Practical Examples in Data Oriented Design

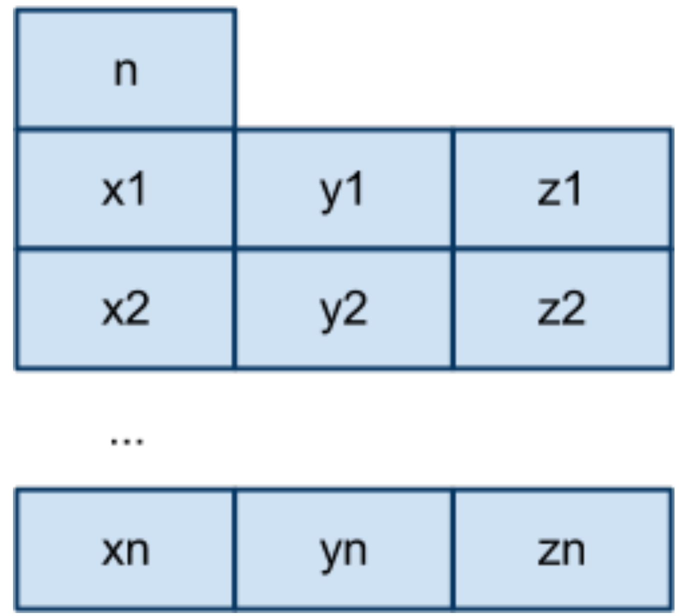
Niklas Frykholm, BitSquid

What is Data-Oriented Design?

Focus on **data**!

Collection<Vector3>

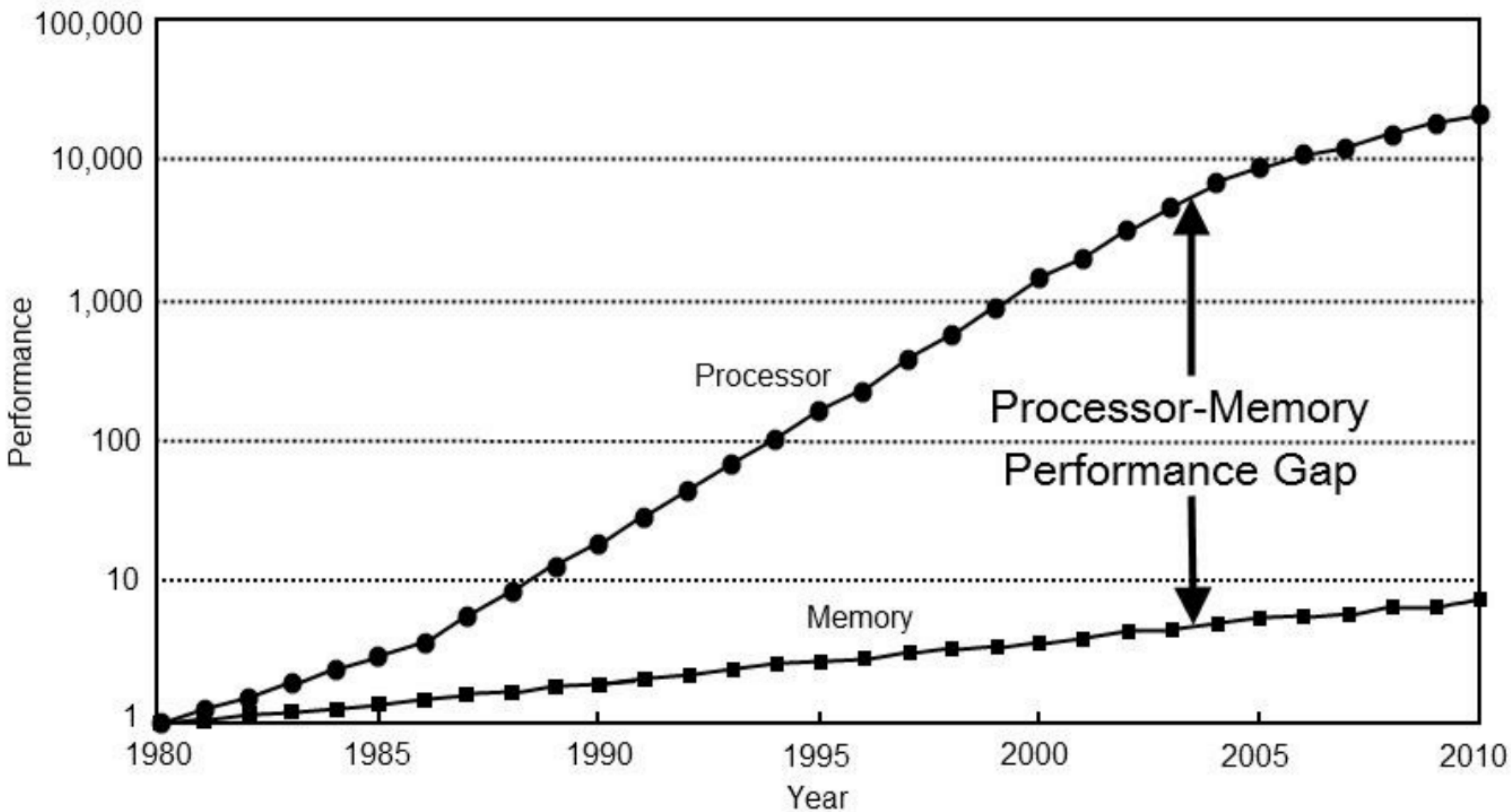
Add() Remove()



How is data represented, moved, shared and transformed?

