Flexible Rendering for Multiple Platforms

tobias.persson@bitsquid.se

bitsquid

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9
2012

Breakdown

- Introduction
- Bitsquid Rendering Architecture
- Tools

Bitsquid

- High-end game engine for licensing
- Multi-platform: PC, MAC, PS3, X360, High-end mobile
- Currently powering 10 titles in production
  - Production team sizes 15-40 developers

Bitsquid

- Key design principles
  - Simple & lightweight code base (~200KLOC)
    - Including tools
  - Heavily data-driven
  - Quick iteration times
  - Data-oriented design

- Highly flexible…

“War of the Roses”
Courtesy of Fatshark and Paradox Interactive

“War of the Roses”
Courtesy of Fatshark and Paradox Interactive

"Krater"
Courtesy of Fatshark

“Krater”
Courtesy of Fatshark

“The Showdown Effect”
Courtesy of Arrowhead Game Studios & Paradox Interactive

“Hamilton’s Great Adventure”
Courtesy of Fatshark


“Stone Giant”
DX11 tech demo

Flexible rendering

- Bitsquid powers a broad variety of game types
  - Third-person, top-down, 2.5D side-scrollers and more
- Different types of games can have very different needs w.r.t rendering
  - 30Hz vs 60Hz
  - Shading & Shadows
  - Post effects, etc..
- Game context aware rendering
  - Stop rendering sun shadows indoors, simplified rendering in split-screen

Flexible rendering

- Also need to run on lots of different HW-architectures
- Cannot abstract away platform differences, we need stuff like:
  - Detailed control over EDRAM traffic (X360)
  - SPU offloading (PS3)
  - Scalable shading architecture (forward vs deferred, baked vs real-time)
- What can we do?
  - Push the decisions to the developer!
    - But, make it as easy as possible for them…

Data-driven renderer

- What is it?
  - Shaders, resource creation / manipulation and flow of the rendering pipe defined entirely in *data*
- In our case *data* == *json* config files
  - Hot-reloadable for quick iteration times
  - Allows for easy experimentation and debugging

