GDC 2020

Writing Tools Faster

Design Decisions to Accelerate Tool Development

Niklas Gray

CTO, Our Machinery, @niklasfrykholm

Who am I?

My name is Niklas Gray, I write game engines:

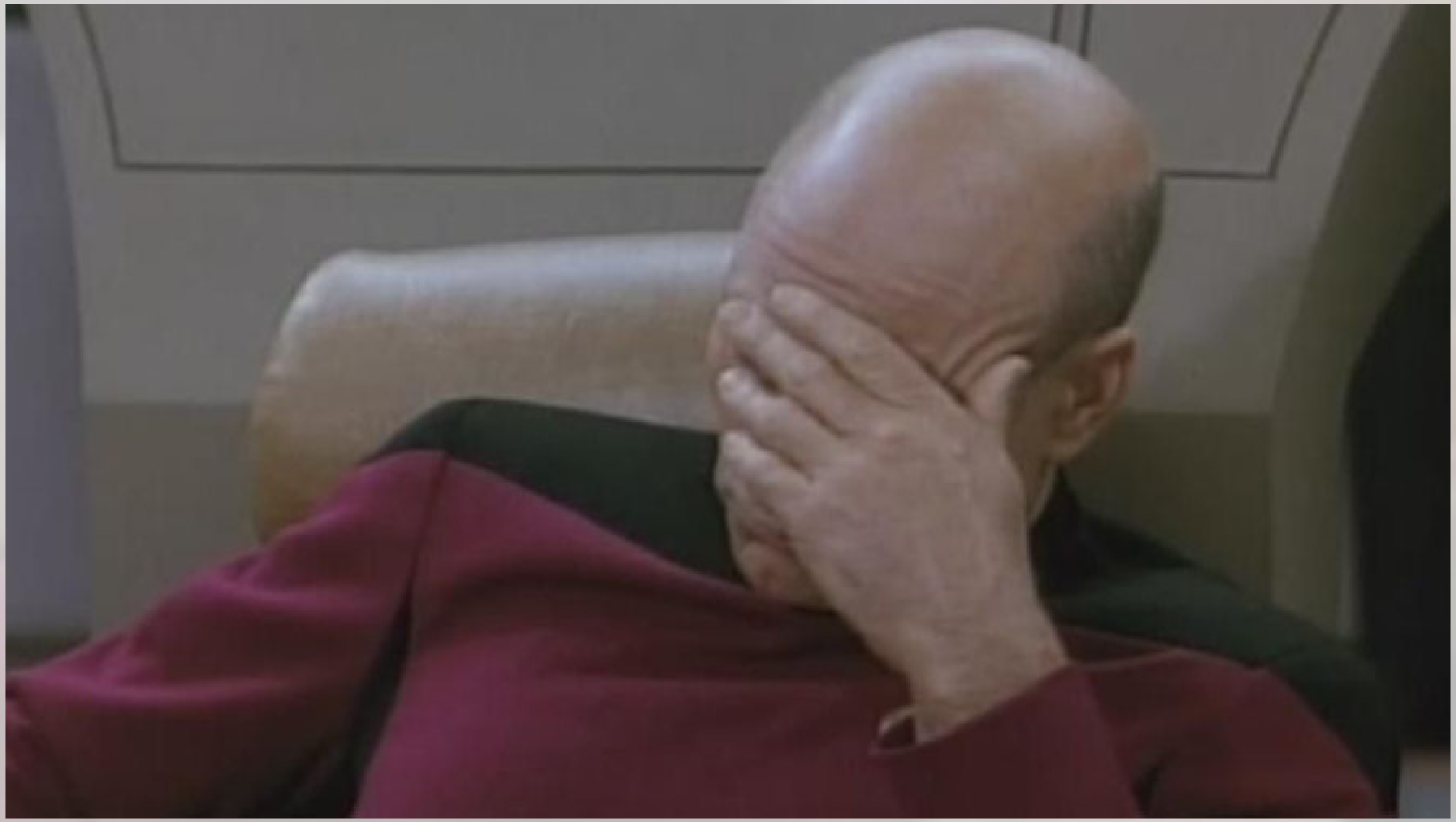
Diesel	In-house engine at Grin (Ghost Recon, Payday)
Bitsquid	Commercial game engine (Vermintide, Helldivers)
Stingray	Bitsquid rebranded by Autodesk
The Machinery	Let's make another game engine!

In This Talk

- Why is writing tools so hard? (for us)
- What can we do about it?

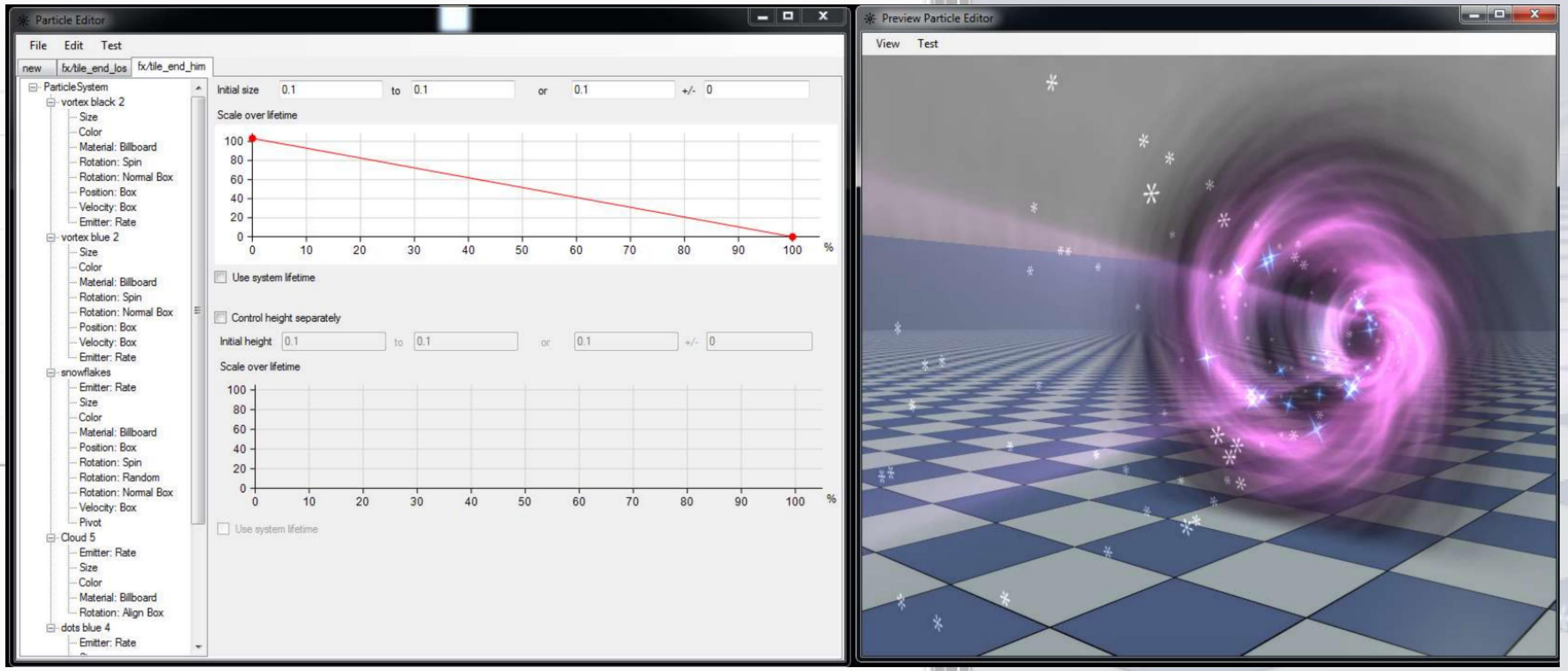
Tools: A Brief History of Failure

Bitsquid 1.0: Our Users Can Make Their Own Tools!



