Data-oriented design in practice

Stoyan Nikolov

@stoyannk
Who am I?

- In the video games industry for 10+ years
- Software Architect at Coherent Labs
- Working on game development technology
- Last 6.5 years working on
  - chromium
  - WebKit
  - Hummingbird - in-house game UI & browser engine
- High-performance maintainable C++

Games using Coherent Labs technology
Images courtesy of Rare Ltd., PUBG Corporation
DEMO video of performance on Android
Agenda

- Basic issue with Object-oriented programming (OOP)
- Basics of Data-oriented design (DoD)
- Problem definition
- Object-oriented programming approach
- Data-oriented design approach
- Results & Analysis
OOP marries data with operations...

- ...it’s not a happy marriage
- Heterogeneous data is brought together by a “logical” **black box** object
- The object is used in vastly different contexts
- Hides **state** all over the place
- Impact on
  - Performance
  - Scalability
  - Modifiability
  - Testability
- **YMMV** but a lot of code-bases (even very successful) do - how do we fix it?
Data-oriented design

- Data A
  - Field A
  - Field B
  - Field C

- Data B
  - Field D
  - Field E
  - Field F

- System α

- System β

- Data C
  - Field G

- System γ

- Data D
  - Field H

- Logical Entity 0
  - Field A[0]
  - Field B[0]
  - Field C[0]

- Logical Entity 1
  - Field A[1]
  - Field B[1]
  - Field C[1]

- OOP data layout
- DoD layout
Data-oriented design

- **Separates** data from logic
  - Structs and functions live independent lives
  - Data is regarded as information that has to be transformed
- **Build for a specific machine**
  - Improve *cache utilization*
- **Reorganizes data according to its usage**
  - The logic embraces the data
  - Does not try to hide it
  - Leads to functions that work on arrays
  - If we aren’t going to use a piece of information, why pack it together?
  - Avoids “hidden state”
- **Promotes deep domain knowledge**
- **References** at the end for more detail
Data-oriented design & OOP

- “Good” OOP shares a lot of traits with data-oriented design
  - But “good” OOP is hard to find
- Thinking in a data-oriented framework will improve your OOP code as well!

*Mature programmers know that the idea that everything is an object is a myth. Sometimes you really do want simple data structures with procedures operating on them.*

Robert C. Martin
Data-oriented design has been mostly demonstrated in video games.
Let’s apply data-oriented design to something that is not a game..
The system at hand
What is a CSS Animation?
Animation definition

@keyframes example {
  from {left: 0px;}
  to {left: 100px;}
}
div {
  width: 100px;
  height: 100px;
  background-color: red;
  animation-name: example;
  animation-duration: 1s;
}

- Straightforward declaration
  - Interpolate some properties over a period of time
  - Apply the Animated property on the right Elements

- However at a second glance..
  - Different property types (i.e. a **number** and a **color**)
  - There is a DOM API (JavaScript) that requires the existence of some classes (Animation, KeyframeEffect etc.)
Let’s try OOP
The OOP way (chromium 66)

- chromium has 2 Animation systems
  - We’ll be looking at the Blink system
- Employs some classic although “old school” OOP
  - Closely follows the HTML5 standard and IDL
  - Running Animation are **separate objects**
- Study chromium - it’s an amazing piece of software, **a lot** to learn!
What is so wrong with this?
The flow

- Unclear lifetime semantics
The state

- Hidden state
- Branch mispredictions

```cpp
bool Animation::Update(TimingUpdateReason reason) {

    if (!timeline_)
        return false;

    PlayStateUpdateScope update_scope(*this, reason, kDoNotSetCompositorPending);
    ClearOutdated();
    bool idle = PlayStateInternal() == kIdle;

    if (content_) {
        double inherited_time = idle || IsNull(timeline_->CurrentTimeInternal())
            ? NullValue()
            : CurrentTimeInternal();
    }
```
The KeyframeEffect

```csharp
Member<AnimationEffectReadOnly> content_;
Member<DocumentTimeline> timeline_;

if (content_) {
    double inherited_time = idle || IsNull(timeline_)->CurrentTimeInternal()
        ? NullValue()
        : CurrentTimeInternal();

    // Special case for end-exclusivity when playing backwards.
    if (inherited_time == 0 && playback_rate_ < 0)
        inherited_time = -1;
    content_->UpdateInheritedTime(inherited_time, reason);
}
```

