Bitsquid: Behind The Scenes

Building a Game Engine
Design, Implementation & Challenges

Niklas Frykholm

System Architect, Bitsquid

bitsquid

Design Goals

- An engine that is:

    Flexible

    Fast

    And supports good workflows

- Not a click-and-play “game maker”

    Aimed at professionals who want full performance and full control

- Not a repurposed first-person shooter

    For all game types

Flexible

- Engine

  Avoid bit-rot (large software systems get worse over time)

- Any game type

  FPS, platformer, RPG, racing game, architectural visualization, etc

  User must be in full control of game logic

- Wide range of platforms (mobile → high end PC)

  Very different performance characteristics

  Don't try to hide platform differences (abstraction → inefficiency)

  Users must be able to control the entire rendering pipeline (forward/deferred)

- Flexibility puts higher demands on users than click-to-play

Hamilton’s Great Adventure (Fatshark: PC, PS3, Android)

War of the Roses (Fatshark: PC)

Krater (Fatshark: PC, OS X)

The Showdown Effect (Arrowhead: PC, OS X)


War of the Vikings (Fatshark)

Magica Wizard Wars (Paradox North)

Helldivers (Arrowhead)

Avoiding Bit-Rot

- Care about code quality

  Publish your source code! (We give it to our customers)

- Keep the engine small

  Less code is better! Aggressively remove what you don't use!

  Don't do everything that our customers want! (They can do it themselves in the source.)

- Refactor

  When you find a better way to do things → rewrite

- Decouple

  As few dependencies between systems as possible

  Individual systems can be replaced without rewriting everything

DECOUPLING

- Avoid systems directly talking to other systems
  - Physics system should not trigger footstep sounds
- Avoid systems directly referring to data owned by other systems
  - No external system should hold a PlayingSound *
- Avoid "global" systems
  - Serialization, reference counting, message switchboard
- Event stream
  - Physics system publishes stream of events
  - [COLLISION] [COLLISION]
- Use IDs to refer to instances
  - Just an integer
  - Internally, system can represent data how it wants
  - Look-up table
- High-level system mediate connections between low-level systems
  - Read event-stream and take action

Decoupling

```
class ISoundSystem {
    uint32_t play(…);
    bool is_playing(uint32_t id);
    void stop(uint32_t id);
    …
};
```

- Sound system can represent sounds however it wants internally
  - Pack and reorder data for highest performance
  - Double-buffer for multithreading
  - Allocate and free sound objects when it wants
  - External systems don’t have to reference count or worry about dangling pointers

DATA-DRIVEN

- Handle *everything* with configuration files

  Even renderer configuration (forward, deferred, layers, etc)

  Everything is text: Extended JSON-format

```
render_settings = {
    shadow_map_size = [1024, 1024]
    ssao_enabled = true
}
global_resources = [
    {name = "depth_stencil_buffer" type="render_target" depends_on="back_buffer"
format="DEPTH_STENCIL"}
    {name = "albedo" type="render_target" depends_on="back_buffer" format="R8G8B8A8"}
    ...
]
```

Data Management

- Configuration files are converted to data BLOBs by data compiler

    BLOB - raw serialization of C data

version	num_objects	name_hash$_1$	x_1 y_1 z_1	name_hash$_2$	x_2 y_2 z_2	…

    Platform specific

    Uses offset instead of pointers to refer to data internally

    No deserialization necessary, can just be read into memory and used directly

    Refers to other resources by hashed string names

- We are free to change the binary format of any data type

    Version change triggers recompile of data from JSON source

Scripting

- Build gameplay in high-level language

    More productive

    Dynamic languages are better suited to gameplay, more flexible — easier to modify and tweak

    Gameplay bugs do not crash the engine

    Scripts can be reloaded while the game is running (bug fixes, tweaks, etc)

    Gameplay programmers do not need to rebuild the engine for all target platforms

    Clear separation between clean engine code and messy gameplay code

    Engine systems can be made more decoupled and cleaner because the necessary messy connections between them are handled by the scripting layer

We use Lua

- Lightweight, flexible and powerful language

    Similar to Ruby, Python, JavaScript — but (IMHO) nicer

    Small core with "just the right stuff" (closures, etc) to build object systems, et

    Entire specification fits on one page

    Easy to learn, understand, optimize

- LuaJIT2

    Maintained by just one genius (Mike Pall)

    JIT compiler that reaches speed close to native C

    FFI interface (bind directly to C)

    Unfortunately cannot be used on all platforms (JIT not allowed)

