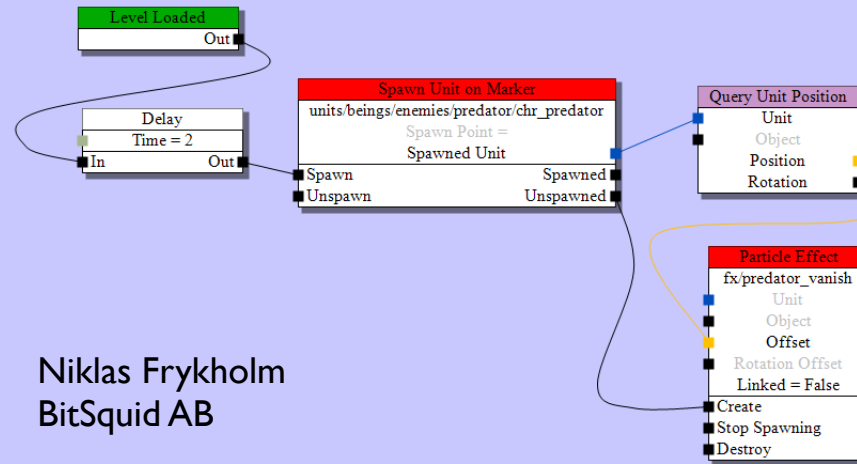


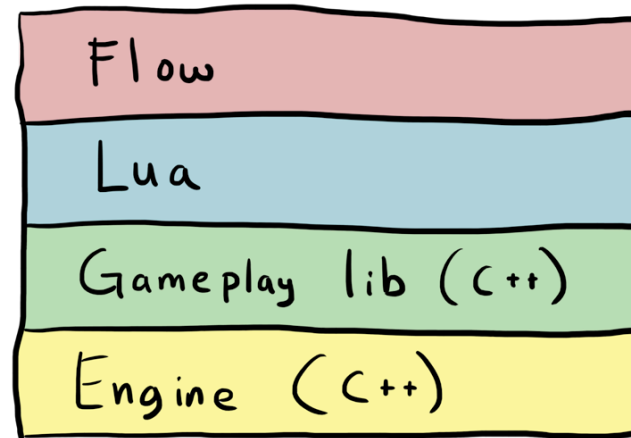
Flow

Data-Oriented Implementation of a Visual Scripting Language



Niklas Frykholm
BitSquid AB

Levels of programming



Visual scripting adds an additional level of programming

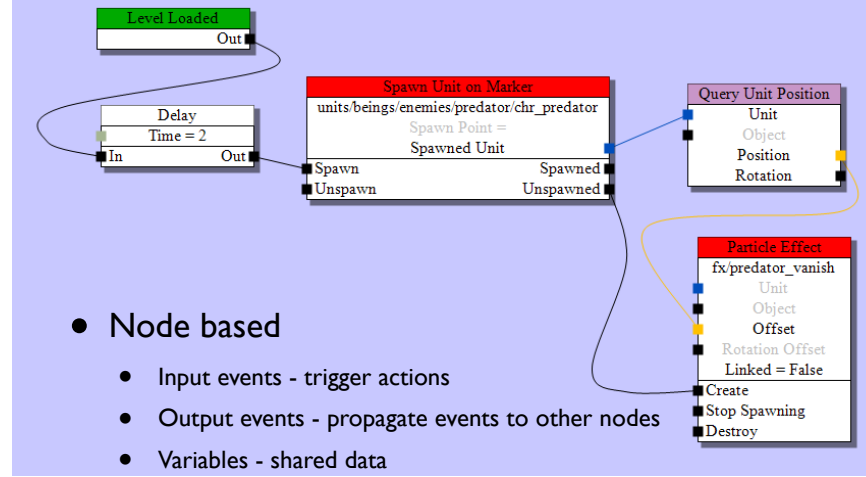
Visual scripting benefits

- To content creators
 - More power: integrate effects, sounds, etc
 - Immediately see results in-game: experiment
- To gameplay programmers
 - Less messy "special purpose" gameplay code
- Performance
 - Faster than Lua (no VM, no GC, typed)

An effect artist adding some gravel to a footstep. Can immediately see it in game, time it, maybe add a secondary effect, etc. This gives faster iterations which is the key to higher quality.

Won't need to write gameplay code for one-off scripts, such as: when you step on this plate, this door opens and five seconds later a zombie comes out of it, etc.