Bitsquid 2.0

- Let's hack together something quickly in WinForms
- Kind of ugly
- No clear overall plan, hard to maintain



Initial size 0.1 to 0.1 or 0.1 +/- 0

☐ Use system lifetime

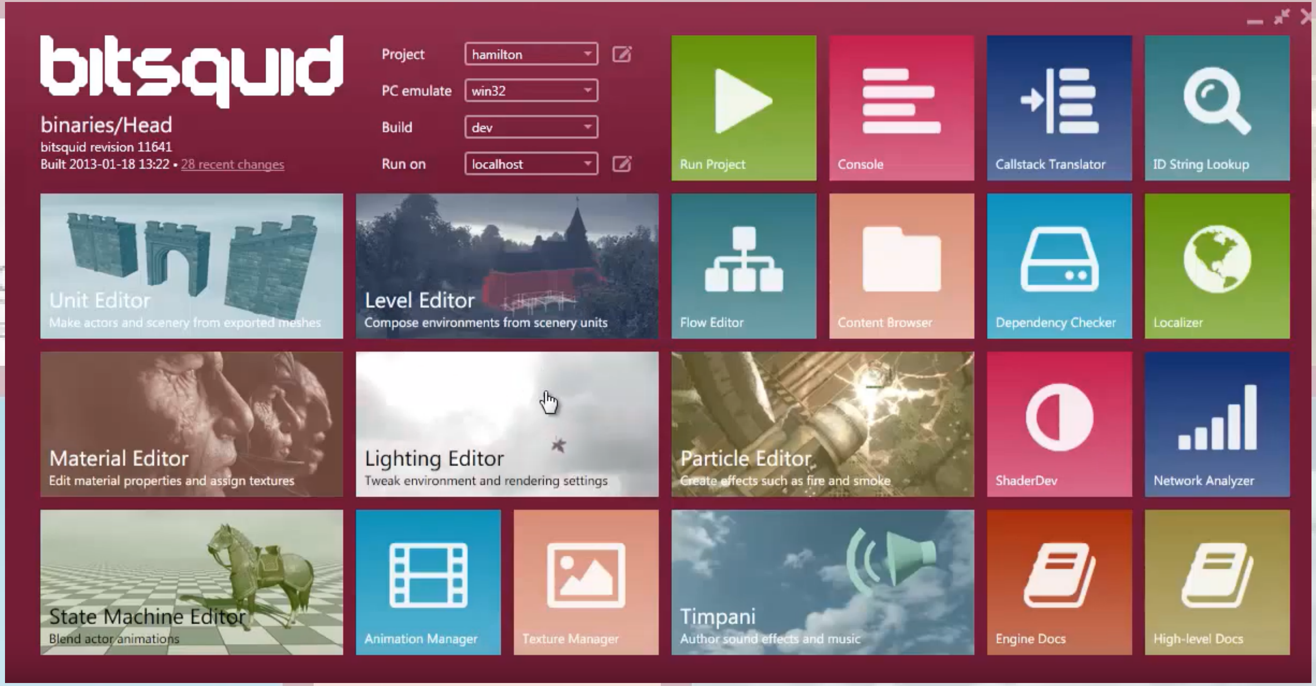
☐ Control height separately

Initial height 0.1 to 0.1 or 0.1 +/- 0

☐ Use system lifetime

Bitsquid 3.0

- WPF is prettier!
- Tools take longer to write
- Barrier of entry: WPF, XAML, ...
- Never fully completed rewrite from WinForms