- Cache misses
Updating time and values

- Jumping contexts
- Cache misses (data and instruction)
- Coupling between systems (animations and events)
Interpolate different **types** of values

- **Dynamic type erasure - data and instruction cache misses**
- **Requires testing combinations of concrete classes**
Apply the new value

- Coupling systems - Animations and Style solving
- Unclear lifetime - who "owns" the Element
- Guaranteed cache misses

Walks up the DOM tree!
SetNeedsStyleRecalc

Miss!
Miss!
Miss!
Miss!
Miss!
SetNeedsStyleRecalc
Recap

- We used more than 6 non-trivial classes
- Objects contain smart **pointers** to other objects
- Interpolation uses **abstract classes** to handle different property types
- CSS Animations directly **reach out** to other systems - coupling
  - Calling events
  - Setting the value in the DOM Element
  - How is the lifetime of Elements synchronized?
Let’s try data-oriented design
Back to the drawing board

- Animation data operations
  - Tick (Update) -> 99.9%
  - Add
  - Remove
  - Pause
  - ...

- Animation Tick Input
  - Animation definition
  - Time

- Animation Tick Output
  - Changed properties
  - New property values
  - Who owns the new values

- Design for many animations
The AnimationController

AnimationController

Active Animations
- AnimationState
- AnimationState
- AnimationState

Tick(time)

Inactive Animations
- AnimationState
- AnimationState

Animation Output
- Left: 50px
- Opacity: 0.2
- Left: 70px
- Right: 50px
- Top: 70px

Elements
- Element*
- Element*
- Element*
Go flat!

```c
struct AnimationStateCommon {
    AnimationId Id;
    mono_clock::time_point::seconds StartTime;
    mono_clock::time_point::seconds PauseTime;
    Optional<mono_clock::time_point::seconds> ScheduledPauseTime;
    float IterationsPassed = 0.f;
    float PlaybackRate = 1.0f;
    mono_clock::duration::seconds Duration;
    mono_clock::duration::seconds Delay;
    AnimationIterationCount::Value Iterations;
    AnimationFillMode::Type FillMode;
    AnimationDirection::Type Direction;
    AnimationTimingFunction::Timing Timing;
    AnimationPlayState::Type PlayState;
};
```
Two approaches to keep the definition

Shared pointers & Copy-on-write

Animation State
Animation State
Animation State
Animation State
Animation State

Animation Definition
Animation Definition
Animation Definition
Animation Definition
Animation Definition

Multiplicated data - no sharing

Animation State
Animation State
Animation State
Animation State
Animation State

Animation Definition
Animation Definition
Animation Definition
Animation Definition
Animation Definition
Avoid type erasure

```cpp
template<typename T>
struct AnimationStateProperty : public AnimationState
{
    AnimatedDefinitionFrames<T> Keyframes;
};
```

Per-property vector for every Animation type!

```cpp
// -- Auto-generated -- /
CSSVector<AnimationStateProperty<BorderWidth>> m_BorderTopWidthActiveAnimState;
CSSVector<AnimationStateProperty<BorderWidth>> m_BorderLeftWidthActiveAnimState;
// ...
CSSVector<AnimationStateProperty<ZIndex>> m_ZIndexActiveAnimState
```

Note: We know every needed type at compile time, the vector declarations are auto-generated
Memory layout comparison

Animation State<BorderLeft>

Animation State<BorderLeft>

Animation State<BorderLeft>

Animation State<Opacity>

Animation State<Opacity>

Animation State<Opacity>

Animation State<Transform>

Animation State<Transform>

Animation State<Transform>

Heap

Heap
Ticking animations

- Iterate over all vectors

```
AnimationState<BorderLeft> AnimationState<BorderLeft> AnimationState<BorderLeft> AnimationState<BorderLeft>
```

```
AnimationState<Opacity> AnimationState<Opacity> AnimationState<Opacity>
```

```
AnimationState<Transform> AnimationState<Transform>
```