How Flow works



- Node based
 - Input events - trigger actions
 - Output events - propagate events to other nodes
 - Variables - shared data
- Inert - pay for what you use

Based on nodes... every node performs some task.

Input events make the node perform some kind of action.

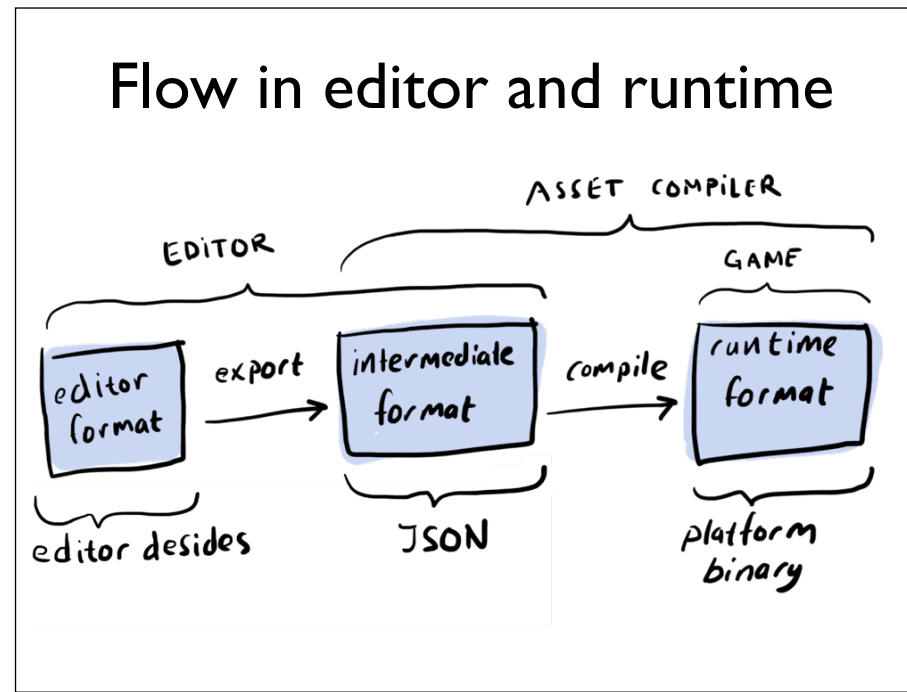
Output events are raised by a node when it knows something has happened. Links connect input events to output events.

Variable links allow nodes to share data.

There is no constant polling or data feeding in the system. The system is inert. If no external events are triggered the system does nothing. You only pay for what you use.

(Query nodes: A hidden event is used to tell the query node to update its output data when it is needed.)

Running the system: An external event triggers an output event in some node. The connected input events fire. As a result output events in those nodes may fire, further propagating the impulse.



A quick picture explaining how resources work in the engine.

The game works with a powerful high-performance, platform specific binary format.

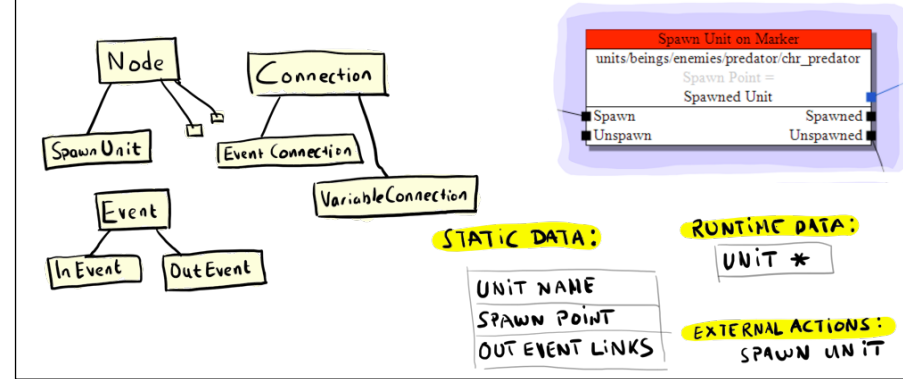
The editor exports to a generic, flexible, text based format.

An asset compiler compiles from the generic to the platform specific format.

This is how all resources work in the BitSquid engine.

Data-oriented design

- Instead of thinking about classes
 - Think about data layouts and transforms
 - Actions on real-world items (bits and bytes) not relations between abstract concepts (classes and objects)



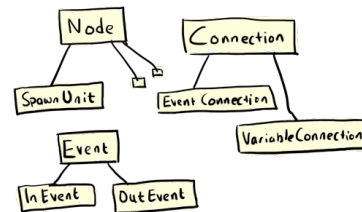
What are the data we need to perform what we want to do. Input data, output data? Static data, dynamic data? How do we best lay out the data in memory?

