INTRODUCTION TO GAME PROGRAMMING & GAME ENGINES

Guillaume Bouyer, Adrien Allard

v4.3

guillaume.bouyer@ensiie.fr

www.ensiie.fr/~bouyer/

Objectives and schedule

Be aware of the technical problems and existing solutions that underpin the development of video games and other real-time interactive simulations

Understand the theoretical and technical components of game engines

Operate a high-level but relatively closed game engine (Unity).

Being able to start a project that looks like a game

Monday		Tuesday		Wednesday		Thursday		Friday	
Am	Pm	Am	Pm	Am	Pm	Am	Pm	Am	Pm
JIN Intro		Course Part. 1 + Project Part. 1		Course Part. 2 + Project Part. 2	SHS	ENV5001	Course Part. 2 + Project Part. 2	Adrien Allard Frog Collective, Talk + Project Part. 3	

Homeworks 1 2 3 4

1. Prerequisites : Unity installed, several completed Unity tutorials (ex. introduction from ENSIIE S4 course)
2. Continue project
3. Finish project part 1 & 2
4. Finish project part 3

bouyer@ensiie.fr

http://www.ensiie.fr/~bouyer/jin.html

Office 111 @ ENSIIE

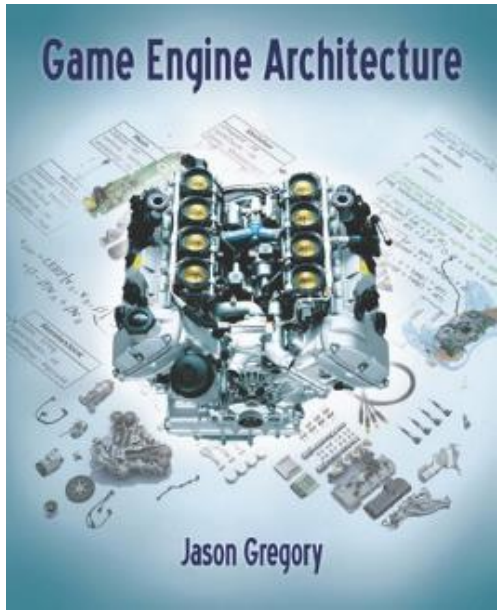
References

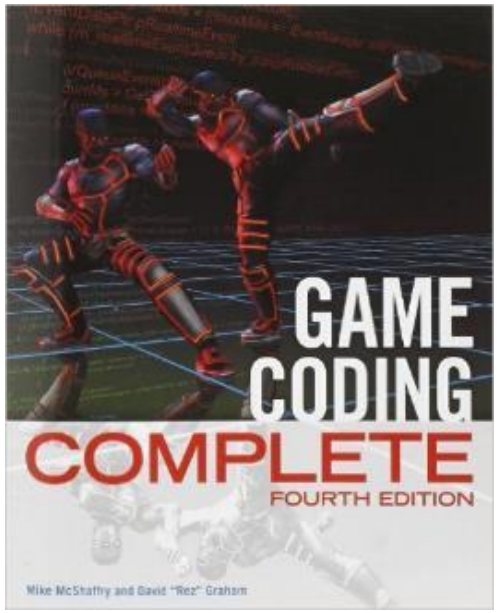
Game Engine Architecture, Jason Gregory, A K Peters/CRC Press, 2009-2015-2019 (http://www.gameenginebook.com/)

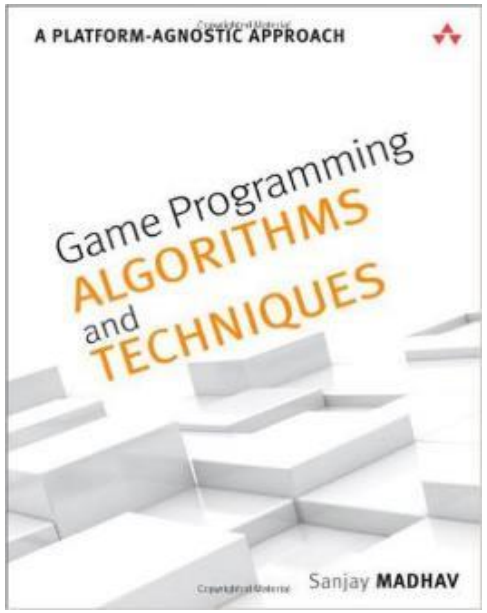
Game Coding Complete, 4th Edition, Mike McShaffry and David Graham, Course Technology, 2013

