIteratorObject QV4::ForEachIteratorObject
        ▶ FunctionObject QV4::FunctionObject
          JsonObject QV4::JsonObject
          MathObject QV4::MathObject
          NamedNodeMap QV4::NamedNodeMap
        ▶ Node QV4::Node
          NodeList QV4::NodeList
          NodePrototype QV4::NodePrototype
        ▶ NumberObject QV4::NumberObject
          ObjectPrototype QV4::ObjectPrototype
          QObjectWrapper QV4::QObjectWrapper
          QQmlDelegateModelGroupChange
```

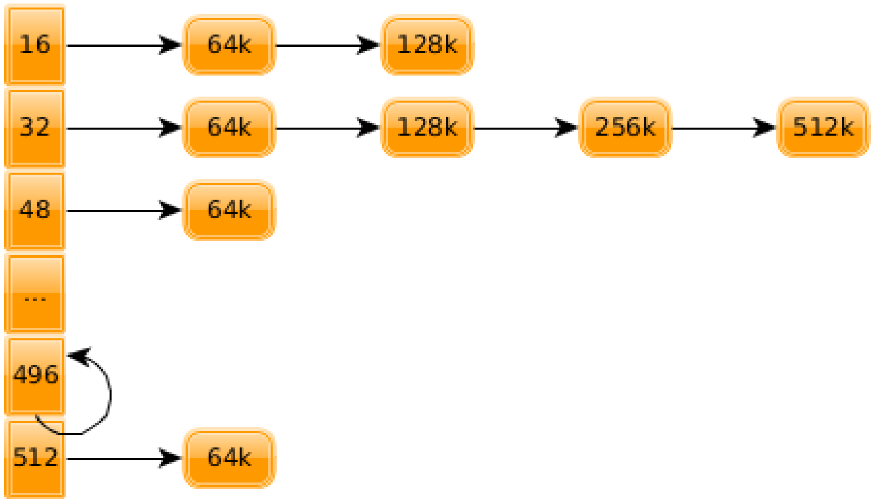
The JavaScript heap

- Implemented in QV4::MemoryManager
- QV4::MemoryManager::allocData(std::size_t) allocates storage for JS objects
  - There are 32 buckets (16, 32, 48, ..., 512 bytes)
  - Allocations are rounded up to the next multiple of 16
  - "Segregated-fits-allocation”
- Buckets are backed by chunks of memory which are allocated on demand



The JavaScript heap

- Memory for the buckets is not aquired through malloc
- The WTF::PageAllocation platform abstraction is used instead
  - mmap'd for a POSIX system
  - VirtualAlloc on Windows
- Exception: anything larger than 512 bytes is a special case and just malloc'd/free'd

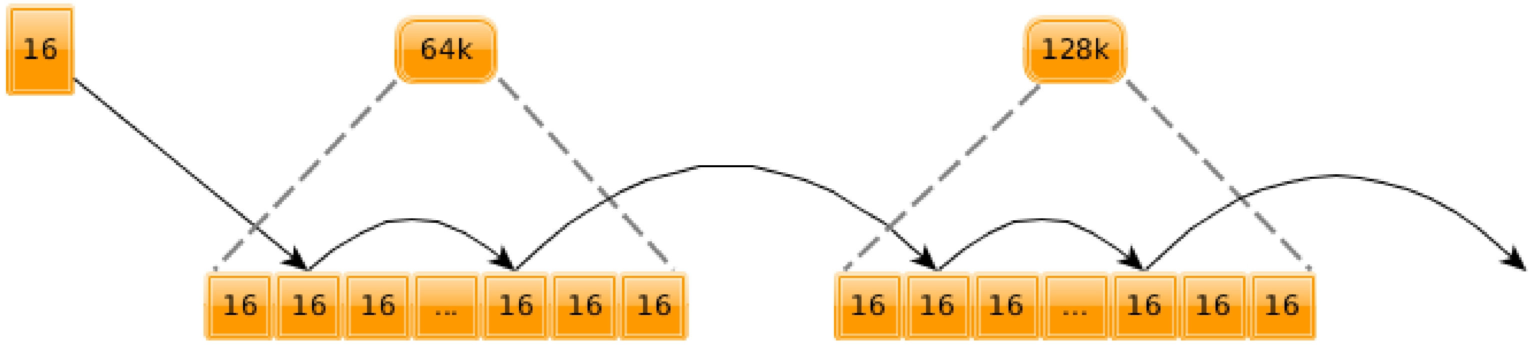


- "Segregated-fits-allocation”:
  - Robust against external fragmentation
  - Some internal fragmentation

Bucket management

- Chunks are chopped into n-sized items which are put on the freelist for a given bucket
- When the freelist is empty
  - either a new chunk is allocated from the OS
  - or the garbage collector is triggered
- A newly allocated chunk is committed memory

- The only way to deallocate JS objects is to run the GC



JavaScript heap: interesting properties

- The size of chunks being allocated for a certain bucket follows a growth strategy
  - The first chunk has 64KB
  - Size of each new allocation for a certain bucket is always doubled
- In recent Qt versions (Qt5.3) this series is capped at 2MB, earlier versions would only cap at 64MB
  - high potential to waste (committed!) memory
- Since Qt5.3 the exact behaviour can be fine tuned
- QV4_MM_MAXBLOCK_SHIFT
  - Allows to modify the growth cap
- QV4_MM_MAX_CHUNK_SIZE
  - Allows to set the size from where chunk growth starts

How does the GC work?