Once those questions are answered, writing the code that "does the stuff" is simple.

OOD is "idealistic". DoD is "realistic". OOD is concerned with models, DoD with data.

Motivation for DoD

- Performance
 - Memory is slower than CPU, think about caches
 - Focus on what the computer *actually does*
- "Objects" do not always give a good design
 - Does EventConnection : Connection make sense?
 - Unnecessary abstractions
- Less coupling
- More freedom



On SPUs and maybe future hardware we don't have "automatic cache"... we must load the caches ourselves. This means that a bad memory layout will also mean a lot of extra work.

Does there even have to be a class for an EventConnection... or is that just a pointer... Do EventConnection and VariableConnection really have something in common?

With the abstractions you don't know what is actually happening to the data. You don't know how you can multithread, copy objects, etc. Code that promises "I do this on this data" is easier to integrate, multithread, etc... than code that works on an abstract model "ball of objects".

Using 3rd party libraries is usually easier the less OO they are.

”More freedom”

- Say you want to store data about a network game
 - String keys and values
- One possible implementation:
`name\0Fight!\0map\0canyon\0players\020\0\0`
- Not OOD because ”there are no objects”
 - But this *might* be a good solution (depending on use)
 - DoD gives a wider design space

This is not necessarily absolutely the best solution, but it is one possibility and it is beyond the reach of OOD.

Of course you could implement this solution in C++ and wrap it in a class, but that doesn't make it OOD. You are still doing DoD because you are focusing on data storage (real world bits and bytes) not OOD (relations between abstract objects).

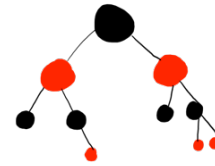
Static and dynamic data

- Very different requirements
 - Dynamic data must change, grow and shrink
 - Dynamic data must (often) allocate memory
 - Dynamic data must be thread-safe
- Standard data structures are made for dynamic data
 - `std::vector`, `std::map`, `std::string`
- *Most game data is static*
 - Textures, animations, vertex data, etc
- **DO NOT USE STANDARD DATA STRUCTURES FOR STATIC DATA!**

Think about what you need to do with the data and store it appropriately.

Dynamic/static example

- `std::map` (red-black tree)
 - Lots of pointers, cache unfriendly
 - To make fast *modification* possible
- Sorted array
 - Find item with binary search
 - Simple, cache friendly, single memory allocation



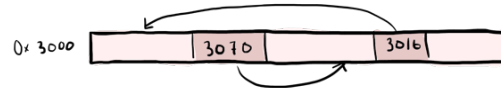
example:



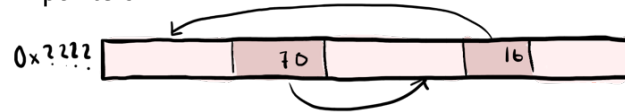
You could do other things as well... round up the size to a power of two and store the three depth-first. Whatever you do it will be better than a `std::map`.

Data "blobs"

- Since static data doesn't need to grow we can always store it in a single memory block



- Idea: make the data relocatable by using offsets instead of pointers

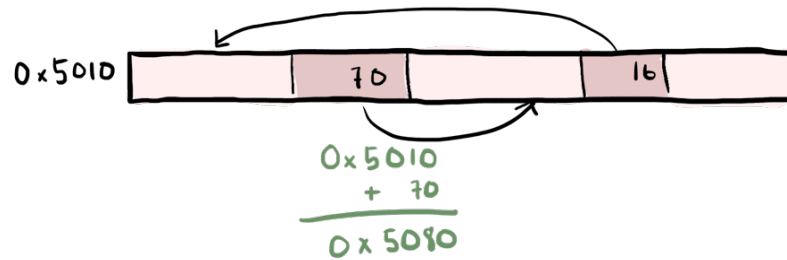


- Now we can move the data wherever we want
 - And read it directly of disk
- I call this "The Blob"

A big system is made up of many small pieces. We can just add them one after another and load all the static data for a big system as a single blob. I.e. we can store an entire Flow graph as a single blob.

Blob disadvantages

- Slightly more inconvenient
 - You must compute pointers from offsets




Of course you can write a function that does this.

Note: You can use relative offsets from the location of the pointer... or absolute offsets from the start of the blob. Doesn't really matter that much, but if you use the start of the blob, you must pass that around of course.