Game Programming Algorithms and Techniques, Sanjay Madhav, Addison-Wesley, 2013

Game Programming Patterns, Robert Nystrom, Paperback, 2014 (gameprogrammingpatterns.com/)







PART 1:

THE BASICS

TEAM / ROLES

Typical Game Team

Software developers

Runtime programmers:

Single engine/game system: rendering, AI, physics, UI…

Low level: memory, network…

Gameplay/3C : Character-Controls-Camera

Tools programmers: off-line tools for the team

=> Lead programmer (+ management), Technical director (high level)… Chief technical officer (for the entire studio)

Artists Produce visual and auditory content

Concept artists, 3D modeler, Animators, Texture & lighting artists, Actors (mocap, voice), Sound designers & composers, Technical artists…

=> Lead artists, art directors

when an engineer meets artists

Typical Game Team

Game designers Design the gameplay

- Macro level
  - Story arc, sequence of levels, high-level objectives of the player
- Individual levels or areas of the game world
  - Static background geometry, enemies spawning, items placement, puzzle elements…
- Technical level
  - Close to gameplay engineers and code (high-level scripting language)
- => Game director

Quality Assurance (QA)

- Tester, analyst, engineer

Producers

- Manage the schedule, the human resources, link between the dev. and the business units…

Publishers

- Marketing, manufacture and distribution (usually not the studio)

Our team

SCHMUP !