- Triggered either through
  - an allocation (depending on usage metrics)
  - manually (JS/C++)
- Runs in the main thread, blocks the application
- The implementation can be found in QV4::MemoryManager

- Tracing GC/mark&sweep
- Two phases

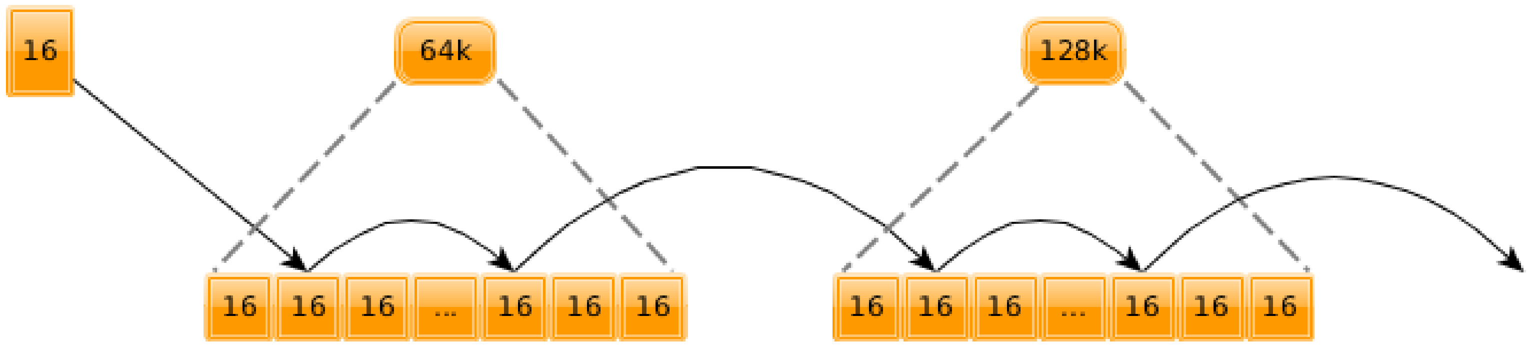
GC: Phase 1

- Starting from certain known „roots“ all reachable objects are marked
  - "Mark" sets a marker bit in each object
  - everything not marked is garbage and can be free'd

- JS stack allows for a non-recursive implementation
- Initially a conservative GC, now an exact GC (the default since Qt5.2)

GC: Phase 2

- Sweep is now walking all chunks
  - All objects marked, have their mark cleared
  - All objects not marked are destroyed, nulled and put back into a freelist
- Chunks which become empty can be given back to the OS
  - New with Qt5.5, earlier versions are not able to ever get rid of a peak!
- On engine shutdown a last sweep is done without a mark



Objectives of the GC

- The GC is freeing unused objects from the JS heap
- It does not take into account the overall memory usage of the host process

- Works as expected, but can exhibit some interesting behaviour:
  - A QV4::String holds internally a QStringData*, the actual string data is on the C++ heap
  - A large string will look small to the GC, but will have a considerable footprint on the C++ heap
  - The GC will never clean up, the host memory usage will go through the roof
  - This has improved with Qt5.5
  - The GC metric is extended to take into account the real weight of QV4::Strings

Should I manually trigger the GC?

