THE ASSET BUILD SYSTEM OF FARCRY5

Rémi QUENIN
Engine Architect – Far Cry
Ubisoft Montréal
@azagoth

THE FARCRY PROJECT

- About 1000 people at peak
  Across 5 studios
- Over 3 Years
  Started after FC4
- Gigantic open world, 2x FC4/Primal
  80 km2 playable

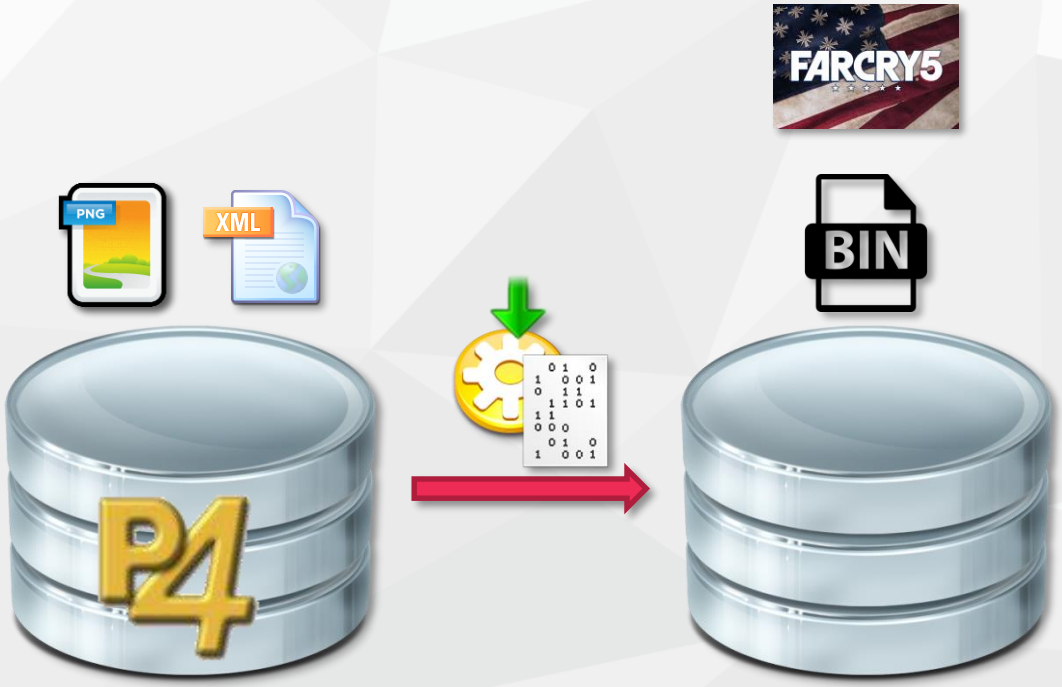
UBISOFT

THE FARCRY PIPELINE

- 1 nightly build = 560 GB
  3 Platforms, Official + Test Maps
- Over 4.5 TB of I/O to produce a build
  Read+Write, 3 platforms
- Over 20 Millions node in dep. graph
  Spread in more than 120 asset types

UBISOFT

DATA FLOW IN THE FARCRY PIPELINE



UBISOFT

BINARIZING *Assets*

1. THE PIPELINE
2. THE ASSET BUILD SYSTEM
3. FEATURES

Runtime & Tool Separation

THE PIPELINE

FC4 PIPELINE

- Fair amount of C++ tools code
  Mixed C++/C# with CLI
- Strong tools/runtime coupling
  Edition code "sneak" into engine / final game
- Big monolithic editor
  Long load time

UBISOFT

FC5 PIPELINE

- Modular specialized tools
  Few dependencies, fast loading
- Clean data separation
  Tools data != RunTime data
- No direct dependencies on engine
  Focus on user experience

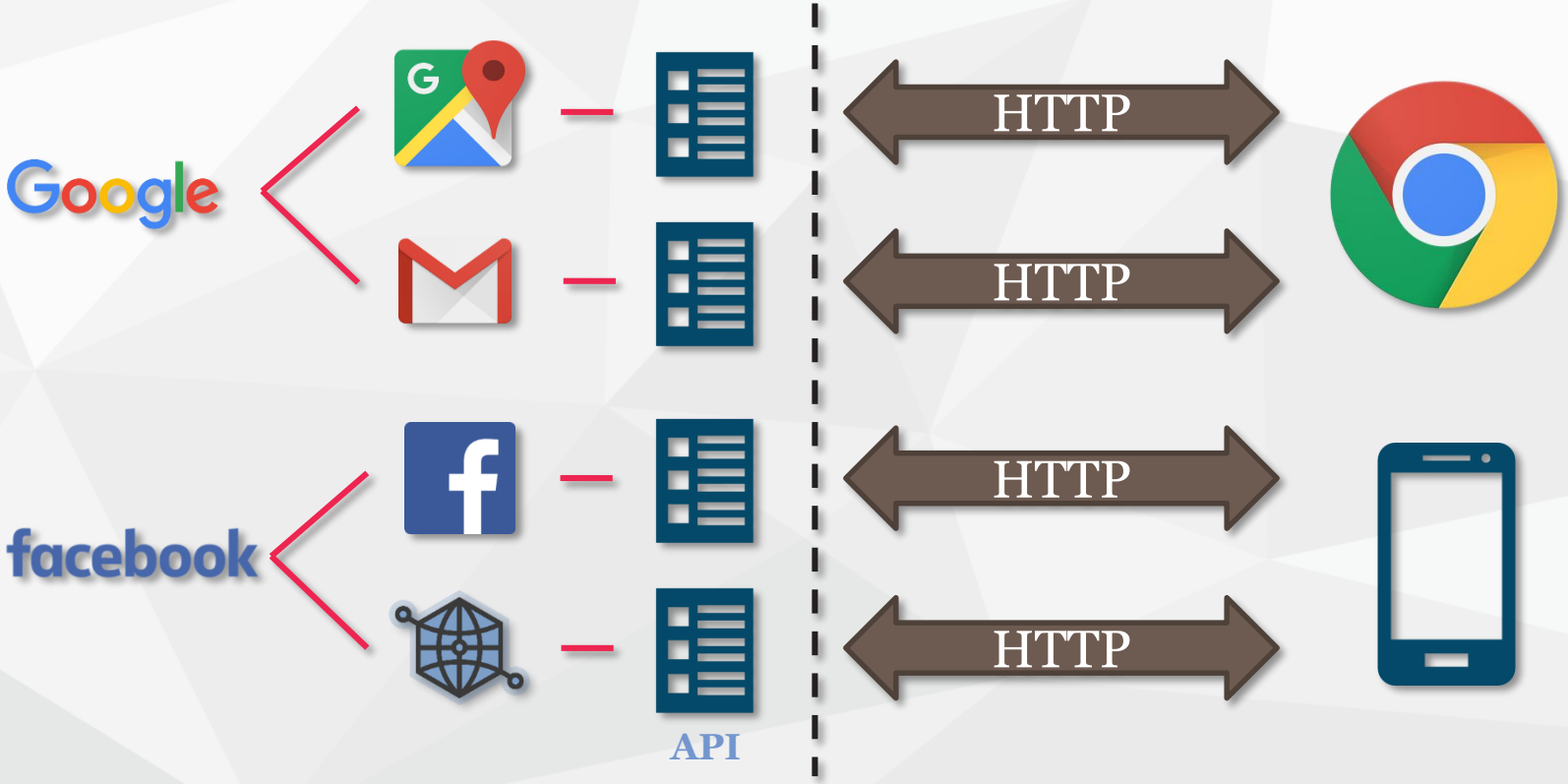
UBISOFT

HOW ?