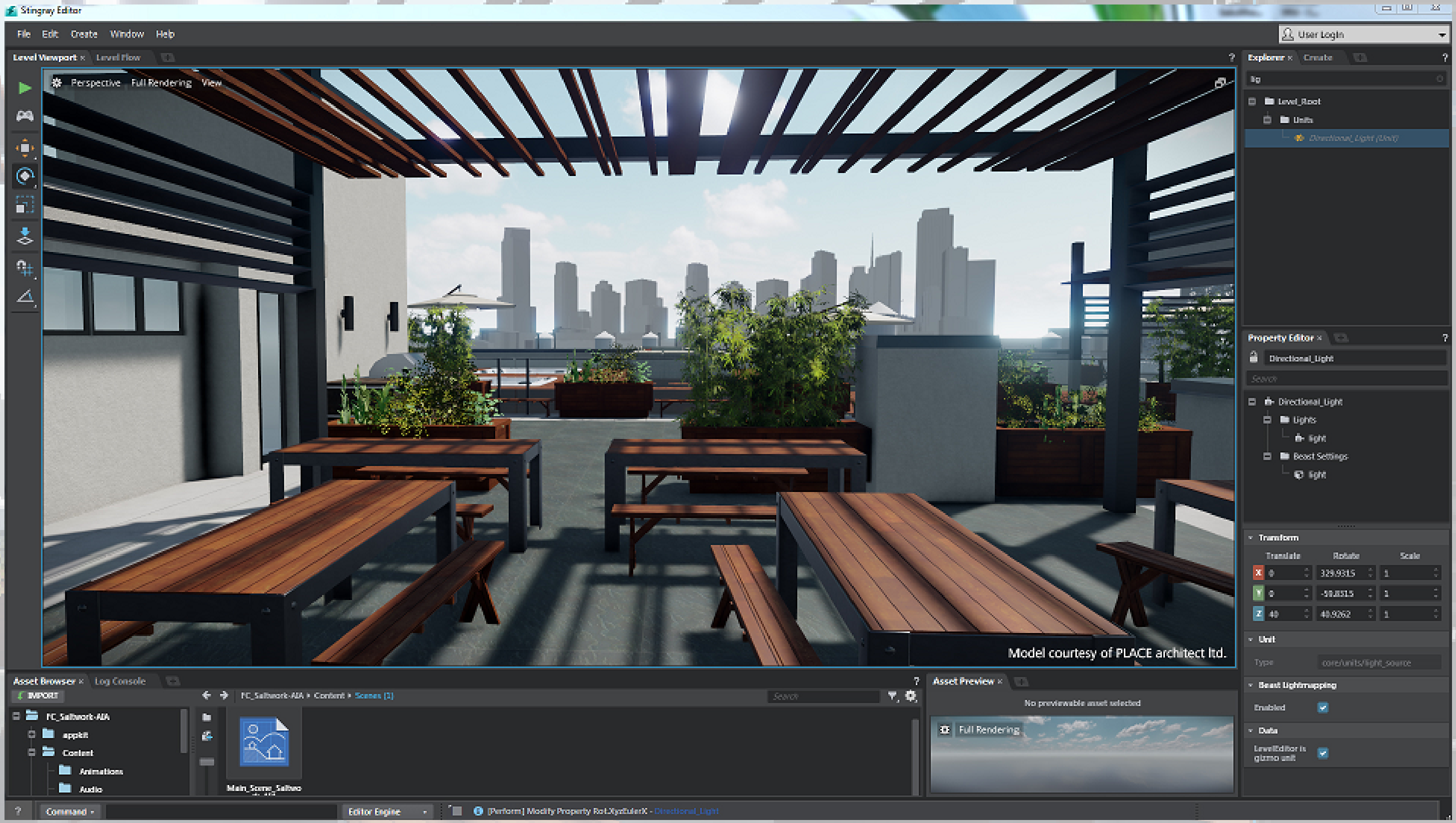


binaries/Head
bitsquid revision 11641
Built 2013-01-18 13:22 • 28 recent changes

Project	hamilton
PC emulate	win32
Build	dev
Run on	localhost

Stingray

- Web platform (in theory)
  - Platform independent
  - Reuse web expertise
- Tech stack getting crazy
  - C#, Lua, C++, WPF, WinForms, Chromium, Qt, JavaScript, Angular, WebSockets
- Tools take even longer
- Never completed this rewrite either!



Bitsquid/Stingray Problems

1. Keep changing frameworks
2. Tools take too long to write
3. Lackluster performance

End result: Bad tools!

How do we fix it?

Why Change Frameworks?

- Sometimes: bad decisions
- Sometimes: tech gets outdated or abandoned
    - Swing, Delphi, Hypercard, Flash, NaCl, Python 2, ...
- Running on abondoned tech gets painful

Why Did Writing Tools Take So Long?

- Every little thing needed an UI (designed, coded, tested)
- Features: Undo, copy/paste, serialize, drag-and-drop, ...
- A deep tech stack is hard to understand
  - Bug in Angular, JavaScript, WebSocket, Chromium, C#, Lua or C++?
  - Complicates everything!
- Only tool people understood the tool stack: silos

Why Did We Have Performance Problems?

- Standard web practices didn't always work
    - Not always a performance mind set
    - Game development has more stuff!
- Fixing performance often required a full rewrite
- The deep stack made the issues harder to find

How Do We Fix it?

- Automate undo, copy/paste, etc with a well-defined data model
  - Less busy-work
- Minimize and own the tech stack
  - Make things explicit and easy to understand
  - Avoid changing frameworks
  - Control performance
- Reuse UIs and generate them automatically from data
  - Properties, Tree, Graph, etc
  - Don't have to create an UI for everything.

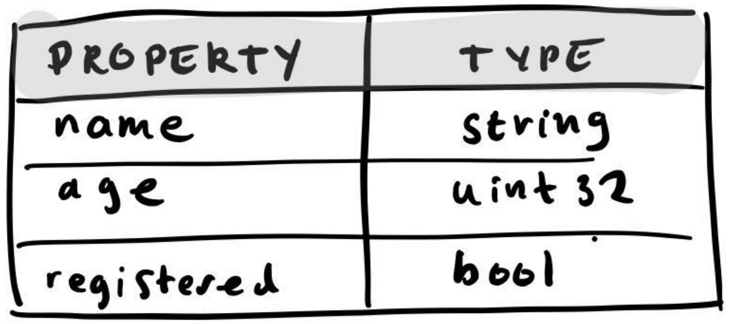
Data Model

The Truth

- Represent all data in a uniform way
- Operations (Undo, etc) can be defined on the data model

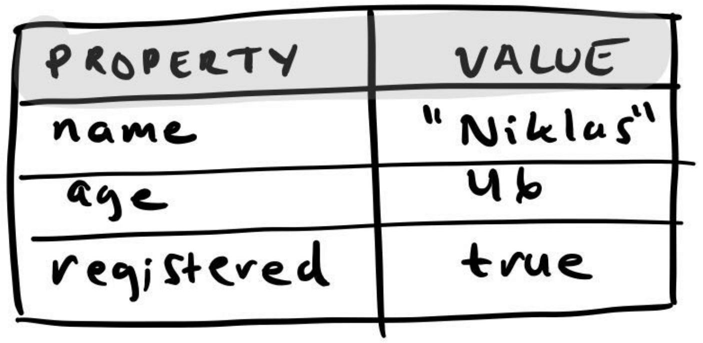
Objects with Types and Properties: (reference, subobject)

OBJECT TYPE



PROPERTY	TYPE
name	string
age	uint32
registered	bool

OBJECT

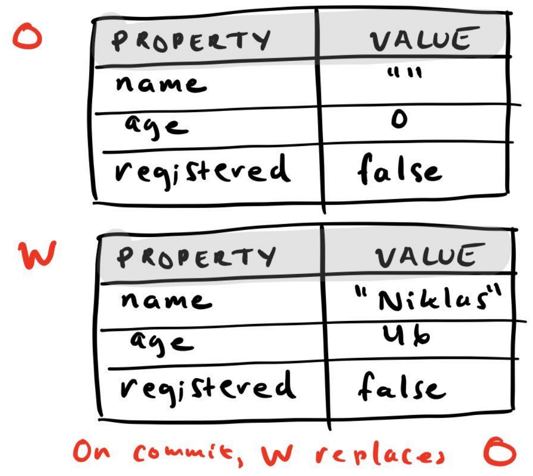


PROPERTY	VALUE
name	"Niklas"
age	46
registered	true

Lock-Free Multithread Access

- Changing the data is a two phase process: write/commit
- *Write* creates a new copy of the object for modification
- *Commit* atomically switches the old copy for the new
- Readers can read the data without locking
    - Old read copies eventually garbage collected

```
W = begin_write(O)
set_property(W, NAME, "Niklas")
set_property(W, AGE, 46)
commit(W)
```



O

PROPERTY	VALUE
name	""
age	0
registered	false

W

PROPERTY	VALUE
name	"Niklas"
age	46
registered	false

On commit, W replaces O

Undo

- On *Commit* – save the old and new object versions in current undo scope
- On *Undo* – reinstate the old data
- An undo scope can contain multiple changes to different objects

```
US = create_undo_scope(T)
W = begin_write(O)
set_property(W, NAME, "Niklas")
set_property(W, AGE, 46)
commit(W, US)

undo(T, US)
```