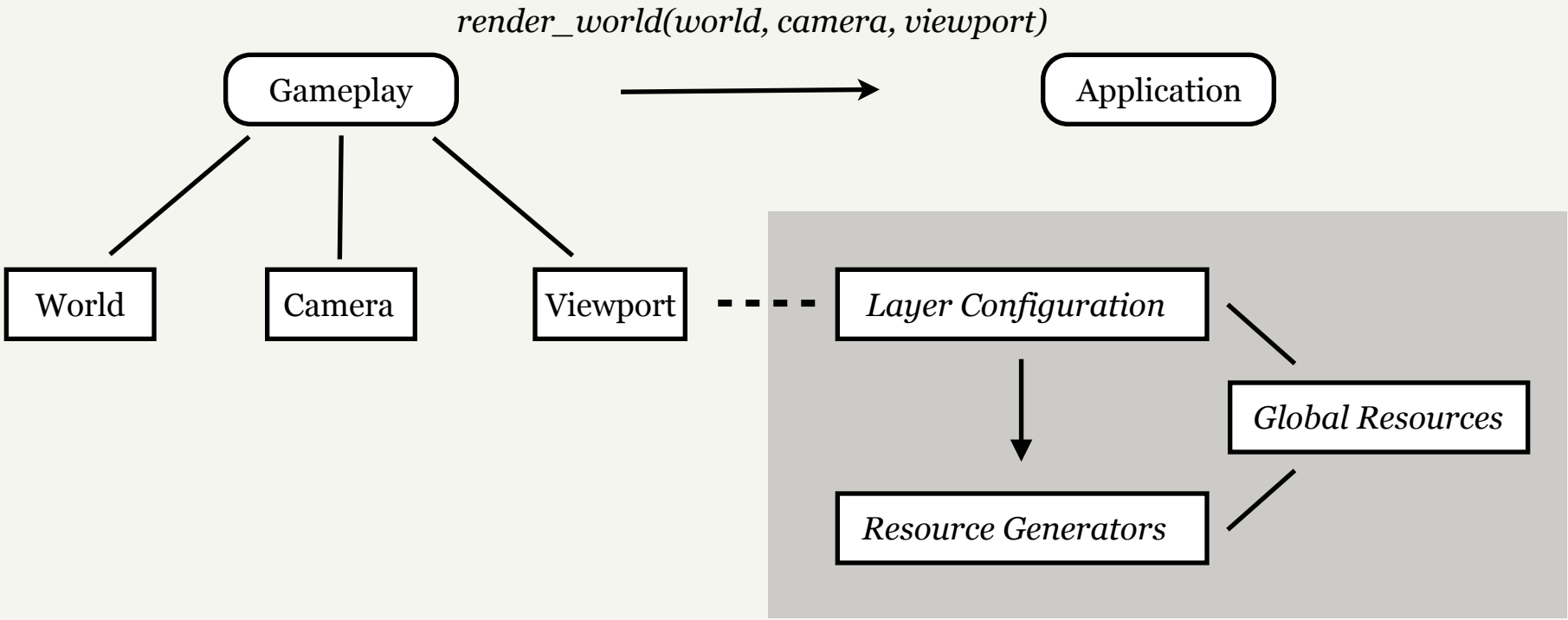
Meet the *render_config*

- Defines simple stuff like
  - Quality settings & device capabilities
  - Shader libraries to load
  - Global resource sets
    - Render Targets, LUT textures & similar
- But it also drives the entire renderer
  - Ties together all rendering sub-systems
  - Dictates the flow of a rendered frame

Gameplay & Rendering

- GP-layer gets callback when it's time to render a frame
  - Decides which Worlds to render
  - What Viewport & Camera to use when rendering the World
- GP-layer calls Application:render_world()
  - Non-blocking operation – posts message to renderer
  - Renderer uses its own world representation
    - Don't care about game entities and other high-level concepts
    - State changes pushed to state reflection stream

Gameplay - Renderer Interaction



Layer Configurations

- Dictates the final ordering of batch submits in the render back-end
- Array of layers, each layer contains
  - Name – used for referencing from shader system
    - Shader dictates into which layer to render
  - Destination RTs & DST
  - Batch sorting criteria within the layer
  - Optional *Resource Generator* to run
  - Optional Profiling scope
- Layers are rendered in the order they are declared

A Simple Layer Configuration

```
simple_layer_config = [
   // Populate gbuffers
   { name = "gbuffer" render_targets="gbuffer0 gbuffer1" depth_stencil_target="ds_buffer"
      sort="FRONT_BACK" profiling_scope="gbuffer"}

   // Kick resource generator ‘linearize_depth’
   { name = "linearize_depth" resource_generator = "linearize_depth"
      profiling_scope="lighting&shadows" }

   // Render decals affecting albedo term
   { name = "decal_albedo" render_targets="gbuffer0" depth_stencil_target="ds_buffer"
      sort="BACK_FRONT" profiling_scope="decals"}

   // Kick resource generator ‘deferred_shading’
   { name = "deferred_shading" resource_generator = "deferred_shading"
      profiling_scope="lighting&shadows" }
]
```

Resource Generators

- Minimalistic framework for manipulating GPU resources
  - Array of Modifiers
  - A Modifier can be as simple as a callback function provided with knowledge of when in the frame to render
  - Modifiers rendered in the order they are declared
- Used for post processing, lighting, shadow rendering, GPU-driven simulations, debug rendering, etc..

A simple Modifier: fullscreen_pass

- Draws a single triangle covering entire viewport
- Input: shader and input resources
- Output: Destination render target(s)

```
// Example of a very simple resource generator using a single modifier (fullscreen_pass)
linearize_depth = [
   // Converts projected depth to linear depth
   { type="fullscreen_pass" shader="linearize_depth" input="ds_buffer" output="d32f" }
]
```

