UNIVERSITÀ DEGLI STUDI
DI MILANO

What is a Game Engine?

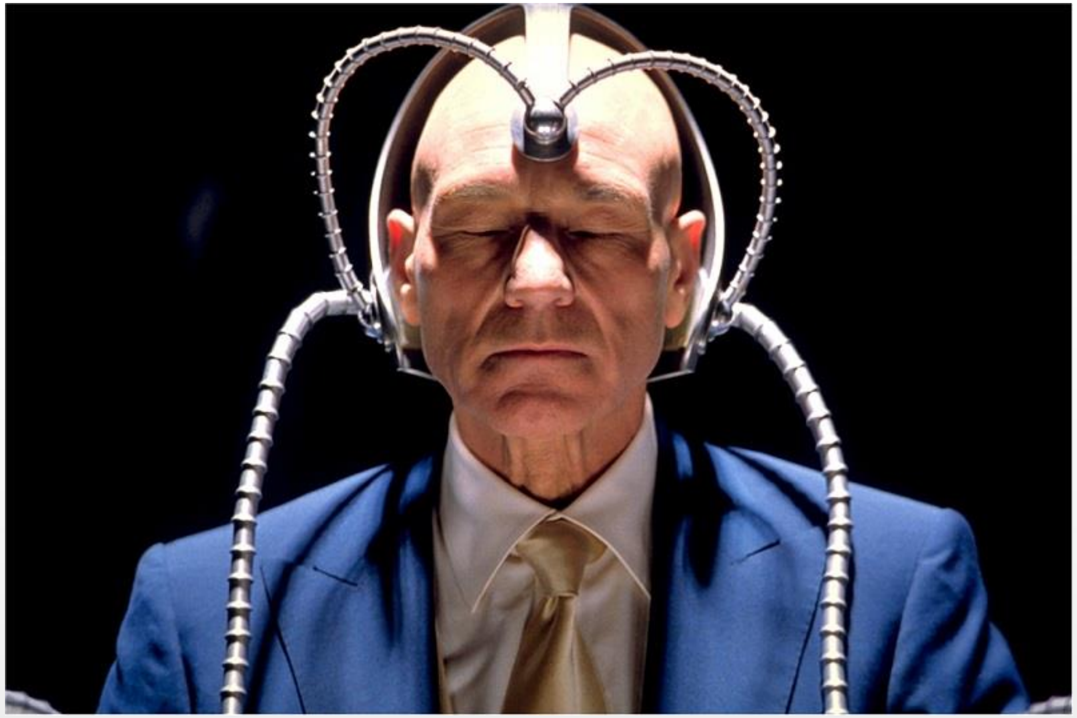
Lesson AI01

Dario Maggiorini (dario@di.unimi.it)

AI for Video Games – A.A. 2020 – 2021

Big Question: What is a Game Engine?

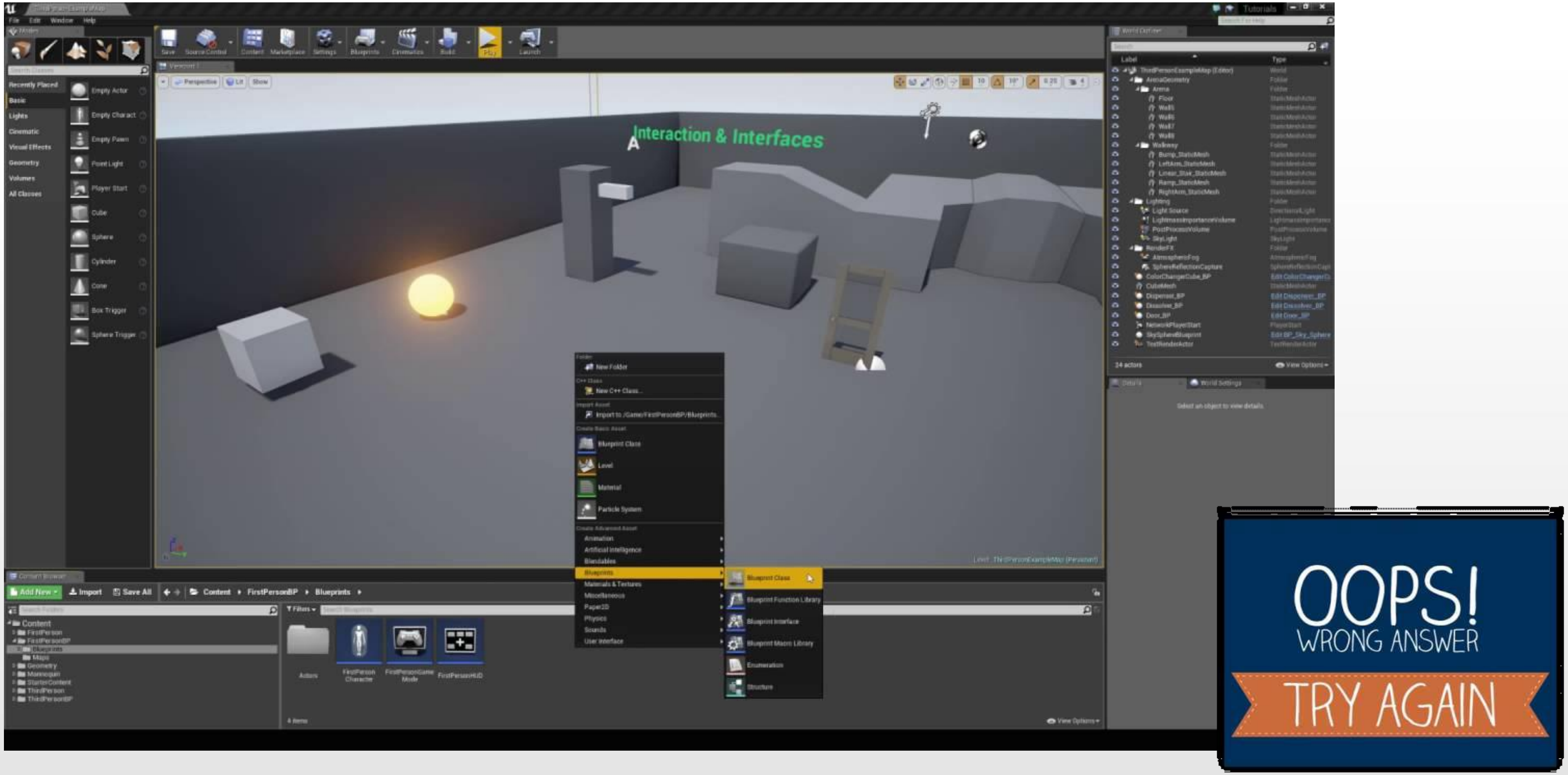
Reading Your Mind ...



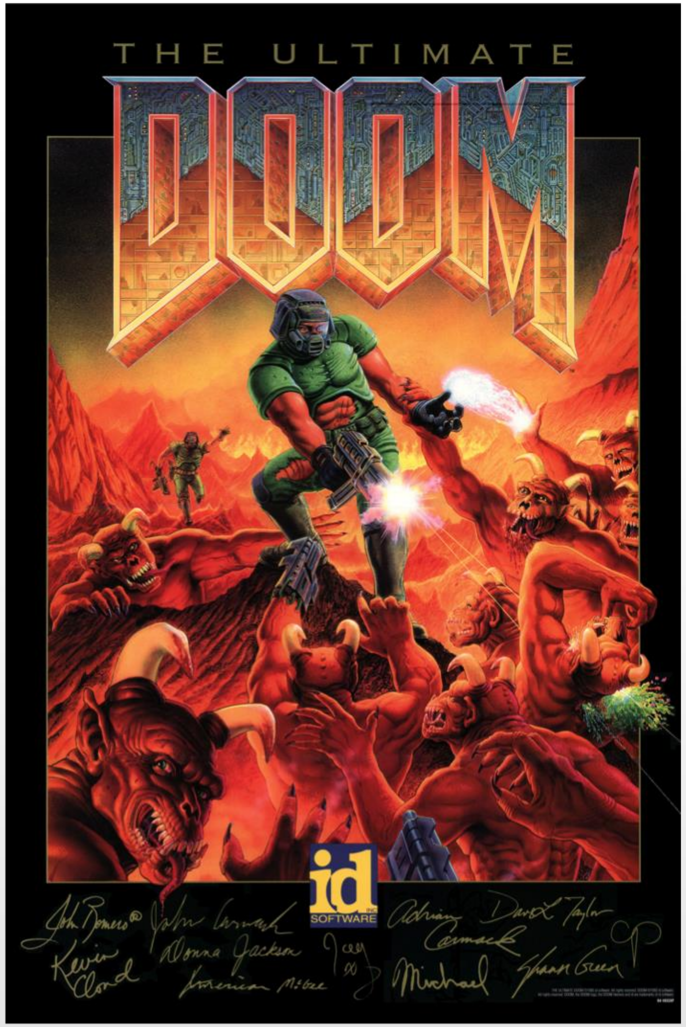
Most probably you though about Unity or Unreal

Wrong Answer!

Not 100% wrong … but a game engine is NOT this!



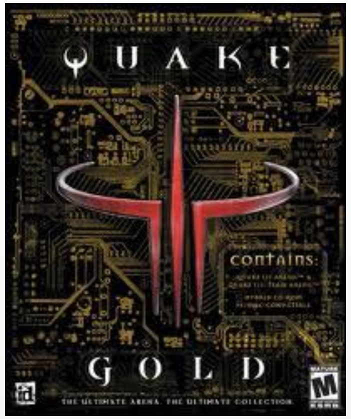
A (Very) Brief History of Game Engines

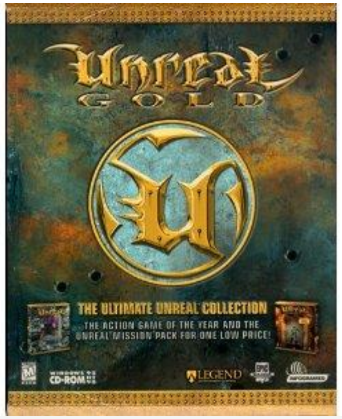


- … dates back to 1994 with DOOM
- Strict separation between core software components
- Strict separation between software and data assets
- Strong code reusability enforced during development
- Yet … The word “engine” was not there

Later on … in 90-Something

- Other games have been designed with with a modular architecture allowing modding and focusing on code reuse




Unreal
GOLD
THE ULTIMATE UNREAL COLLECTION
THE ACTION GAME OF THE YEAR AND THE UNREAL MISSION PACK FOR ONE LOW PRICE!

- Scripting language (such as quake C) started to be part of the distribution
- The game engine is now a standalone product (and a profitable one!)
  - E.g., Quake]I[can be considered as a (paid for) demo of the actual product (the Quake engine)
  - Customer are no longer players but developers!

What is a Game Engine

(Really) Technical description of game:

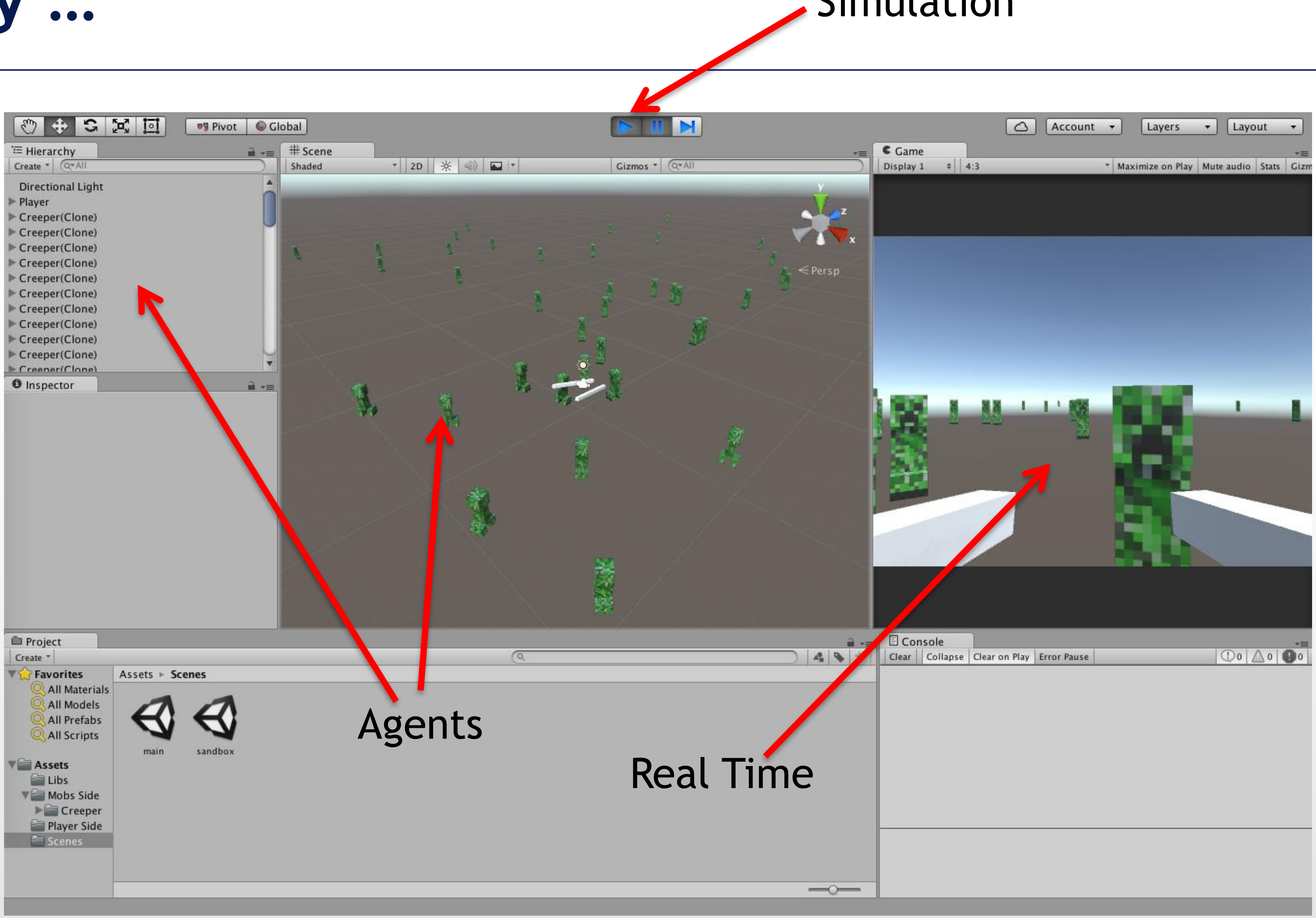
A real-time interactive agent-based computer simulation

A game engine is a software made to implement such a system

More Into Details

- Real-time (and interactive)
  - Must respond to player input in a timely-bound manner
- Agent-based
  - Independent entities (agents) live and interact with each other within the engine
- Simulation
  - It is capable to describe a model representation of a virtual world
  - It must be a MATHEMATICAL description