O

PROPERTY	VALUE
name	""
age	0
registered	false

W

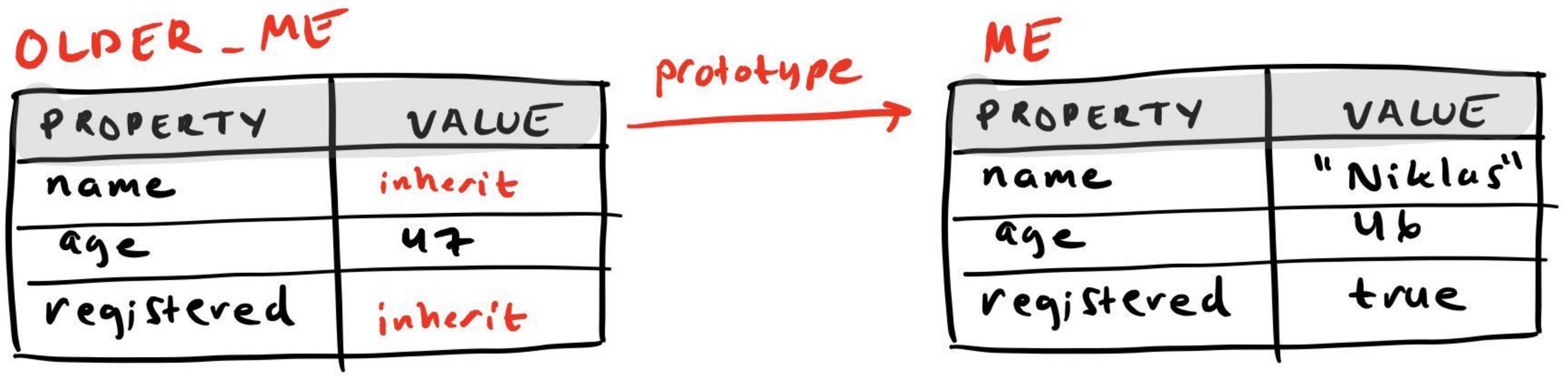
PROPERTY	VALUE
name	"Niklas"
age	46
registered	false

On commit, W replaces O

Prefabs/Prototypes

- An object can specify another object as its prototype
- "Inherits" properties, but can "override" them

```
US = create_undo_scope(T)
OLDER_ME = create_object_from_prototype(T, ME, US)
W = begin_write(OLDER_ME)
set_property(W, AGE, 47)
commit(W, US)
```



OLDER_ME

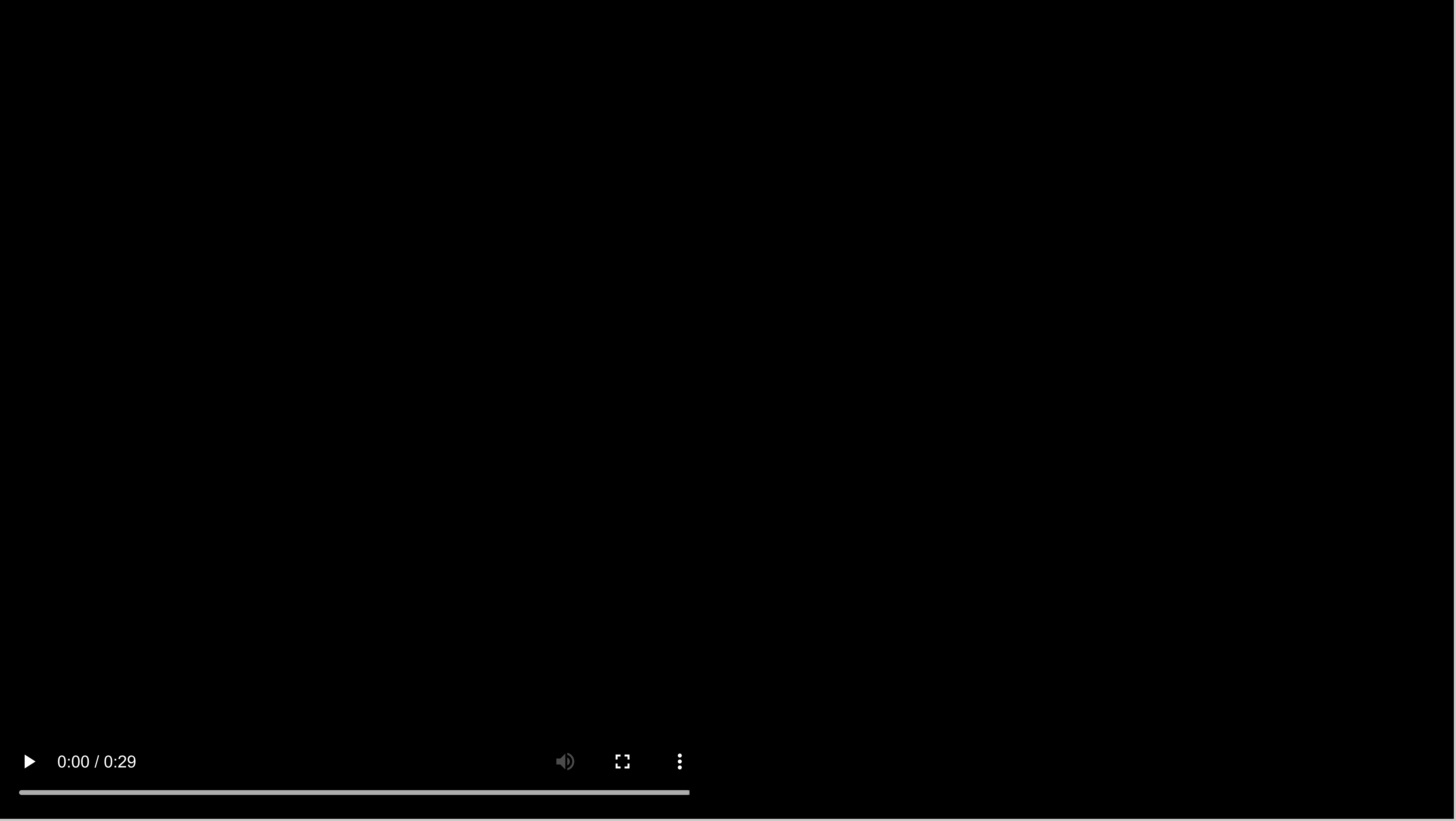
PROPERTY	VALUE
name	inherit
age	47
registered	inherit

ME

PROPERTY	VALUE
name	"Niklas"
age	46
registered	true

Live Collaboration

- On *commit* – compute a delta between old and new object versions
- Transmit delta over wire to other collaborators


0:00 / 0:29

The Truth: Pros & Cons

- Lots of functionality "for free"
- Even advanced features: collaboration, prototyping