Scripting Disadvantages

- Multiple languages used (C++ and scripting language)

    Separate debuggers — stepping through the code is trickier

- Performance (scripting languages are slower)
- Lack of static typing
- Garbage collection
- No multithreading support

- Still a clear win over writing gameplay in C++ (IMHO)

Scripting in the Bitsquid Engine

- C++ code exposes a Lua API

  Application.create_world(…)
  World.spawn_unit(…)
  …

- Script has full control

  Scripts are not attached to individual units

  Script does not consist of individual disjoint "snippets"

  More like a full program

  Single update() call from engine

  Responsible for loading data, creating worlds, updating and rendering them, etc

Visual Scripting

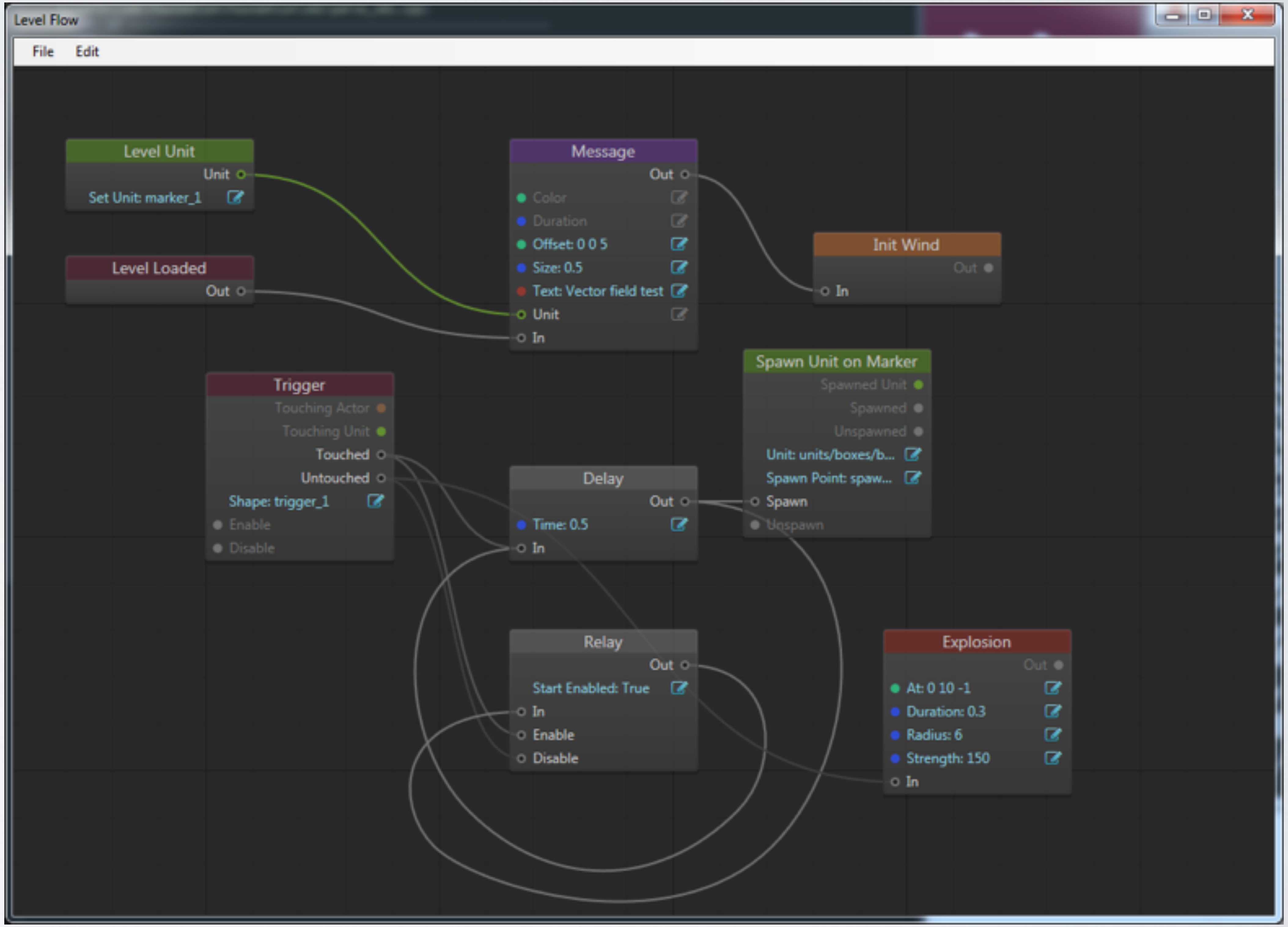
- Based on nodes and connections
- Event-based

  No update

- Allow artists & level designers to create custom logic

  Without learning a scripting language

- Gets compiled to BLOB



FAST

- Some simple games do not care that much about performance

    Fast enough

- But for *a lot* of games, performance is everything

    Getting nice graphics, lots of content, etc all depends on having good performance

- How do we make it fast?