- In general: no

- Exceptions to the rule:
- the application is idle (and no one is looking)

- after unloading a large QML component
  - Ensure to pass through the eventloop once, before calling gc()
  - Try to run malloc_trim(0) to encourage malloc to give memory back to the OS

Wrap up

- JavaScript objects
  - are allocated from a separate JavaScript heap
    - with the exception of large items
  - deallocated only via the GC
    - also large items are gc'd
- The GC is triggered either
  - through utilisation metrics
  - manually

Tools for memory profiling

Tools for memory profiling

- How much memory is used overall?
- How much memory is used on the QML-side?
- How much memory is used on the JavaScript-side?
- What caused an allocation?

- Let's review the tools...

- Usage overall
  - Various means offered by your specific OS
  - /proc/$pid/smaps on Linux for example
  - Understand what you are actually measuring
    - Virtual memory vs. RSS vs. PSS

Built-in

- QV4_MM_STATS

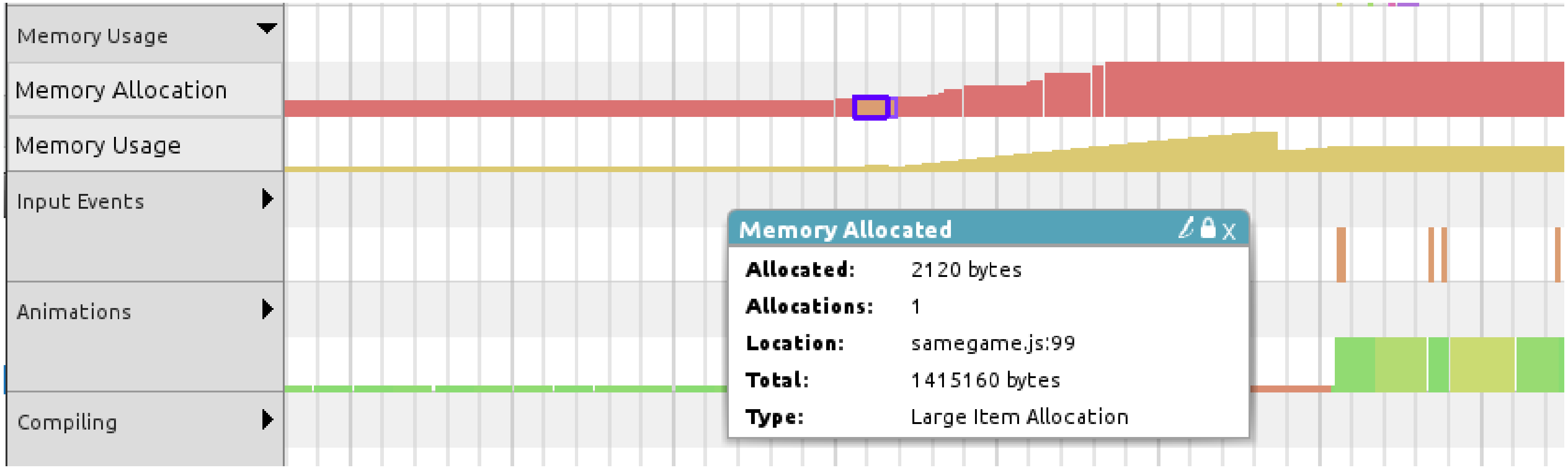
- ~2.8MB of memory has been acquired from the OS for the JS heap
- ~700KB of it are in use
- 3 mappings have been given back to the OS (must be a Qt >= 5.5)
- Large items (>512 bytes) are not shown
  - Added in Qt5.6

- Note: QV4_MM_AGGRESSIVE_GC is an internal developer tool

```
$ export QV4_MM_STATS=1
$ ./myQmlApp
========== GC ==========
Marked object in 6 ms.
Sweeped object in 3 ms.
Allocated 2883584 bytes in 21 chunks.
Used memory before GC: 1313984
Used memory after GC: 698736
Freed up bytes: 615248
Released chunks: 3
======== End GC ========
...