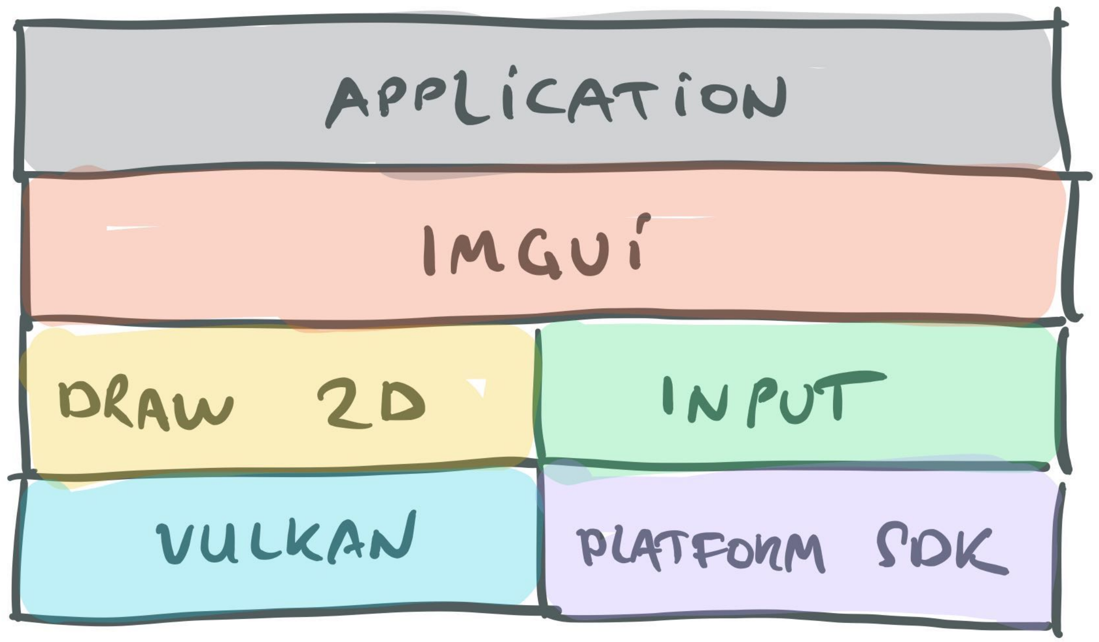
Cons

- Some data is not represented well in key-value format (e.g. long text)
- The system is complex and sits at the center of everything
  - No easy way for other systems to "opt-out"
  - Scary to make modifications

Minimized Tech Stack

Our Stack

- Everything is written in C
- Very few external dependencies



Draw 2D: 2D Drawing Library For UI

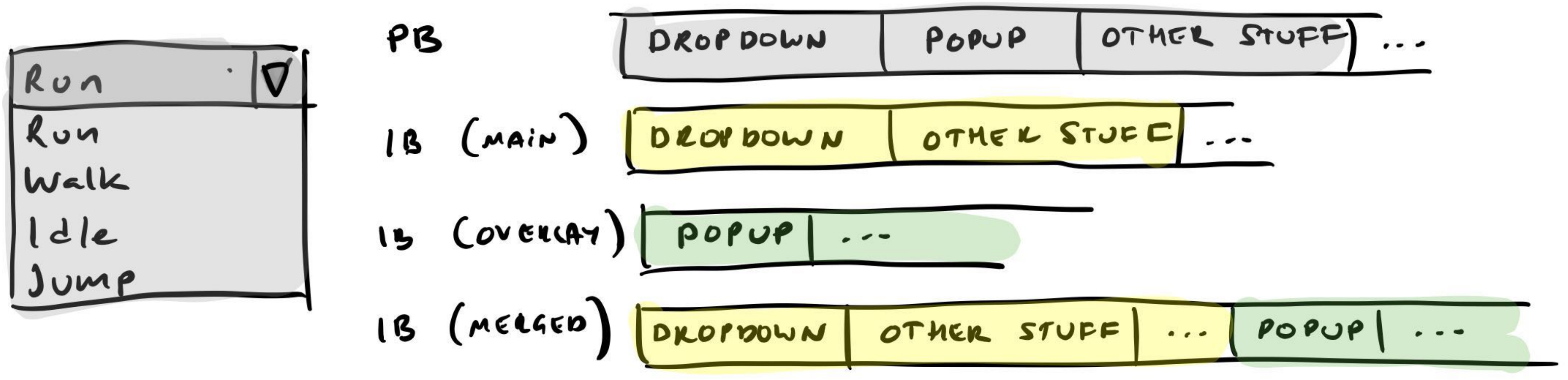
- `stroke_rect()`, `fill_rect()`, etc
- Writes data directly into vertex buffer & index buffer
- Entire UI rendered in a single draw call
- https://ourmachinery.com/post/ui-rendering-using-primitive-buffers/

Draw 2D: Clipping

- Clip rects are written to the vertex buffer
- Pixel shader clips against rect

Draw 2D: Overlays

- Overlay images (popups) are drawn to a separate index buffer
- Concatenated before submitting draw call
- Note: overlay will be clipped to window



UI

- Immediate mode GUI – no create/destroy
- Single call to draw control and handle interaction

```
if (ui_api->button(ui, &(ui_button_t){.rect = button_r, .text = "OK"}))
    logger_api->printf(LOG_TYPE_INFO, "OK was pressed!");

bool cb = false;
ui_api->checkbox(ui, &(ui_checkbox_t){ .rect = box_r, .text = "Check!" }, &cb);
```

- Every control is drawn every frame
- Controls don't have permanent existence, but they're identified by an ID
- We keep track of the ID the user is hovering over or interacting with

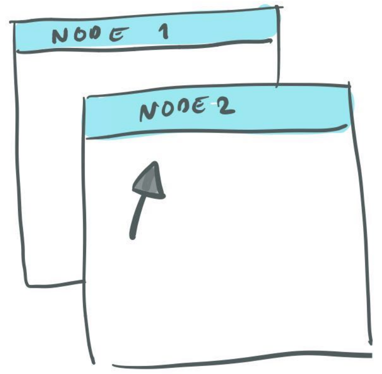
IMGUI: Pros & Cons

- More straightforward code flow (debugging, profiling)
- No need to synchronize state
- Redraw every frame -- expensive?
    - Viewport typically wants to render every frame anyway
    - Can do it just on mouse/keyboard events
    - Easy to match performance to what is shown on screen
- New mindset: no objects to talk to
    - Can usually find ways around it

0:00 / 0:48

IMGUI Gotchas Example: Overlap

- In retained: we would just loop over all nodes
- We can't do: `if (in_rect(mouse,r) && button_down)`
    - Node 1 would get click that should go to Node 2
- Fix: frame delay

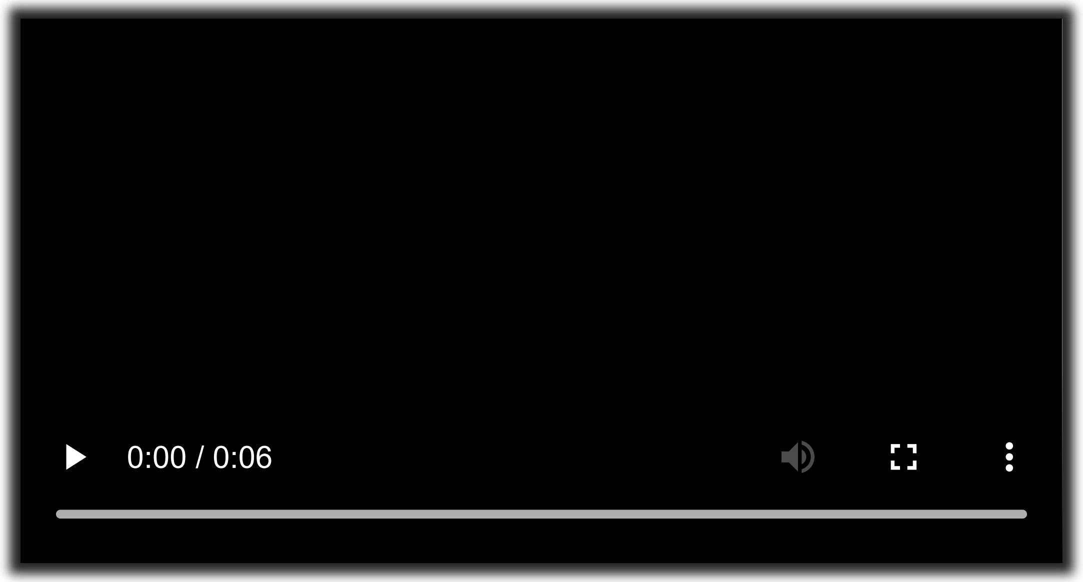


```
if (in_rect(mouse, r))
    ui.next_hover = id;
if (ui.hover == id && button_down)
    ...;
```