Blob advantages

- **Automatically cache friendly**
 - All data allocated at the same place
- **Automatically memory system friendly**
 - Few large allocations are better than many small
 - All allocation sizes known up front
 - No fragmentation during data load
 - Can be moved for defragmentation
- **Automatically loads fast**
 - Copy data directly from disk to memory
 - No pointer patching or other data fixup

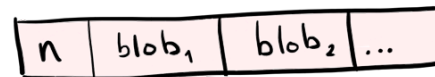
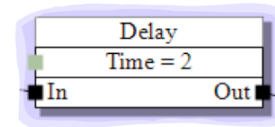
Blob advantages 2

- Less work
 - Don't have to think about serialization format
 - No need to write serializer/deserializer
- Data can be copied
 - To SPU for off-core processing
 - For instancing
- I  the blob
 - And you should too!

Using blobs for Flow

- struct for each node's data

```
struct DelayNode {  
    float time;  
};
```

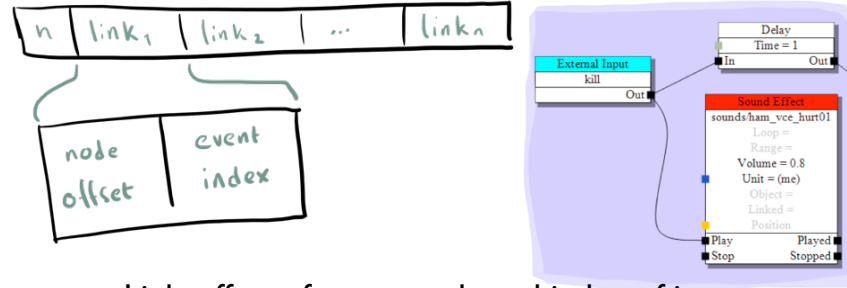


Make a big blob by concatenating the blob for every node.

Every node has a type indicator, node specific data, and links for its out events.

Storing the event links

- For each out event, store linked events

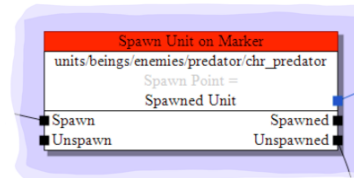
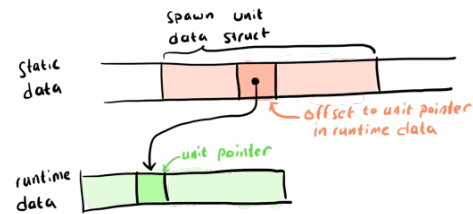


- Link: offset of target node and index of in event
 - Can loop over links to trigger an impulse

ADD NODE IMAGE

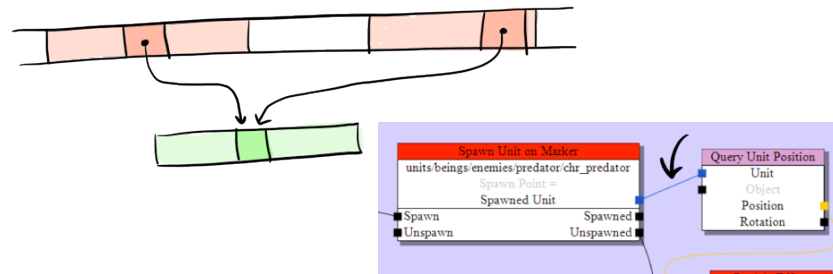
Runtime data

- Each node needs to store runtime data
 - Id of instances
 - Data on wires
- Idea: Runtime data is non-const blob
 - During asset compile nodes reserve memory
 - Get an offset into the runtime data



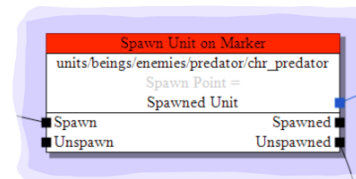
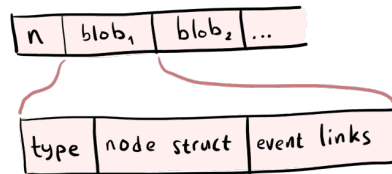
Data connections

- Represent shared data
- Implementation: Share runtime data
 - Two offsets pointing to the same location