- Use **implementation-level** templates (in the .cpp file)

```cpp
template<css::PropertyTypes PropType>
AnimationRunningState TickAnimation(mono_clock::time_point::seconds now, AnimationStateProperty<
typename css::PropertyValue<
PropType>::type_t>& state)
{
```

32
Avoiding branches

- Keep lists per-boolean “flag”
  - Similar to database tables - sometimes called that way in DoD literature
- **Separate** **Active** and **Inactive** animations
  - Active are currently running
    - But can be stopped from API
  - Inactive are finished
    - But can start from API
- Avoid “if isActive)”!
- Tough to do for every bool, prioritize according to branch predictor chance
A little bit of code

template<css::PropertyTypes PropType>
AnimationRunningState::TickAnimation(mono_clock::time_point::seconds now,
AnimationStateProperty<
typename css::PropertyValue<
PropType> type_t>& state)
{
    using Type = typename css::PropertyValue<
    PropType>::type_t;

    AnimationRunningState transition;
    const auto t = CalculateAnimationPoint(now, state, transition);
    assert(!std::isnan(t));

    const typename AnimatedDefinitionFrames<
    Type>::Frame* from = nullptr;
    const typename AnimatedDefinitionFrames<
    Type>::Frame* to = nullptr;

    size_t firstFrameIndex;
    auto interpolator = DetermineKeyFrameInterval(t, state, from, to, firstFrameIndex);

    interpolator = ApplyEase(interpolator, state.Timing, state.Duration);

    const auto newValue = GetInterpolatedValue(state,
    firstFrameIndex,
    interpolator,
    from->Value,
    to->Value);

    state.Output->template SetValue<
    Type, PropType>(newValue);

    return transition;
}
Adding an API - Controlling Animations

- The API requires having an "Animation" object
  - play()
  - pause()
  - playbackRate()
- But we have no "Animation" object?!
- An Animation is simply a **handle** to a bunch of data!
- **AnimationId** (unsigned int) wrapped in a JS-accessible C++ object
Implementing the DOM API cont.

- AnimationController implements all the data modifications
- “Animation” uses the AnimationId as a simple handle

```cpp
void PauseAnimation(AnimationId animationId);
void PlayAnimation(AnimationId animationId);
void PlayFromTo(AnimationId animationId,
                mono_clock::duration::milliseconds playTime,
                mono_clock::duration::milliseconds pauseTime);
void SetAnimationSeekTime(AnimationId animationId, mono_clock::duration::milliseconds seekTime);
mono_clock::duration::milliseconds GetAnimationSeekTime(AnimationId animationId);
void SetAnimationPlaybackRate(AnimationId animationId, float playbackRate);
float GetAnimationPlaybackRate(AnimationId animationId);
void ReverseAnimation(AnimationId animationId);
```
## Analogous concepts comparison

<table>
<thead>
<tr>
<th>OOP (chromium)</th>
<th>DoD (Hummingbird)</th>
</tr>
</thead>
<tbody>
<tr>
<td>blink::Animation inheriting 6 classes</td>
<td>AnimationState templated struct</td>
</tr>
<tr>
<td>References to Keyframe data</td>
<td>Read-only duplicates of the Keyframe data</td>
</tr>
<tr>
<td>List of dynamically allocated Interpolations</td>
<td>Vectors per-property</td>
</tr>
<tr>
<td>Boolean flags for “activeness”</td>
<td>Different tables (vectors) according to flag</td>
</tr>
<tr>
<td>Inherit blink::ActiveScriptWrappable</td>
<td>Animation interface with Id handle</td>
</tr>
<tr>
<td>Output new property value to Element</td>
<td>Output to tables of new values</td>
</tr>
<tr>
<td>Mark Element hierarchy (DOM sub-trees)</td>
<td>List of modified Elements</td>
</tr>
</tbody>
</table>
Key points