Engine as Service

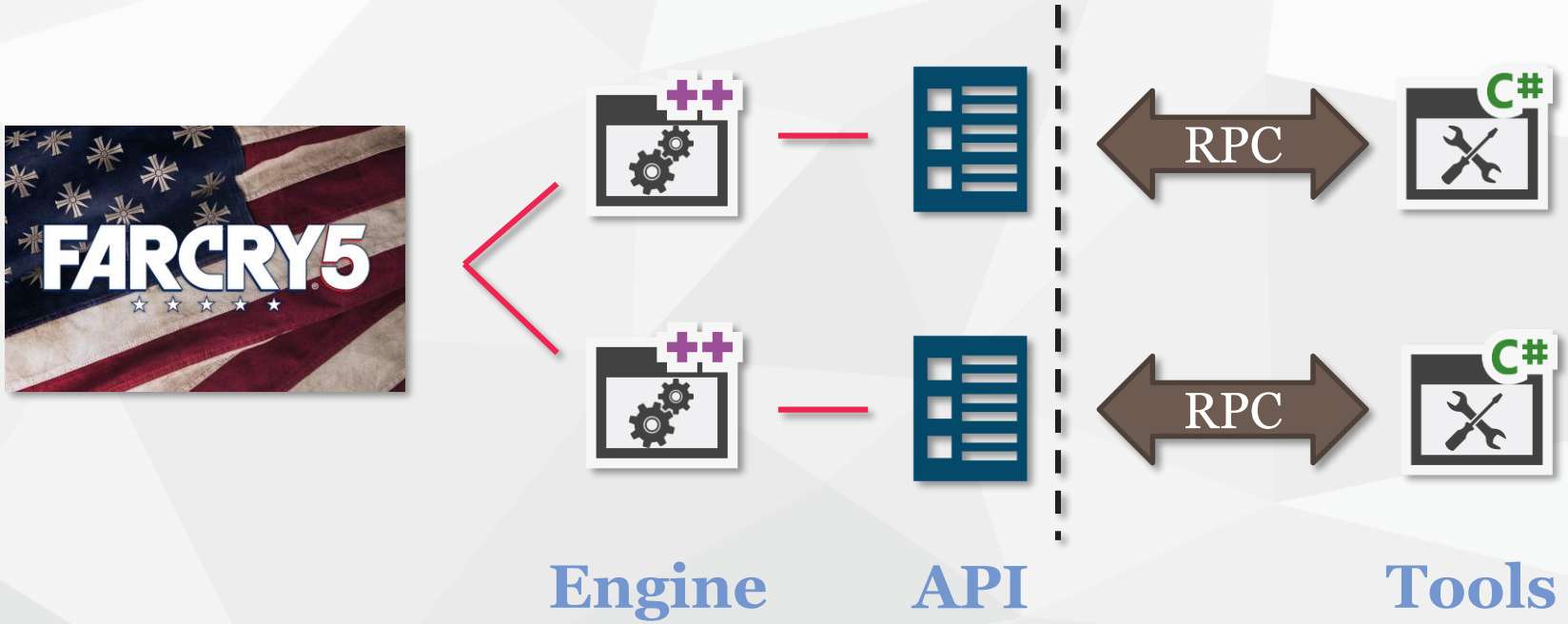
UBISOFT

INSPIRATION



UBISOFT

ENGINE AS SERVICE



UBISOFT

BENEFITS

- Separation of concerns
  Proper dependencies
- Use the right technologies
  C++ for engine, C# for tools
- Fast dev. and iteration
  Key for quality

UBISOFT

FEEDING DATA TO THE ENGINE

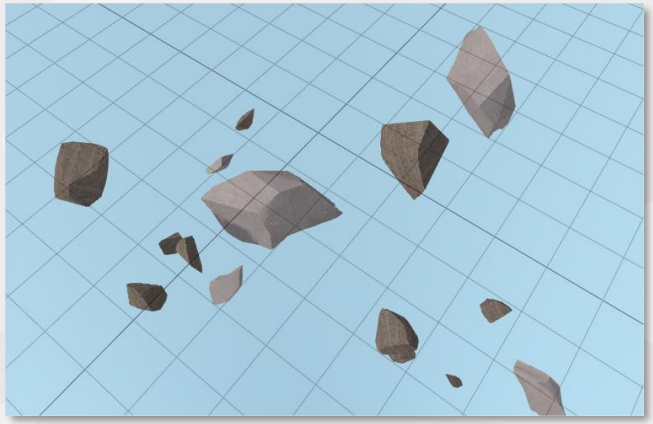
- Engine only consume binarized data
  No code path for "tool" data
- Binarization delegated to the AssetBuildSystem
  Separation of concerns
- Just in Time "JIT" compilation
  Only if necessary

UBISOFT

EXAMPLE

Tool.cs

```
var engine = new Engine();
var particle = engine.Create<IParticleSystem>();
particle.Spawn(“Rock.Destruction”);
```

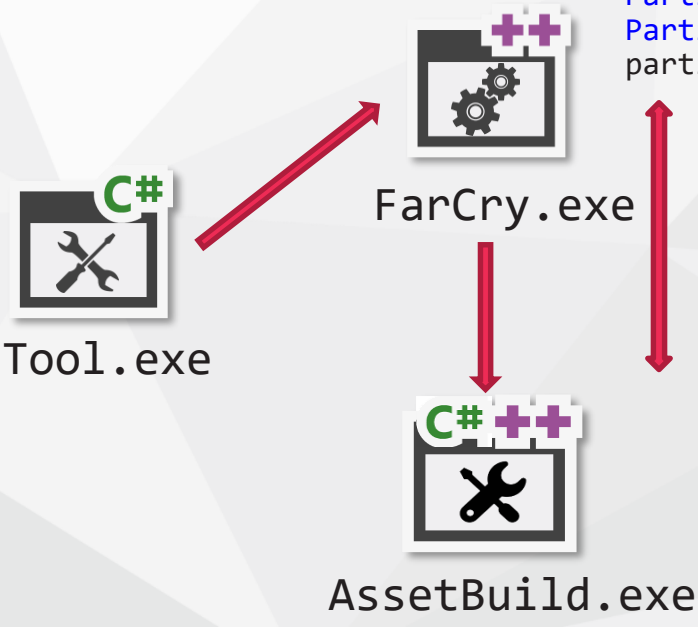


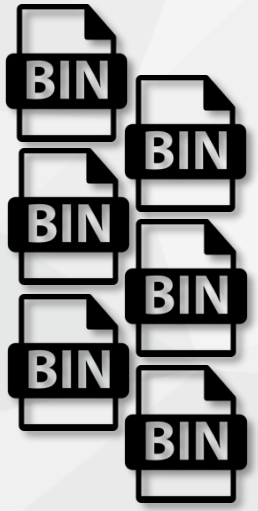
UBISOFT

EXAMPLE

```
var engine = new Engine();
var particle = engine.Create<IParticleSystem>();
particle.Spawn(“Rock.Destruction”);
```

```
ParticleSystem* particle = new ParticleSystem();
ParticleDesc* desc = FileSystem::Stream(“Rock.Destruction”);
particle->Spawn(desc);
```



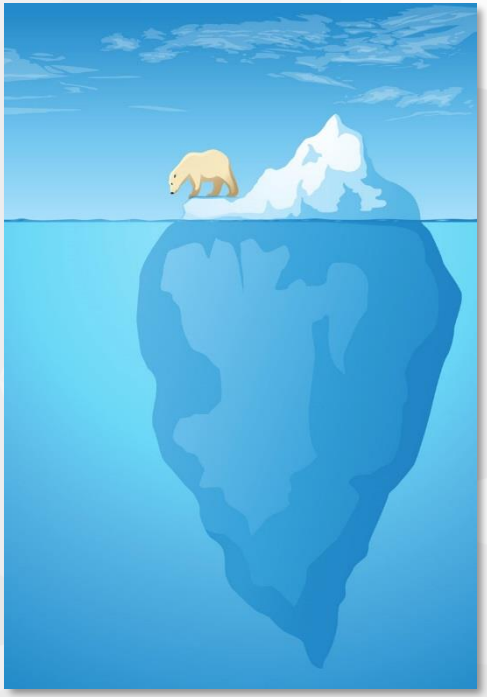


- Particle
- Sounds
- Textures
- …etc.