Running a node

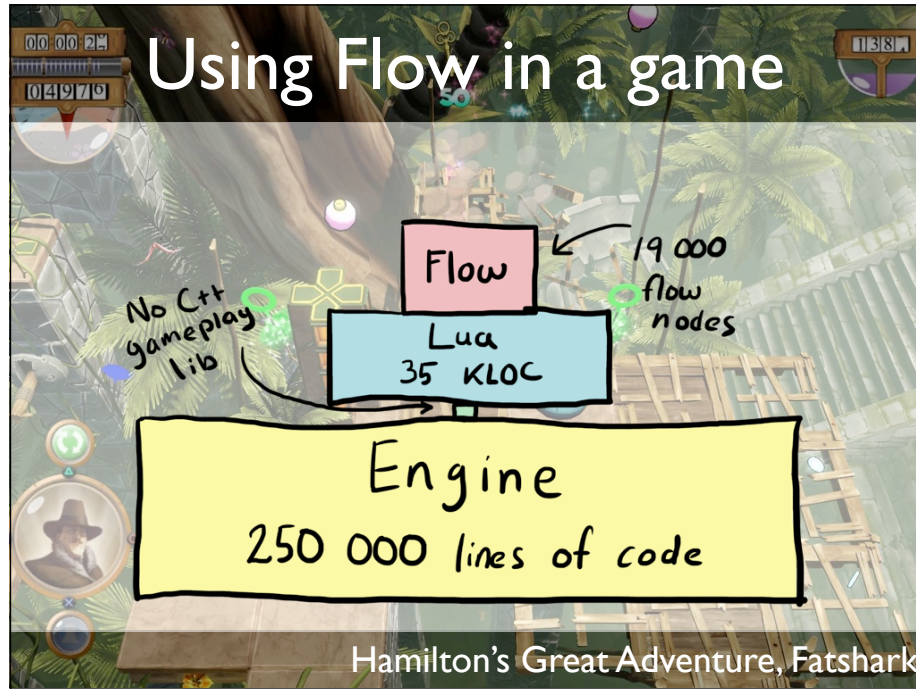
- Node actions implemented by functions
 - `void spawn_unit(const SpawnUnit *u, int event, char *runtime_data)`
- Lookup function pointer based on type
 - `node_functions[node_type](blob, event, runtime_data)`
 - Function table acts as vtable





Hamilton is a puzzle game made by Fatshark. Will soon be released.

Using Flow in a game



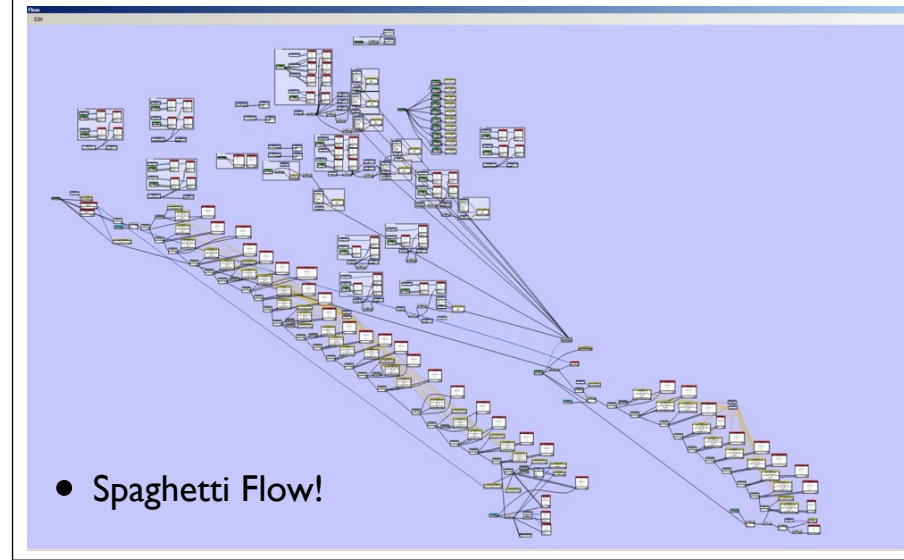
Hamilton's Great Adventure, Fatshark

The Good



- Able to do more and work faster
 - Easier than traditional scripting or "mission XML files"
 - Add quality: delays, randomization, multiple effects, polish
 - Test "crazy ideas" without code support
 - Everyone can experiment with "behaviors" and "rules"

The Bad & the Ugly



Spaghetti Flow became an issue. But only because flow was so useful that it was used a lot. This is actually a simple cutscene system implemented in flow. A similar work implemented as a combination of Lua and configuration JSON files would have been more work.

Q&A

- Questions?
 - Email: niklas.frykholm@bitsquid.se
 - Twitter: [niklasfrykholm](#)