- **Keep data flat**
  - Maximise cache usage
  - No RTTI
  - Amortized dynamic allocations
  - Some read-only duplication improves performance and readability

- **Existence-based predication**
  - Reduce branching
  - Apply the same operation on a whole table

- **Id-based handles**
  - No pointers
  - Allow us to rearrange internal memory

- **Table-based output**
  - No external dependencies
  - Easy to reason about the flow
What about something more complex - style solving?
Style solving

- Doesn’t map well to the “by the book” data-oriented design idea
- Traverse a tree of potentially large objects
- Complex rules to apply for each style type

Node

**Specificity(0 0 0 1)**
- left: 2em;
- opacity: 0.5;

**Specificity(0 0 1 1)**
- left: 10px;
- color: inherit;

**Animation**
- left: 50px;

Node

**Computed**
- left: 50px;
- opacity: 0.5;
- color: orange;
- ...
The DOM tree styling walk

- Styling of children can depend on parents due to inheritance of styles
- Classic top-down algorithm
  - If Node or its children have something changed - re-style
  - Walk children
  - Node & Elements have different rules. Nodes (Text usually) take directly the style of their parent
Issues with top-down algorithm

- Requires marking Node/Element parents when their children have changed styles
  - Saw this in chromium
- Requires walking a tree of heap-allocated large objects
  - Nodes and Elements have interface requirements and usually have a lot of data
- Nodes and Elements (inherit Node) implement different styling logic
- There are hundreds of styles
  - We would like to compute only what is changed
Data-oriented design approach

● Input
  ○ List of Nodes with potentially changed styling
  ○ Bitset for each Node of potentially changed styles

● Split the algorithm in 3 phases
  ○ Gather children and sort by DOM level
    ■ We have to keep the order of elements - remember children can depend on parent style
    ■ Separate Element and Node objects
  ○ Compute styles on the sorted list of Elements
    ■ Nodes can be directly iterated at the end - they are always **leaves** in the tree
  ○ Compute final output
    ■ Shown/Hidden nodes
    ■ Nodes with new styles
    ■ etc.
Phase 1 - Gather children and sort

- **Input**
  - List of Nodes

- **Additional data needed**
  - IsElement
  - Children
  - DOM level

- **Output**
  - Sorted list of Elements
  - List of Nodes

```plaintext
for each Node in Input:
    Push Node in Queue
while !Queue.empty():
    if Node !IsElement(Node):
        Put in NodesOutput;
    else
        Put in ElementOutput;
    Push Children in Queue;
Sort ElementOutput By DOM Level;
```
Phase 2 - Compute styles for Elements and Nodes

- **Input**
  - List of Elements sorted by DOM Level
  - List of Nodes
- **Additional data needed**
  - Potentially changed styles
  - List of matched styles for each
  - Type classification of styles (transform, layout etc.)
- **Output**
  - Modified computed styles
  - Elements with changed style and type of change
  - Nodes with changed style and type of change
Phase 3 - Classify changes for next steps in pipeline

● Input
  ○ List of changed Nodes & Elements
  ○ Type of change class for each

● Additional data needed
  ○ None

● Output
  ○ Classified lists
    ■ Nodes/Elements with changed Layout styles
    ■ Nodes/Elements with changed Transform styles
    ■ Nodes/Elements shown/hidden
    ■ etc.
Each phase uses different data

- Different Input/Output
- Different **additional** needed data
- In classic OOP DOM all the data will be in Node/Element
  - With a bunch of stuff unused by our algorithm!
  - Low cache occupancy
- Idea -> **Split** the Node/Element in Components
  - A version of Entity-Component System (ECS)
  - We don’t need dynamically adding/removing components!
  - Maximise cache occupancy in each phase
# Nodes with Components