Not: What can objects do? How do they interact?

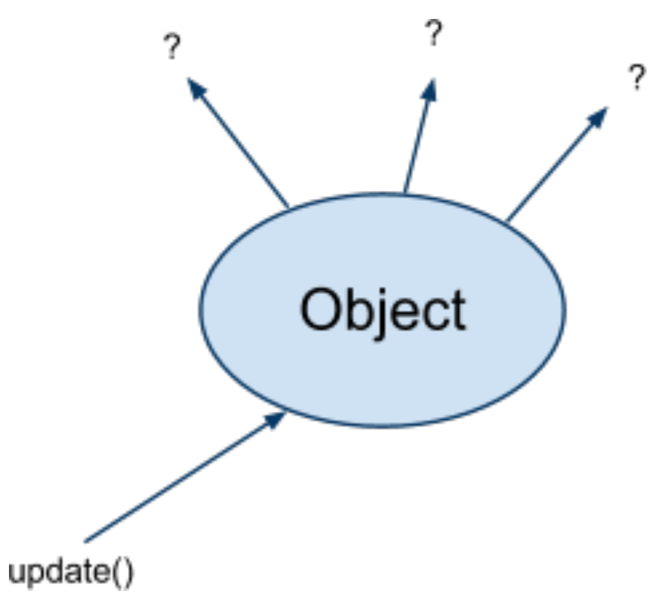
Why - Performance

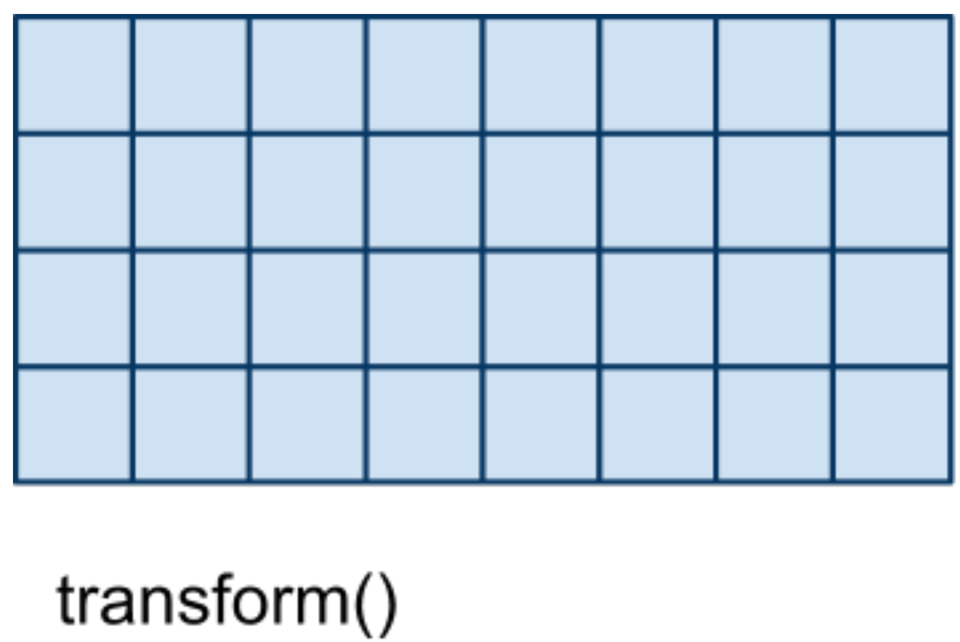


- you **cannot** be fast without knowing how data is touched
- virtual calls
- scattered individual objects