UBISOFT

USING ENGINE AS A SERVICE

- Asset Preview
- Asset Editor
- Augmented debugging
- Automated testing
- Tech. prototyping
- …and More



UBISOFT

FC5 PIPELINE

3 POINTS

RUNTIME USED AS SERVICE
No « tool » code shipped

INDEPENDENT MODULAR TOOLS
Fast iteration

ASSET BUILD SYSTEM
Junction between both

UBISOFT

2

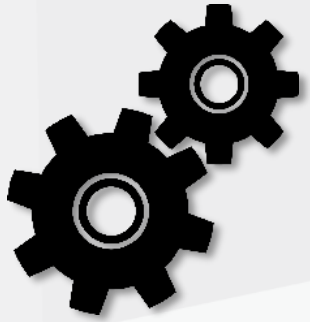
Fast and scalable

THE ASSET BUILD SYSTEM

WHAT IS THE PURPOSE OF A BUILD ?

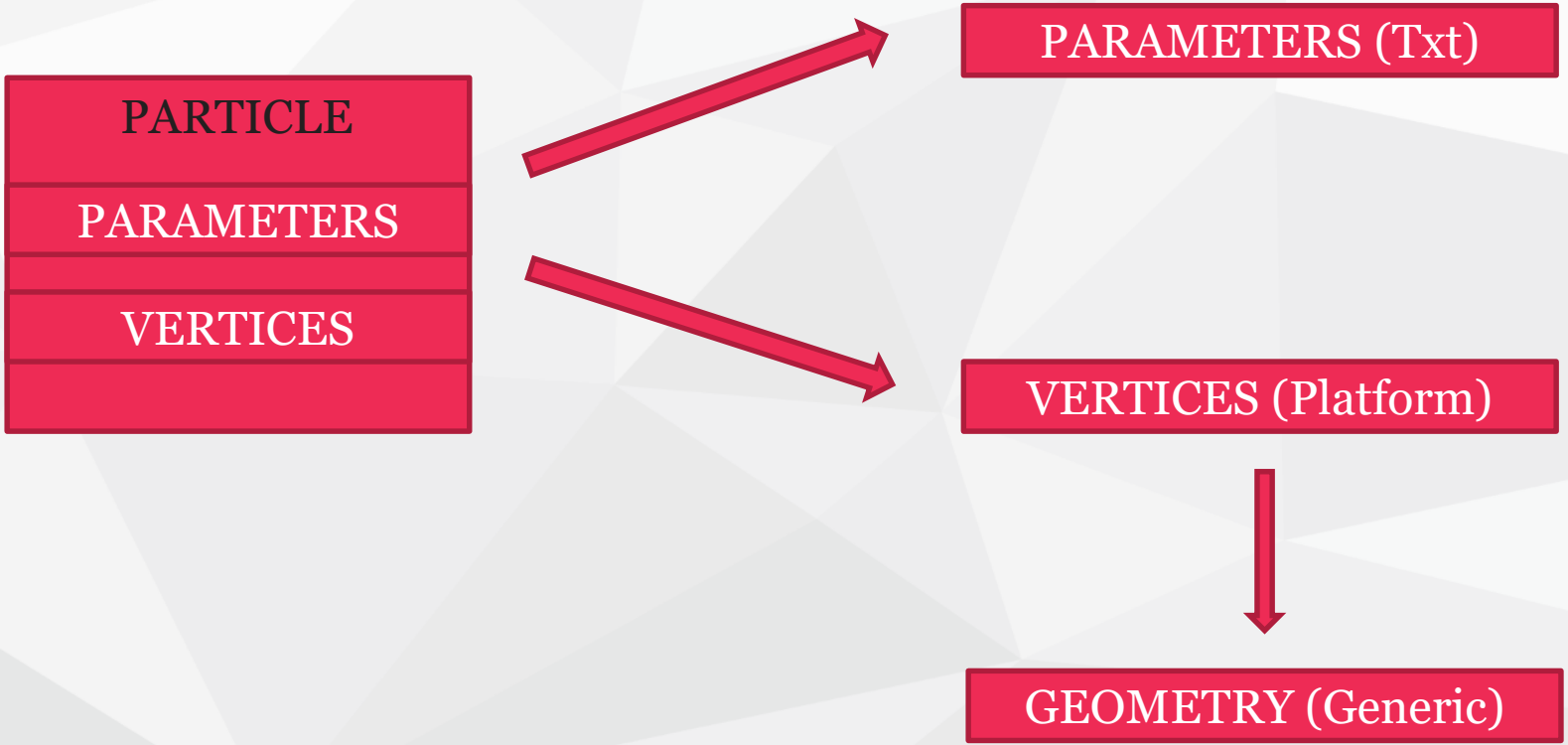
- Source asset to RunTime optimized asset
  Binary format, strip useless info

- Faster load, platform specific
  Best possible result at runtime



UBISOFT

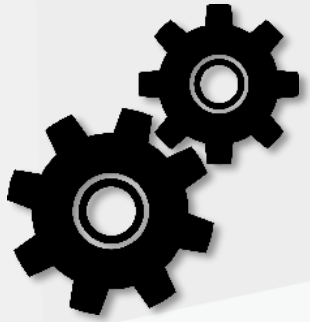
WHAT IS A BUILD ?



WHAT IS A BUILD ?

- A set of independent build actions
  Has dependencies, can produce output

- Graph of build actions
  Dependencies needs to satisfied before processing node



UBISOFT

DEPENDENCIES

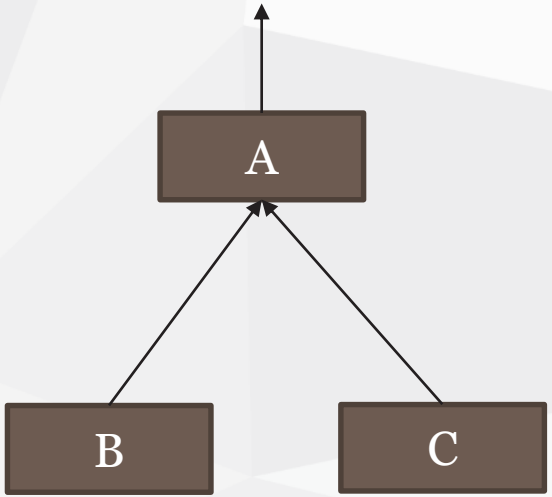
DEFINES

THE *SPEED*

OF THE BUILD

WHAT IS A BUILD ?

- Want to Build asset “A”
  - List dependencies
  - Build dependencies
  - Build
  - Notify dependencies