```

QtCreator memory profiler

- The commercial version of Qt has a JavaScript memory profiler
- Upper bar (Memory Allocation) visualizes the memory acquired from the OS
  - – Mappings and LargeItems

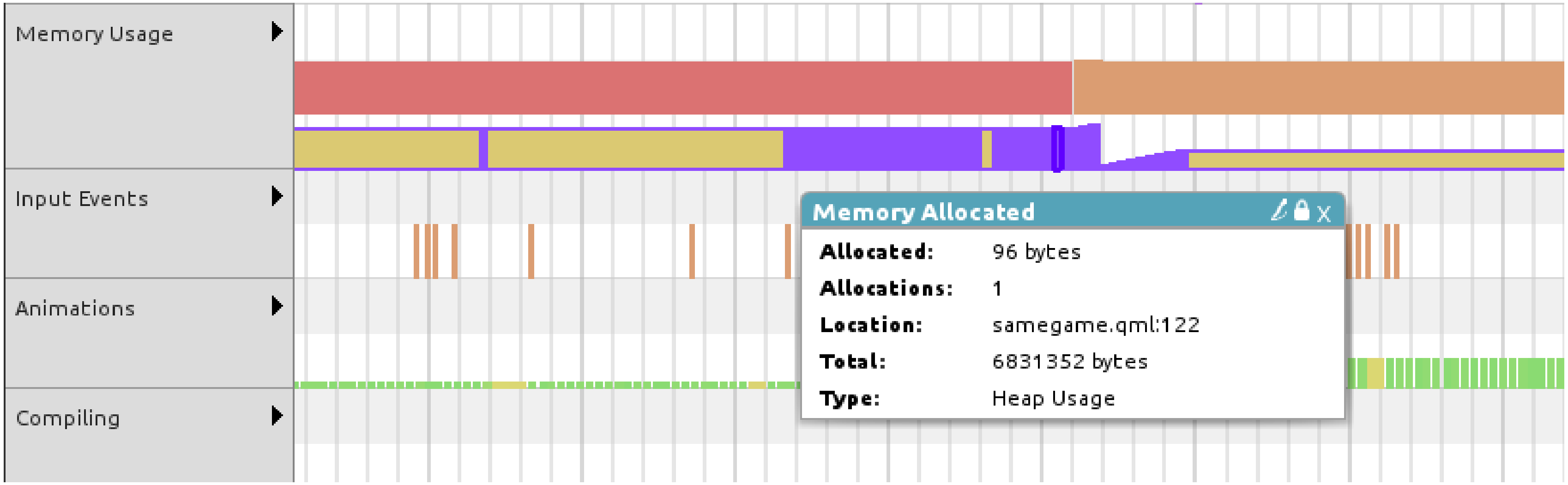


Memory Allocated

Allocated:	2120 bytes
Allocations:	1
Location:	samegame.js:99
Total:	1415160 bytes
Type:	Large Item Allocation

QtCreator memory profiler

- Lower bar (Memory Usage) visualizes the actual usage by the application

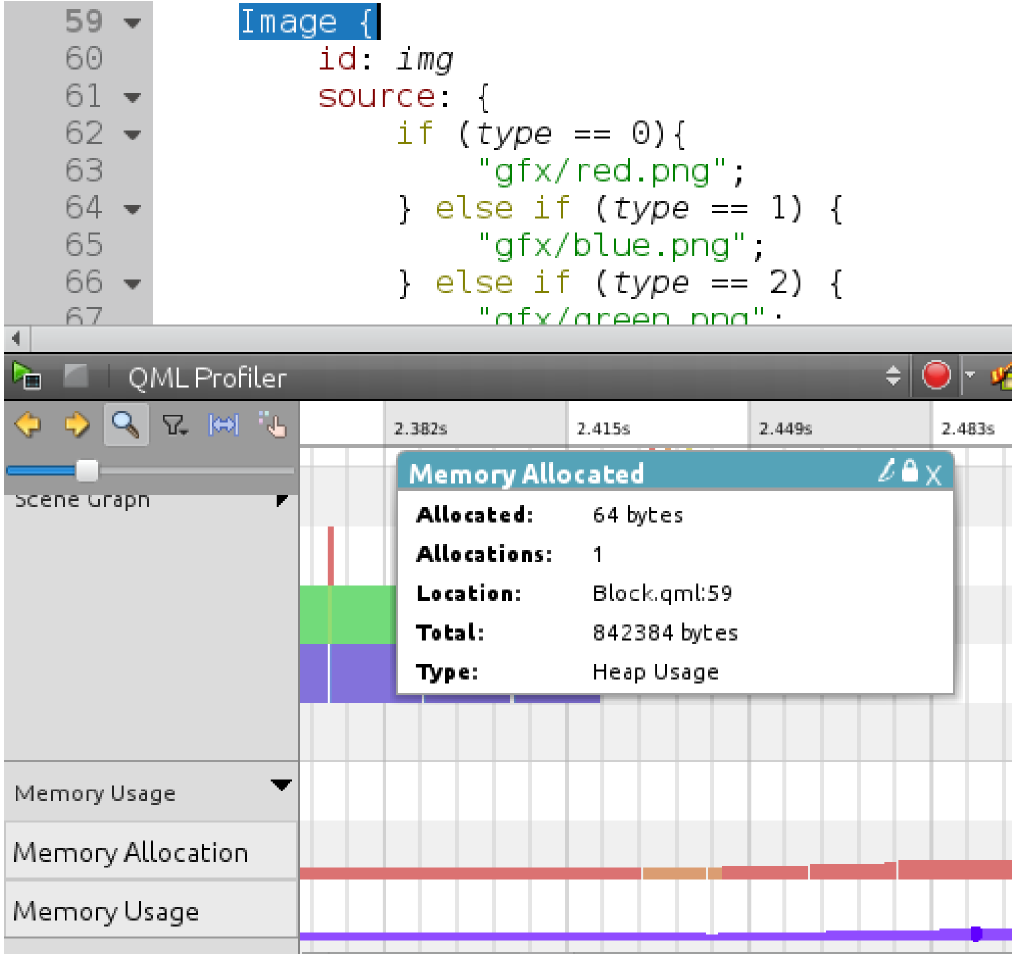


Memory Allocated

Allocated:	96 bytes
Allocations:	1
Location:	samegame.qml:122
Total:	6831352 bytes
Type:	Heap Usage

QtCreator memory profiler

- Profiling information links back to the source
  - Often no obvious mapping between an allocation and the responsible source location
  - Inherent: Qt/JavaScript plumbing, primitives of the JS runtime
  - Not so clear how to act on this information
- Shines when combined with the other timeline information
  - Animation
- Does not show the QML-side
  - It is a JavaScript profiler after all!



