SCRIPTING PARTICLES

Getting Native Speed from a Virtual Machine

Niklas Frykholm

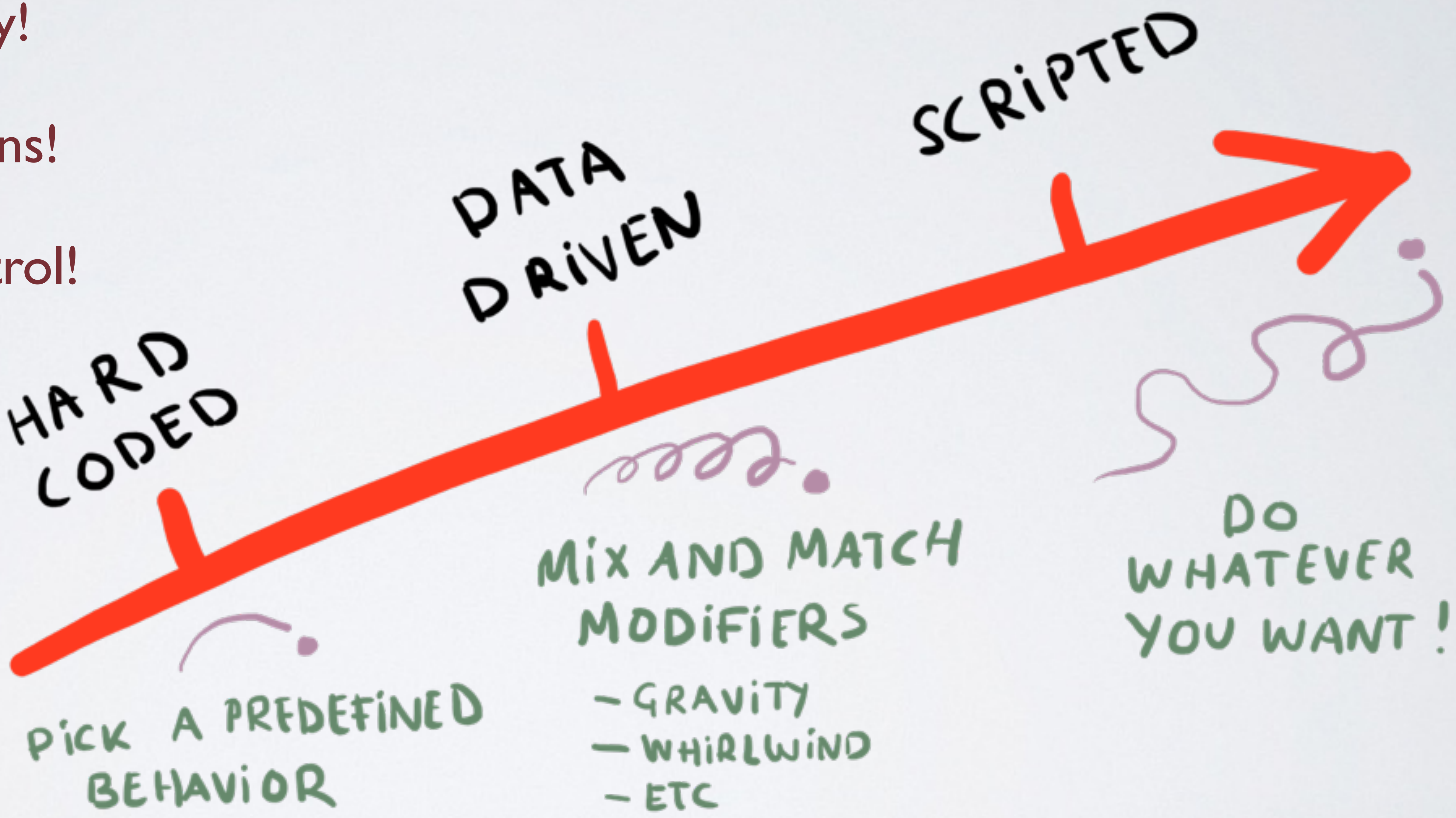
System Architect, Bitsquid

GAME DEVELOPERS CONFERENCE®

SAN FRANCISCO, CA
MARCH 17–21, 2014
EXPO DATES: MARCH 19–21
2014

The Case For Dynamic Content

- More flexibility!
- Faster iterations!
- Artists in control!