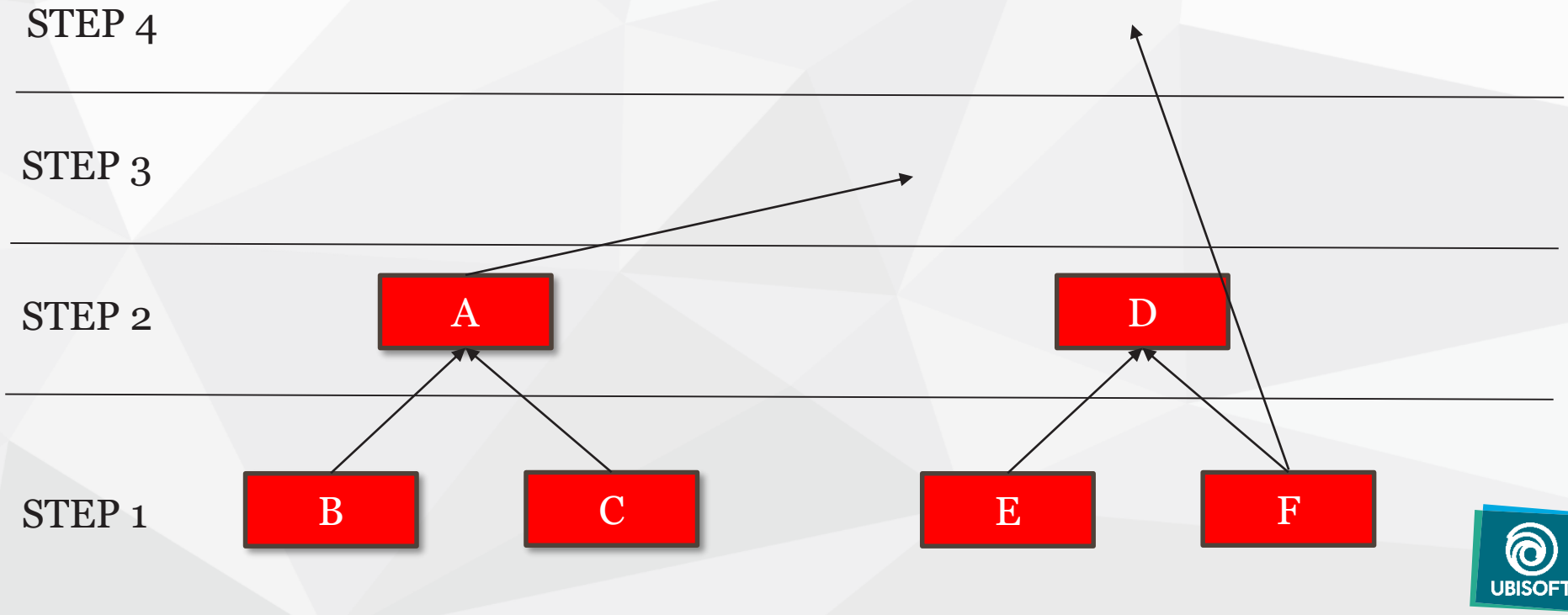


UBISOFT

BUILD DEPENDENCIES: PARALLELIZATION

UBISOFT

BUILD DEPENDENCIES: PARALLELIZATION



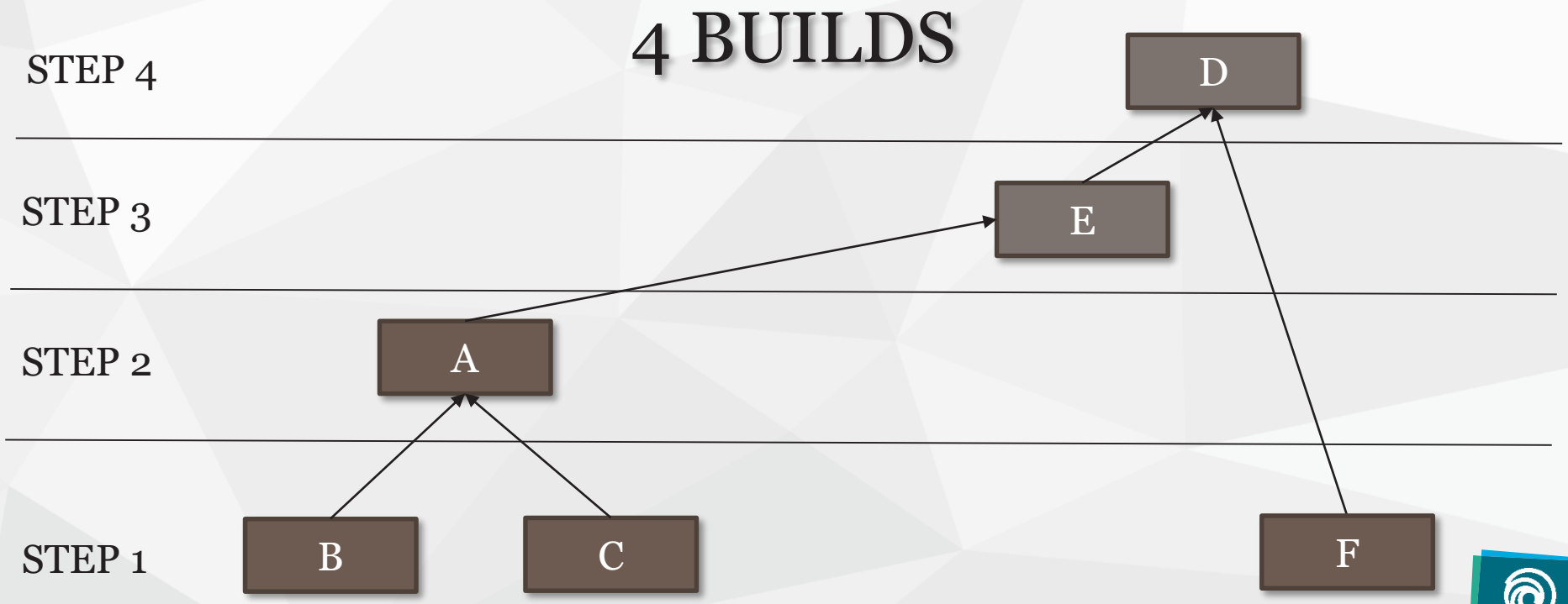
UBISOFT

BUILD DEPENDENCIES: INCREMENTAL

2 BUILDS

UBISOFT

BUILD DEPENDENCIES: INCREMENTAL



DEPENDENCIES

DEFINES

THE *SPEED*

OF THE BUILD

DEPENDENCIES MAKE IT FAST

1. Parallelization
   For non-dependent rules

2. Incrementality
   Just do what's need to be done

UBISOFT

DEPENDENCIES

DEPENDENCIES

DEPENDENCIES

UBISOFT

DEPENDENCIES

COMPILE DEP
Required to build

RUNTIME DEP
Required to Run

UBISOFT

1 COMPILE DEPENDENCIES

- Static
  Can be listed upfront
- Dynamic
  Needs analysis of the static deps

UBISOFT

1 COMPILE DEPENDENCIES: EXAMPLES

Code object file (.obj)

- Static
  cpp file
- Dynamic
  Header files included

Particle

- Static
  Definition file
- Dynamic
  Vertex buffer

UBISOFT

1 COMPILE DEPENDENCIES: EXAMPLES

Texture

- Static
  .png file
- Dynamic
  Texture profile

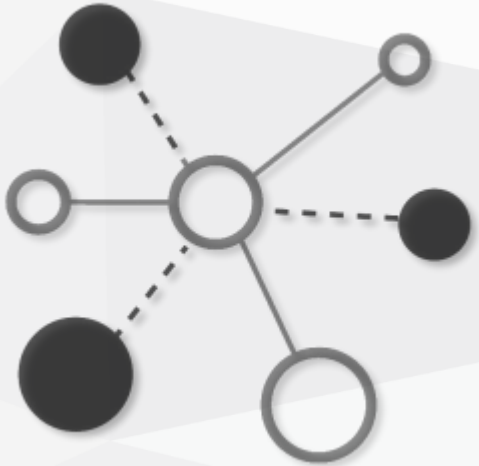
Animation

- Static
  Anim. source file
- Dynamic
  Skeleton

UBISOFT

2 RUNTIME DEPENDENCIES

- Required to run, not to build: “Reference”
  Discovered during build
- Emitted during the build
  Weak link: does not block emitting graph



UBISOFT

2 RUNTIME DEPENDENCIES : EXAMPLES

- Material
  => Textures
- Geometry
  => Materials
- Dialog
  => Sounds, facial animations
- Animation
  => Particles, sounds
- Particles
  => Textures, sounds

UBISOFT

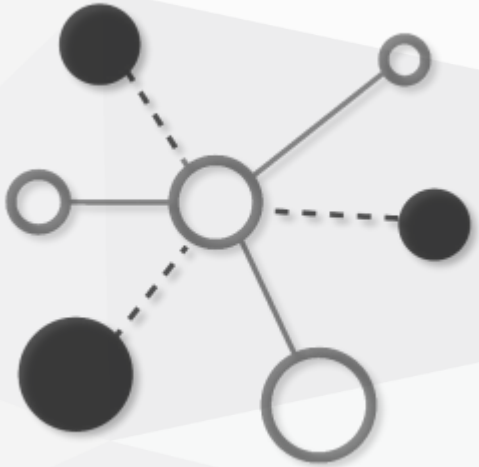
FC5 ASSET BUILD SYSTEM

Prepare **P**latform **D**ata

UBISOFT

PPD: PREPARE PLATFORM DATA

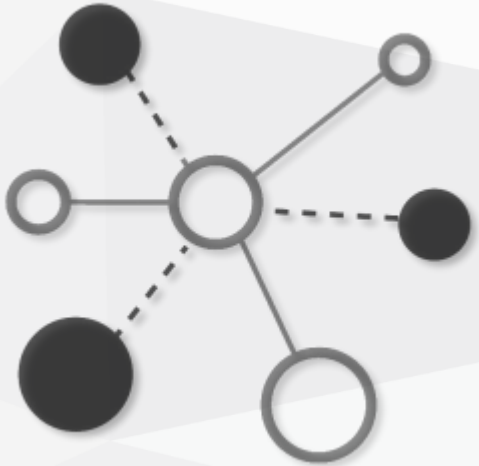
- Dependency graph
  Static/Dynamic compile dep., Runtime dep.
- Everything is a node
  Physical or virtual: asset file, src file, file list…
- Each node is access only once
  Only one I/O op. per file