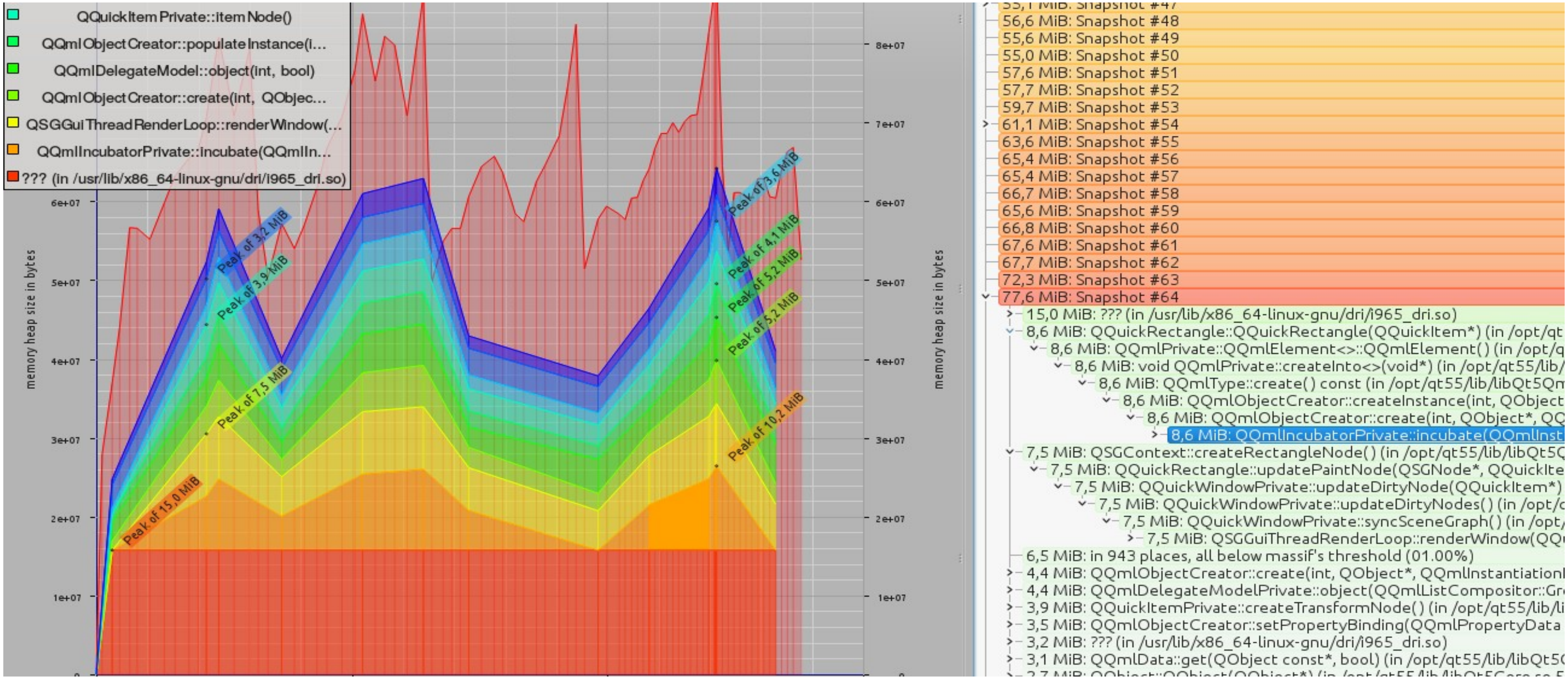
```
59      Image {
60          id: img
61          source: {
62              if (type == 0){
63                  "gfx/red.png";
64              } else if (type == 1) {
65                  "gfx/blue.png";
66              } else if (type == 2) {
67                  "gfx/green.png";
```

Memory Allocated

Allocated:	64 bytes
Allocations:	1
Location:	Block.qml:59
Total:	842384 bytes
Type:	Heap Usage

valgrind/massif/massif-visualizer

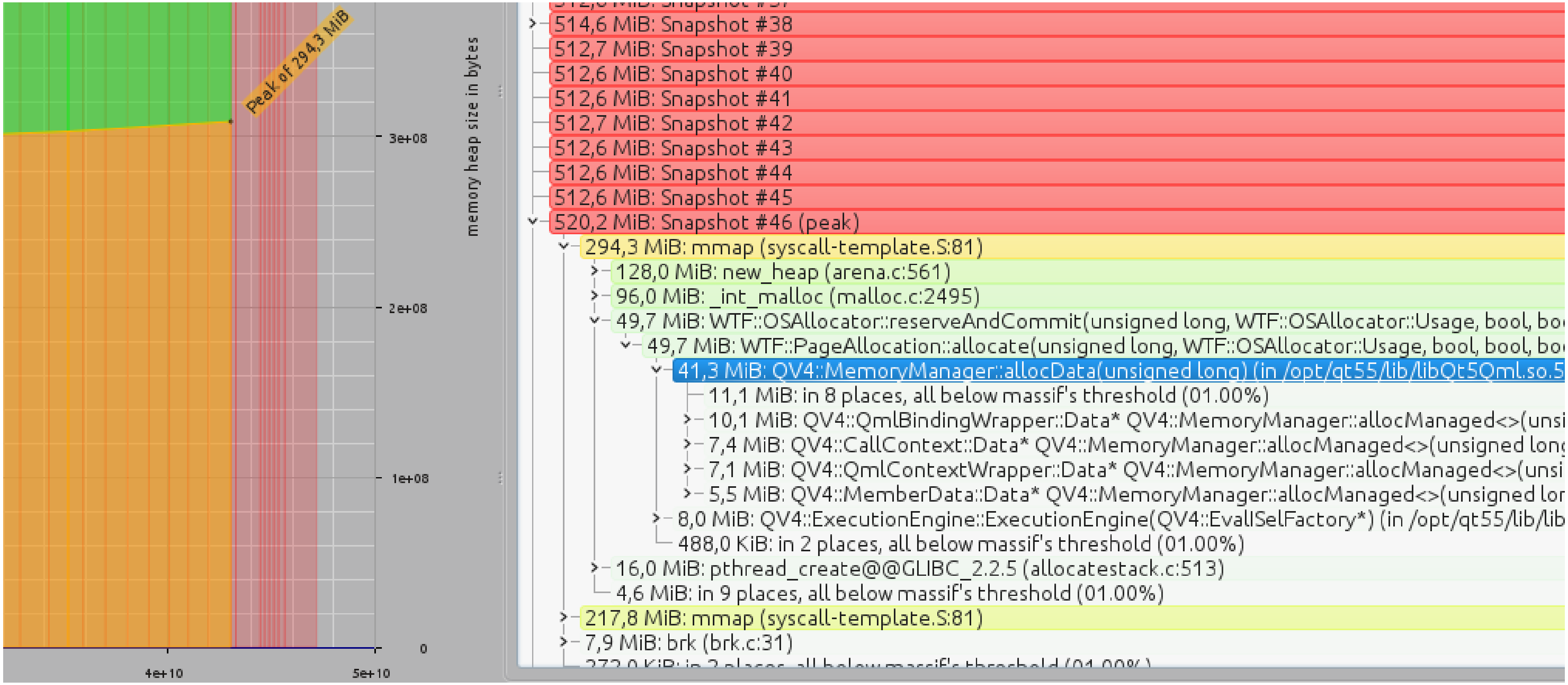
- Shows allocations on the process heap
  - QML objects are visible
  - No link back to the QML source
- No visibility of objects on the JS heap!


QQuickItemPrivate::itemNode()
QQmlObjectCreator::populateInstance(i...
QQmlDelegateModel::object(int, bool)
QQmlObjectCreator::create(int, QObjec...
QSGGuiThreadRenderLoop::renderWindow(...
QQmlIncubatorPrivate::incubate(QQmlIn...
??? (in /usr/lib/x86_64-linux-gnu/dri/i965_dri.so)
memory heap size in bytes
6e+07
5e+07
4e+07
3e+07
2e+07
1e+07
0
8e+07
7e+07
Peak of 3,2 MiB