Flexibility vs Performance

- Scripting is great, but too slow for high performance tasks
  - Even with JIT
- Some high performance areas that would benefit from increased flexibility
  - Particle simulation
  - Wind simulation (and other vector field effects)
  - Sound processing
  - ...
- How can we make scripting work for them?

What We Want:
Fully Scripted FX with Near Native Performance
(programmer art)

EXISTING SOLUTIONS

- Stack of C “modifiers” or “filters”

  Good performance, limited flexibility

  Must get C programmers to add new filters

  Bad for generic & reusable engine

Artist selects filters and parameters in tool

gravity(0,0,-9.82), whirlwind(0, 0, 5)

Loop with switch statement in C code

Apply one filter at a time

- Runtime code compile

  Promising but tricky to get right

  Need compilers for all platforms (server?)

  Need runtime linking on all platforms (iOS)

  Must be converted to static code for “final” release

Artist creates effect in tool

Tool generates C code for running the effect

C code gets compiled for target platforms

Runtime linked with running executable

Why Are Script Interpreters Slow?

Virtual machine

Decode instruction
Jump to opcode
Execute instruction

Decode instruction
Jump to opcode
Execute instruction

Decode instruction
Jump to opcode
Execute instruction

...

These parts are identical: same machine instruction

addps xmm0, xmm1

This part is the overhead of using bytecode instead of native

Native

Execute instruction

Execute instruction

Execute instruction

Execute instruction

Execute instruction

...

DATA WIDE VIRTUAL MACHINE

- Perform each instruction on MANY data items
- Cost of decoding & branching is amortized
- Byte code just as fast as native?

  Can't keep data in registers

  More loads & stores

  Touches more cache memory

```
movaps xmm0, xmmword ptr [edx]
addps xmm0, xmmword ptr [ecx]
movaps xmmword ptr [eax], xmm0
```

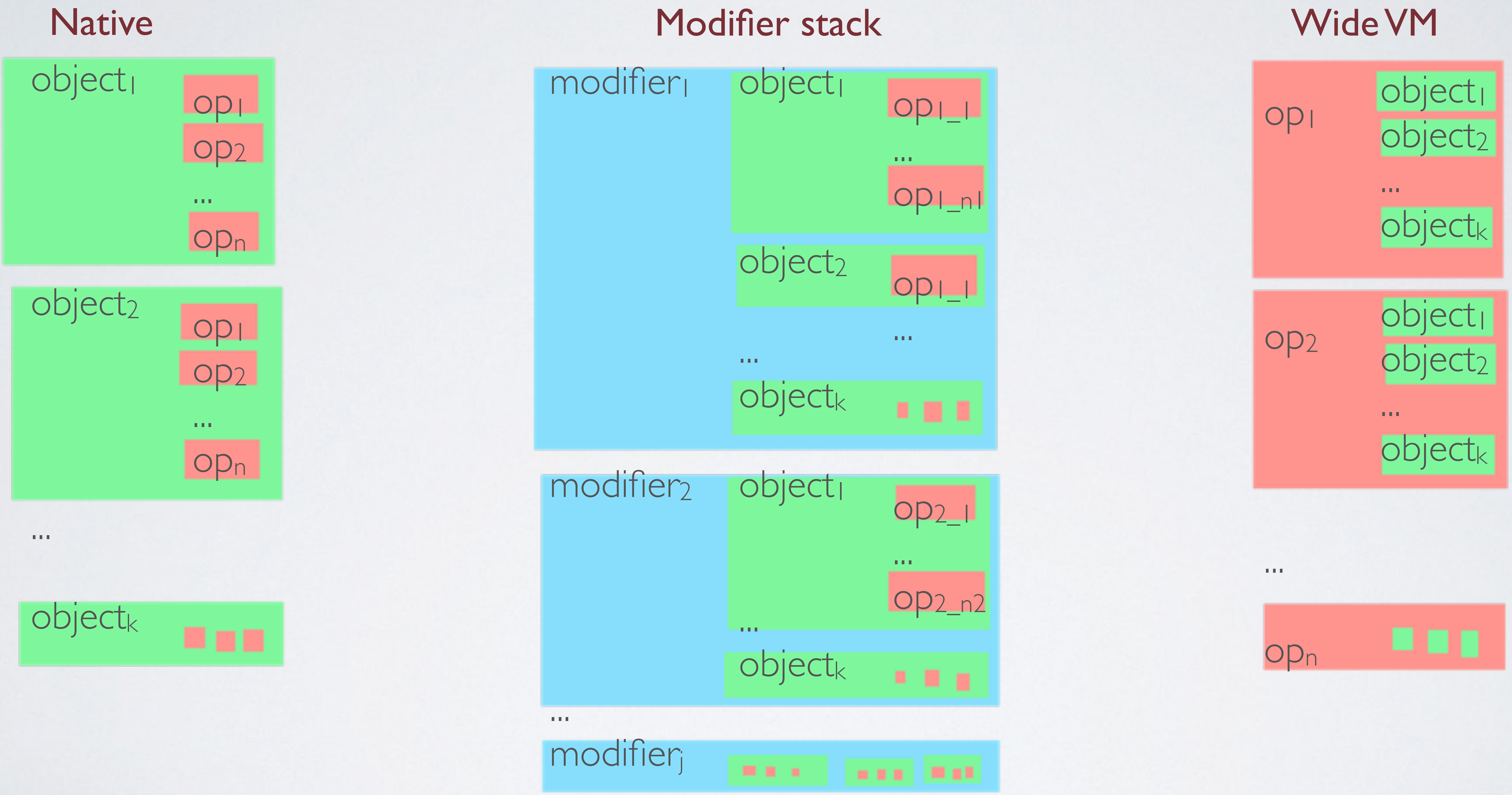
Virtual machine

```
Decode instruction
Jump to opcode
    Execute instruction
    Execute instruction
    Execute instruction
    ...

Decode instruction
Jump to opcode
    Execute instruction
    Execute instruction
    Execute instruction
    ...
...
```