Producer : me

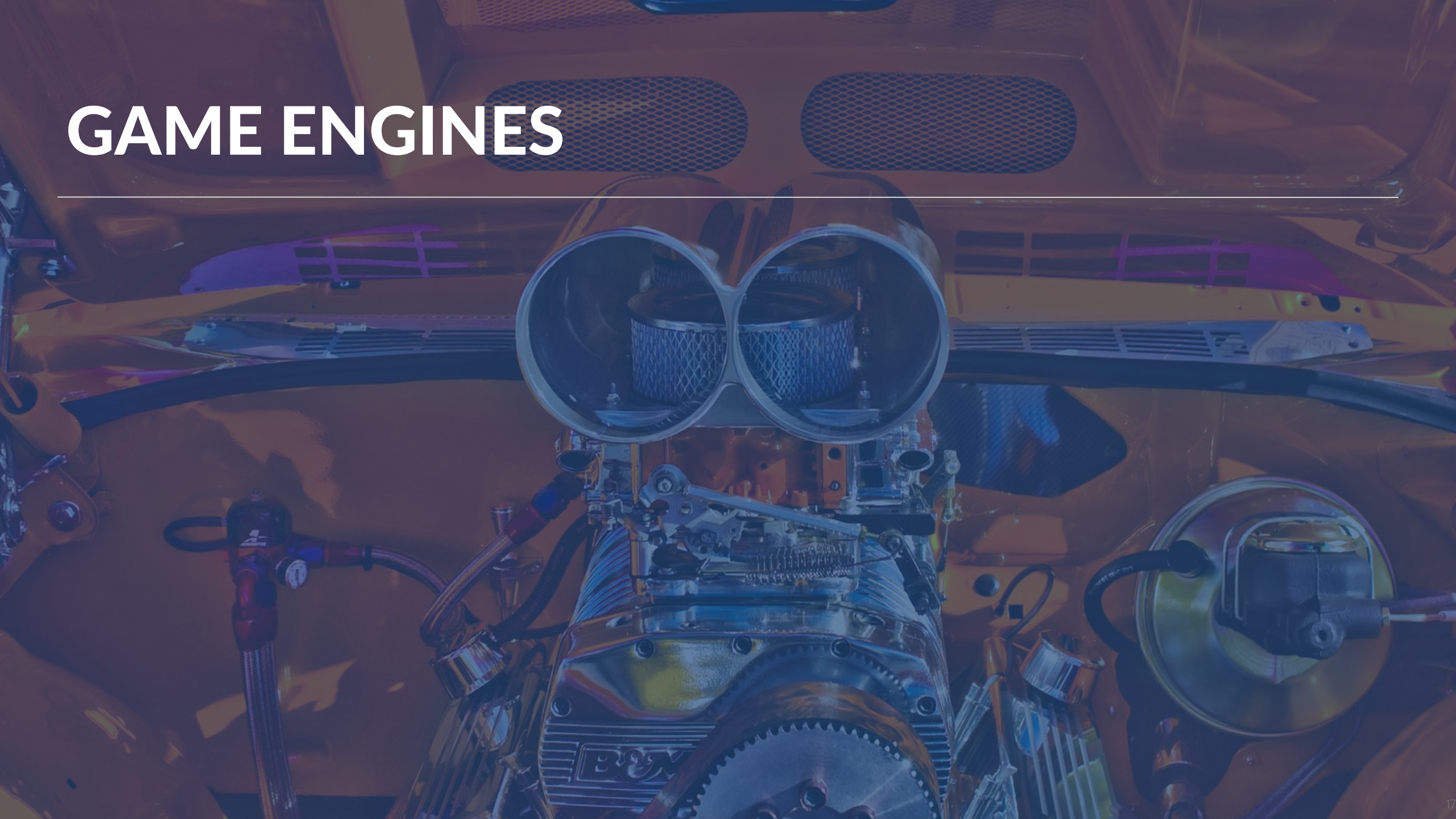
Game designer : contractor

Artists : internet…

Engineers : you

Self-published

GAME ENGINES



Game Engine

Extensible set of software that can be used as a basis for different games

Based on the division between:

- **Core runtime components**: 3D graphics rendering, collision detection, audio…
- **Art** assets, game worlds, **gameplay** that constitute the gaming experience

Benefits

- Create new games with new contents & "minimal" changes to reusable core software
- Mod community
- Engine licensing = additional income

Engine names?

Game Engine Examples

Doom & Quake Engines, ID tech (Id Software)

- Castle Wolfenstein 3D (92), Doom, Quake 1-4 (96-05), HalfLife (98), Medal of Honor…

Unreal Engines (Epic Games)

- Unreal (98-08), Deus Ex (00-03), Gears of War (06-13), Bioshock (07)…

Source Engine (Valve)

- Half-life 2, Team Fortress, Portal…

CryEngine (Crytek), Lumberyard (Amazon)