Peak of 3,9 MiB
Peak of 7,5 MiB
Peak of 15,0 MiB
Peak of 3,6 MiB
Peak of 4,1 MiB
Peak of 5,2 MiB
Peak of 5,2 MiB
Peak of 10,2 MiB

```
55,1 MiB: Snapshot #47
56,6 MiB: Snapshot #48
55,6 MiB: Snapshot #49
55,0 MiB: Snapshot #50
57,6 MiB: Snapshot #51
57,7 MiB: Snapshot #52
59,7 MiB: Snapshot #53
61,1 MiB: Snapshot #54
63,6 MiB: Snapshot #55
65,4 MiB: Snapshot #56
65,4 MiB: Snapshot #57
66,7 MiB: Snapshot #58
65,6 MiB: Snapshot #59
66,8 MiB: Snapshot #60
67,6 MiB: Snapshot #61
67,7 MiB: Snapshot #62
72,3 MiB: Snapshot #63
77,6 MiB: Snapshot #64
  15,0 MiB: ??? (in /usr/lib/x86_64-linux-gnu/dri/i965_dri.so)
  8,6 MiB: QQuickRectangle::QQuickRectangle(QQuickItem*) (in /opt/qt
    8,6 MiB: QQmlPrivate::QQmlElement<>::QQmlElement() (in /opt/q
      8,6 MiB: void QQmlPrivate::createInto<>(void*) (in /opt/qt55/lib/
        8,6 MiB: QQmlType::create() const (in /opt/qt55/lib/libQt5Qm
          8,6 MiB: QQmlObjectCreator::createInstance(int, QObject
            8,6 MiB: QQmlObjectCreator::create(int, QObject*, QQ
              8,6 MiB: QQmlIncubatorPrivate::incubate(QQmlInst