In Unity …

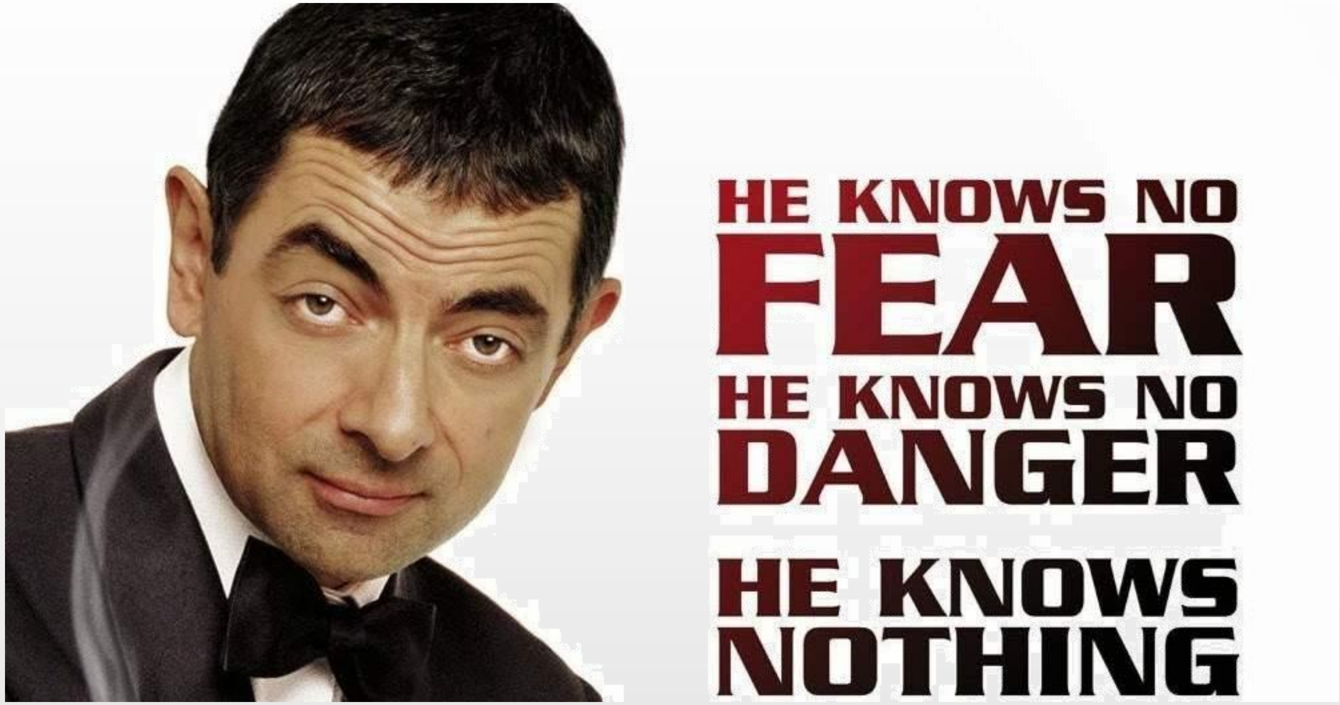

Simulation

Agents

Real Time

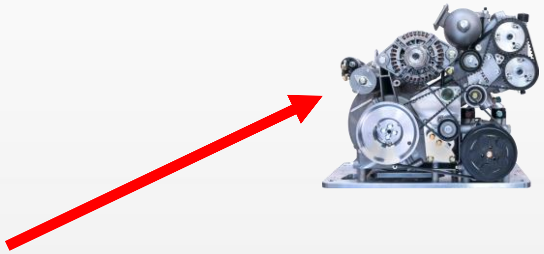
About the Unity Interface

- No fear: we will delve in the Unity interface and architecture later in the course



Building a Game Engine

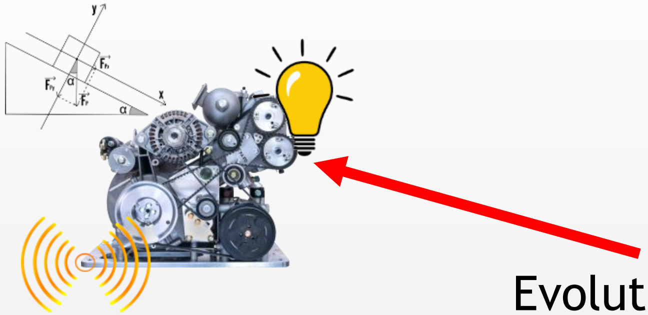
What is making your game “run” is a very small piece of code in charge to manage the basic simulation



Core
Resource management and coordination

Building a Game Engine

The core does not know how to make the system evolve.
It just applies “evolution rules” provided with the game



Evolution rules

Core is focused on performances.
It must be able to apply any kind of rule

Building a Game Engine

Core and basic evolution rules (e.g., physics and lighting) are bundled and hidden from the user. Like in a black box

Runtime

This black box (no one knows its content for commercial engines) is referenced as “the Runtime component”

Building a Game Engine

Easy to use interface

There is no need to know the content of the black box to create a game.

All we need to know is how to USE it! Possibly, by means of a convenient user interface

Runtime and Tool Set

- A game engine is made of two parts:
  - Tool Set
    - To compile software to work within the game engine
    - To help you describe rules
    - To manage assets
    - To create content

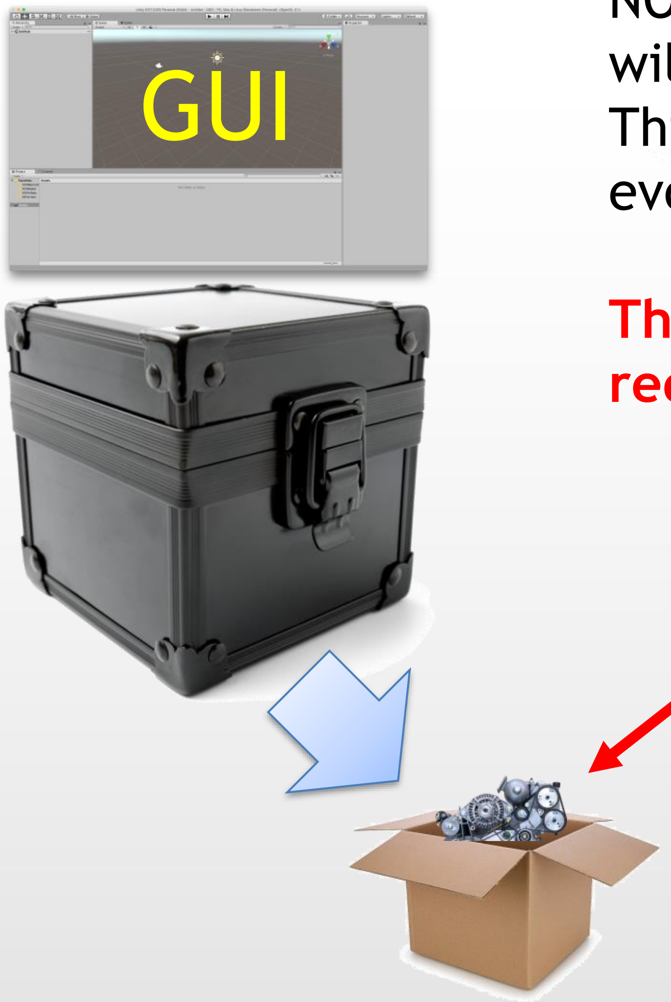
The GUI is usually the front-end to the toolset

And THIS was your (wrong) answer to the initial question

  - Runtime
    - A library/middleware/sandbox/virtual machine
    - This will run your rules on your assets
    - MUST be distributed with the game

Building a Game Engine

You use the GUI to put "stuff" (assets) inside the black box and then ask it to create an executable file



NOTE: a (large) piece of the core will be inside each executable. This is because the game must evolve also outside the black box

This is why we need redistributable licenses

Your game

Focus on the Black Box

- All we need to understand is how the black box is working
  - As a matter of fact, it is not required for a game engine to have a GUI (e.g., Source from Valve)

LEFT 4 DEAD

HALF-LIFE

... Have been made with Source

A Black Box for Rules and Assets

- A Game engine is a container for RULES
  - You **explain** how the world is evolving
  - You **do not** create the code to make it evolve

- The black box will follow the rules and apply them on the assets you provided
  - At every step, these rules will change the box status and its assets, making your world magically evolve

What is an Asset?

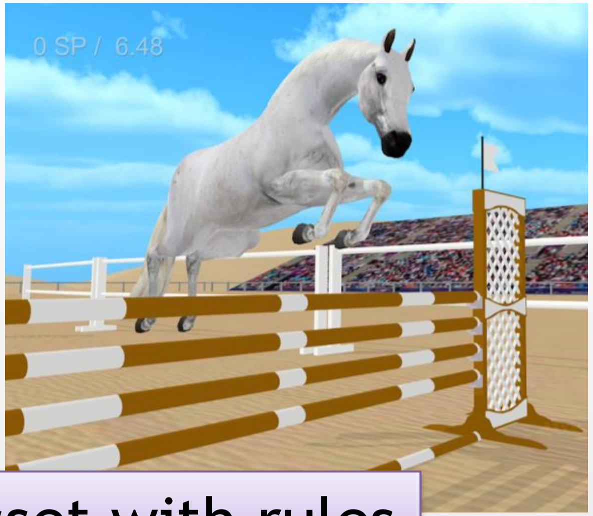
- An asset is whatever you may think about to show, listen, or feel while playing
  - Texture
  - 3D meshes
  - Material definition
  - Particles
  - Visual effects (shaders)
  - Music
  - *Scraps of code* (?)

What is a Rule?

- A rule is the definition of a behavior you attach to an asset in order to define:

  1. How to interact with the user
     e.g., reacting to user controls
  2. How to interact with the environment
     e.g., falling and casting shadows
  3. How to interact with other assets
     e.g., collisions

- Creating a believable NPC means creating the “right” rules based on the surrounding context



Asset with rules

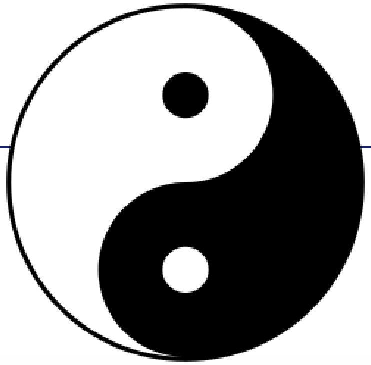
Script and Rules

- Of course, the easiest way (for a computer scientist) to describe a rule is by means of a scrap of code

… but

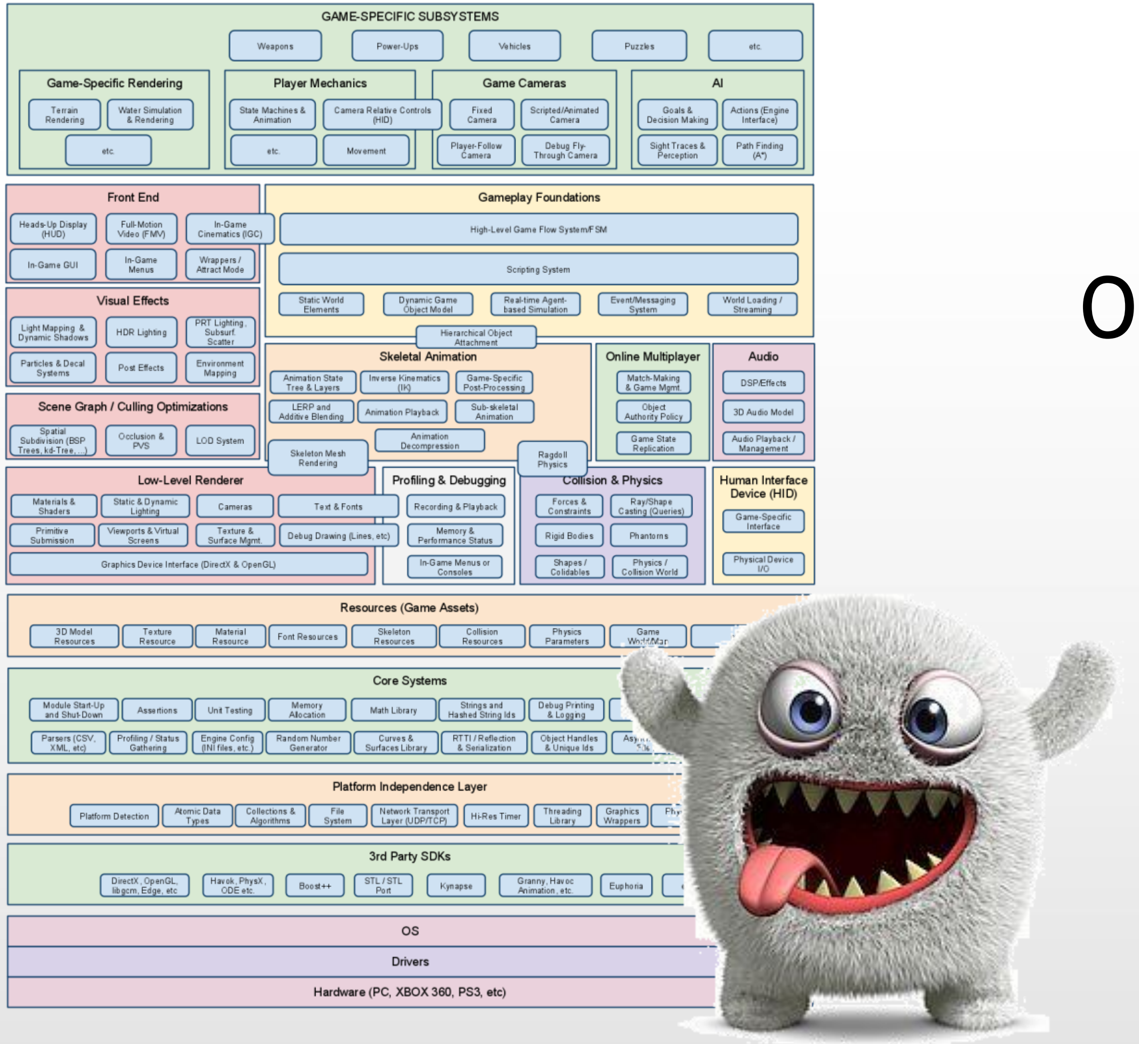
- A script is technically an asset (!)
- A script turns into a rule when:
  1. Is compiled
  2. Is run inside another asset

There are Two Kinds of Rules



- Built-in
  - Wired in the black box for everyday swiss-army knife use
- Provided by the user
  - This is what is making you game truly unique
  - You will call then “gameplay programming”
- There is NO ACTUAL DIFFERENCE between the two
  - You can change built-in rules if you wish
    (e.g., switch to havok or bullet physic engines)

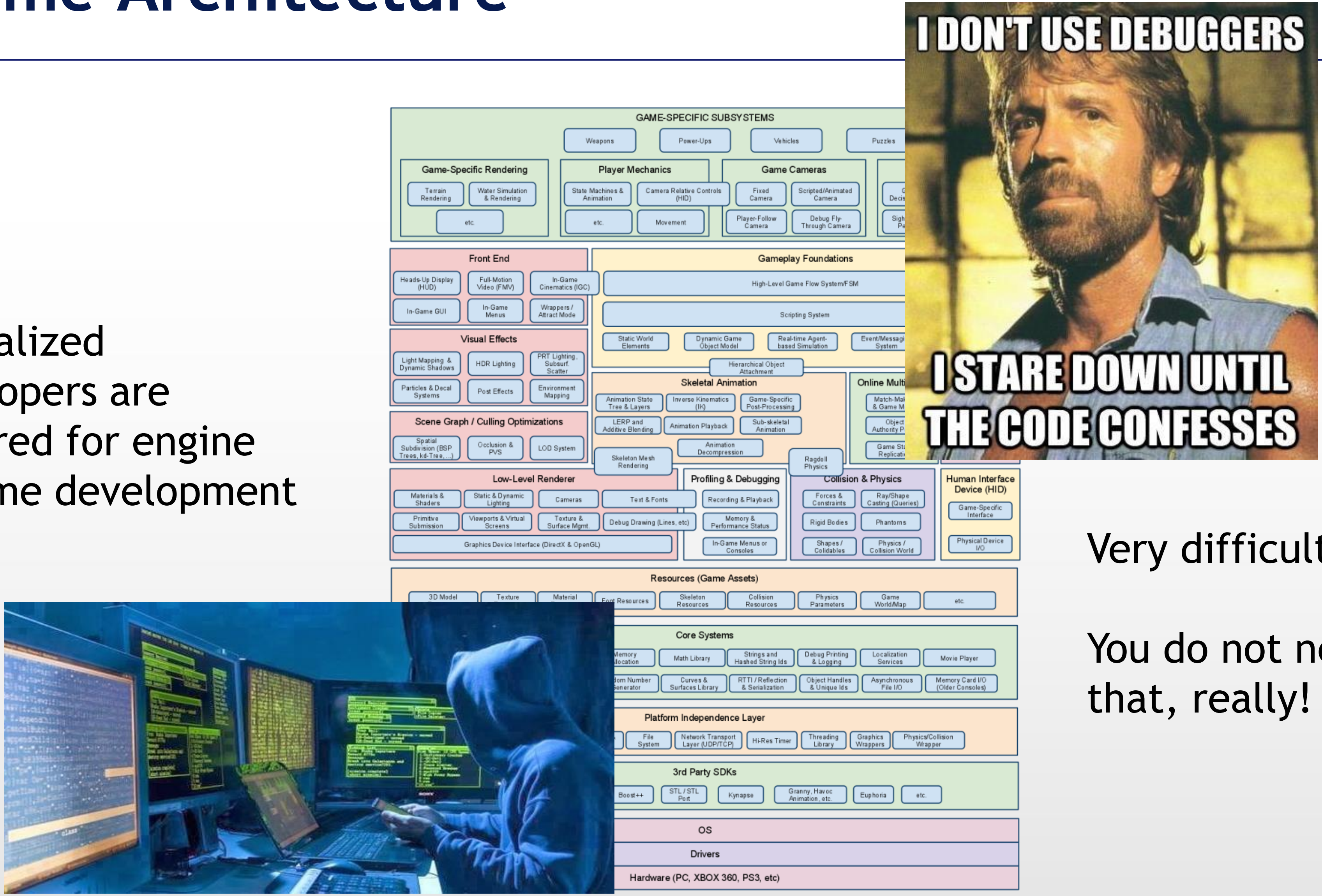
Runtime Architecture



Ouch!