- FarCry (2004), Crysis (2007), Crysis 2 (2011), Crysis 3 (2013), Evolve (2015)…

Unity 3D

Gamemaker, Construct 2, RPG Maker…

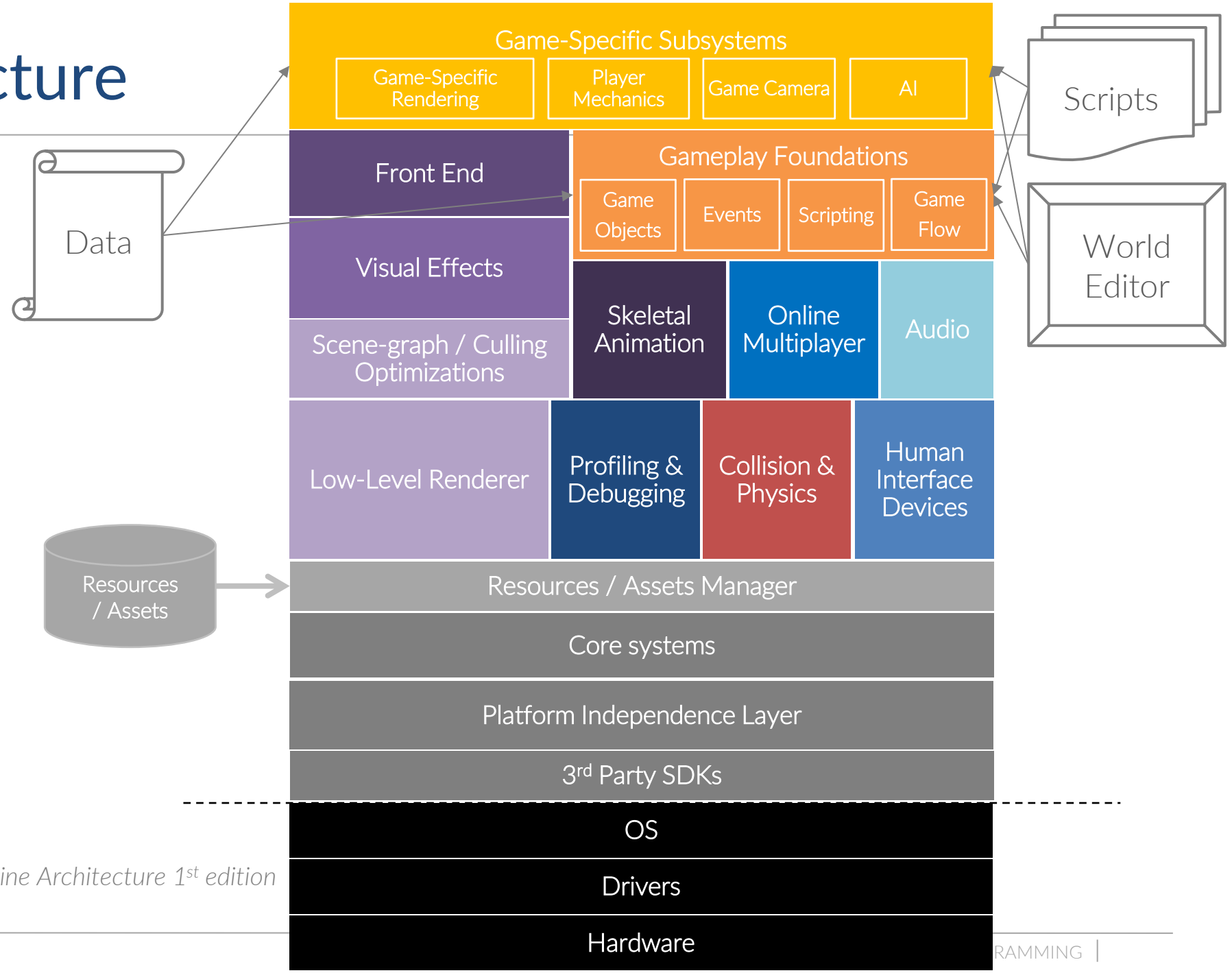
Open Source Engines

- Godot, Ogre 3D, Panda3D, Yake, Crystal Space, Torque, Irrlicht…

Proprietary in-House Engines

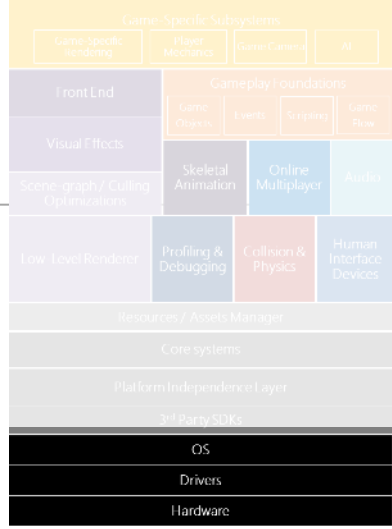
More at https://en.wikipedia.org/wiki/List_of_game_engines

Engine Architecture



Source: J. Gregory, Game Engine Architecture 1st edition

Platform



Hardware

PC, console, mobile…

Drivers

Shield the OS and upper layers from the communication details

Manage hardware resources

Operating System (OS)