UBISOFT

PPD: MODULARITY

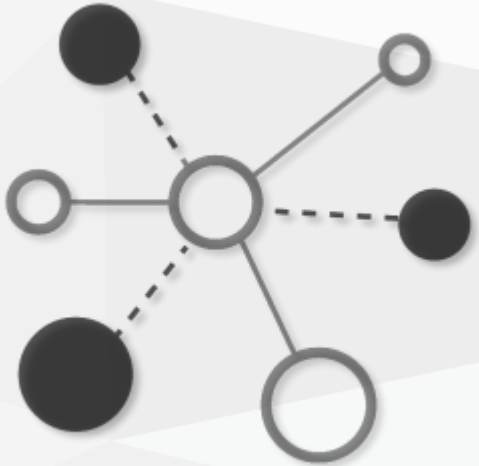
- "Nodes" bound to "Processor"
  "Data" separated form "Processing"
- User only write the "compilation"
  ...and callback to extract dependencies
- Framework does the graph evaluation
  ...and invoke user callbacks when necessary
- Easiness of implementation
  Users are not afraid to add processors



UBISOFT

PPD: REVERSE « USER API »

- “Build this asset”
  ie. : Build this node
- Don’t need to know the asset
  User just invoke ppd.exe on target
- Unroll dependencies
  From target to sources



UBISOFT

DEPENDENCIES

DEFINES

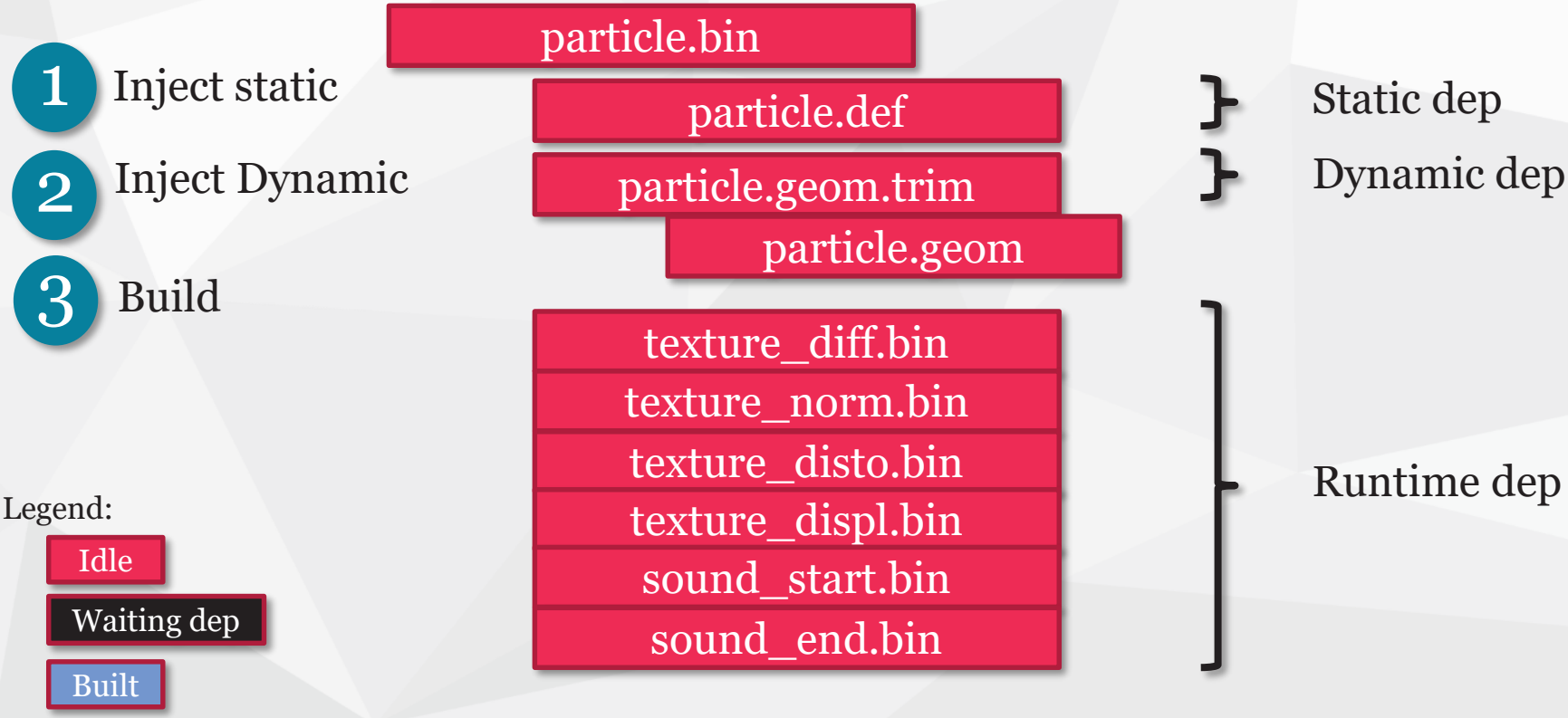
THE *SPEED*

OF THE BUILD

EXAMPLE: PARTICLE

- <particle>.bin: parameters + vertex buffer
  Lifetime, spawn rate, emitting direction & speed…
- Static dep: <particle>.def
  Particle definition
- Dynamic dep: A “trimmed” geometry
  Needs to be built from a full “fat” geometry
- Runtime dep
  Textures and sounds

EXAMPLE: PARTICLE



UBISOFT

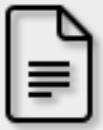
EXAMPLE: SETTINGS

- Set of parameters, edited through a Property Grid
  Typically, the fields of a C++ struct/class instance
- Fields exposed through C++ Reflection
  Lots of online material on the subject
- Can automate serialization
  Many more usage

UBISOFT

EXAMPLE: SETTINGS

```
struct Paramaters
{
  int iVal = 0;
  float fVal = 10.f;
};
```



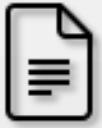
```
<DuniaObject ClassType="Parameters">
  <Member Name="iVal" Type="int" DefaultVal="0"/>
  <Member Name="fVal" Type="float" DefaultVal="10.f"/>
</DuniaObject>
```

UBISOFT

EXAMPLE: SETTINGS

EXAMPLE: SETTINGS

Tool « instance » } Static dep



Definition file } Dynamic dep

FarCry.exe } Static dep

UBISOFT

PPD: VERSIONING

- Each processor has a version number
  Defined in code, saved in state
- At state-load time, discard any changed processors
  Version number, or settings
- Discard all nodes attached to this proc.
  …and recursively discard users of discarded nodes