Runtime Architecture

Specialized developers are required for engine runtime development



GAME-SPECIFIC SUBSYSTEMS

Weapons | Power-Ups | Vehicles | Puzzles

Game-Specific Rendering: Terrain Rendering, Water Simulation & Rendering, etc.

Player Mechanics: State Machines & Animation, Camera Relative Controls (HID), etc., Movement

Game Cameras: Fixed Camera, Scripted/Animated Camera, Player-Follow Camera, Debug Fly-Through Camera

Front End: Heads-Up Display (HUD), Full-Motion Video (FMV), In-Game Cinematics (IGC), In-Game GUI, In-Game Menus, Wrappers / Attract Mode

Visual Effects: Light Mapping & Dynamic Shadows, HDR Lighting, PRT Lighting, Subsurf. Scatter, Particles & Decal Systems, Post Effects, Environment Mapping

Scene Graph / Culling Optimizations: Spatial Subdivision (BSP Trees, kd-Tree, ...), Occlusion & PVS, LOD System

Gameplay Foundations: High-Level Game Flow System/FSM, Scripting System, Static World Elements, Dynamic Game Object Model, Real-time Agent-based Simulation, Event/Messagi System, Hierarchical Object Attachment

Skeletal Animation: Animation State Tree & Layers, Inverse Kinematics (IK), Game-Specific Post-Processing, LERP and Additive Blending, Animation Playback, Sub-skeletal Animation, Animation Decompression, Skeleton Mesh Rendering

Online Multi: Match-Mak & Game M, Object Authority P, Game St Replicati

Ragdoll Physics

Low-Level Renderer: Materials & Shaders, Static & Dynamic Lighting, Cameras, Text & Fonts, Primitive Submission, Viewports & Virtual Screens, Texture & Surface Mgmt., Debug Drawing (Lines, etc), Graphics Device Interface (DirectX & OpenGL)

Profiling & Debugging: Recording & Playback, Memory & Performance Status, In-Game Menus or Consoles

Collision & Physics: Forces & Constraints, Ray/Shape Casting (Queries), Rigid Bodies, Phantoms, Shapes / Collidables, Physics / Collision World

Human Interface Device (HID): Game-Specific Interface, Physical Device I/O

Resources (Game Assets): 3D Model, Texture, Material, Font Resources, Skeleton Resources, Collision Resources, Physics Parameters, Game World/Map, etc.

Core Systems: Memory Allocation, Math Library, Strings and Hashed String Ids, Debug Printing & Logging, Localization Services, Movie Player, Random Number Generator, Curves & Surfaces Library, RTTI / Reflection & Serialization, Object Handles & Unique Ids, Asynchronous File I/O, Memory Card I/O (Older Consoles)

Platform Independence Layer: File System, Network Transport Layer (UDP/TCP), Hi-Res Timer, Threading Library, Graphics Wrappers, Physics/Collision Wrapper

3rd Party SDKs: Boost++, STL / STL Port, Kynapse, Granny, Havoc Animation, etc., Euphoria, etc.

OS

Drivers

Hardware (PC, XBOX 360, PS3, etc)

Very difficult to debug

You do not need to do that, really!

Runtime Engine Architecture for Beginners

- A game engine is:
  - huge
  - complex
  - made of layers
    - Like many other complex softwares, such as O.S. kernels

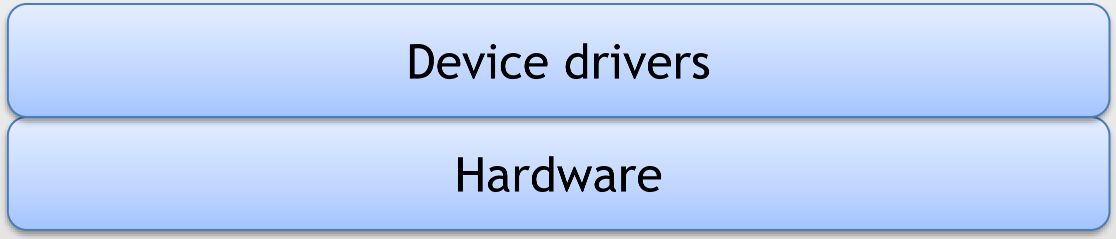
Let’s Try Again

Hardware is there.
You cannot go without

Hardware

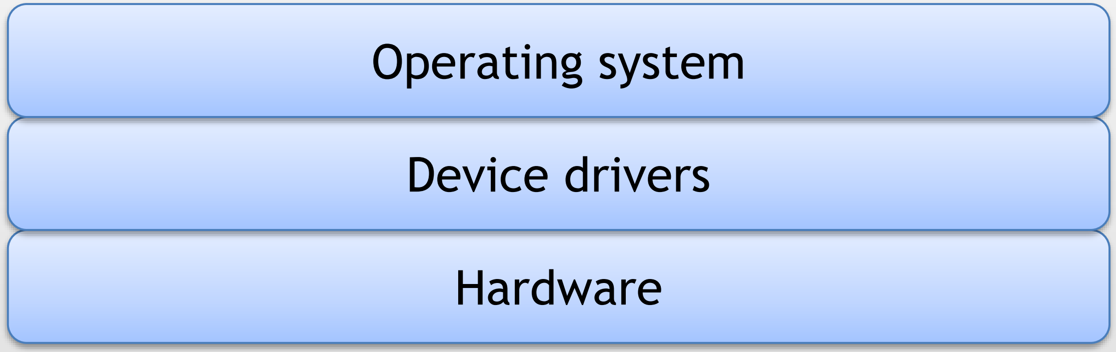
Let’s Try Again

As in any operating system, device drivers will provide uniform and manageable interfaces to device controllers



Let’s Try Again

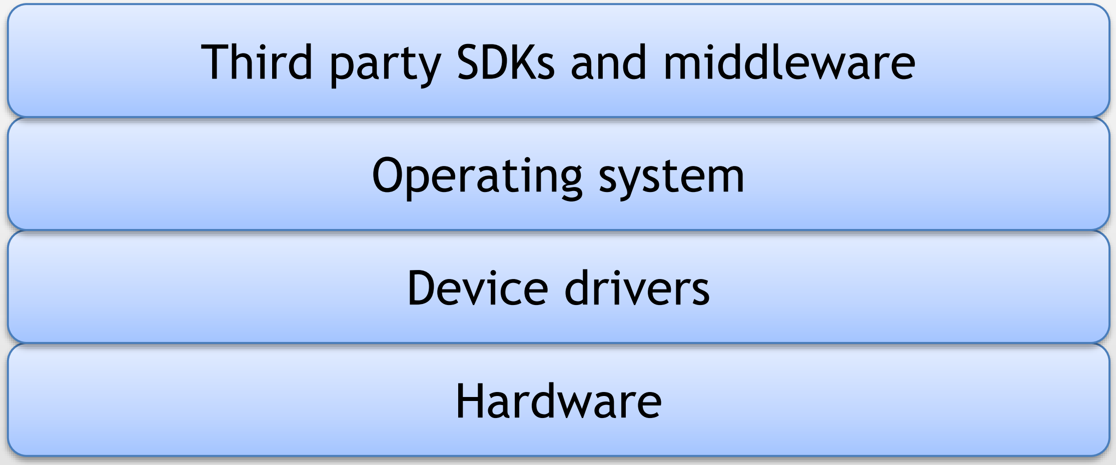
The operating system will allow your application (game) to use the hardware via the device drivers. This is true also for consoles!



Let’s Try Again

The O.S. is hosting third parties’ SDK and middleware to let you use the O.S. in a way which us convenient for a game.

This is usually different from the way ordinary applications are using the same O.S.



Third party SDKs and middleware

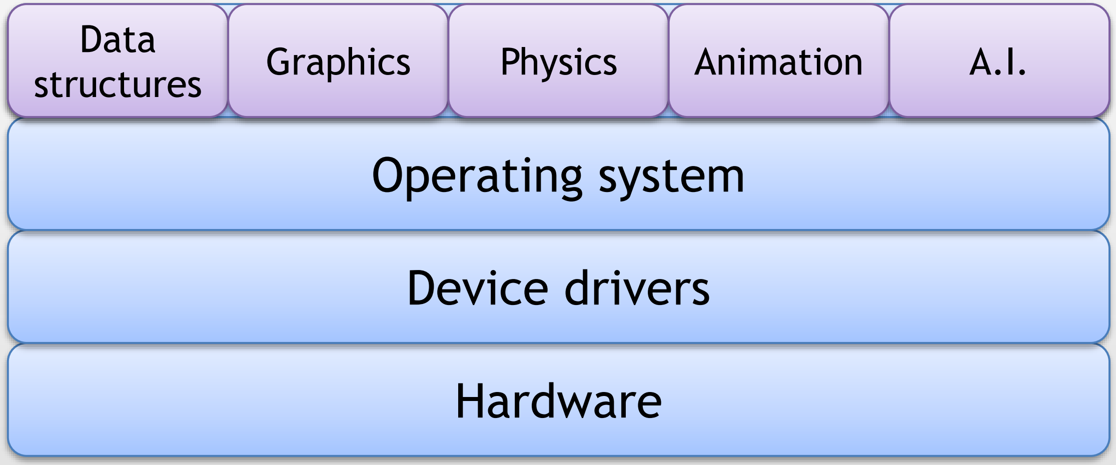
Operating system

Device drivers

Hardware

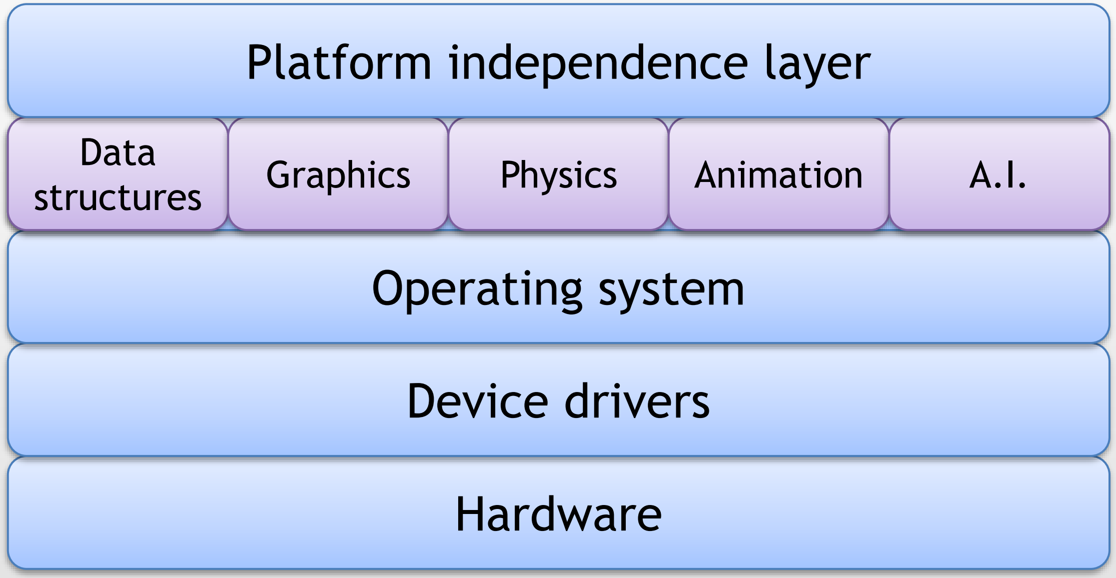
Let’s Try Again

The middleware is implementing data structures and logics (extremely optimized for the current platform) useful for any kind of game. Or, at least, for a very large variety of games



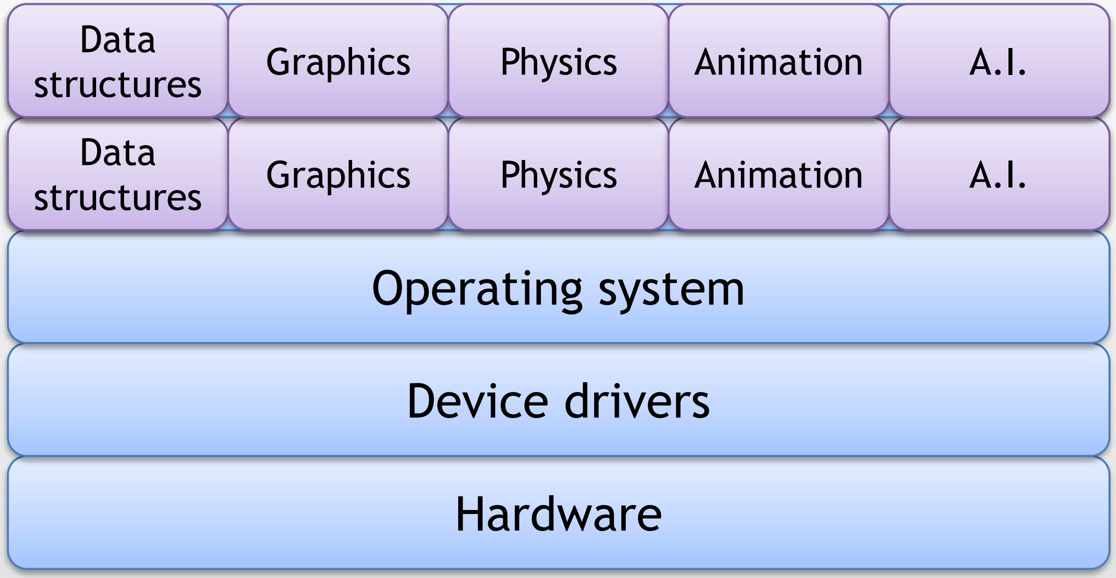
Let’s Try Again

The middleware (and its component) are architecture-specific.
We DO NOT WANT to ask the developer to create code to be used only once (on only one architecture).
So, we must provide an additional middleware to create a platform-independent access to the engine inner functionalities



Let’s Try Again

In the platform-independent layer all platform-dependent data structures and logics are replicated to offer a set of standard (and stable) APIs to developers



Data structures | Graphics | Physics | Animation | A.I.

Data structures | Graphics | Physics | Animation | A.I.