Native

```
Execute instruction

Execute instruction

Execute instruction

Execute instruction

Execute instruction

...
```

Loop Orders



Native
object1
op1
op2
...
opn
object2
op1
op2
...
opn
...
objectk
Modifier stack
modifier1
object1
op1_1
...
op1_n1
object2
op1_1
...
...
objectk
modifier2
object1
op2_1
...
op2_n2
...
objectk
...
modifierj
Wide VM
op1
object1
object2
...
objectk
op2
object1
object2
...
objectk
...
opn

How it Works: Data-Wide Interpreter

- Built on top of Vector4 intrinsic abstraction

- Input/output data are channels (arrays of SIMD vectors)

- Byte code contains instructions for operating on channels

  pos = ADD pos move

- After decoding instruction, interpreter applies it to *n* objects at a time


```
Vector4 *a = (decode channel ref);
const  Vector4 *b = (decode channel ref);
const  Vector4 *c = (decode channel ref);
Vector4 *ae = a + n;
while (a < ae) {
    *a = *b + *c;
    ++a; ++b; ++c;
}
```

LOOP CAN BE UNROLLED TO FIXED VALUE OF n

Bytecode Details: Constants

- Constants

constant for all particles during frame

age = age + *delta_time;*

- Stored as a Vector4 in-place in the byte code

age = $\text{ADD}_{\text{constant}}$ age $(0.0\ 0.0\ 0.0\ 0.0)_{\text{delta\_time}}$

Note: The bytecode uses a separate $\text{ADD}_{\text{constant}}$ opcode when adding a channel and a constant

- The compiler keeps track of location of all bytecode constants

Before running the bytecode you patch the values of all constants

patch_constant(bytecode, hash("delta_time"), vector4(0.33, 0.33, 0.33, 0.33));

age = $\text{ADD}_{\text{constant}}$ age $(0.33\ 0.33\ 0.33\ 0.33)_{\text{delta\_time}}$

When the bytecode runs, no lookup is needed for constants → maximum speed

BYTECODE DETAILS: TEMPORARY VARIABLES

r0 = MUL vel delta_time

pos = ADD pos *r0*

- We use temporary Vector4 buffers for temporary (and local) variables

  To virtual machine, no distinction between temp buffers and channels

  Temp buffers do not have to be as big as the input channel

  Only as big as *n*, the number of items we process at a time

- Balance between memory use and performance

  We want high *n* to amortize the cost of instruction decoding

  We want low *n* to minimize temporary memory use

  n = 128 is a decent compromise

THE BIG PICTURE

- Offline

  Data compiler parses code

  pos = pos + vel * delta_time

  Generates bytecode, introduces temporary variables as necessary

  $r0 = \text{MUL vel } (0.0\ 0.0\ 0.0\ 0.0)_{\text{delta\_time}}$

  pos = ADD pos r0

  Bytecode is optimized (Temporary variable elimination)