- At end of frame: `ui.hover = ui.next_hover`
- Node 2 will overwrite `ui.next_hover`

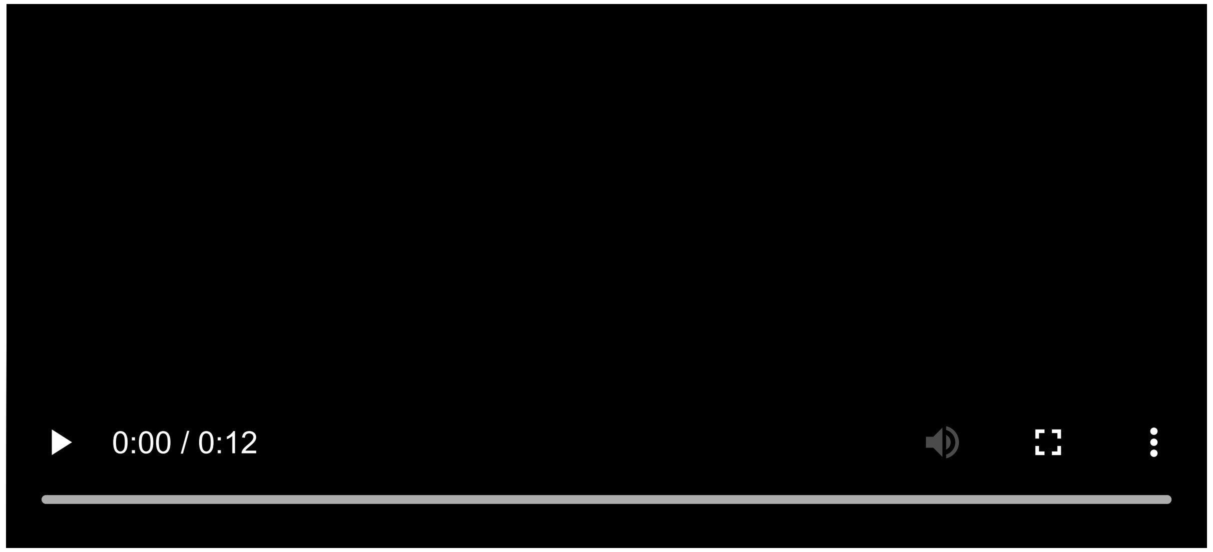
Layouting

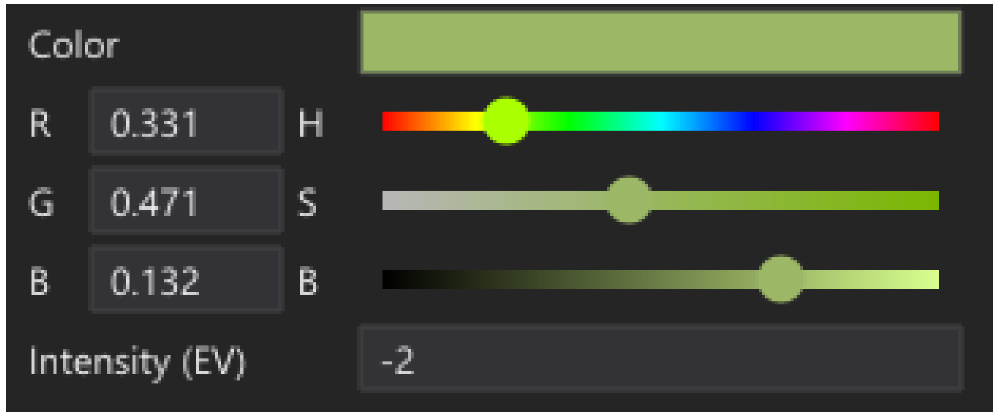
```
// No need for "layout managers" -- instead we split rects directly in code

rect_t header_r = rect_split_off_top(r, header_height, margin);
rect_t search_r = rect_split_off_right(header_r, search_width, margin);
rect_t footer_r = rect_split_off_bottom(r, footer_height, margin);

rect_t tree_r, browser_r;
ui_api->splitter_x(ui, &(ui_splitter_t){.rect = r}, &bias, &tree_r, &browser_r);
```



Custom Controls

- Easy to implement custom control: draw + input interaction
- No distinction between "built-in" and "custom" controls




Color
R 0.331 H
G 0.471 S
B 0.132 B
Intensity (EV) -2

```
static void ui_drag_number(ui_o *ui, uistyle_t *style, const ui_drag_number_t *c, float *value)
{
    ui_buffers_t uib = ui_api->buffers(ui);
    const uint64_t id = c->id ? c->id : ui_api->make_id(ui);

    if (vec2_in_rect(uib.input->mouse_pos, c->rect) && !uib.activation->next_hover_in_overlay)
        uib.activation->next_hover = id;

    if (uib.activation->hover == id && uib.input->left_mouse_pressed)
        ui_api->set_active(ui, id);

    if (uib.activation->active == id) {
        const float dx = uib.input->mouse_delta.x;
        *value = active->original_value + dx / 50.0f * fabsf(active->original_value);
        if (uib.input->left_mouse_released)
            ui_api->set_active(ui, 0);
    }

    if (uib.activation->active == id || uib.activation->hover == id)
        style->color = colors[UI_COLOR_SELECTION];

    char text[32];
    sprintf(text, "%.7g", *value);
    draw2d_api->draw_text(uib.vbuffer, *uib.ibuffers, style, c->rect, text, n);
}
```

In Summary

- Full control of the stack – easier to understand
- Same language/API as rest of engine, no artificial barriers

Cons:

- You start from scratch (~6 man-months of work)
    - Initial cost is soon recouperated
    - Could use Dear IMGUI
- Lots of design decisions
- IMGUI requires new thinking