Operating system

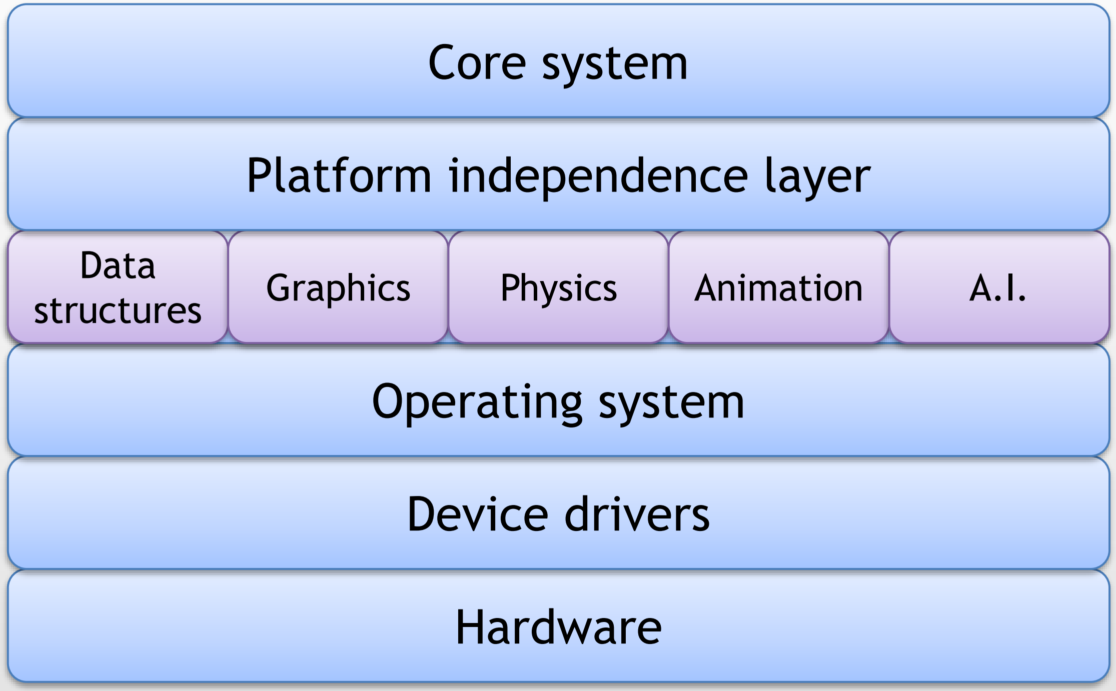
Device drivers

Hardware

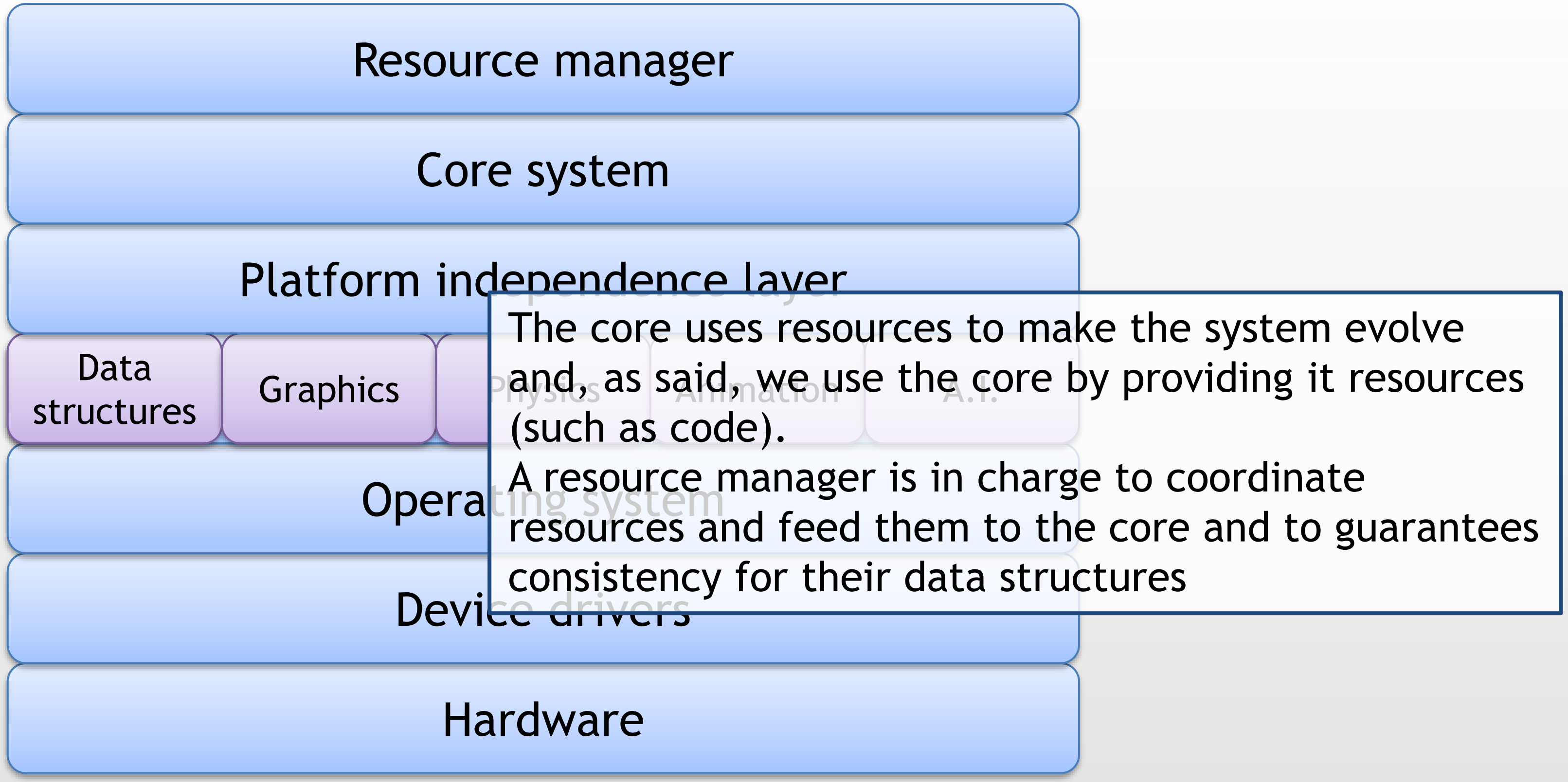
Let’s Try Again

Implementing the core in a lower lever could improve performances.
Nevertheless, will also require more effort to keep all the platforms in sync and introduce (additional) platform-specific bugs

And, at last, we can build a (platform-independent) core system on top it



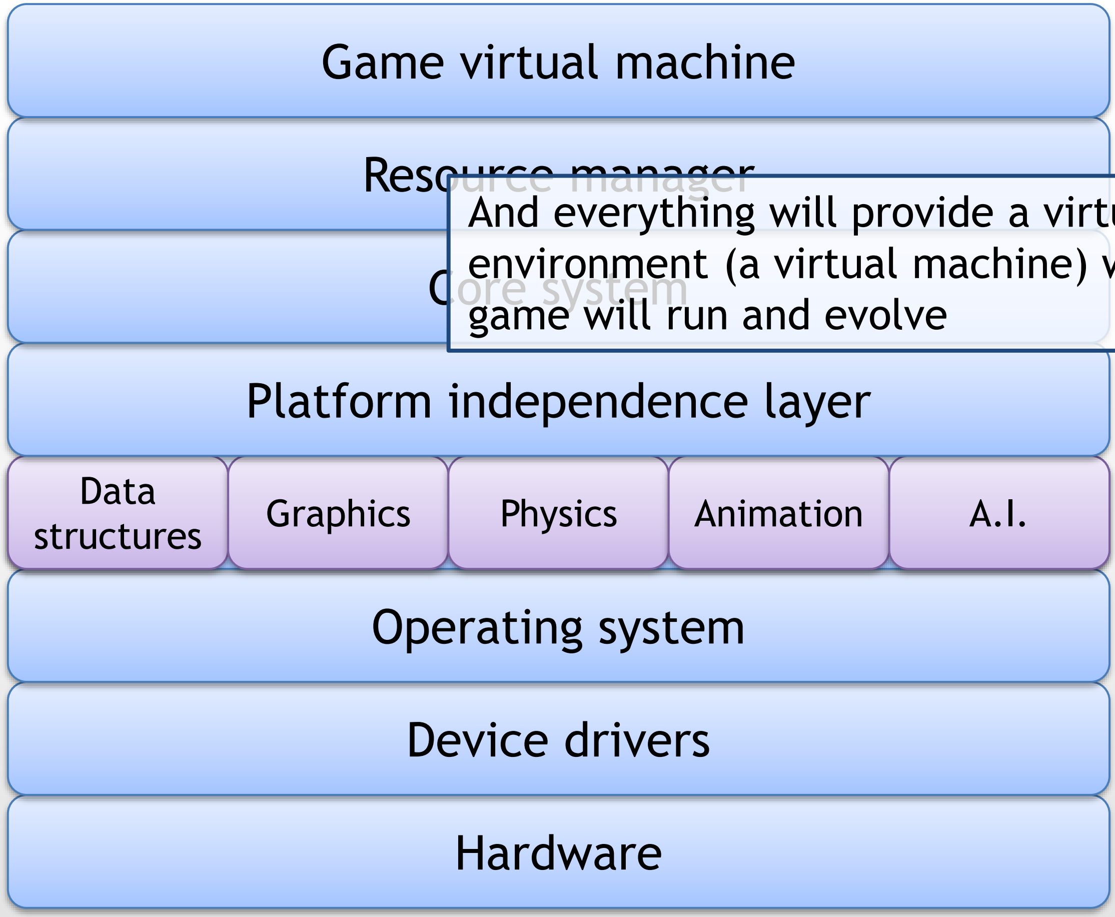
Let’s Try Again



The core uses resources to make the system evolve and, as said, we use the core by providing it resources (such as code).

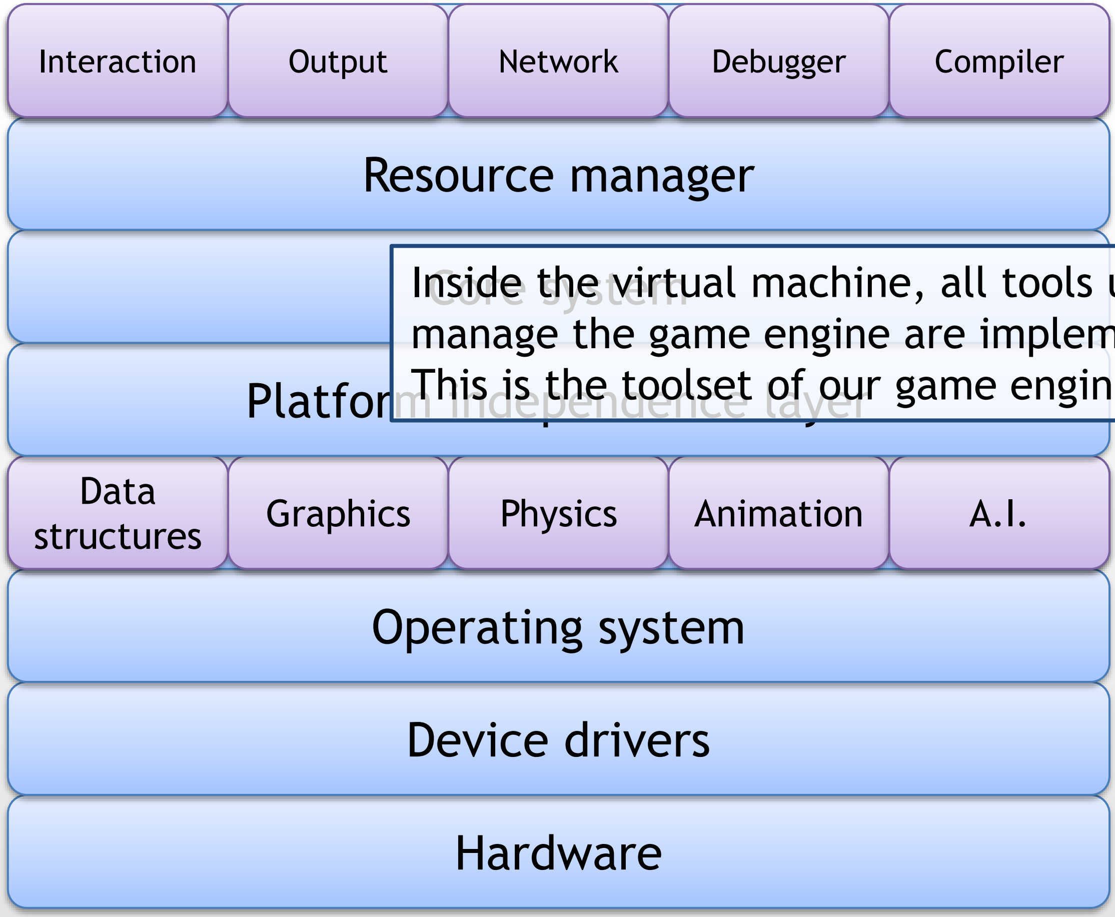
A resource manager is in charge to coordinate resources and feed them to the core and to guarantees consistency for their data structures

Let’s Try Again



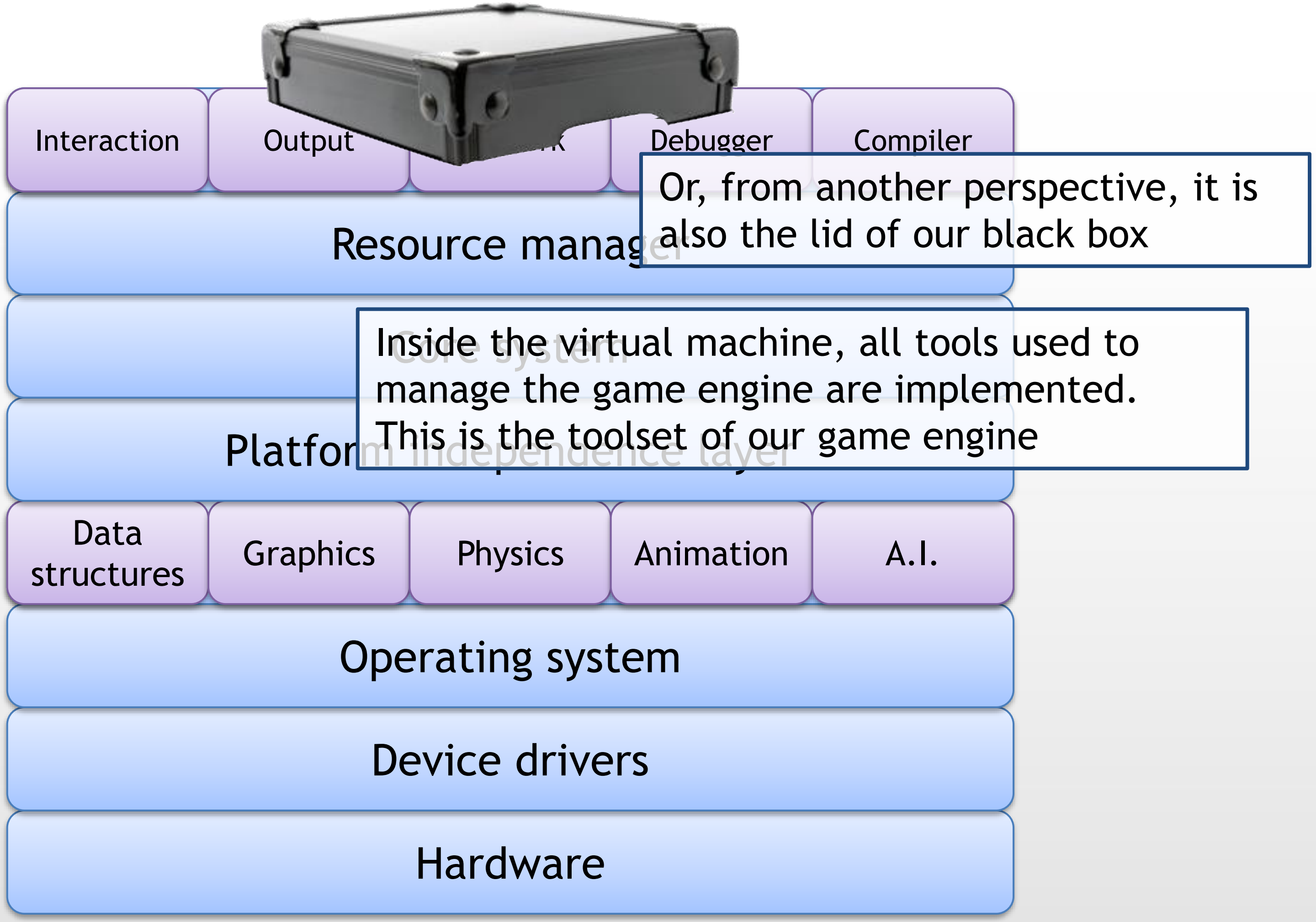
And everything will provide a virtual execution environment (a virtual machine) where your game will run and evolve

Let’s Try Again



Inside the virtual machine, all tools used to manage the game engine are implemented. This is the toolset of our game engine