    No stupid stuff! $O(n)$ — usually. $O(n \log n)$ — rarely. $O(n^2)$ — never.

    Think about cache behavior!

    Multithread!

    Measure your performance

MEMORY VS CPU PERFORMANCE

- CPU performance is rising much faster than memory performance

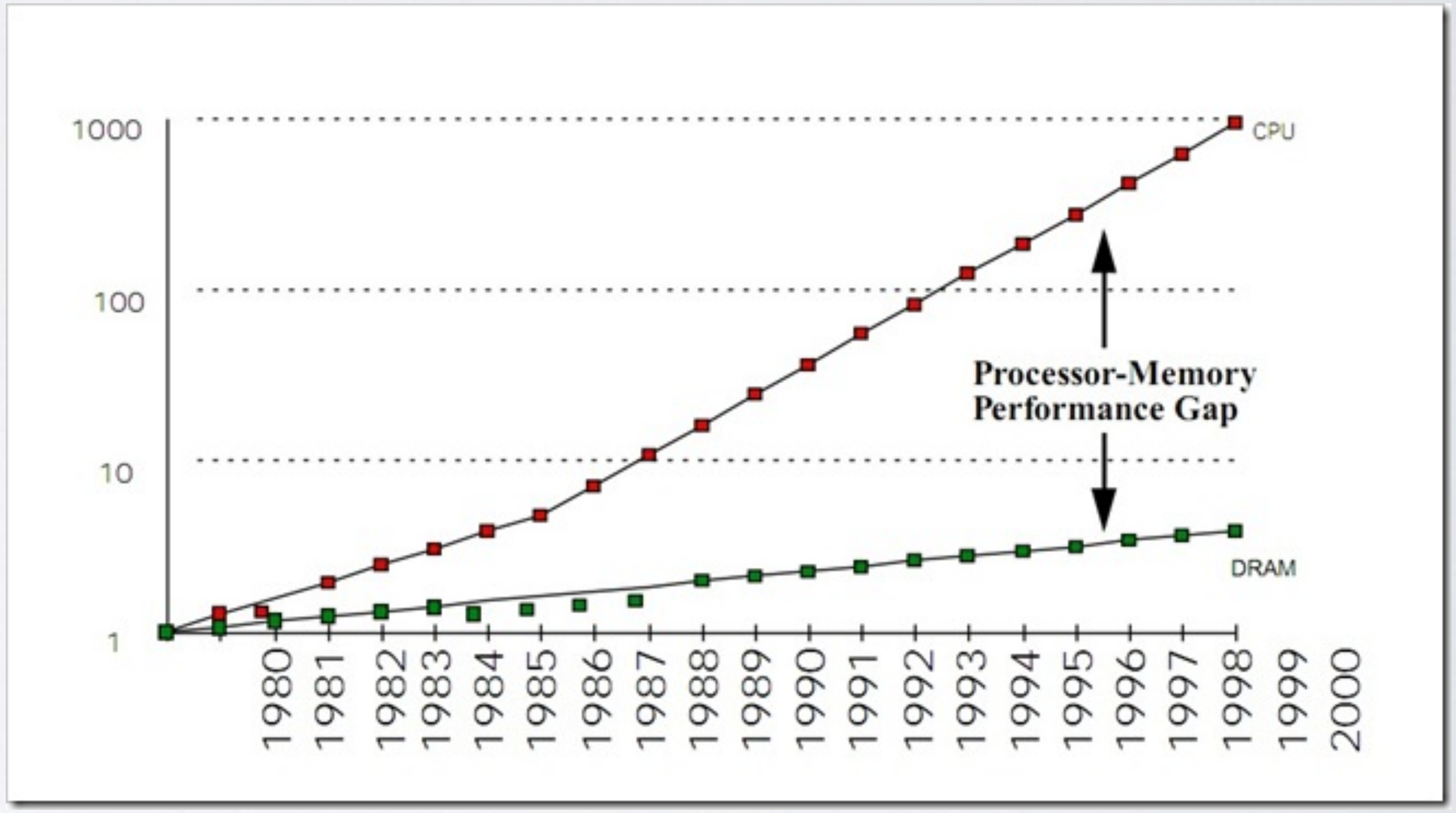
    Used to be one memory fetch per cycle

    Only gets worse

- LLC cache miss → 200 cycles or more

    Lots of times performance is memory rather than CPU-limited

- Make sure you hit the cache!