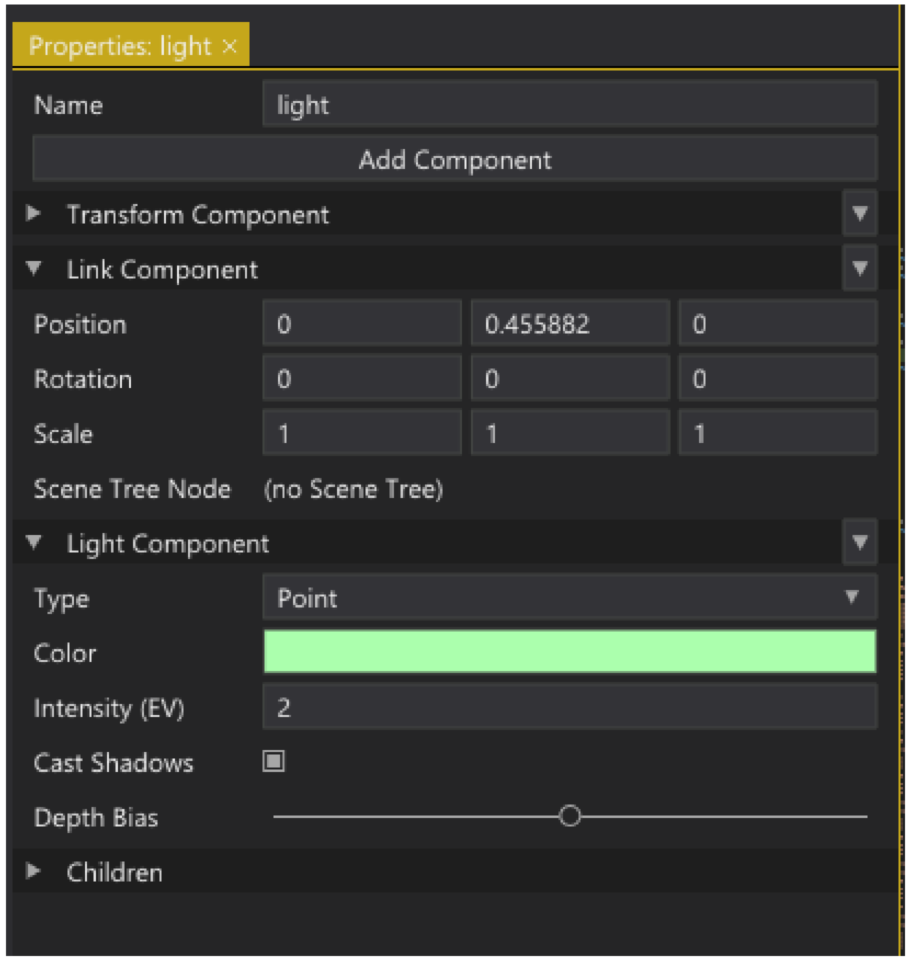
Generating UIs

Motivation

- Reduce the work of creating UIs for everything

Example: Properties Panel

- Our default object editor
- Loop over the properties of a focused object
- Draw an appropriate editor for each property
  - Bool: Checkbox
  - String: Textbox
  - ...
- This doesn't always work (color)



Properties: light ×

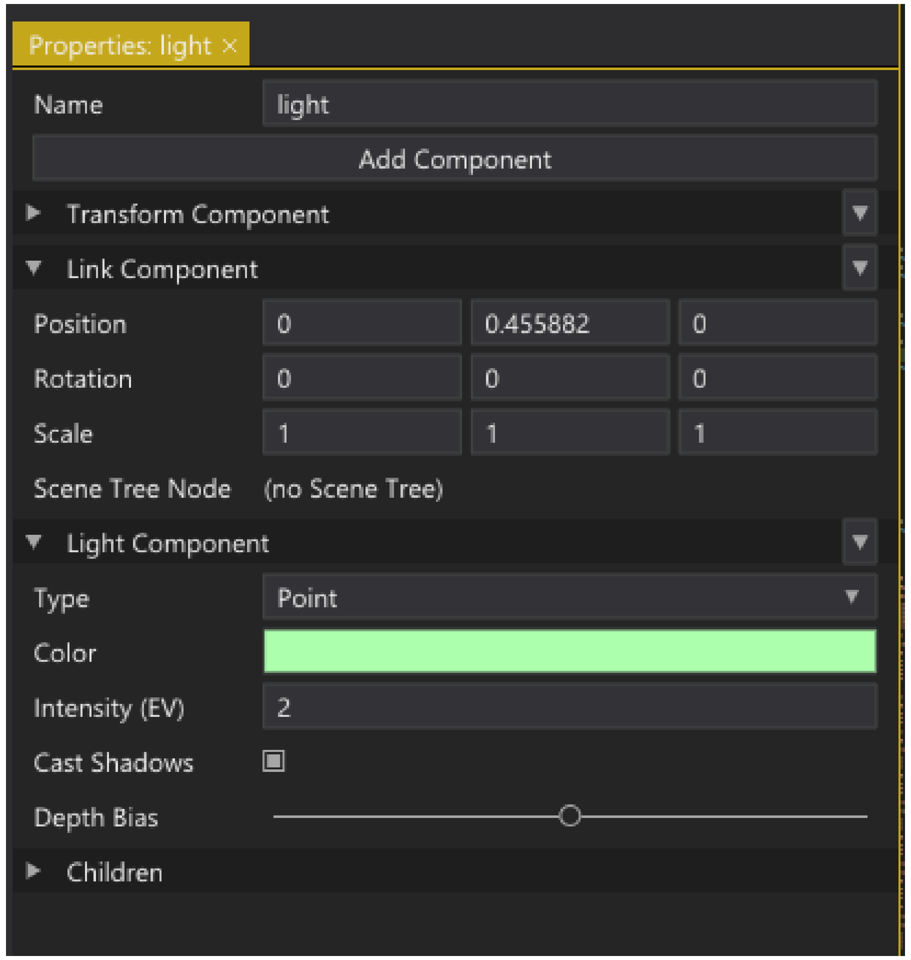
Name	light		
Add Component			
▶ Transform Component			
▼ Link Component			
Position	0	0.455882	0
Rotation	0	0	0
Scale	1	1	1
Scene Tree Node	(no Scene Tree)		
▼ Light Component			
Type	Point		
Color	🟩		
Intensity (EV)	2		
Cast Shadows	▣		
Depth Bias			
▶ Children			

Custom Properties

- We can customize how objects in The Truth behave by adding *Aspects*
- Basically a callback identified by an ID
- Draw `vec3` on a single line:

```
the_truth_api->set_aspect(
    tt, vec3_type, TT_ASPECT__CUSTOM_PROPERTIES,
    ui_vec3);
```

- Objets without aspect get the default panel



Properties: light

Name	light		
Add Component			
▸ Transform Component			
▾ Link Component			
Position	0	0.455882	0
Rotation	0	0	0
Scale	1	1	1
Scene Tree Node	(no Scene Tree)		
▾ Light Component			
Type	Point		
Color	🟩		
Intensity (EV)	2		
Cast Shadows	▣		
Depth Bias			
▸ Children			