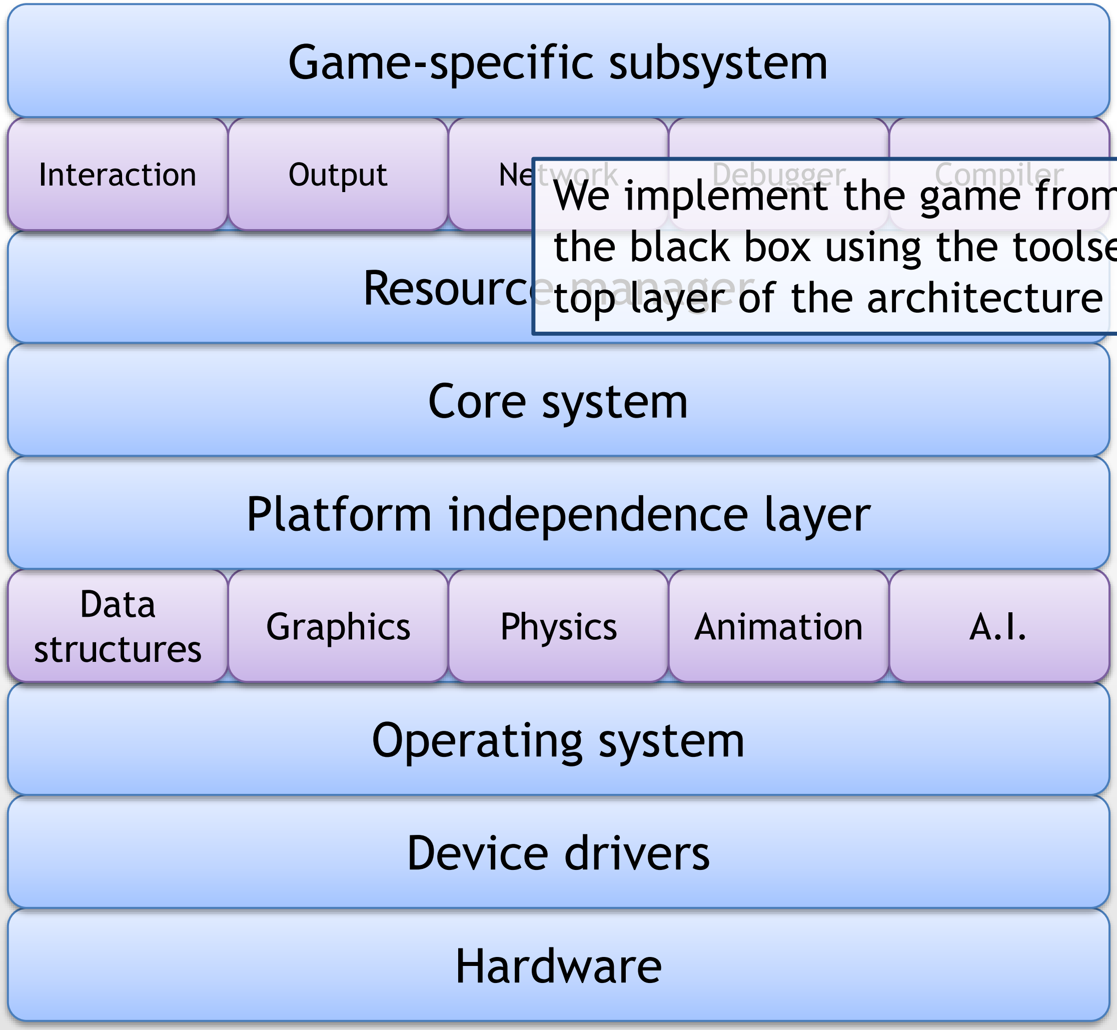
Let’s Try Again



Or, from another perspective, it is also the lid of our black box

Inside the virtual machine, all tools used to manage the game engine are implemented. This is the toolset of our game engine

Let’s Try Again



Game-specific subsystem

Interaction | Output | Network | Debugger | Compiler

Resource manager

Core system

Platform independence layer

Data structures | Graphics | Physics | Animation | A.I.

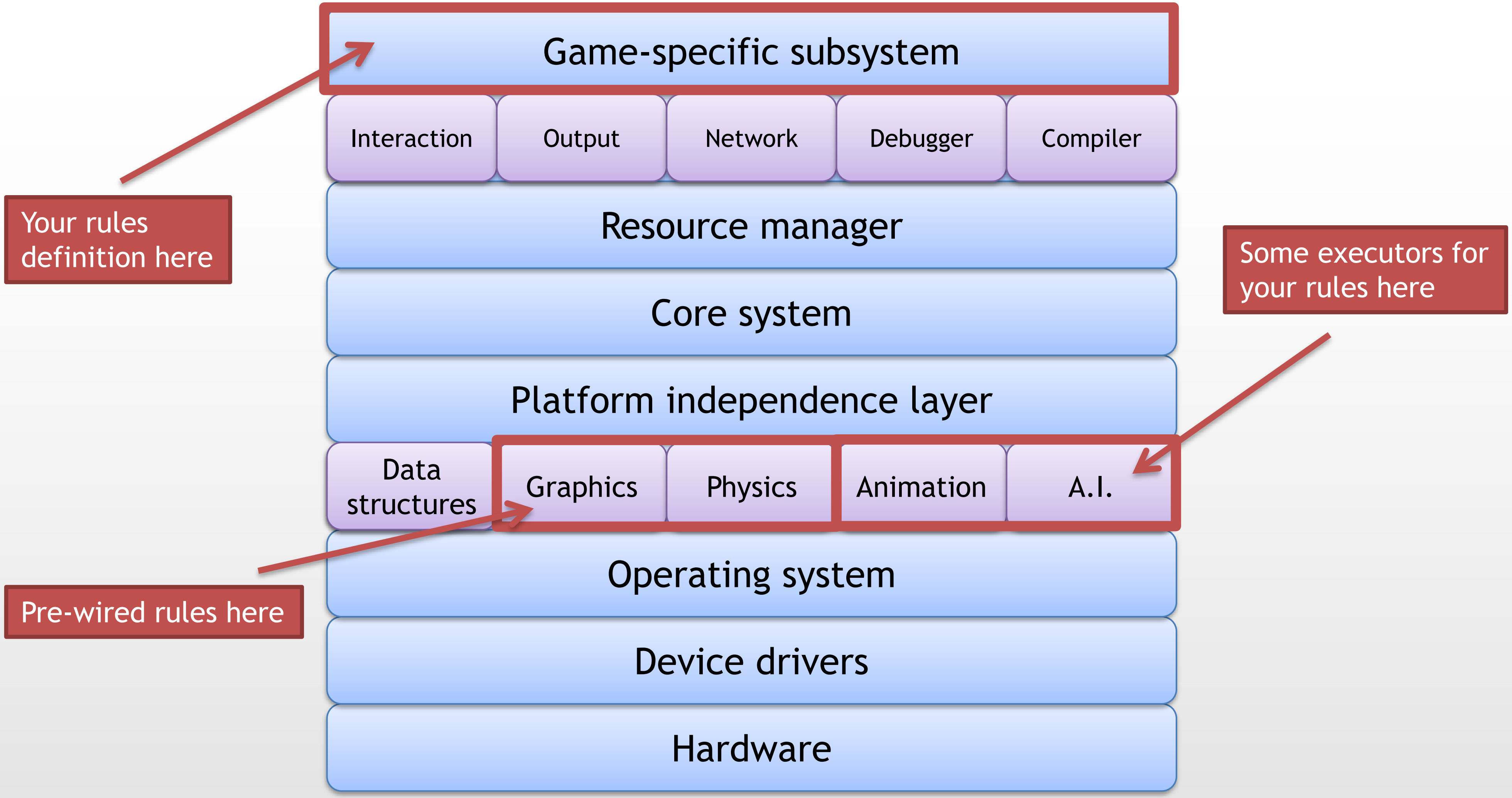
Operating system

Device drivers

Hardware

We implement the game from outside the black box using the toolset in the top layer of the architecture

Where are the Rules?

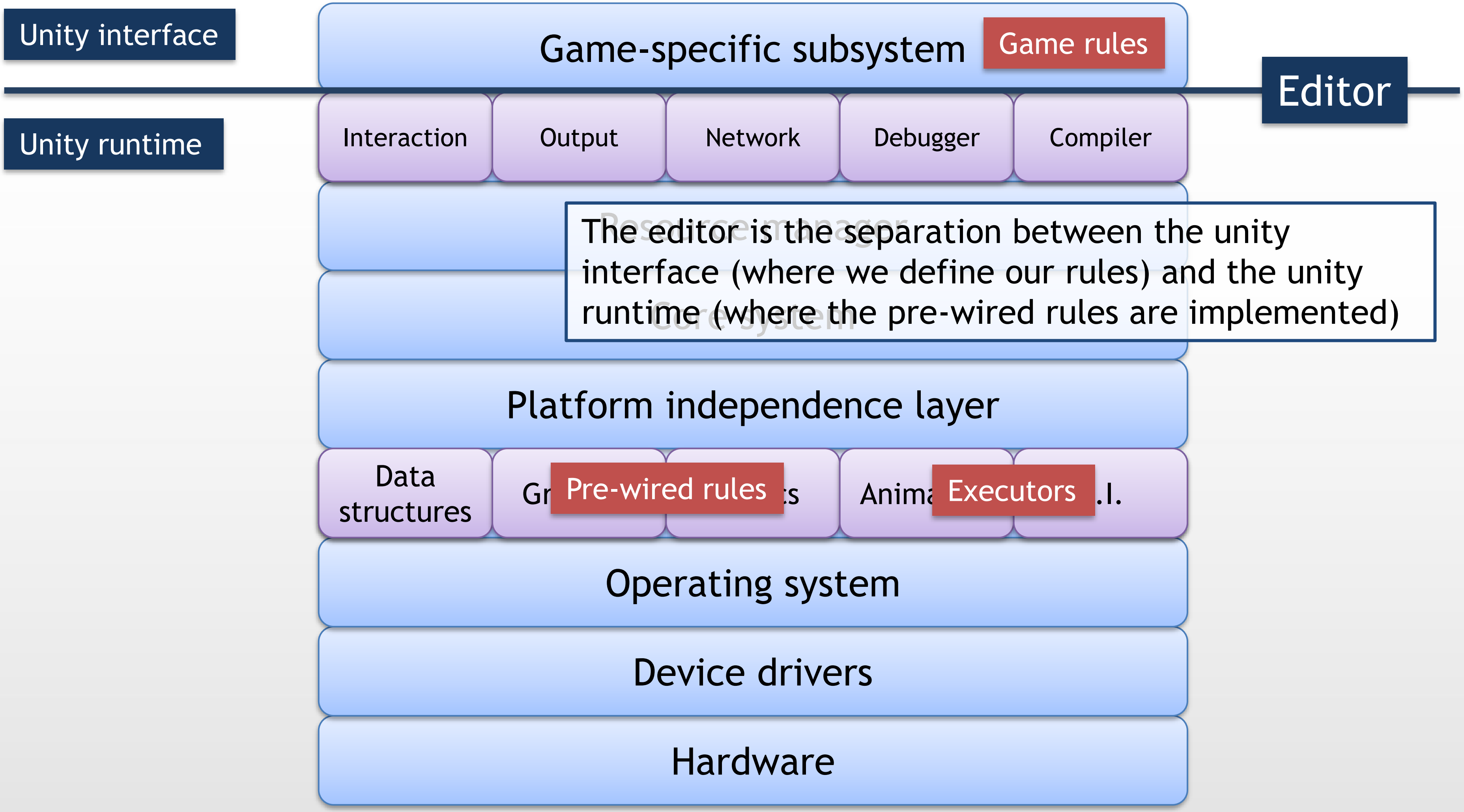


Your rules definition here

Some executors for your rules here

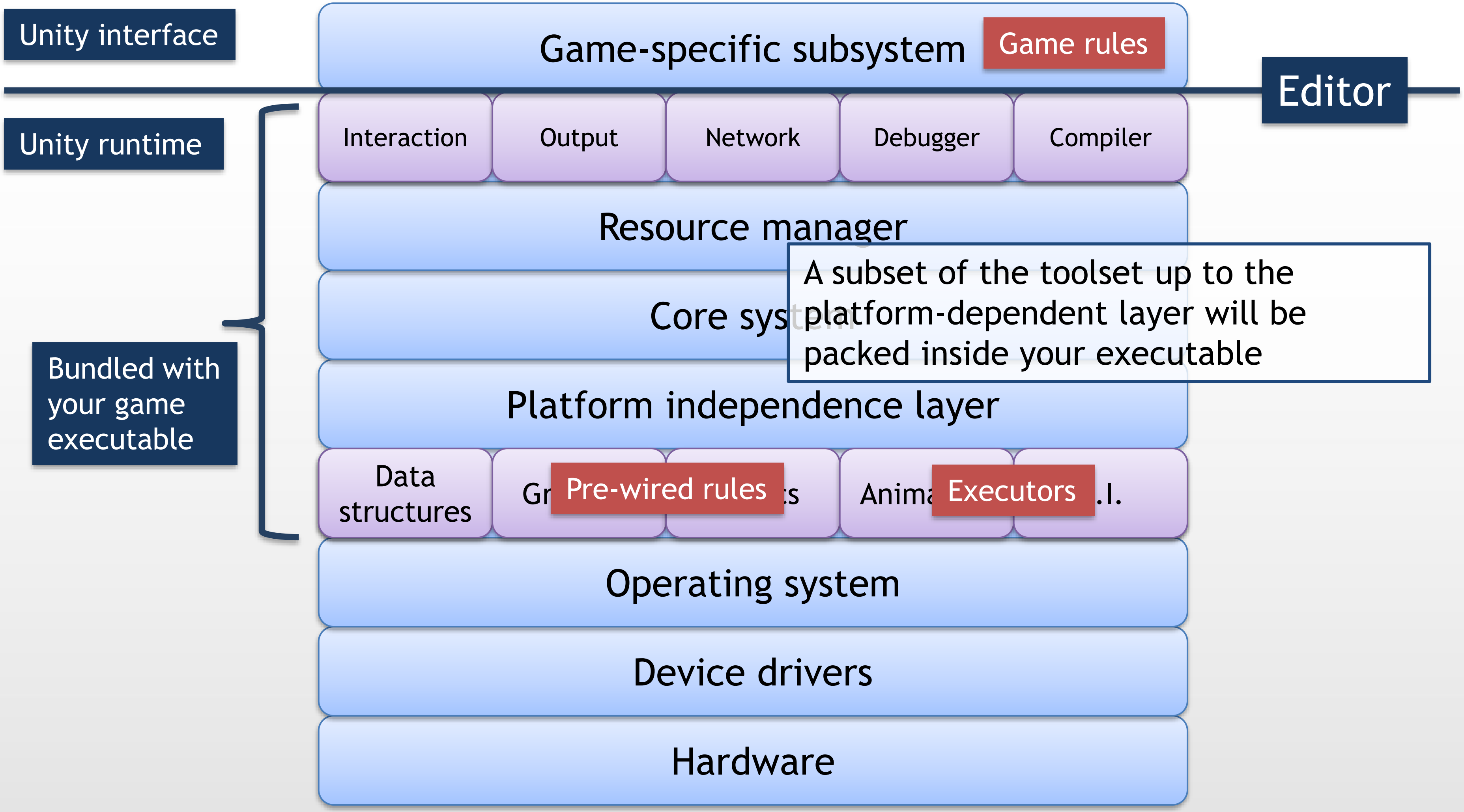
Pre-wired rules here

How do the Engine Uses Them?