Data-Oriented Design

- New methodology to reflect memory performance (alternative to OOP)

    OOP — "Everything is an object"

    OOP — That lives separately on the heap

- Organize code with memory access patterns in mind

    Think about how data flows between systems and is transformed

    Avoid "pointer chasing" (malloc and new)

    Prefer flat arrays as data structures

        Create trees and list *within* an array instead of with pointers

    Process arrays in-order, one element at a time

Example: Object Update

```
std::vector<Effect *> e;
...
for (int i = 0; i<e.size(); ++i) {
    if (e[i]->visible())
        e[i]->update();
}
```

- Each effect allocated separately

    LLC cache miss when fetching

    LLC cache miss even if not updated

    (Accessing the visible flag)

```
unsigned num_visible;
unsigned num_effects;
Effect *e;
...
for (int i=0; i<num_visible; ++i)
    update_effect(e[i]);
```

- Effects allocated together

    No cache miss when fetching

- Array sorted with visible first

    No cost at all for invisible effects

EXAMPLE: SCENE GRAPH

```
class Node
{
    Matrix4x4 _local, _world;
    vector<Node *> _children;
};

void Node::transform(const Matrix4x4
&parent)
{
    _world = parent * _local;
    for (auto child : _children)
        child->transform(_world);
}
```

```
struct Graph
{
    unsigned num_nodes;
    Matrix4x4 *local, *world;
    unsigned *parent;
};

void transform(Graph &sg)
{
    for (unsigned i=0; i<sg.num_nodes; ++i) {
      sg. world[i] = (sg.parent[i] == UINT_MAX)
            ? sg.local[i]
            : sg.world[parent[i]] * sg.local[i];
    }
}
```

Multi-Threading

- Multi-core is the future

    x16 speedup — cannot be ignored

- Most of games is trivial to parallelize

    Lots of independent objects (particle effects, animations, entities)

    You don't need a "distributed algorithms" course!

- Strategy for multithreading?

    One thread per system? One thread per object? Locks, semaphores?

    Locking is expensive and easy to mess up → avoid as much as possible

    Context switching is expensive → avoid as much as possible

JOB PARALLELISM

- Create as many threads as there are cores in the system

    No over- or under-subscription, minimizes context switches

- Post jobs to a job pool

    Job: Function pointer, input data, output data

    Job only touches its own data, no locking needed

    Job threads pull next free job from job pool and execute it

    Systems split their updates into suitable number of jobs

    Mechanism for waiting for jobs

Multithreading Data Flow

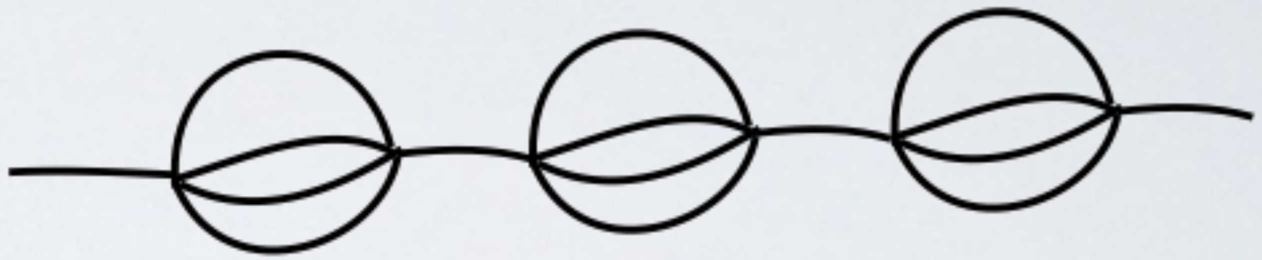
- Main track that goes wide for each system