  7,5 MiB: QSGContext::createRectangleNode() (in /opt/qt55/lib/libQt5Q
    7,5 MiB: QQuickRectangle::updatePaintNode(QSGNode*, QQuickIte
      7,5 MiB: QQuickWindowPrivate::updateDirtyNode(QQuickItem*)
        7,5 MiB: QQuickWindowPrivate::updateDirtyNodes() (in /opt/c
          7,5 MiB: QQuickWindowPrivate::syncSceneGraph() (in /opt/
            7,5 MiB: QSGGuiThreadRenderLoop::renderWindow(QQ
  6,5 MiB: in 943 places, all below massif's threshold (01.00%)
  4,4 MiB: QQmlObjectCreator::create(int, QObject*, QQmlInstantiation
  4,4 MiB: QQmlDelegateModelPrivate::object(QQmlListCompositor::Gr
  3,9 MiB: QQuickItemPrivate::createTransformNode() (in /opt/qt55/lib/li
  3,5 MiB: QQmlObjectCreator::setPropertyBinding(QQmlPropertyData
  3,2 MiB: ??? (in /usr/lib/x86_64-linux-gnu/dri/i965_dri.so)
  3,1 MiB: QQmlData::get(QObject const*, bool) (in /opt/qt55/lib/libQt5
  2,7 MiB: QObject::QObject(QObject*) (in /opt/qt55/lib/libQt5Core.so.5
```

Another perspective

- valgrind –tool=massif –pages-as-heap=yes
- Objects on the JS heap?