UBISOFT

DEPENDENCIES

DEFINES

THE *SPEED*

OF THE BUILD

PPD: MULTITHREADING

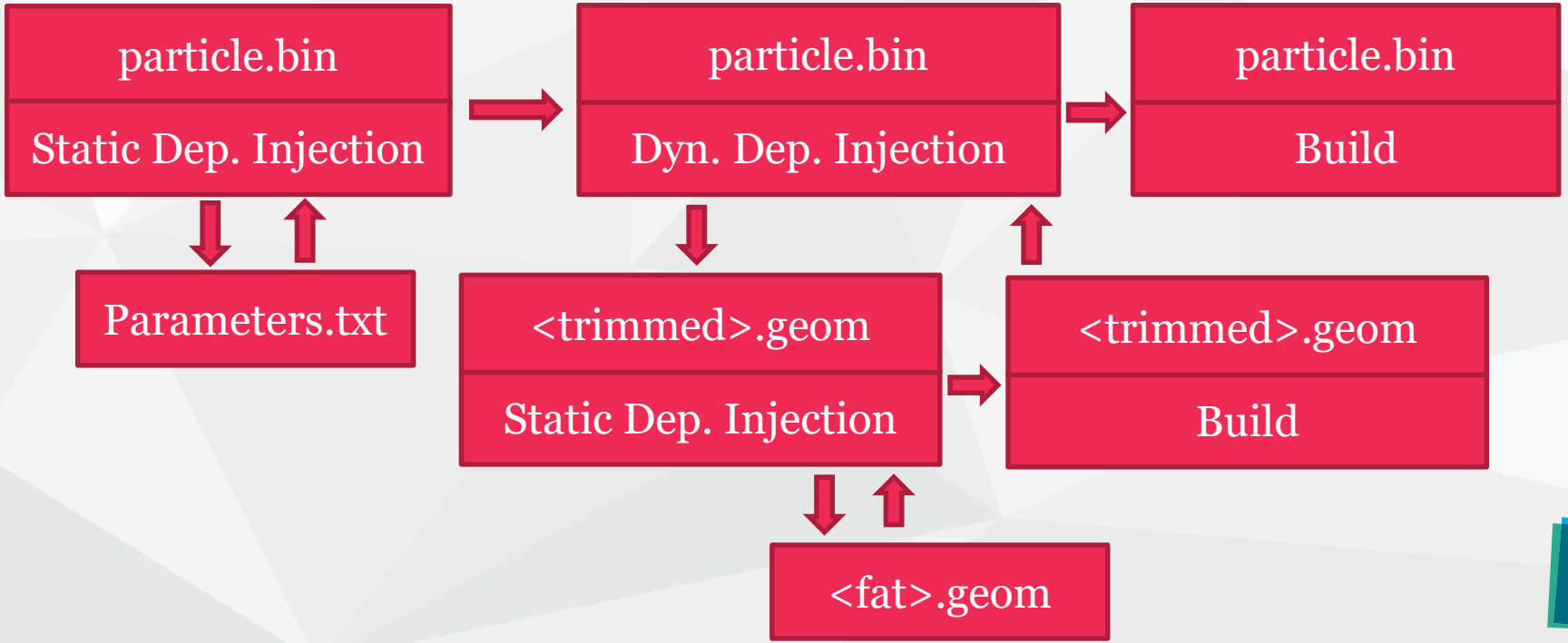
- Every stage is a task: all parallelized
  Static dep, Dynamic dep, Build

- Notification driven
  New task spawned upon completion of last dependent one

UBISOFT

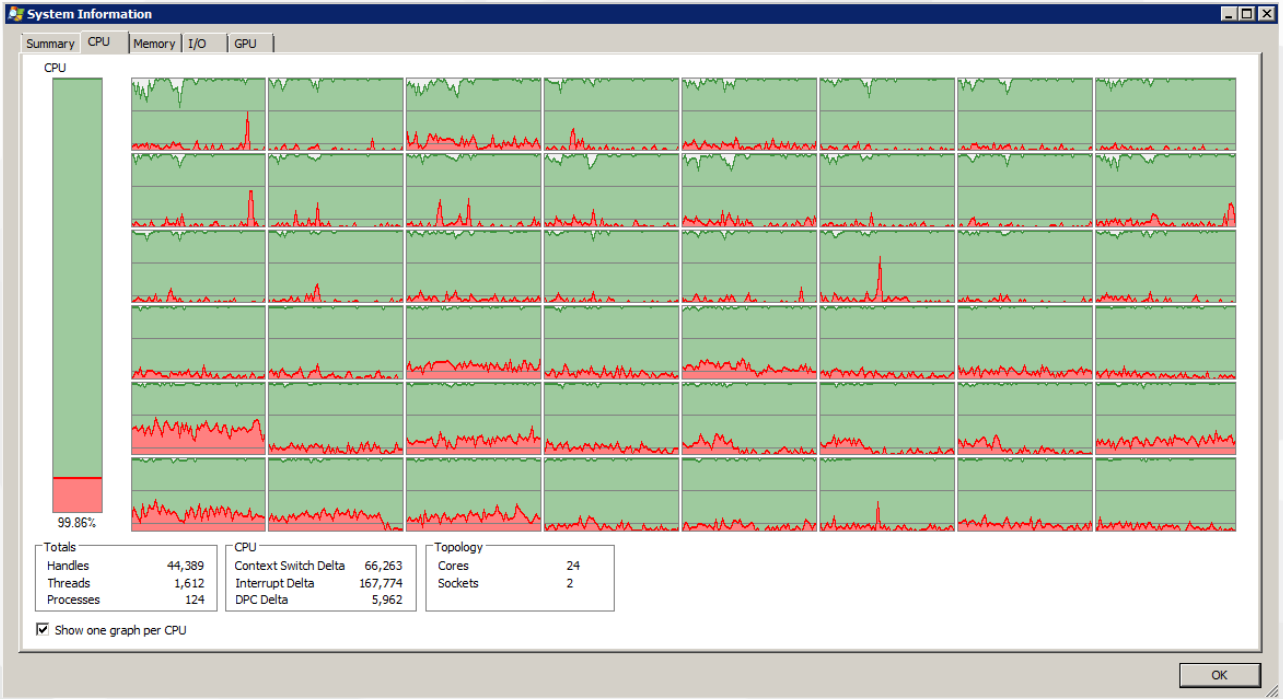
PPD: MULTITHREADING

```
ppd.exe particle.bin
```



PPD: MULTITHREADING

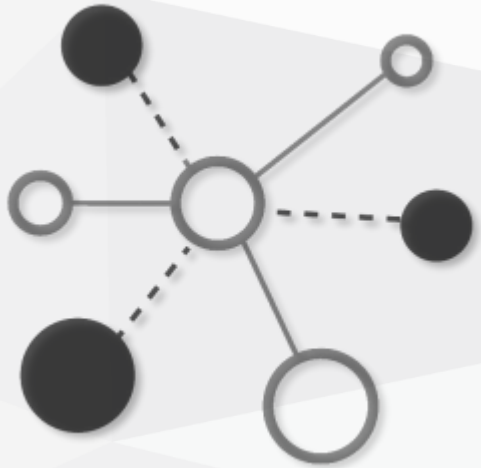
48 cores, 256GB RAM, 2x 8GB VRAM, 2x RAID0 NVMe SSDs



UBISOFT

PPD: BLAZING FAST NO-OP

- Graph is parsed by all the threads
  Multithreaded logic
- Written in highly optimized native code
  No script or external exec. => 1 exec. all in C++
- Cache friendly, low alloc. count
  State loaded in single alloc., pooled Node
- Hammering file system
  Local disk IOPS bound



UBISOFT

FC5 BUILD SYSTEM

3 POINTS

DEPENDENCY GRAPH
Compile dependencies

STAGED PARALLEL EVALUATION
Dependencies injection, Build

MINIMAL INCREMENTAL BUILDS
Primary speed factor

UBISOFT

DEPENDENCIES

DEFINES

THE *SPEED*

OF THE BUILD

3

Faster

FEATURES

PPD: FEATURES

- 99% of build time spent in build
  Further optim. needs to focus on build

- Features are just alternatives to ::Build()
  No impact on graph parsing logic

UBISOFT

PPD *Features*

1. PROCESS ISOLATION
2. DISTRIBUTION
3. CACHING
4. FILE SYSTEM HOOKING
5. BUILD SYSTEM AS AS SERVICE

1 PPD: PROCESS ISOLATION

- Run the code in another process
  Separate memory space
- Execute non thread safe code
  Only one build per isolated process
- Execute non-trusted code
  Crash tolerancy

UBISOFT

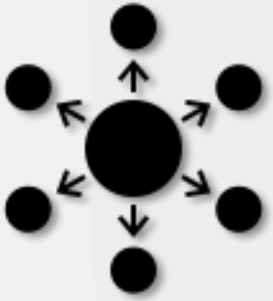
1 PPD: PROCESS ISOLATION

- Start a sub-process in “worker” mode
  ppd.exe -worker
- Send relevant information by RPC
  Processor settings, node data
- Gather results
  Logs, emitted runtime deps