    Easy to reason about

    Temporary memory for job output is short lived

    Low latency, from start of frame to end result

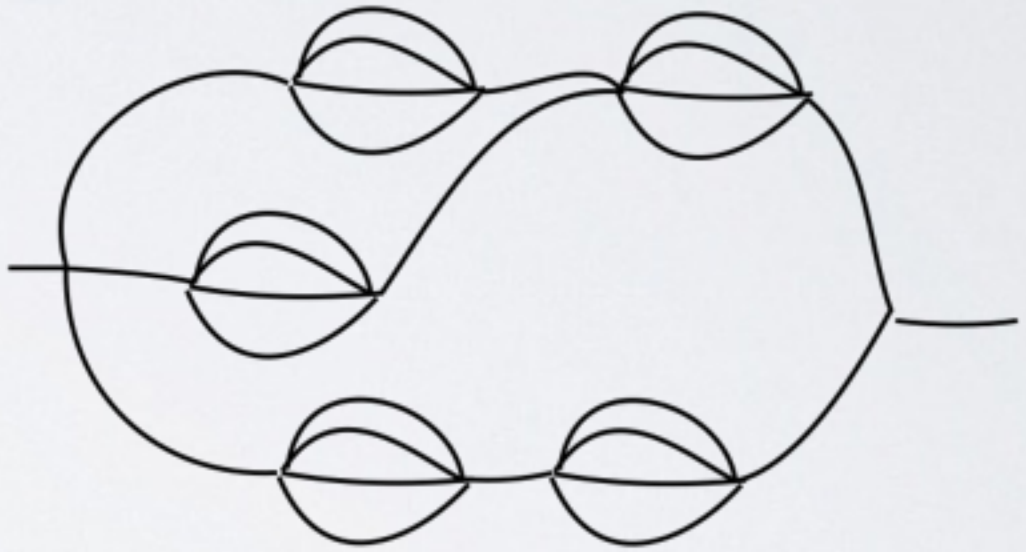
    In single threaded portions — CPU efficiency is low



- Multiple tracks

    Need to think more about what data is needed when

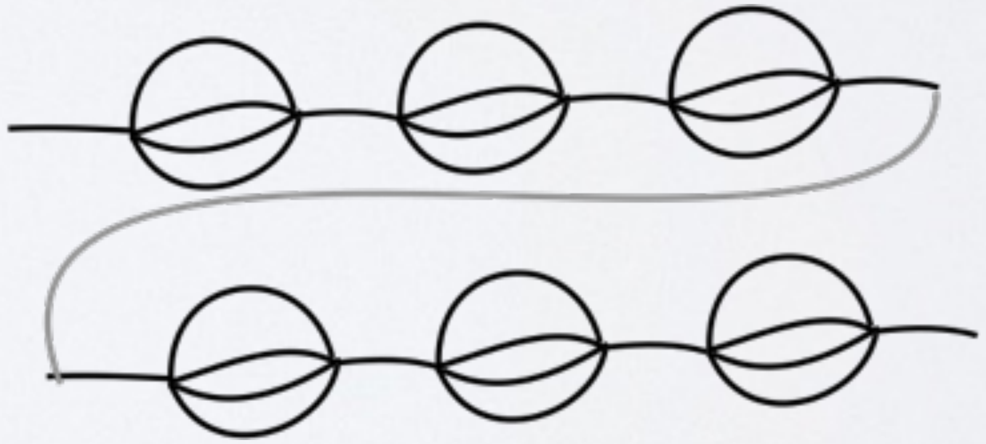
    Job priority may be needed in order to optimize critical path (load dependent)



- Pipelined

    Data takes several frames to move through system

    Higher frame rate, but higher latency



Challenge: Scripting Performance

- Scripting without JIT is slow compared to C++
  - Platforms without JIT are especially slow (PS3, X360, iOS)
  - Gameplay is inherently slow (access patterns)
- Scripting is tricky to multithread
  - Multithreading is *hard* — tricky for gameplay programmers
  - Lua does not natively support multithreading

Challenge: Multi-Threading Scripting

- Actor model
    - Separate Lua state per job thread
    - Lua states communicate with messages
    - No shared data between Lua states
        - More memory use
- How do the states communicate with the engine
    - Need to expose a "multithread-safe" interface
    - Probably using some sort of double buffering of data
    - Read reads constant data from last frame, write adds action to queue
- Complicated — but may be necessary as cores increase

Good Workflows

- Games are expensive to make
  - Lots of people, lots of time
  - Make sure game development is *efficient*
- Key concepts
  - Make it easy to run the *actual game*
  - On the *actual hardware*
  - Make it easy to *iterate over content*, tweak properties and make changes
  - Make it easy to *collaborate*