- Careful: shows only what triggered the initial allocation, not what is currently stored!



```
512,6 MiB: Snapshot #37
514,6 MiB: Snapshot #38
512,7 MiB: Snapshot #39
512,6 MiB: Snapshot #40
512,6 MiB: Snapshot #41
512,7 MiB: Snapshot #42
512,6 MiB: Snapshot #43
512,6 MiB: Snapshot #44
512,6 MiB: Snapshot #45
520,2 MiB: Snapshot #46 (peak)
  294,3 MiB: mmap (syscall-template.S:81)
    128,0 MiB: new_heap (arena.c:561)
    96,0 MiB: _int_malloc (malloc.c:2495)
    49,7 MiB: WTF::OSAllocator::reserveAndCommit(unsigned long, WTF::OSAllocator::Usage, bool, boo
      49,7 MiB: WTF::PageAllocation::allocate(unsigned long, WTF::OSAllocator::Usage, bool, bool, boo
        41,3 MiB: QV4::MemoryManager::allocData(unsigned long) (in /opt/qt55/lib/libQt5Qml.so.5
          11,1 MiB: in 8 places, all below massif's threshold (01.00%)
          10,1 MiB: QV4::QmlBindingWrapper::Data* QV4::MemoryManager::allocManaged<>(unsi
          7,4 MiB: QV4::CallContext::Data* QV4::MemoryManager::allocManaged<>(unsigned long
          7,1 MiB: QV4::QmlContextWrapper::Data* QV4::MemoryManager::allocManaged<>(unsi
          5,5 MiB: QV4::MemberData::Data* QV4::MemoryManager::allocManaged<>(unsigned lor
        8,0 MiB: QV4::ExecutionEngine::ExecutionEngine(QV4::EvalISelFactory*) (in /opt/qt55/lib/lib
        488,0 KiB: in 2 places, all below massif's threshold (01.00%)
    16,0 MiB: pthread_create@@GLIBC_2.2.5 (allocatestack.c:513)
    4,6 MiB: in 9 places, all below massif's threshold (01.00%)
  217,8 MiB: mmap (syscall-template.S:81)
  7,9 MiB: brk (brk.c:31)
  272,0 KiB: in 2 places, all below massif's threshold (01.00%)
```

Wrap up

- Overall memory usage => OS specific methods
- JavaScript memory usage => QV4_MM_STATS, QtCreator
- QML memory usage => Overall usage – JavaScript usage?
  - Misleading: Counts all other memory usage as QML memory usage...
  - Valgrind/massif can help to break this down further

- No clear mapping between a line of code and the resulting allocation

basysKom

Conclusion

Conclusion

- A conceptual understanding how QML memory management works
- QML: allows you to control the life-time of objects
- JavaScript: No direct control over object life-time

- Memory management has improved throughout Qt5
- Use an up to date version of Qt
  - If you can't, be aware of version specific behaviour
  - E.g. avoid memory peaks with a Qt < 5.5

- For memory constrained environments
  - Less is more (especially for delegates)
  - Plan for dynamic object loading/unloading
  - Limit the ammount of JavaScript

Questions?

basysKom

Contact

Frank Meerkötter
Development Lead

frank.meerkoetter@basyskom.com
+49 (6151) 870 589 0

Company

basysKom GmbH
Robert-Bosch-Str. 7
64293 Darmstadt
Germany

sales@basyskom.com
+49 (6151) 870 589 0

www.basyskom.com