UBISOFT

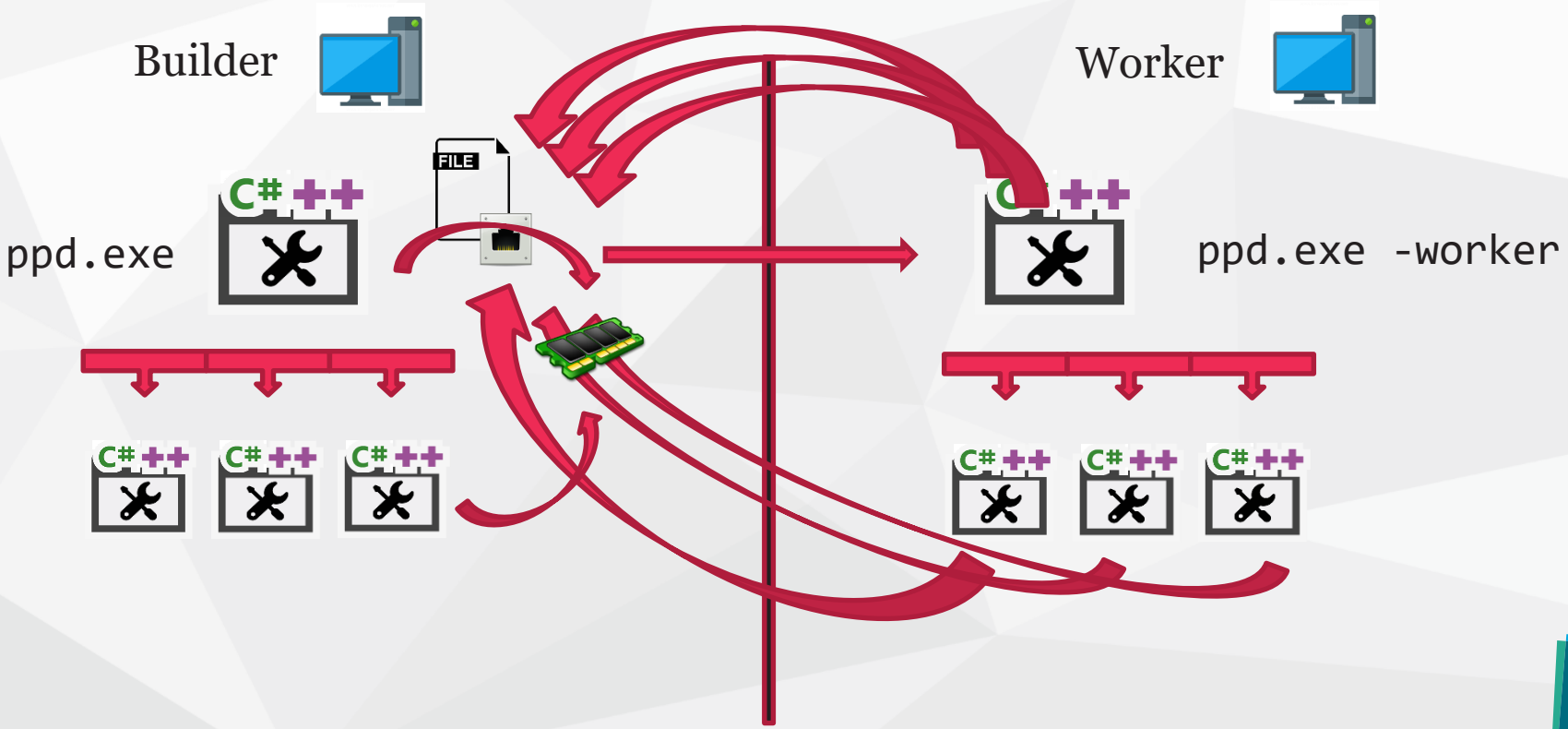
2 PPD: DISTRIBUTION

- Virtualize RPC transport
  TCP/IP instead of MMAP
- Contextualize file accesses
  Virtualized file system (see GDC 2015)
- Worker management
  Binaries transport, thread reservation. ...etc



UBISOFT

2 PPD+: DISTRIBUTION+ISOLATION



3 PPD: CACHING

- Snapshot of “sources”: MD5 Hash
  Everything that can affect result
- Save result on shared network location
  Asset store – see GDC 2015
- Try to download prior to build
  If not found, build locally
- Not always interesting
  Only when build time > build key + download time

UBISOFT

3 PPD: CACHING

- Key: MD5 of several information
  - Node (file) name
  - Version of processor
  - Setting of processor
  - CRC of source file content

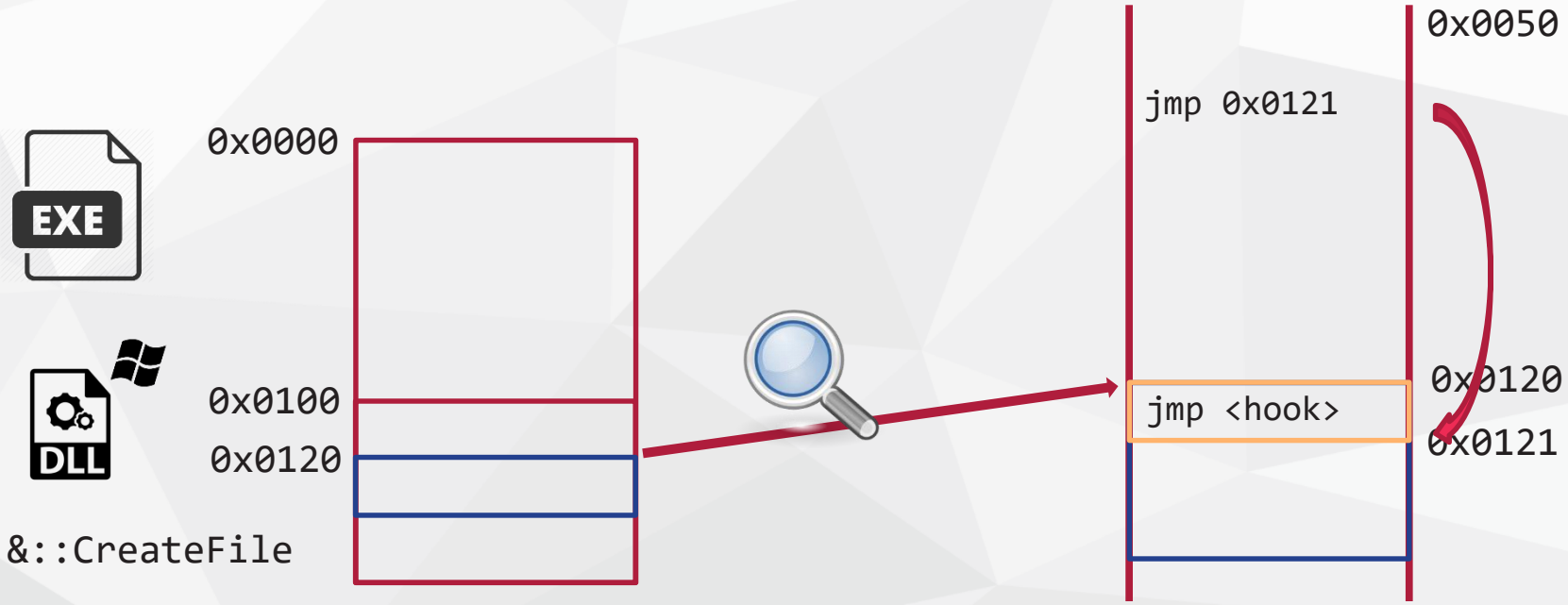
- Contains output(s) and emitted RT dep
  Packed in a single buffer

4 FILESYSTEM HOOKING: GOAL

- Sandboxing
  Redirect outputs
- Distribution of 3rd party tools
  Havok binarization

UBISOFT

4 DETOURING



4 HOOKING REMOTE PROCESS

```
void Hacked_CreateFile(const char* path)
{
  const char* newPath = DoSomething(path);
  Original_CreateFile(newPath);
}

void HookFileSystem()
{
  Original_CreateFile =
       HookManager::Hook(&CreateFile, &Hacked_CreateFile);
}
```