Example

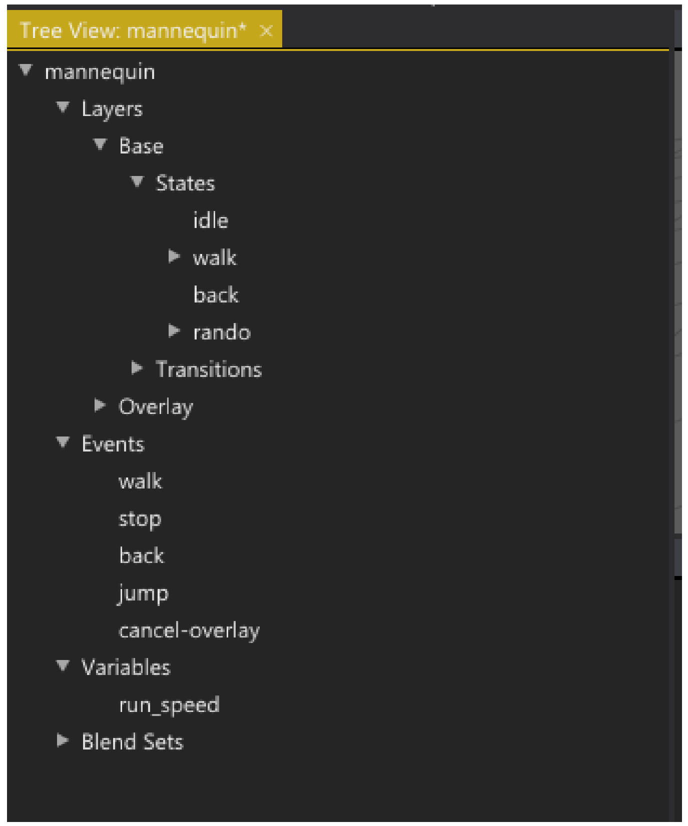
```
static float ui_vec3(properties_ui_args_t *args, rect_t item_rect, const char *name,
    const char *tooltip, uint64_t vec3)
{
    const rect_t label_r = rect_split_left(item_rect, label_width, margin, 0);
    const rect_t control_r = rect_split_left(item_rect, label_width, margin, 1);

    private__ui_tooltip_label(args->ui, args->uistyle,
        &(ui_tooltip_label_t){ .text = name, .rect = label_r, .tooltip = tooltip });

    for (uint32_t i = 0; i < 3; ++i) {
        const rect_t component_r = rect_divide_x(control_r, margin, 3, i);
        private__ui_float_box(args, component_r, vec3, i);
    }
    return item_rect.y + item_rect.h + margin;
}
```

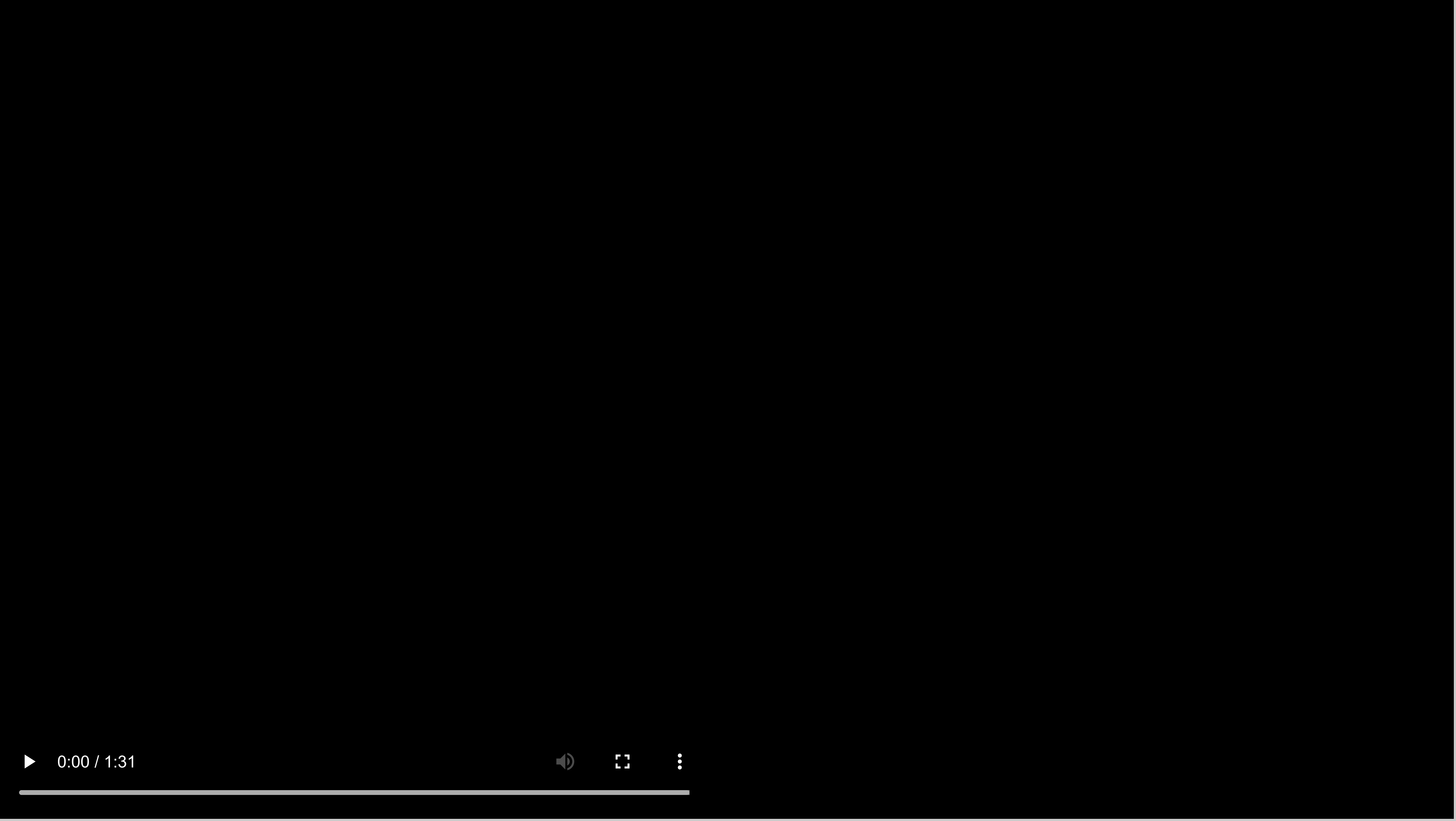
Generated UI: Preview

- Tab that allows preview of assets
- Controlled by a `PREVIEW` aspect – spawns entities, draws UI

Generated UI: Tree View

- By default, all *subobjects* are rendered as children
- `TREE_VIEW` aspect for customizing




0:00 / 1:31

Conclusion / Post-Mortem I

- Creating UIs feels faster
  - Not "blocked" by UI tasks
- Full engine built by two people in two years
- Data model: awesome, but scary
  - Each new piece adds more complexity
- Aspects are a great way of customizing object behaviors

Conclusion / Post-Mortem II

- Implementing things yourself is a lot of work
- Making a toolkit requires a lot of "functional design"
  - How should things work?
- We are missing features that you would expect in a full-fledged toolkit
  - Right-to-left text
  - (But note: In Stingray we never even had time to *start* on localization)

All-in-all we're happy with the direction

Questions?

@niklasfrykholm