Why - Multithreading

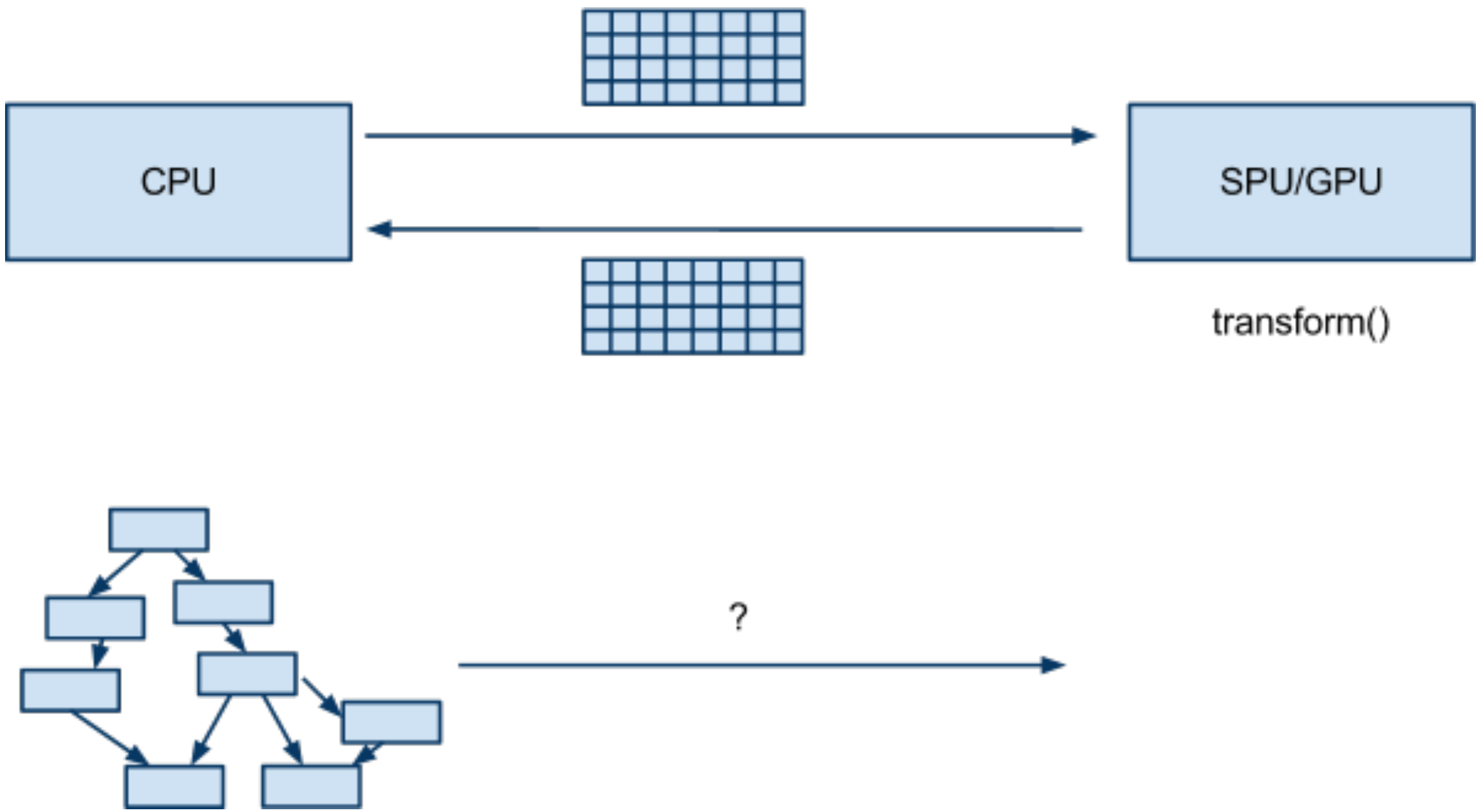
- You **cannot** multithread without knowing how data is touched





Why - Offload to co-processor

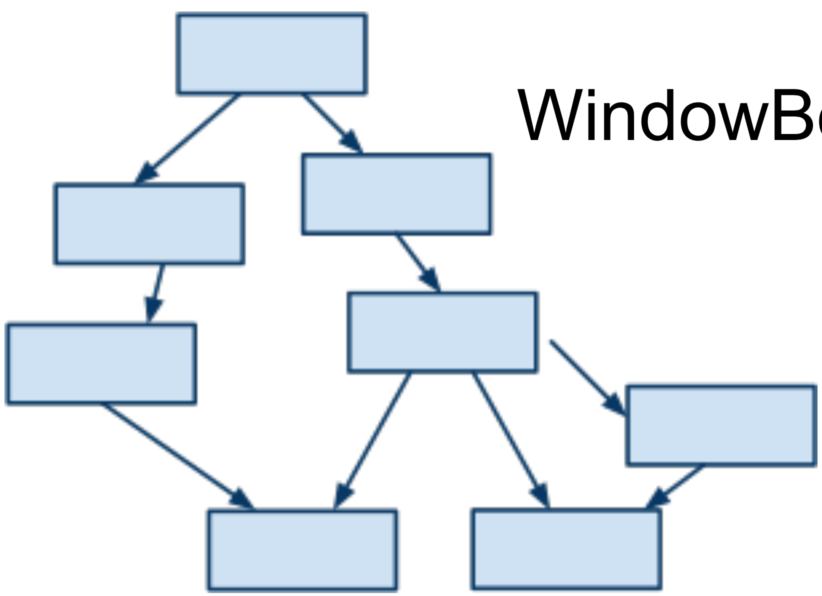
- You **cannot** offload without knowing what data to send



Why - Better design... sometimes

Data focus *can* lead to isolated, self-contained, interchangeable pieces of code and data