UBISOFT

4 HOOKING REMOTE PROCESS

```
LoadDLL("hooking.dll");
FuncProto f = FindProcAdress("HookFileSystem");
f();
```

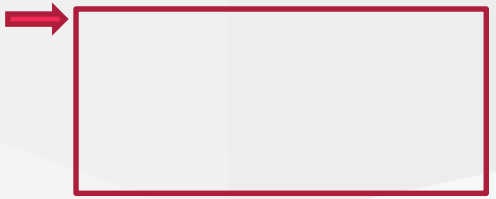
ASM

UBISOFT

4 HOOKING REMOTE PROCESS

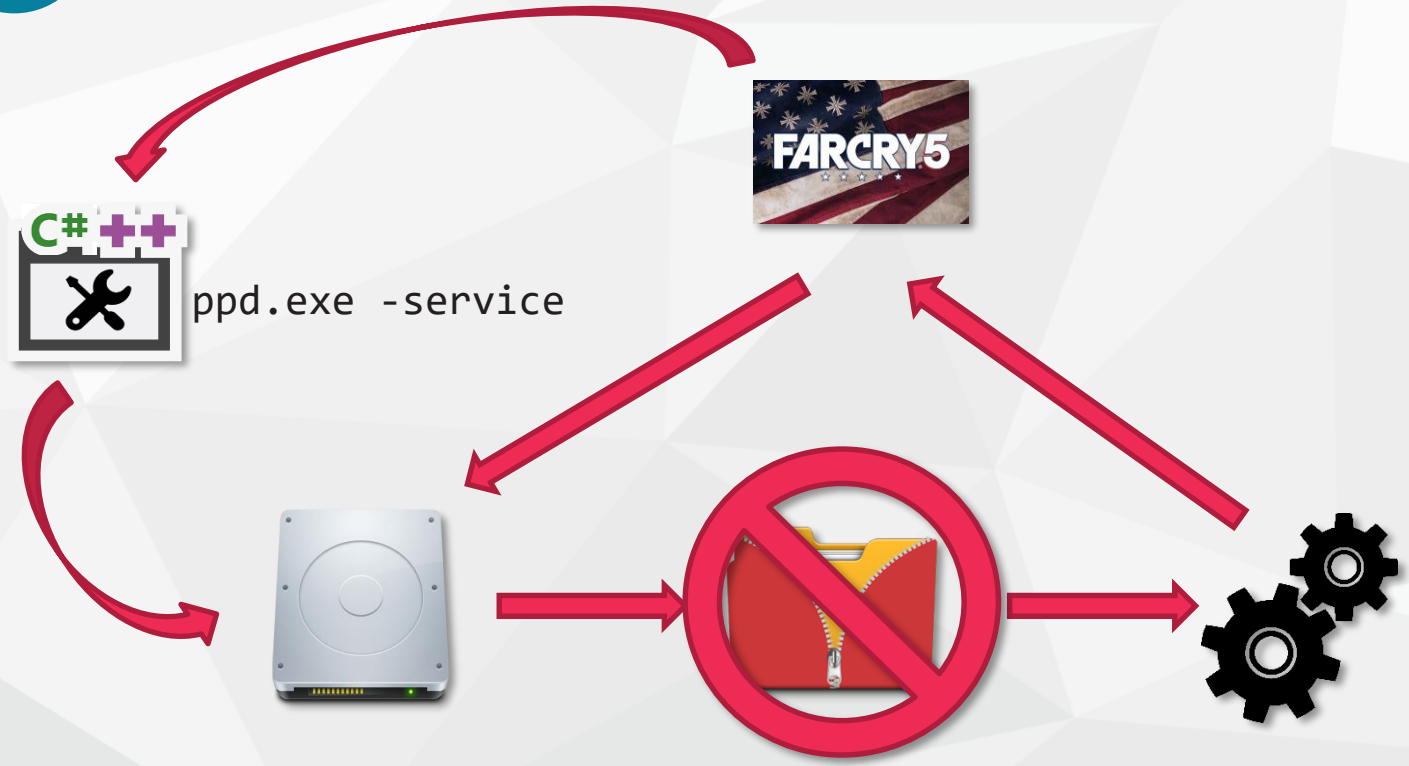
```
void CreateProcessWithHooking()
{
  CreateProcess(CREATE_SUSPENDED);
  VirtualAllocEX();
  WriteProcessMemory();
  CreateRemoteThread();
  Wait();
  VirtualFreeEx();
  ResumeThread();
}
```

ASM

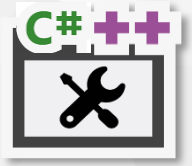


UBISOFT

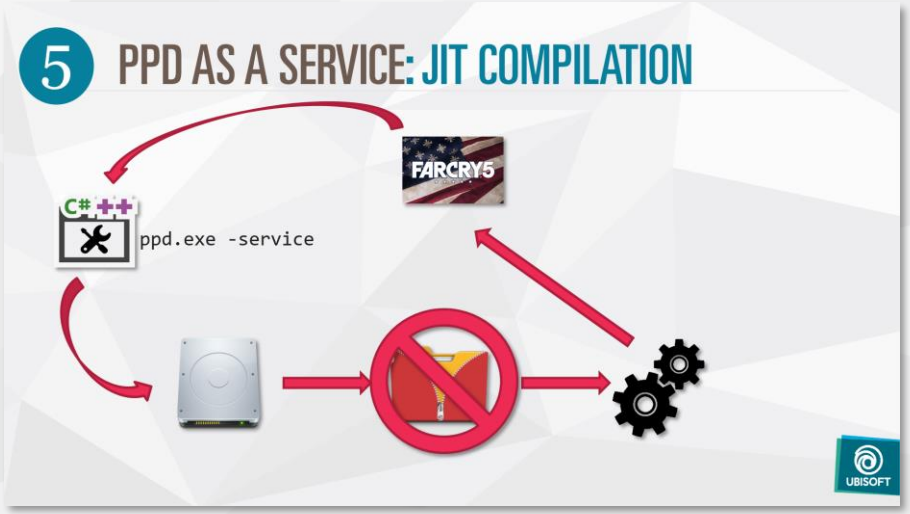
5 PPD AS A SERVICE: JIT COMPILATION



5 PPD AS A SERVICE: LIVE EDITING



ppd.exe -service



UBISOFT

4

Figures

CONCLUSION

THE FC5 PIPELINE

3 POINTS

PIPELINE TRINITY
Tools, Engine, Build System

DEPENDENCY GRAPH
Parallelisation & Incrementality

ADDITIONAL FEATURES
In replacement of ::Build()

UBISOFT

NUMBERS: TYPICAL NIGHTLY

No-Op build(1.5M nodes): 28s (654k timestamps, 267 dirlist)

116 editor exports, 20 at a time

20 millions nodes evaluated

309 world bigfiles (8 languages)

18 FarCry Arcade asset packs

Shaders for the 3 platforms

4.5 TiB of I/O, 560 GiB of outputs

1 machine, 2 h 00 (10 min incremental)

Fc4: 3h00, 13 machines, 1/3 of the work (1h00 incremental)

UBISOFT

THANKS & CREDITS

- Jessy Gosselin-Grant / *Engine as service, code hooking, pipeline architecture*
- Philippe Gagnon / *Pipeline architecture*
- Jean-Francois Cyr / *PPD power user*
- Philippe de Sève / *PPD power user*
- Franta Fulin / *Inspiration from FASTBuild*

THANKS & CREDITS

- Jeremy Moore/ *Peer review*
- Ryan Smith / *Peer review*
- Dominic Couture/ *Peer review*
- Danny Couture / *Peer review*
- Jean-Francois Dube / *Peer review*
- Christian Martin / *Peer review*
- Julien Merceron / *Peer review*
- Audrey Belanger / *Review*

DEPENDENCIES

DEFINES

THE *SPEED*

OF THE BUILD

QUESTIONS?

Rémi QUENIN

@azagoth

remi.quenin@ubisoft.com