- Runtime

  Patch the constants in the bytecode

  Execute the instructions

Implementation Details

- Very simple hand-written tokenizer and recursive decent parser

  ~1000 lines

- Trivial bytecode format

  OPERATOR $operand_1$ $operand_2$

  No packing/unpacking necessary, we do not need to optimize for bytecode size

- Very simple virtual machine implementation

  Big switch statement

  ~250 lines

Real-World Example

```
// Source syntax inspired by HLSL
const float4 center = float4(0,0,0,0);
const float4 up = float4(0,0,1,0);
const float4 speed = float4(1,1,1,1);
const float4 radius = float4(5,5,5,5);

struct vf_in
{
    float4 position : CHANNEL0;
    float4 wind : CHANNEL1;
};

struct vf_out
{
    float4 wind : CHANNEL1;
};

void whirl(in vf_in in, out vf_out out)
{
    float4 r = in.position - center;
    out.wind = in.wind + speed * cross(up, r) / dot(r,r) * radius;
}
```

```
// Resulting bytecode
// r0, r1 correspond to CHANNEL0, CHANNEL1
// r2--r5 are temporary variables
```

$r2 = \text{SUB } r0\ (0,0,0,0)_{center}$
$r3 = \text{CROSS } (0,0,1,0)_{up}\ r2$
$r4 = \text{MUL } (1,1,1,1)_{speed}\ r3$
$r3 = \text{DOT } r2\ r2$
$r5 = \text{DIV } r4\ r3$
$r3 = \text{MUL } r5\ (5,5,5,5)_{radius}$
$r1 = \text{ADD } r1\ r3$