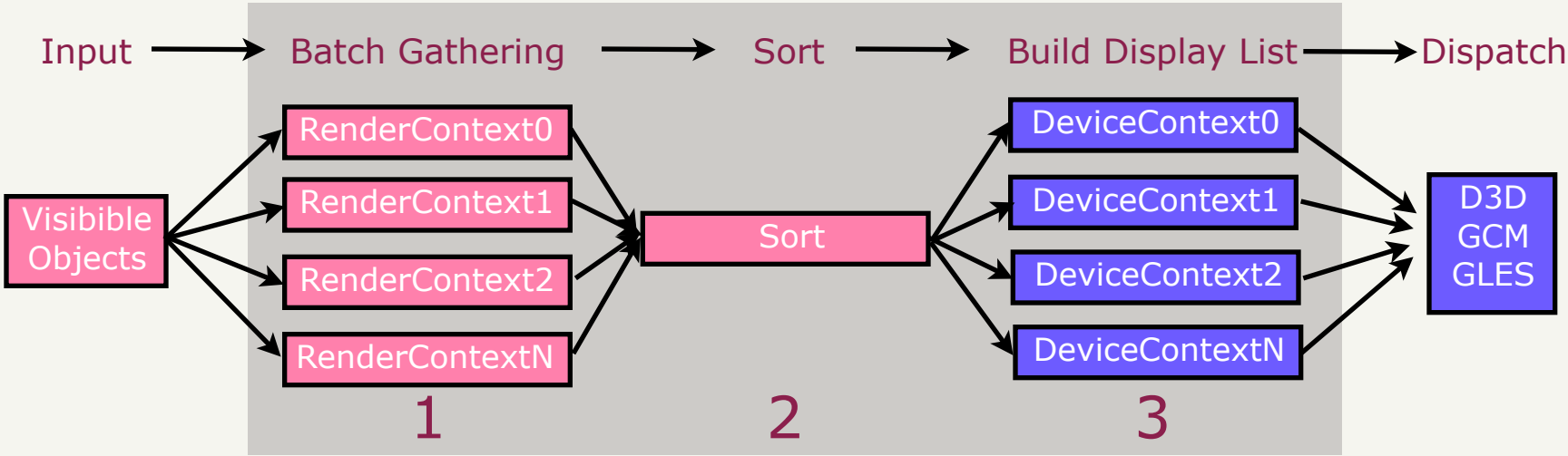
More Modifiers

- Bitsquid comes with a toolbox of different Modifiers
  - shadow_mapping, deferred_shading, compute_kernel (dx11), edram_control (x360), spu_job (ps3), mesh_renderer, branch, loop, generate_mips, and many many more..
- Very easy to add your own..

A peek under the hood

Parallel rendering

- Important observation: only ordering we care about is the final back-end API calls
- Divide frame rendering into three stages



Batch Gathering

- Output from View Frustum Culling is a list of renderable objects

```
struct Object {
   uint type; // mesh, landscape, lod-selector etc
   void *ptr;
};
```

- Sort on type
- Split workload into *n*-jobs and execute in parallel
  - Rendering of an object does not change its internal state
  - Draw-/state- commands written to *RenderContext* associated with each job

RenderContext

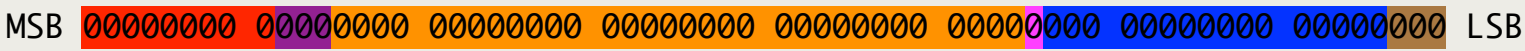
- A collection of helper functions for generating platform independent draw/state commands
- Writes commands into an abstract data-stream (raw memory)
  - When command is written to stream it's completely self-contained, no pointer chasing in render back-end
  - Also supports platform specific commands
    - e.g. DBT, GPU syncing, callbacks etc

Command Sorting

- Each command (or set of commands) is associated with a SortCmd stored in separate “sort stream”

```
struct SortCmd {
   uint64 sort_key;
   uint offset;
   uint render_context_id;
};
```

64-bit Sort Key Breakdown



9 Layers bits (Layer Configuration)
3 Deferred Shader Passes bits (Shader System)
32 User Defined bits (Resource Generators)
1 Instance Bit (Shader Instancing)
16 Depth Bits (Depth sorting)
3 Immediate Shader Passes bits (Shader System)

Dispatch RenderContexts

- When all RenderContexts are populated
  - "sort-streams" are merged and sorted
    - Not an insane amount of commands, we run a simple std::sort
  - Sent to render back-end
- Back-end walks over sort-stream and translates the RC commands into graphics API calls
- If graphics API used supports building "display lists" in parallel we do it