Object focus *can* lead to **FRAMEWORK HELL!**



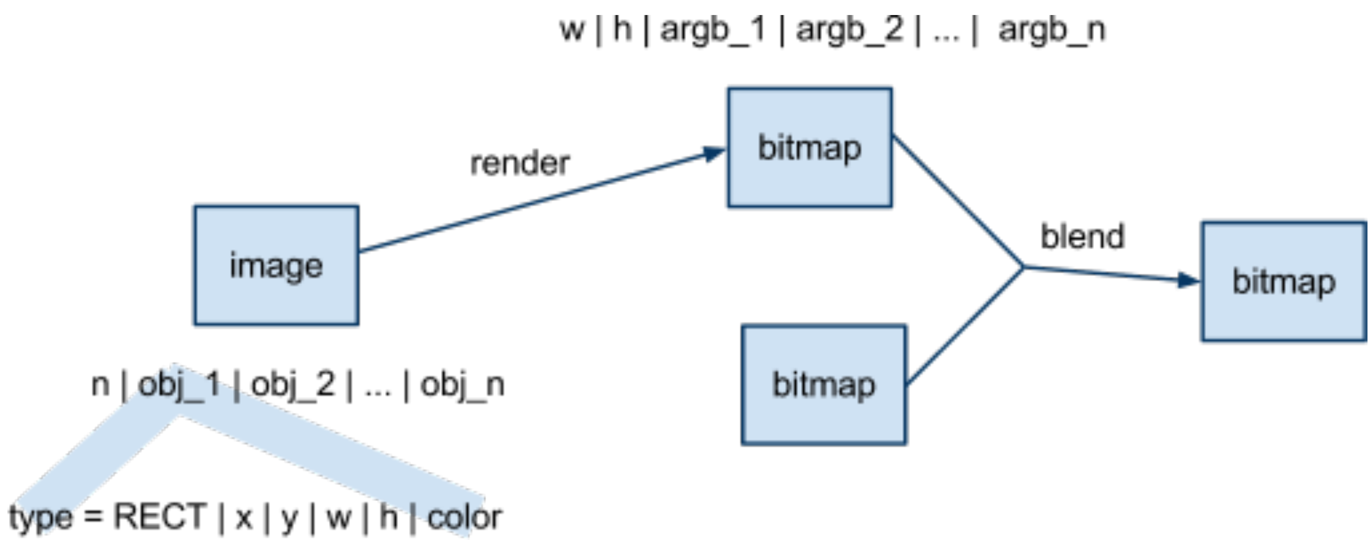
WindowBorderDecoratorAccessorIterator

Begun, the code war has...

Principles of Data-Oriented Design

- Isolate the tasks
  - Do many-at-once
- Find the data objects
- Design data based on access patterns

Never underestimate the power of a linear array!

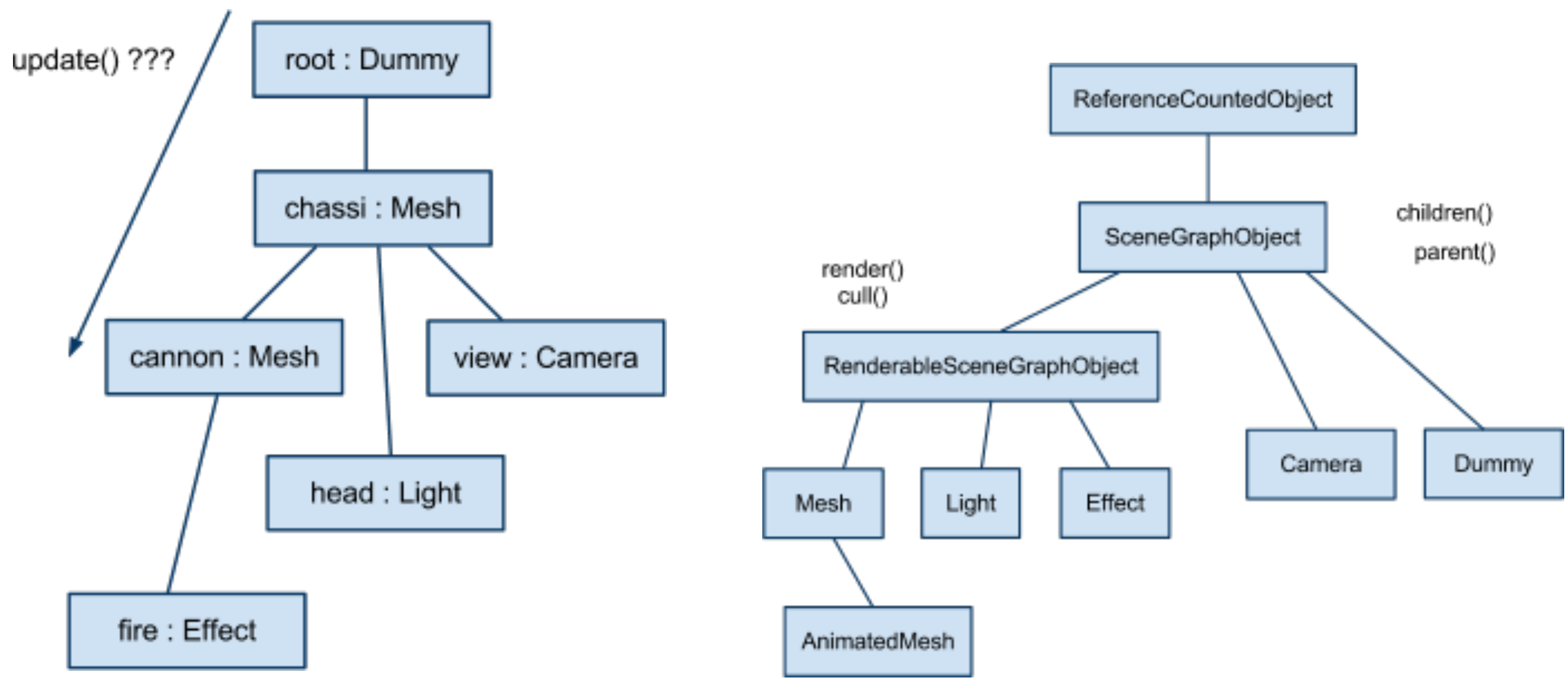


Practical examples

- Scene Graph
- Animation Player

Scene graph

Scary scary OOD:



Isolate tasks

~~Animate~~

~~Cull~~

~~Script callback~~

~~Update effect~~

Handled by their respective subsystems

Scene Graph:

Local-to-world transform for linked objects

Find data objects

Input:

- Local poses for *n* nodes
- Description of link hierarchy

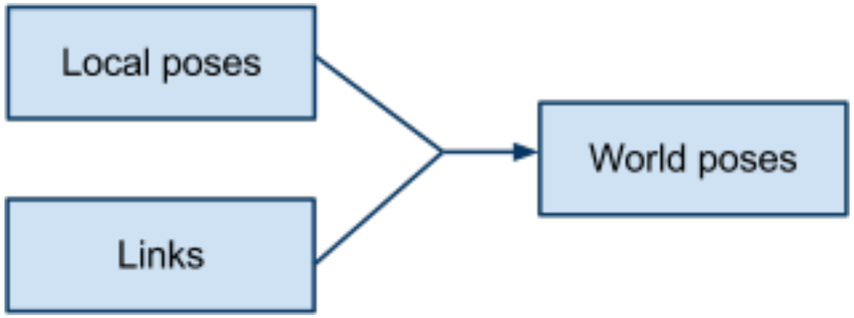
Output:

- World poses for *n* nodes

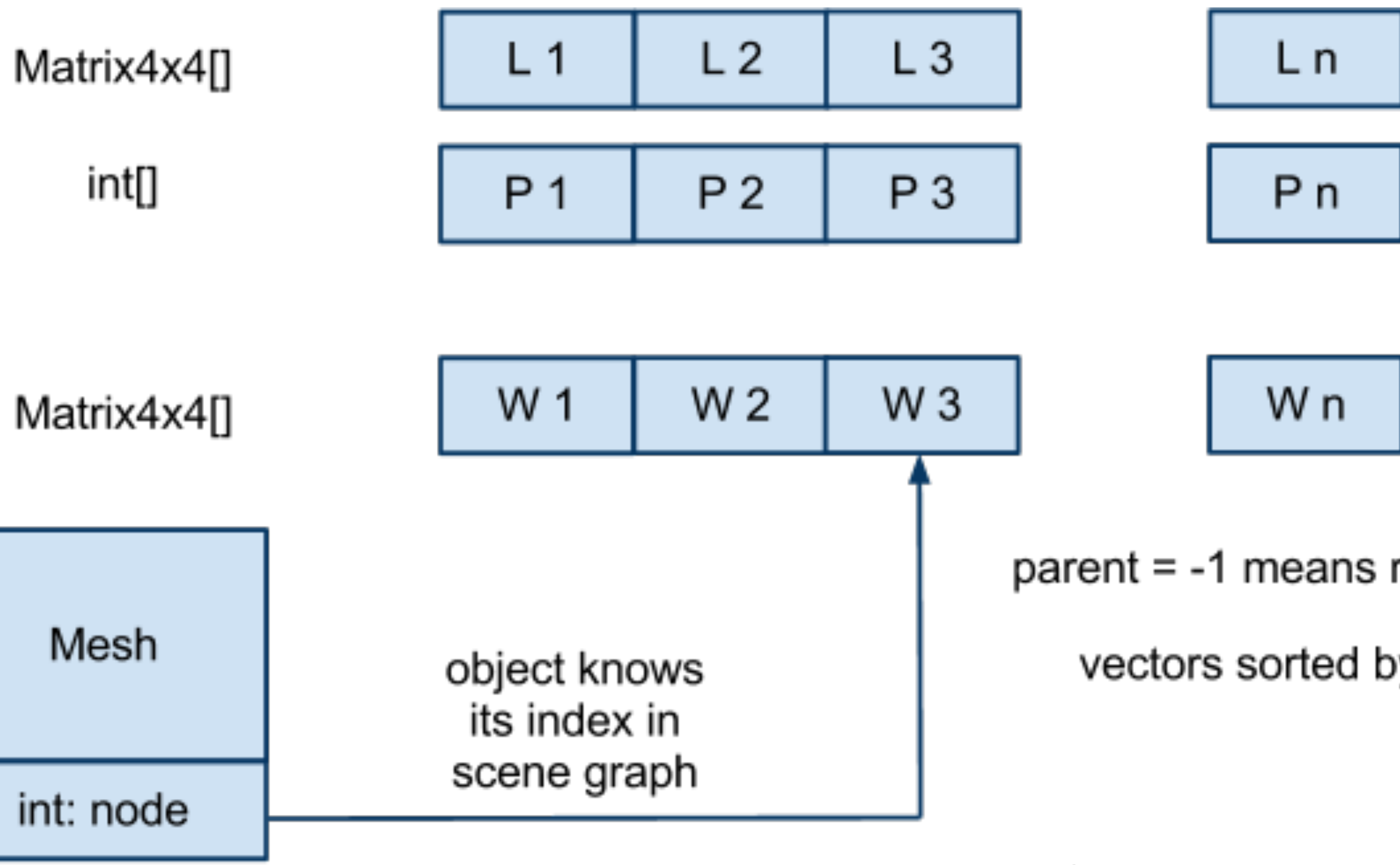
Transform:

$$W = L \qquad \textit{(root object)}$$

$$W = W_{parent} \times L$$



Data design



parent = -1 means root object

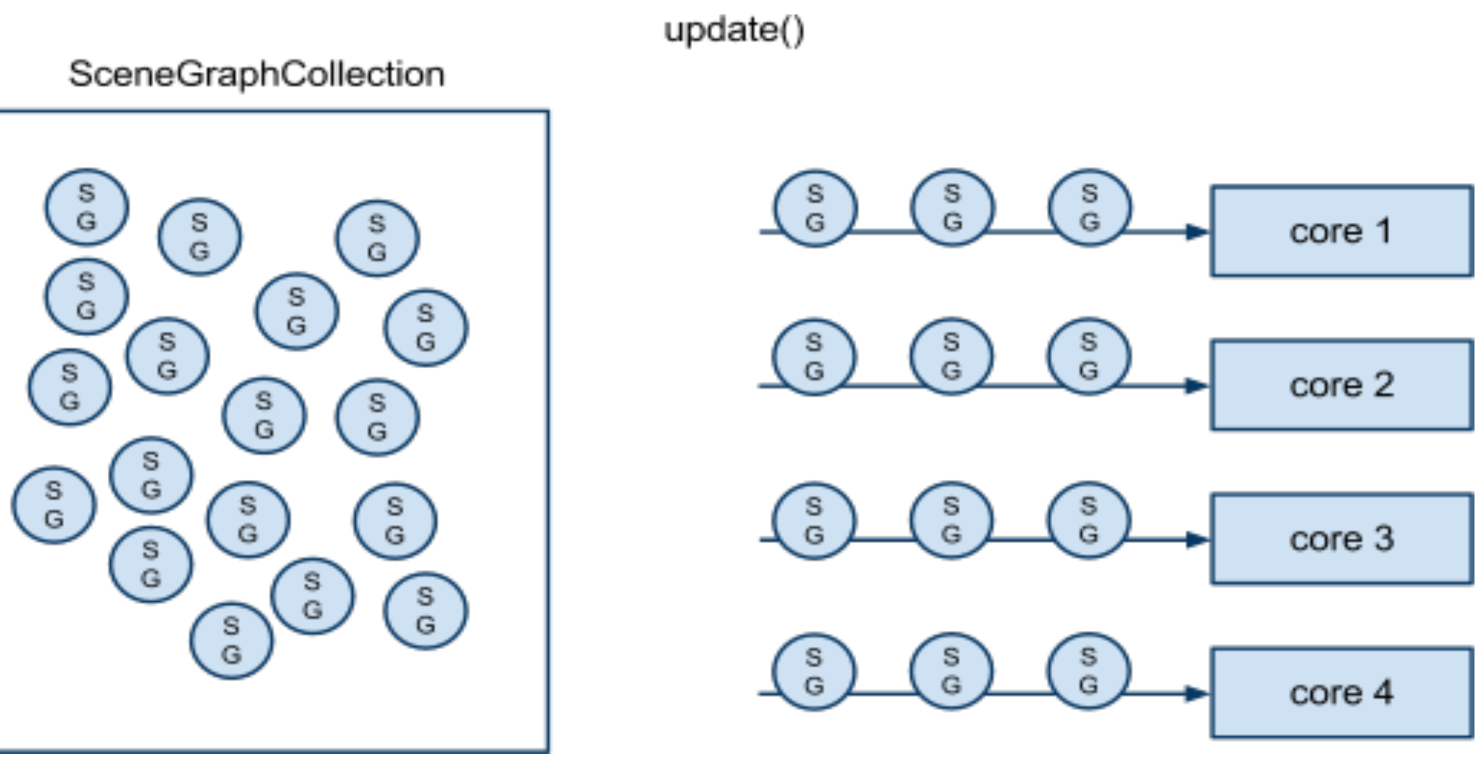
vectors sorted by DAG

Note: Multiple objects can share the same scene graph node!

```
for i = 1,n
    if P[i] = -1
        W[i] = L[i]
    else
        W[i] = W[P[i]] * L[i]
```