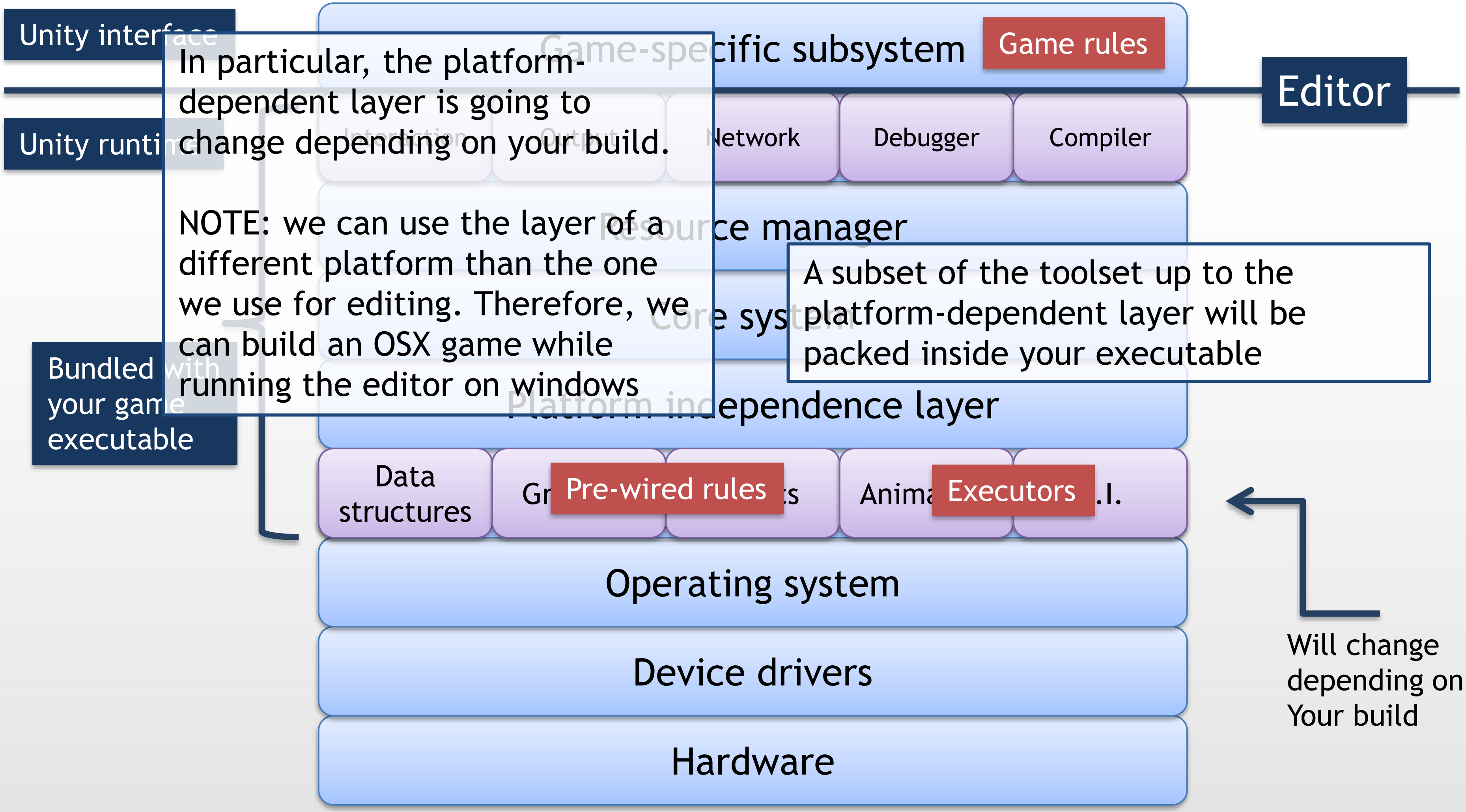
The editor is the separation between the unity interface (where we define our rules) and the unity runtime (where the pre-wired rules are implemented)

How do the Engine Uses Them?



A subset of the toolset up to the platform-dependent layer will be packed inside your executable

How do the Engine Uses Them?

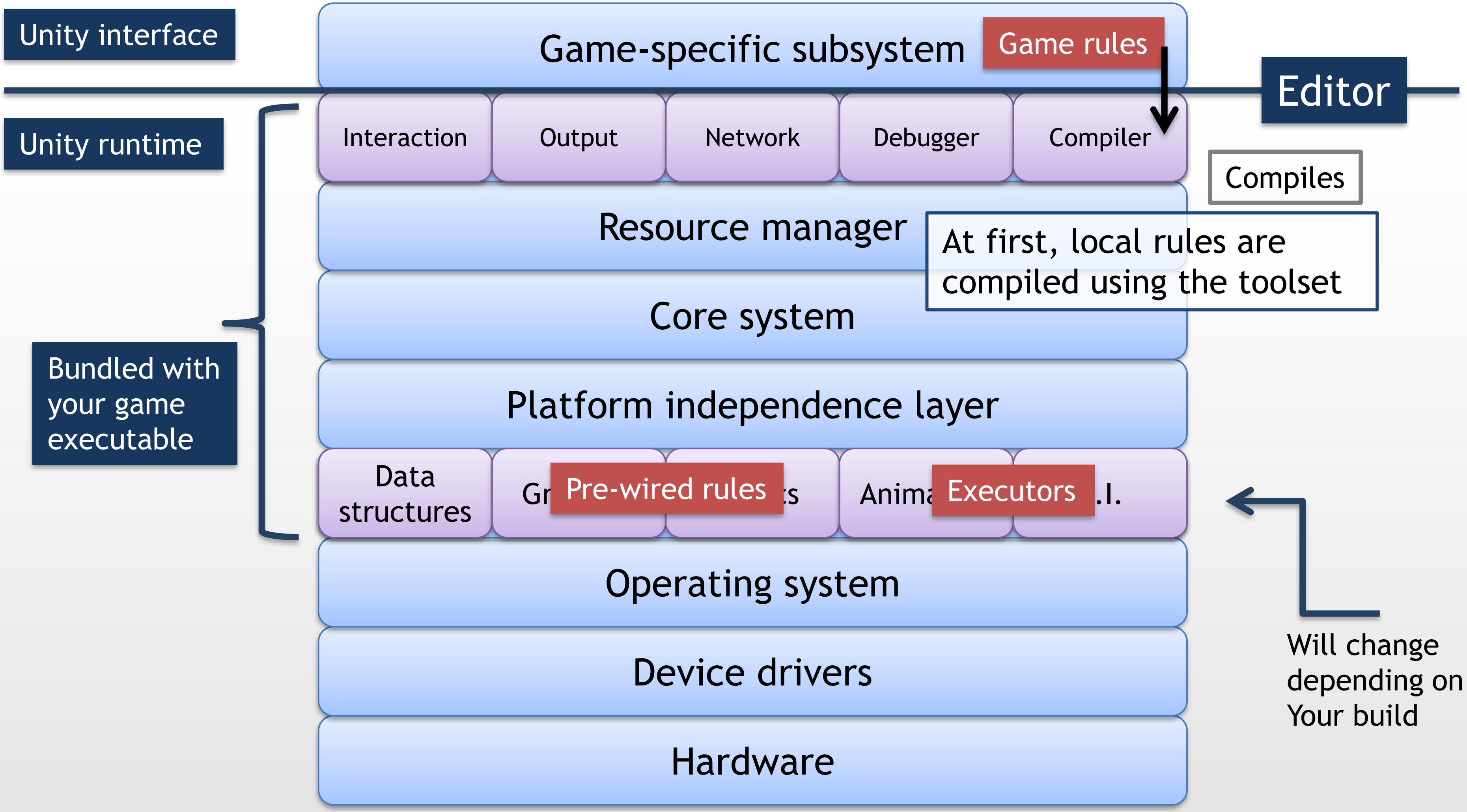


In particular, the platform-dependent layer is going to change depending on your build.

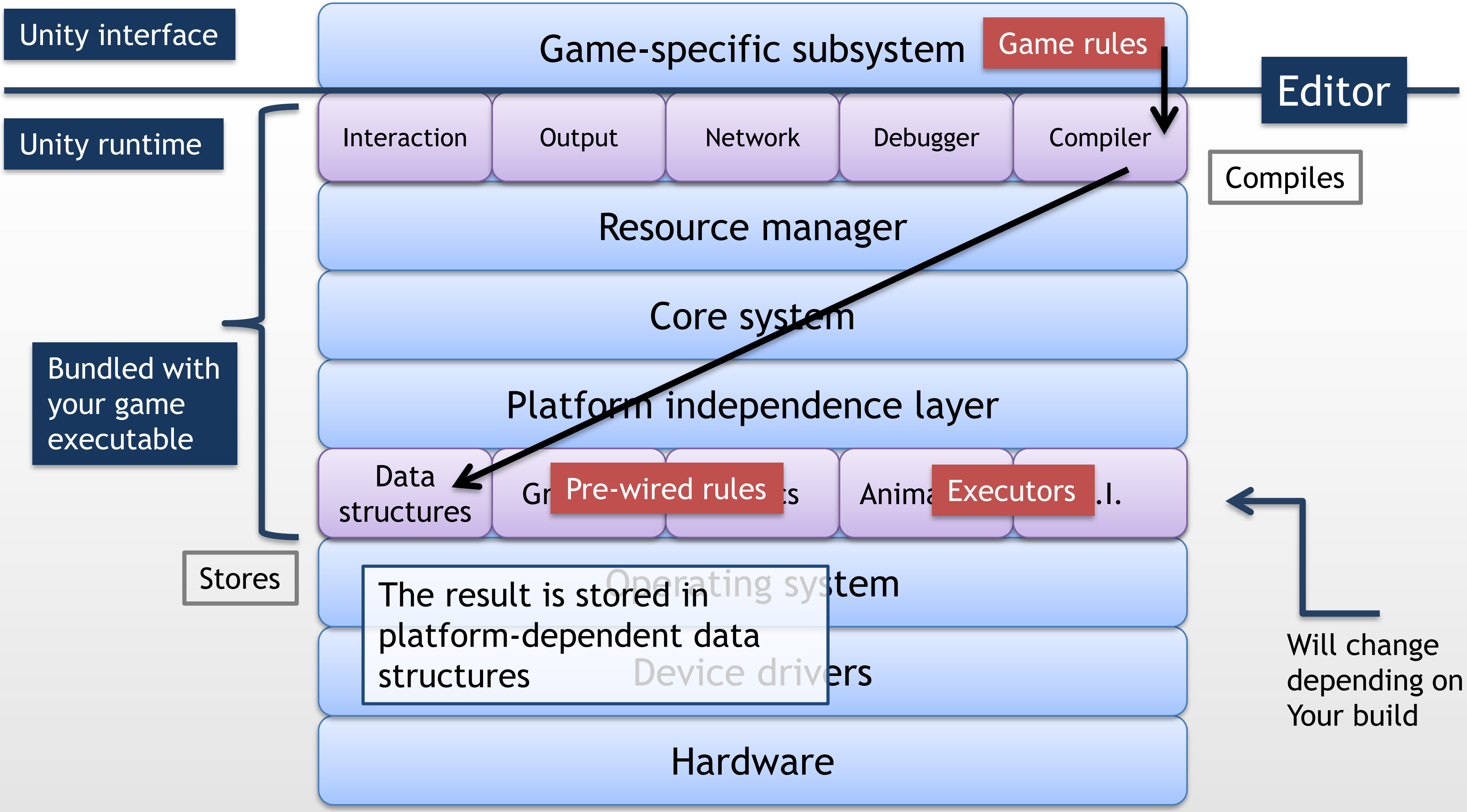
NOTE: we can use the layer of a different platform than the one we use for editing. Therefore, we can build an OSX game while running the editor on windows

A subset of the toolset up to the platform-dependent layer will be packed inside your executable

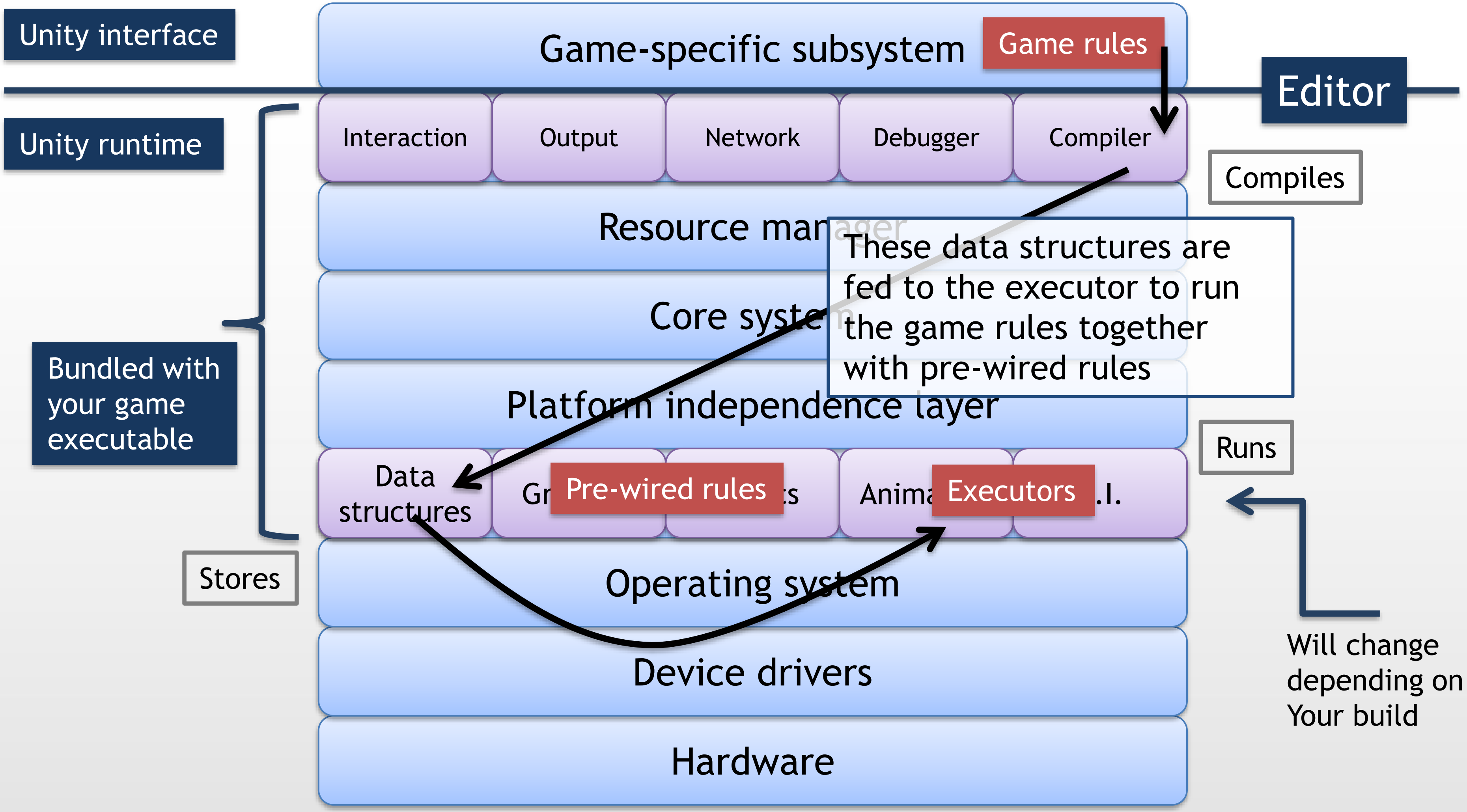
How do the Engine Uses Them?



How do the Engine Uses Them?