Tools

Tools Architecture

- Avoids strong coupling to engine by forcing all communication over TCP/IP
  - Json as protocol
- All visualization using engine runtime
  - Boot engine running tool slave script (LUA)
  - Tool sends window handle to engine, engine creates child window with swap-chain
  - Write tools in the language you prefer

Editor Mirroring

- Decoupling the engine from the tools is great!
  - Better code quality - clear abstraction between tool & engine
  - If engine crashes due to content error - no work is lost
    - Fix content error & reboot exe - tool owns state
- Strict decoupling allows us to run all tools on all platforms
  - Cross-platform file serving from host PC over TCP/IP
  - Quick review & tweaking of content on target platform

Tool slaving

- Running level editor in slave mode on Tegra 3

Working with platform specific assets

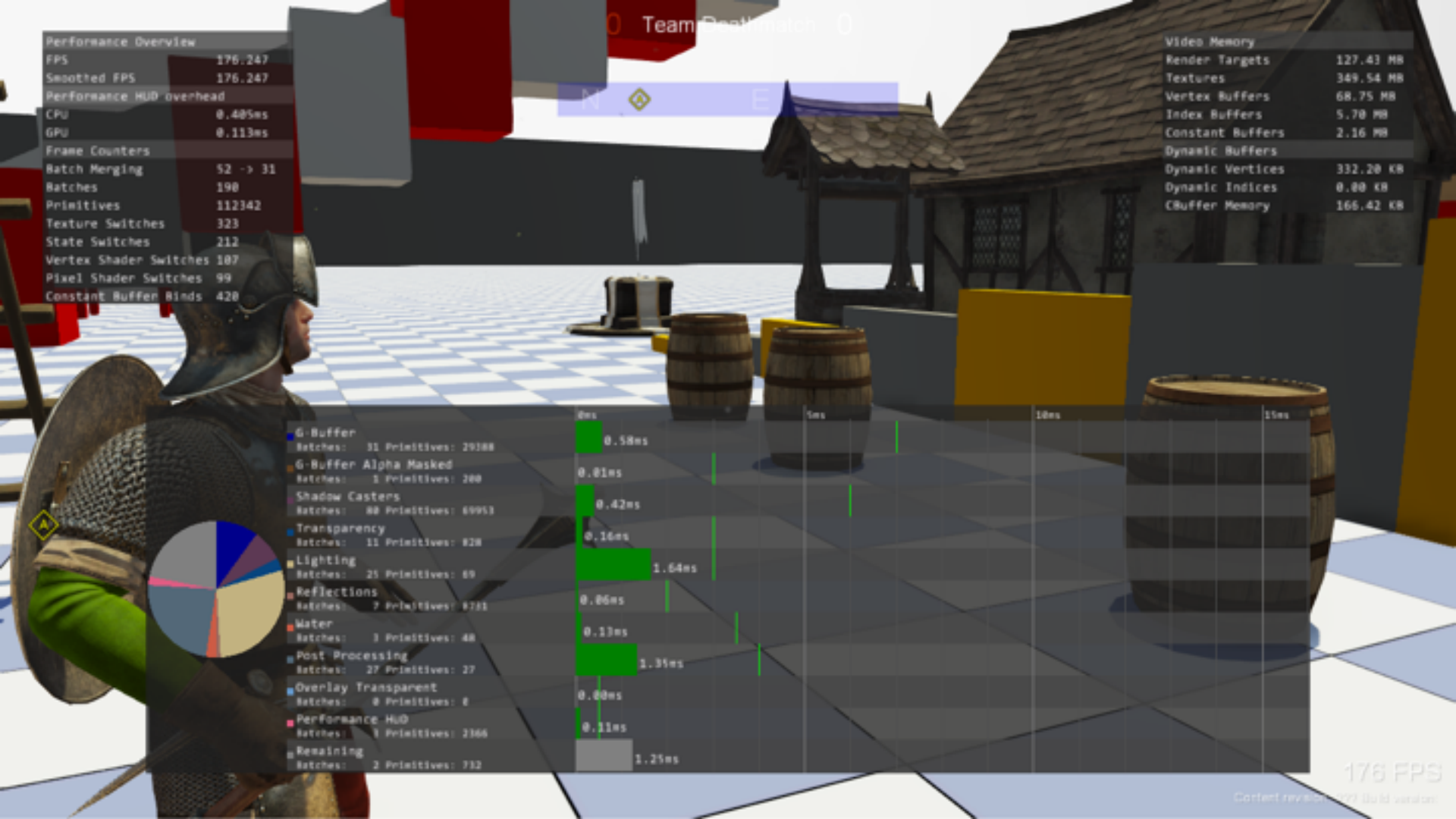
- To make a resource platform specific - add the platform name to it's file extension
  - cube.unit -> cube.ps3.unit
- Data Compiler takes both *input* and *output* platform as arguments
  - Each resource compiler knows if it can cross-compile or not
- Allows for easy platform emulation
  - Most common use case: run console assets on dev PC
  - Also necessary if you need to do any kind of baking.