## OOP

**Node**
- Parent*
- Children[]
- DomLevel
- ChangedStyles*
- ComputedStyles*
- ID
- Classes
- MatchedStyles*
  ...

## DoD

**Node**

**Hierarchy**
- Parent*
- Children[]
- DomLevel

**Styling**
- ChangedStyles*
- ComputedStyles*

**Style Matching**
- ID
- Classes
- MatchedStyles*
  ...

Used in Phase 1
Used in Phase 2
Used in Style matching (not in this talk)
Analysis
Performance analysis

<table>
<thead>
<tr>
<th>Animation Tick</th>
<th>Average Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOP</td>
<td>6.833</td>
</tr>
<tr>
<td>DoD</td>
<td>1.116</td>
</tr>
</tbody>
</table>

DoD Animations are **6.12x** faster

* Data gathered on PC, Intel i7
Scalability

- **Issues multithreading OOP chromium Animations**
  - Collections getting modified during iteration
  - Event delegates
  - Marking Nodes for re-style

- **Solutions for the OOP case**
  - Carefully re-work each data dependency

- **Issues multithreading DoD Animations**
  - Moving AnimationStates to “inactive” (table modification from multiple threads)
  - Building list of modified Nodes (vector push_back across multiple threads)

- **Solutions in the DoD case**
  - Each task/job/thread keeps a private table of modified nodes & new inactive anims
  - Join merges the tables
  - Classic fork-join
Multithreaded animation system

Thread A
Tick Animations
[0..N/3)

Thread B
Tick Animations
[N/3..2N/3)

Thread C
Tick Animations
[2N/3..N)

Output A

Output B

Output C

Output
Testability analysis

● The OOP case
  ○ Needs mocking the main input - animation definitions
  ○ Needs mocking at least a dozen classes
  ○ Needs building a complete mock DOM tree - to test the “needs re-style from animation logic”
  ○ Combinatorial explosion of internal state and code-paths
  ○ Asserting correct state is difficult - multiple output points

● The DoD case
  ○ Needs mocking the input - animation definitions
  ○ Needs mocking a list of Nodes, complete DOM tree is not needed
  ○ AnimationController is self-contained
  ○ Asserting correct state is easy - walk over the output tables and check
Modifiability analysis

- **OOP**
  - Very tough to change base classes
    - Very hard to reason about the consequences
  - Data tends to “harden”
    - Hassle to move fields around becomes too big
    - Nonoptimal data layouts stick around
  - Shared object lifetime management issues
    - Hidden and often fragile order of destruction
  - Easy to do “quick” changes

- **DoD**
  - Change input/output -> requires change in System “before”/“after” in pipeline
  - Implementation changes - local
    - Can experiment with data layout
    - Handles mitigate potential lifetime issues
Downsides of DoD

● Correct data separation can be hard
  ○ Especially before you know the problem very well

● Existence-based predication is not always feasible (or easy)
  ○ Think adding a bool to a class VS moving data across arrays
  ○ Too many booleans is a symptom - think again about the problem

● “Quick” modifications can be tough
  ○ OOP allows to “just add” a member, accessor, call
  ○ More discipline is needed to keep the benefits of DoD

● You might have to unlearn a thing or two
  ○ The beginning is tough

● The language is not always your friend
When OOP?

- Sometimes we have no choice
  - Third-party libraries
  - IDL requirements
- Simple structs with simple methods are perfectly fine
- Polymorphism & Interfaces have to be kept under control
  - Client-facing APIs
  - Component high-level interface
  - IMO more convenient than C function pointer structs
- Remember - C++ has great facilities for static polymorphism
  - Can be done through templates
  - .. or simply include the right “impl” according to platform/build options
Object-oriented programming is not a silver bullet..

..neither is data-oriented design..

..use your best judgement, please.
References

- “Data-Oriented Design and C++”, Mike Acton, CppCon 2014
- “Pitfalls of Object Oriented Programming”, Tony Albrecht
- “Introduction to Data-Oriented Design”, Daniel Collin
- “Data-Oriented Design”, Richard Fabian
- “Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)”, Noel Llopis
- “OOP != classes, but may == DOD”, roathe.com
- “Data Oriented Design Resources”, Daniele Bartolini
- https://stoyannk.wordpress.com/