USE CASE 1: WIND SIMULATION

- Wind is simulated as a superposition of effects

    Effect: Cull box + script with constants

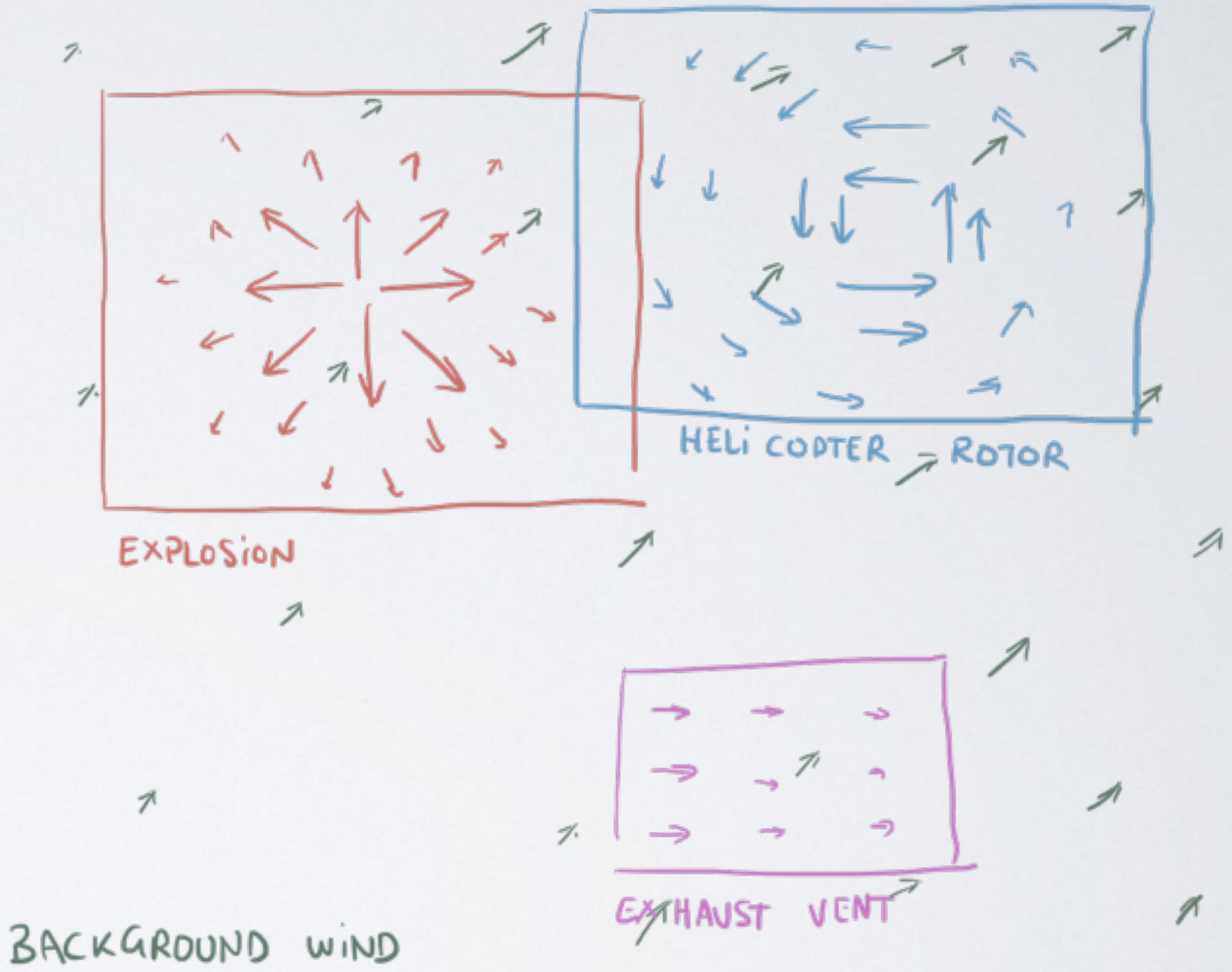
    Script returns wind at position

- Evaluation at a large number of points

    Positions of particles and physics objects

    Apply culling to find relevant effects

    Merge the bytecode to a single function (for performance)



Use Case 2: Particle Simulation

- A "particle" is just a collection of channels

  Position, velocity, color, size, etc

  Editor completely defines what channels exist

  For example there could be two position channels (for a "beam" particle).

- Particle effects written in vector language

  Editor allows using existing effects

  Or writing completely new ones

- We are transitioning to this system

  Runs in parallel with old less dynamic particle system

Comparing Performance to Native

- Example: 64K particles with gravity and one collision surface

- ~34 % overhead over native
  ~18 % overhead over modifiers

  Source: loads & stores

  Compare to typical bytecode overhead: x10 – x20

- On the console, the modifier solution exhausts L2 cache

  With smaller data set x1.28

```
void update(in vf in, out vf out)
{
    float4 vel = in.vel + gravity*dt;
    out.pos = in.pos + vel*dt;
    float4 collide = dot(in.pos - plane_p, plane_n) < 0;
    float4 travelling_down = dot(vel, plane_n) < 0;
    out.vel = vel - 2 * vel * collide * travelling_down;
};
```

	Native	Modifiers	Scripted
Modern PC	0.402 ms	0.455 ms	0.539 ms
	x1.0	x1.13	x1.34
X360 PS3 gen console	5.398 ms	10.196 ms	7.006 ms
	x1.0	x1.89	x1.30

But Wait — We Can Do Better!

- Rewrite the bytecode interpreter in AVX

    Process 8 floats at a time

    Now we run faster than native!

- Fair comparison?

    We could rewrite the native code in AVX as well

    But will you take the time to rewrite all your handwritten code to use AVX?

    Will you maintain multiple versions for SSE, AVX, Neon, etc?

	Native	Modifiers	Scripted	AVX
Modern PC	0.402 ms	0.455 ms	0.539 ms	0.373 ms
	x1.0	x1.13	x1.34	x0.92

CONCLUSIONS

- The "data wide interpreter" model is a viable solution for high-performance scripting

  Completely configurable behaviors

  Fully dynamic: can be quickly reloaded, no engine recompile necessary

  18 % overhead over traditional modifier stack solution (34 % over native)

  AVX enabled scripted solution is *faster* than native solution

- Future

  One channel per component (pos.x, pos.y, pos.z)

  More backends: JIT compiler, GPU Compute, SPU…

QUESTIONS

- www.bitsquid.se
- niklas.frykholm@bitsquid.se
- niklasfrykholm
- www.youtube.com/bitsquid