Profiling Graphics

- Artist friendly profiling of graphics is hard
  - Context dependent
    - That über-model with 300 material splits skinned to 600+ bones might be fine - *if it's only one instance in view!*
    - That highly-unoptimized-super-complicated shader won't kill your performance - *if it only ends up on 5% of the screen pixels!*
  - Can make sense to give some indication of how "expensive" a specific shader is
    - But what to include? Instruction count? Blending? Texture inputs?
- We don't provide any preventive performance guiding
  - Would like to - but what should it be?

Artist Performance HUD

- Graphics profiler scopes defined in Layer Configuration & Resource Generators
  - Start / Stop profiling commands in RenderContext
    - Batch count, triangle / vertices count, state switches
    - GPU timing using D3D11_QUERY_TIMESTAMP
- Artist friendly in-game HUD with break-down of frame
  - Summery of artist relevant profiler scopes
  - Config data-driven


Performance Overview
FPS 176.247
Smoothed FPS 176.247
Performance HUD overhead
CPU 0.405ms
GPU 0.113ms
Frame Counters
Batch Merging 52 -> 31
Batches 190
Primitives 112342
Texture Switches 323
State Switches 212
Vertex Shader Switches 107
Pixel Shader Switches 99
Constant Buffer Binds 420
Video Memory
Render Targets 127.43 MB
Textures 349.54 MB
Vertex Buffers 68.75 MB
Index Buffers 5.70 MB
Constant Buffers 2.16 MB
Dynamic Buffers
Dynamic Vertices 332.20 KB
Dynamic Indices 0.00 KB
CBuffer Memory 166.42 KB
G-Buffer
Batches: 31 Primitives: 29188
G-Buffer Alpha Masked
Batches: 1 Primitives: 200
Shadow Casters
Batches: 80 Primitives: 69953
Transparency
Batches: 11 Primitives: 828
Lighting
Batches: 25 Primitives: 69
Reflections
Batches: 7 Primitives: 9781
Water
Batches: 1 Primitives: 48
Post Processing
Batches: 27 Primitives: 27
Overlay Transparent
Batches: 0 Primitives: 0
Performance HUD
Batches: 1 Primitives: 2366
Remaining
Batches: 2 Primitives: 732
0ms
5ms
10ms
15ms
0.58ms
0.01ms
0.42ms
0.16ms
1.64ms
0.06ms
0.13ms
1.35ms
0.00ms
0.11ms
1.25ms
176 FPS

Conclusions

- It's all about workflows
- A data-driven rendering pipe will
  - Drastically increase your productivity
    - Easy to try out new techniques
    - Simple to debug broken stuff
  - Keep your engine code clean, render system coupling in data
- Clear separation between engine and tools makes
  - Your tools more stable
  - It easy to run your entire tool chain on multiple platforms

Thank you! Questions?

- tobias.persson@bitsquid.se
- @tobias_persson

- slides -> www.bitsquid.se

Bonus Slides

Quick note on shader authoring

- Shaders authored in our in-house meta-language
  - Shader snippets in HLSL/Cg & GLSL
  - Über-shader approach, pre-processor branching and snippet combining
    - Shader permutations needed dictated by project material files
- Works but excludes artist from doing easy R&D :(
- Future: Shader-graph tool that ties in with Resource Generators framework
  - Super powerful but can be problematic from a performance perspective