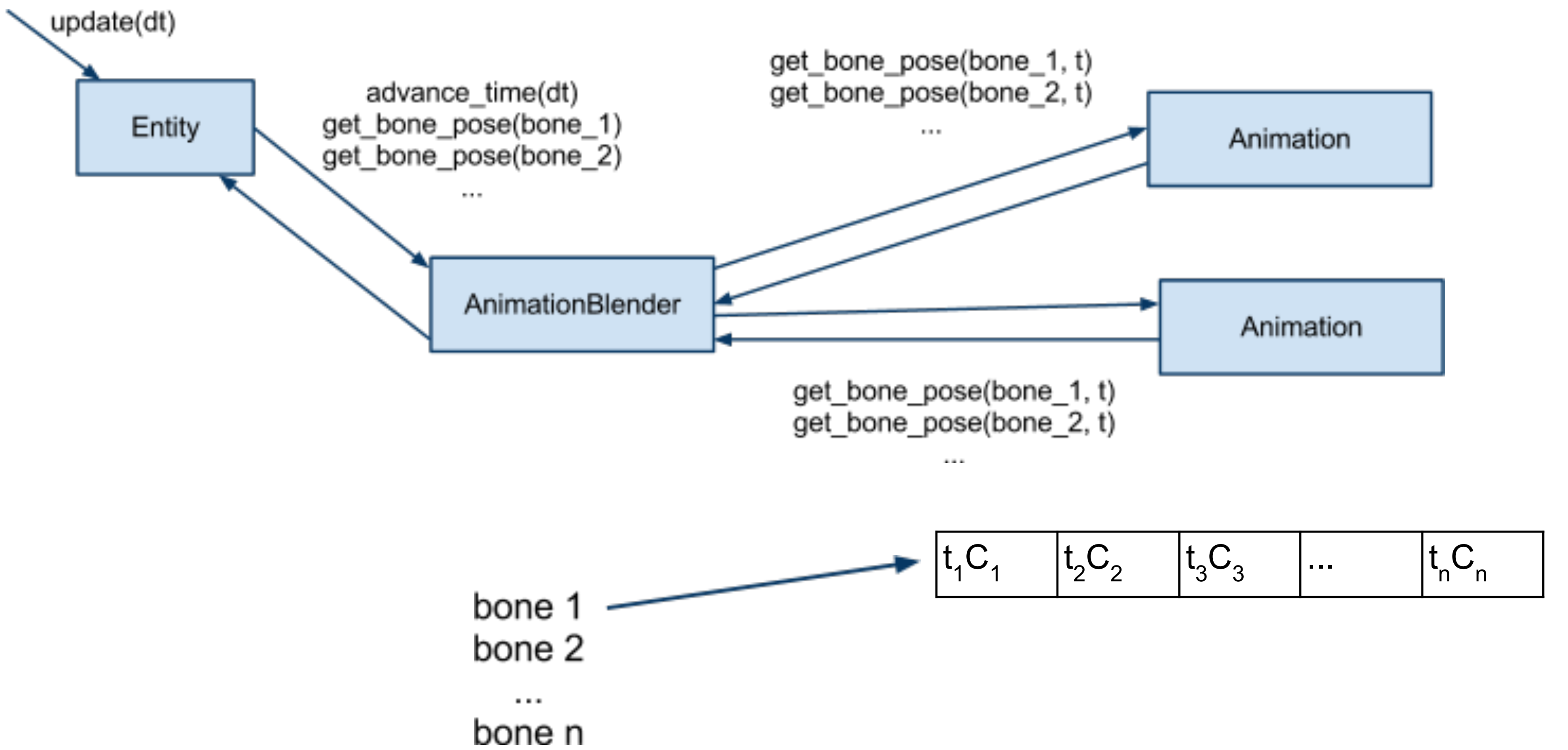
Results:

- No unnecessary cache misses
- Isolated, easy to performance measure
- Trivial to parallelize and/or offload



Next example: Animation player

The OOD, the bad and the ugly:



Isolate tasks

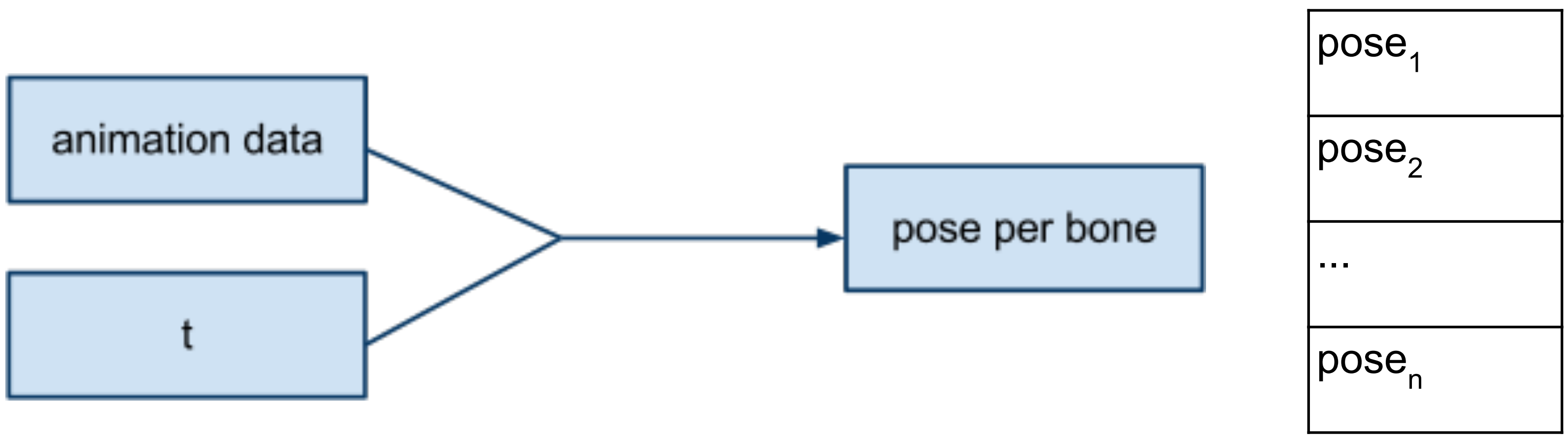
Given an animation:

Find the pose for every bone at time *t*

~~Blend animations~~

~~Other entity updates~~

Identify data objects

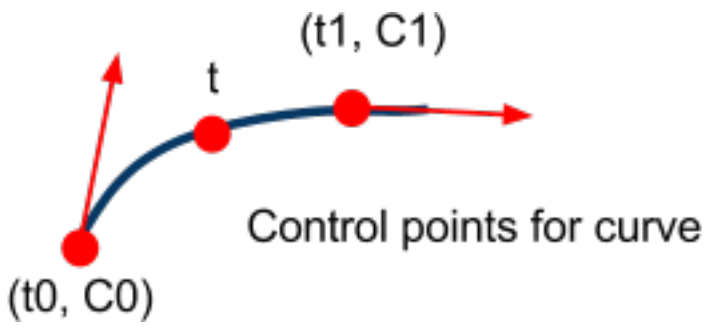


Access patterns

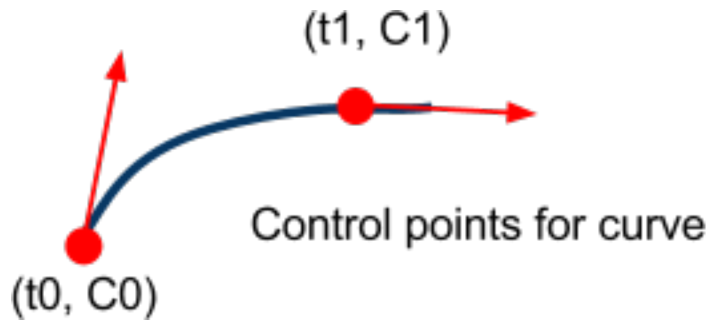
- Animation data is accessed in time order
    - Sort data by time
- To interpolate we need several curve points
    - Keep the active curve points in a separate structure
- Two separate operations
    - Update active curve points when time is advanced
    - Interpolate pose from active curve points

Active curve points:

1	t_0C_0	t_1C_1
2	t_0C_0	t_1C_1
...		
n	t_0C_0	t_1C_1



Data design



Animation data:

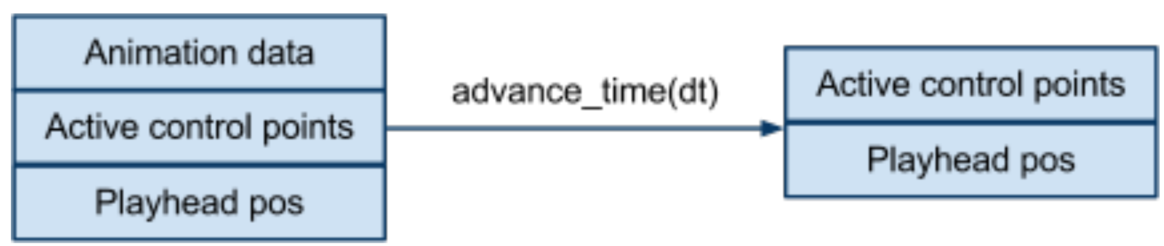
playhead →

bone_index	t	C
bone_index	t	C
...		

Active curve points:

1	t_0C_0	t_1C_1
2	t_0C_0	t_1C_1
...		
n	t_0C_0	t_1C_1

$$\max(t_0) < t < \min(t_1)$$



Original animation curve points

Head	0, A	10, B			
Arm	0, C	2, D	5, E	10, F	
Leg	0, G	3, H	7, I	9, J	10, K

Sorted by time needed

Head	0	A	0
Head	10	B	0
Arm	0	C	0
Arm	2	D	0
Leg	0	G	0
Leg	3	H	0
Arm	5	E	2
Leg	7	I	3
Arm	10	F	5
Leg	9	J	7
Leg	10	K	9

Active curve points at

$t = 4$

Head	0, A	10, B
Arm	2, D	5, E
Leg	3, H	7, I

Results:

- Huge improvement in data-access patterns
- Only the minimal required amount of animation data needs to be touched
- Stream compression of animations possible
- Resulting pose can be reused for different purposes
- Trivial to parallelize or offload computation

Conclusions

Benefits:

- Faster code
  - Cache-friendly
  - Multi-threading
  - Co-processing
- More modular
- Additional benefits
  - Networking
  - Serialization

Methods:

- Isolate tasks
  - Do many at once
- Find data transforms
- Optimize access patterns

- When in doubt, use a linear array!

Questions?

niklas.frykholm@bitsquid.se
www.bitsquid.se
Twitter: niklasfrykholm

www.stonegiant.se