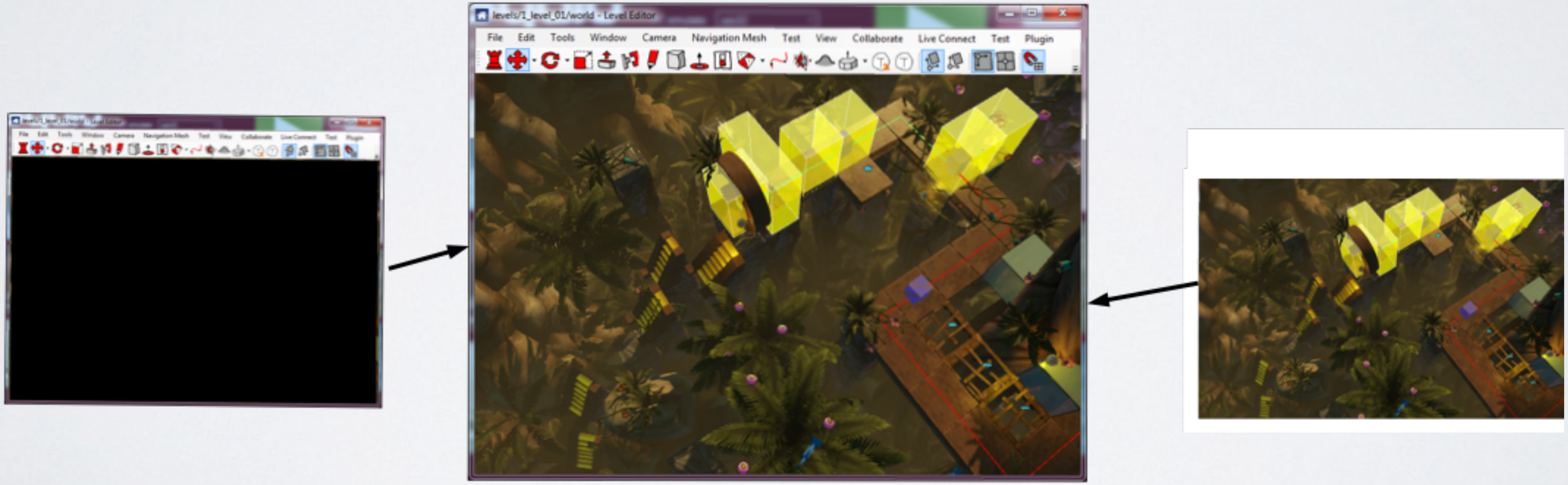
Tool Architecture

- Don't want tools too tightly integrated with the engine

  Tools are written in C# (or any other language)

  Creates a sub window and launches the engine there (as a separate process)

  Communicates with the engine over TCP/IP



What You See is What You Get

- Since engine code and tool code is separated we can run the engine on a separate machine

    Which can be a console or a mobile phone

    Communicate with TCP/IP as if it was a local machine

- See exactly what the game will look like on the final hardware

    On multiple machines at once

Drop-in Play

- From the level editor you can play F5 to immediately play the game
- Engine is already running
  - Cleanly unload all the editor stuff and bring the engine back to where it was just after boot
  - Load the gameplay code and the gameplay stuff
  - You are now playing the game
- Works on all connected machines — immediately try out the game
- Press F5 to go back and forth between editing and playing

Short Iteration Times

- Everything in the engine is reloadable

  This includes shaders, render configurations, etc

  This also includes the Lua gameplay script

  Reload is triggered by a command over the TCP/IP connection

- Optimized process

  Only the changed data is recompiled and reloaded

  See the changes in-game in < 1 second

  On all platforms

- No complicated build process to play the game on different hardware

Collaboration

- Custom merge tool for Json data files

    Can merge changes from different users without destroying the file

    No need for “file locking” — people can work independently and merge their results

- Collaboration in level editor

    Users can share their editing session and others can join

    Edit the world together

Challenge: Tool Extensibility

- How can we make it easy for users to extend tools with custom functionality?
- Tricky
  - Multiple languages: C# (tool), Lua (engine part of tool code), C++ (engine, data compiler)
  - TCP/IP interface must be managed to sync data between tool and engine
  - Quite complicated to get into
- Current experiment: Lua extensions
  - Two Lua files, one extends the editor (through NLua) and one the engine

Challenge: Tool Build Effort

- We spend *a lot of time* on building tools

    Building tools is *more expensive* than adding engine features

    As much code in the tools as in the engine

    And we only support Win32 for the tools

- Are we doing something wrong?

- Or should we just accept that building good tools is expensive?

    Copy/paste, undo, drag & drop, interface design, backwards compatibility, etc…

Questions / Comments