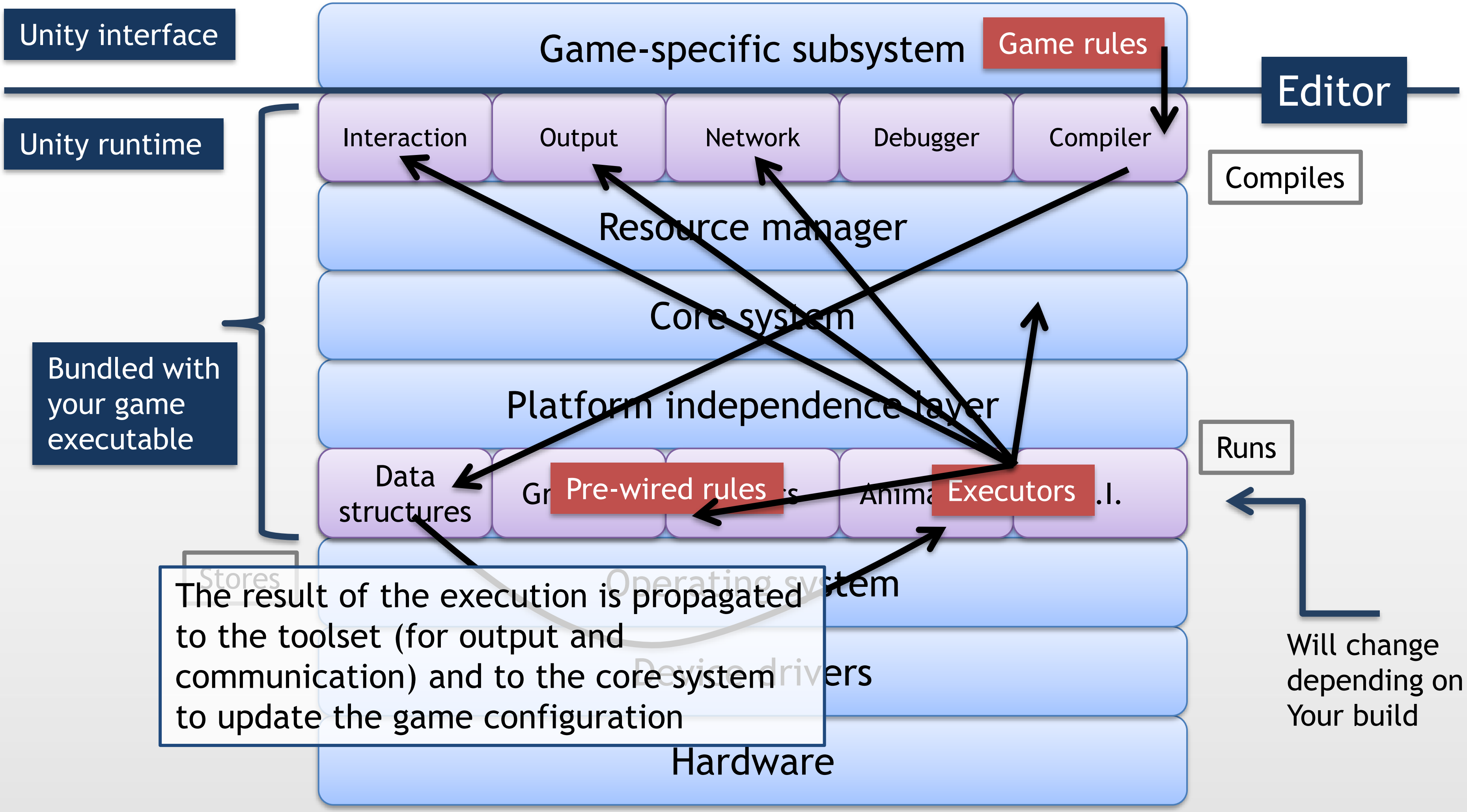


How do the Engine Uses Them?



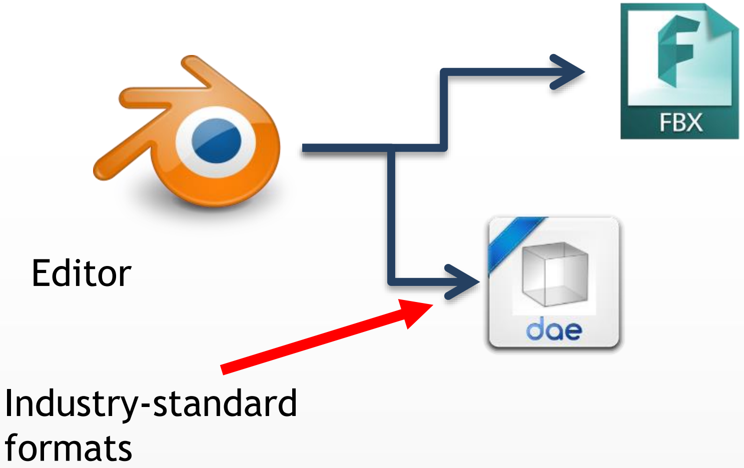
These data structures are fed to the executor to run the game rules together with pre-wired rules

How do the Engine Uses Them?



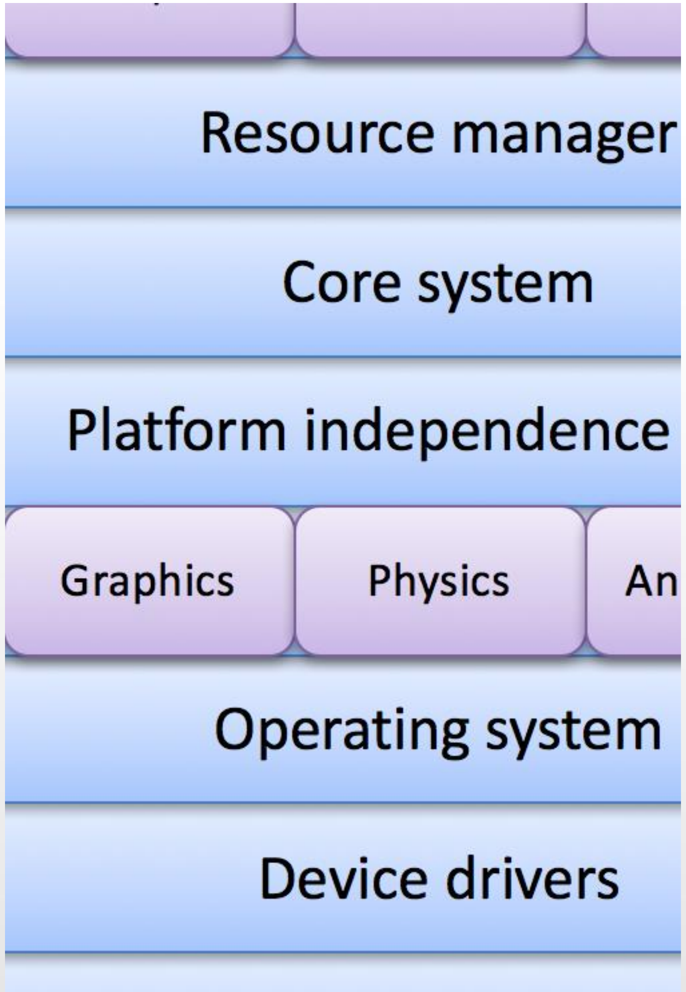
The result of the execution is propagated to the toolset (for output and communication) and to the core system to update the game configuration

A Word About Graphics (Conversion Pipeline)

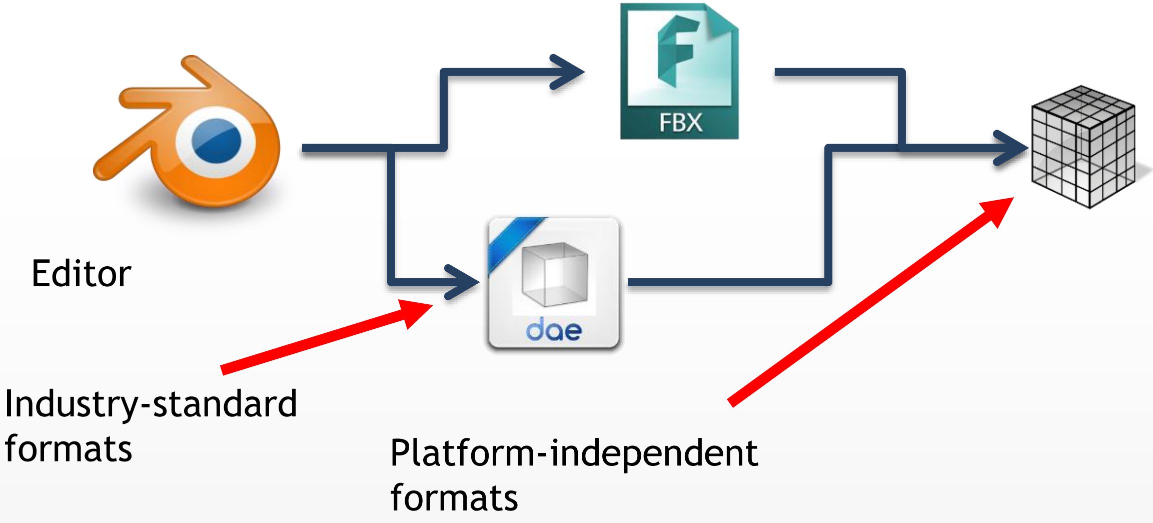


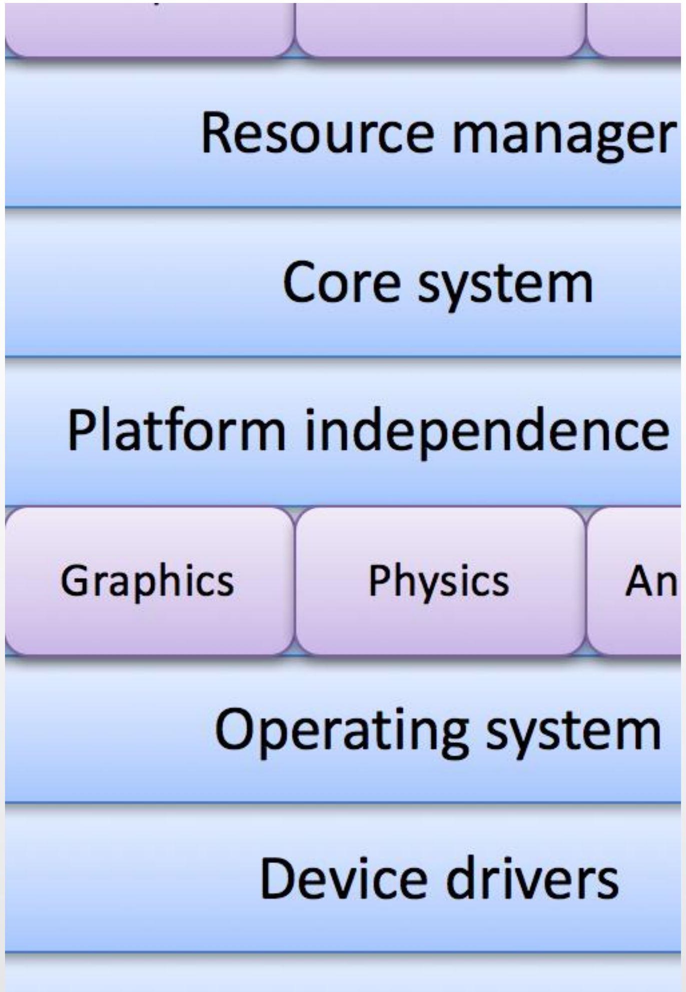
We use editors to create content in industry-standard formats.

The editor is completely unaware we will be using its exports inside a game engine



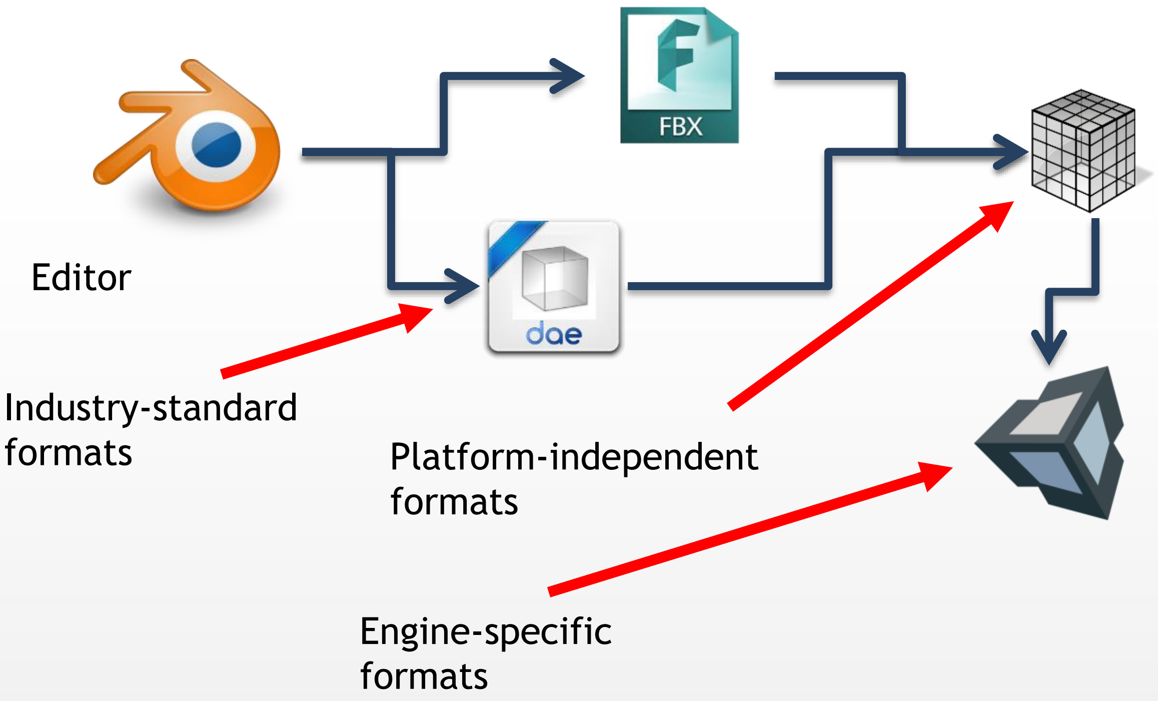
A Word About Graphics (Conversion Pipeline)



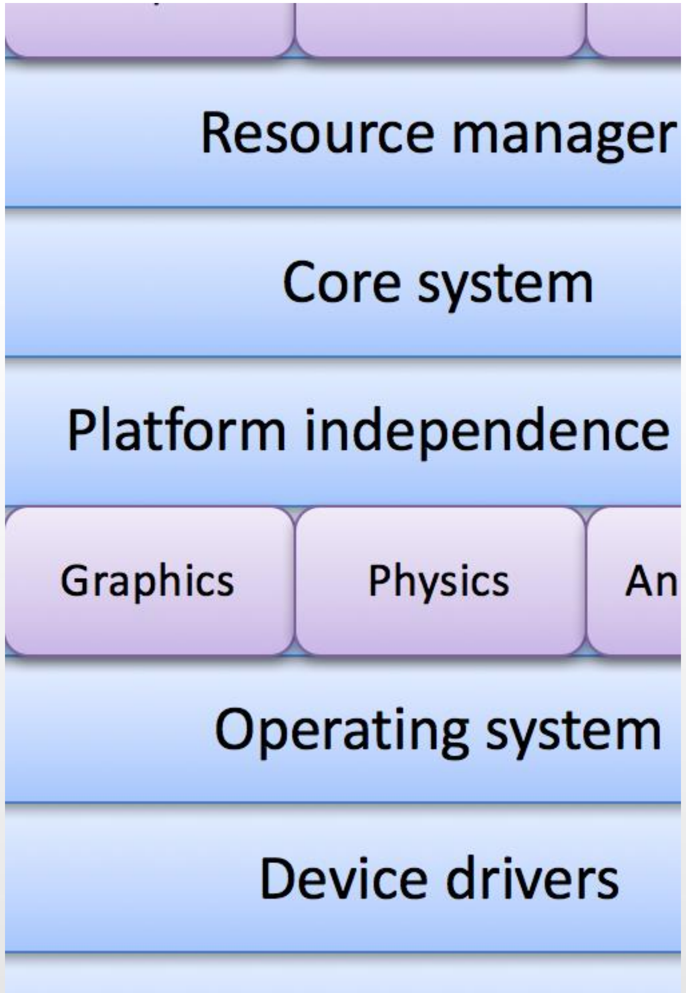


On import, the resource manager will convert (using the toolset) everything into a platform-independent format

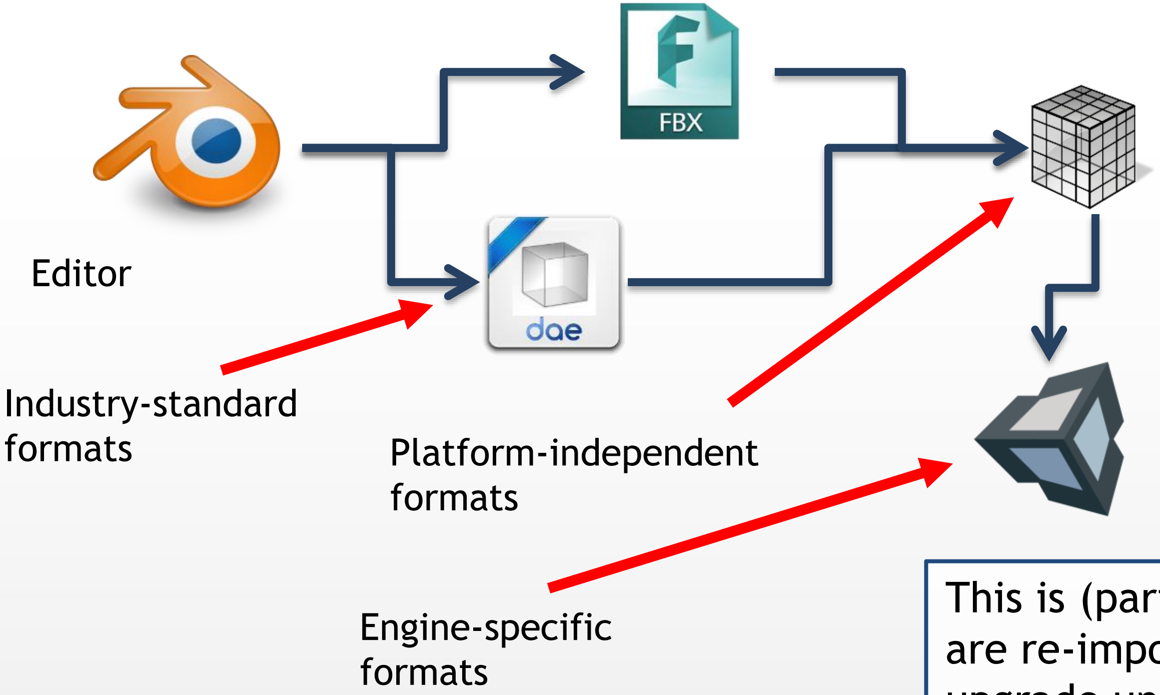
A Word About Graphics (Conversion Pipeline)



Then, this platform-independent format is converted into an engine-specific format to boost storage and usage performances



A Word About Graphics (Conversion Pipeline)

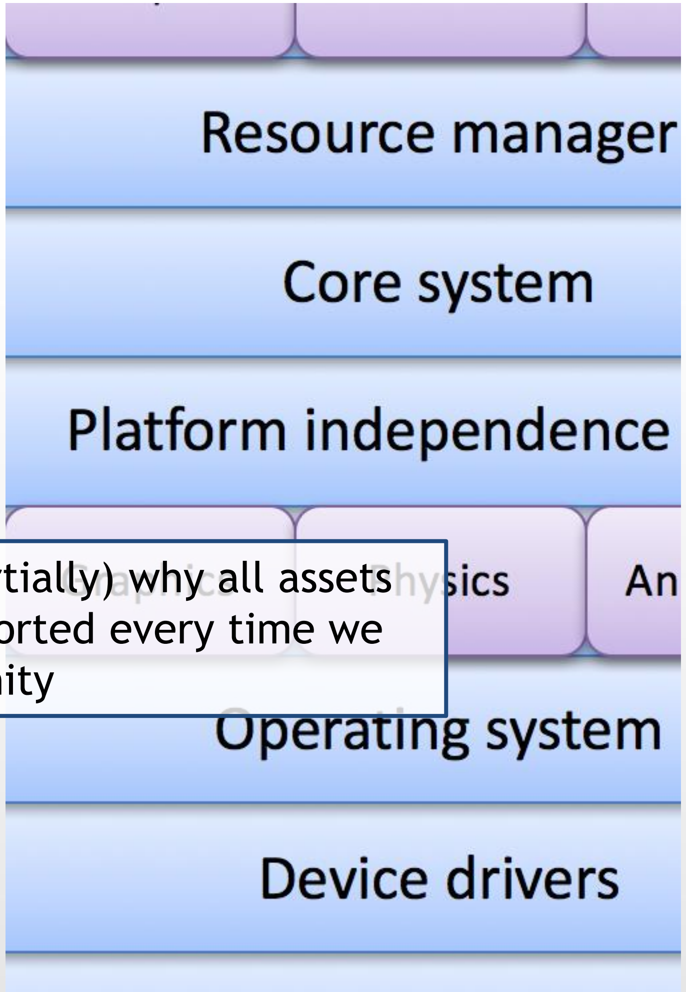


Editor

Industry-standard formats

Platform-independent formats

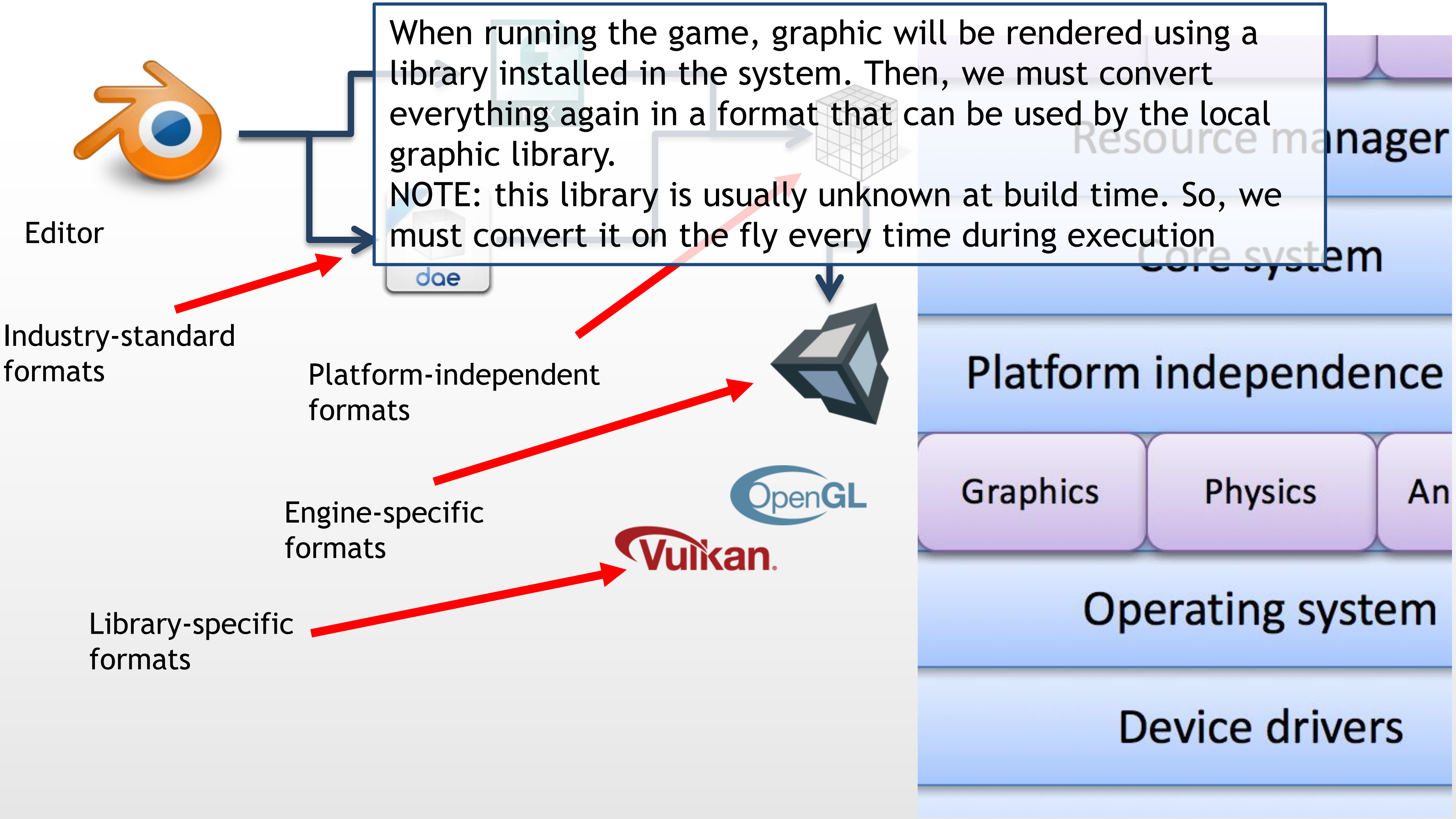
Engine-specific formats



This is (partially) why all assets are re-imported every time we upgrade unity

Then, this platform-independent format is converted into an engine-specific format to boost storage and usage performances

A Word About Graphics (Conversion Pipeline)



When running the game, graphic will be rendered using a library installed in the system. Then, we must convert everything again in a format that can be used by the local graphic library.
NOTE: this library is usually unknown at build time. So, we must convert it on the fly every time during execution

Editor

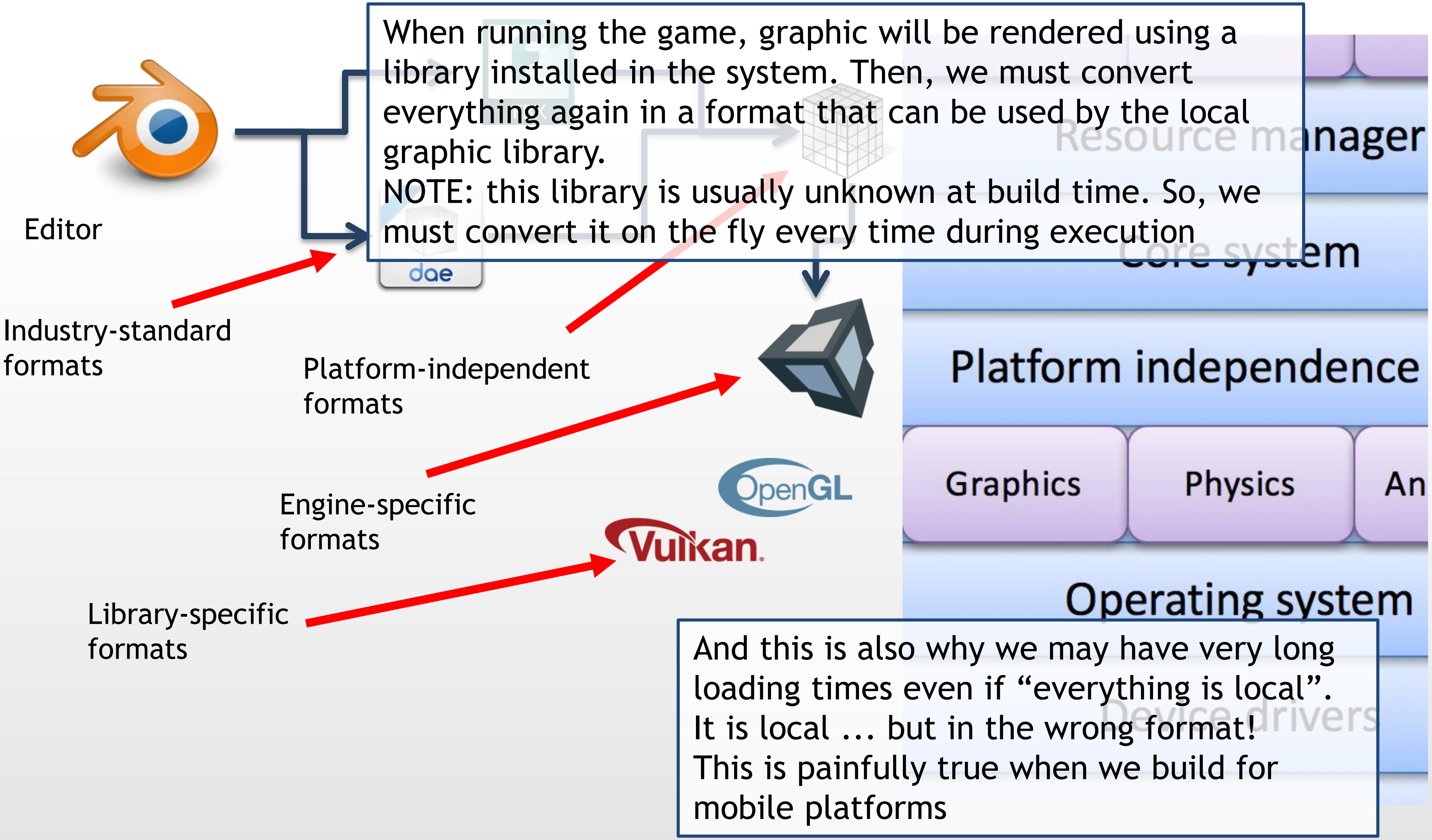
Industry-standard formats

Platform-independent formats

Engine-specific formats

Library-specific formats

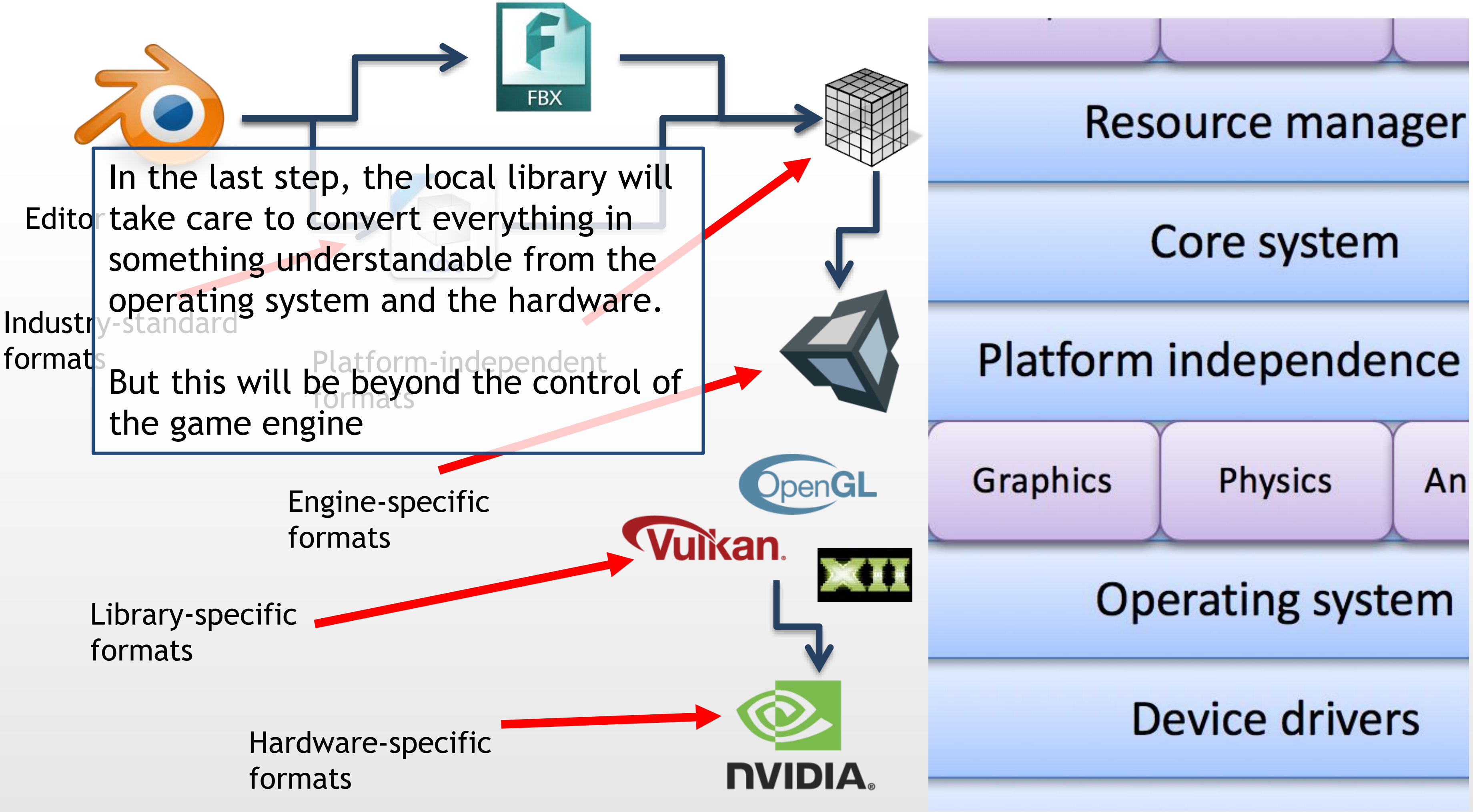
A Word About Graphics (Conversion Pipeline)



When running the game, graphic will be rendered using a library installed in the system. Then, we must convert everything again in a format that can be used by the local graphic library.
NOTE: this library is usually unknown at build time. So, we must convert it on the fly every time during execution

And this is also why we may have very long loading times even if “everything is local”.
It is local … but in the wrong format!
This is painfully true when we build for mobile platforms

A Word About Graphics (Conversion Pipeline)



In the last step, the local library will take care to convert everything in something understandable from the operating system and the hardware.

But this will be beyond the control of the game engine

Beware … distinction is blurry

- There is no strict definition of engine modules
  - The rendered might know how to render a full fledge ogre or may just provide basic functionalities
  - The network manager may implement SOAP or provide just sockets

Study Material

- Game Engine Architecture
  3rd edition, ISBN 1138035459
  by Jason Gregory
  Chapter 1, up to § 1.7.4 included