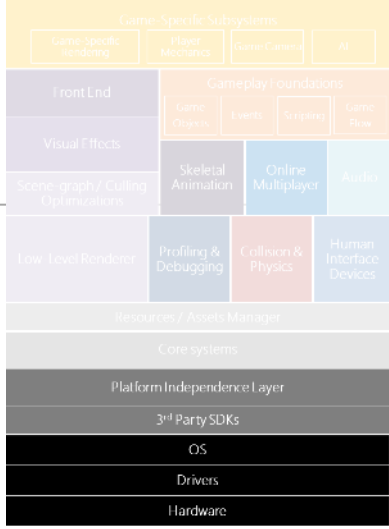
Runs all the time

Orchestrates the execution of multiple programs, including the game

Pre-emptive multitasking: time-sliced approach to sharing hardware

Rq: previously on console only a thin library layer compiled into the game executable (game "owns" the machine)

Engine Architecture



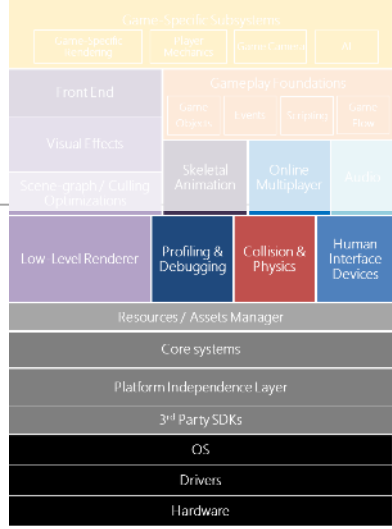
Third-Party SDKs and Middleware

- Data Structures and Algorithms
  - STL, STLport, Boost…
  - Memory allocation performance vs. convenience?
- Graphics
  - OpenGL, DirectX, libgcm (PS3), Edge (Naughty Dog)…
- Collision and Physics
  - Havok, PhysX, ODE, I-Collide, V-Collide, RAPID…
- Character Animation
  - Granny, Havok Animation, Edge…
- Artificial Intelligence

Platform Independence Layer

- Wrap/replace common standard library functions, OS calls, and other foundational APIs
- Shields upper layers from the knowledge of the underlying platform

Engine Architecture



Core Systems: common useful utilities

- Assertions, unit testing…
- Memory allocation
- Custom data structures and algorithms
- Math library, random number generator

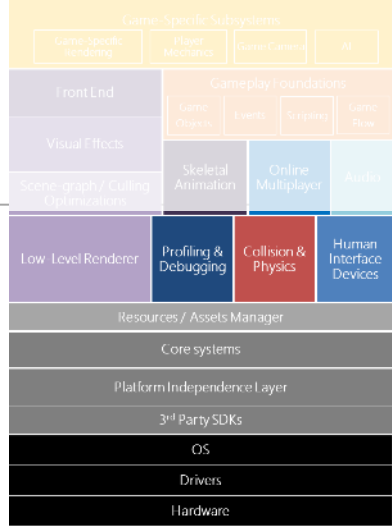
Resources/Assets Manager

- Interfaces for accessing game assets and other input data
- 3D model, texture, material, font, skeleton, collision, map…

Profiling and Debugging Tools

- Profile performance and analyze memory to optimize
- In-game debugging facilities
- Record and play-back gameplay
- Config, stats…
- Commercial or custom

Engine Architecture



Rendering Components/Engine

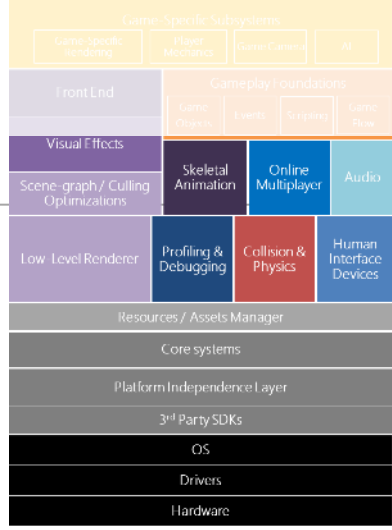
- Low-Level Renderer
- Scene Graph/Culling Optimizations
- Visual Effects
  - Particles, decal, light and environment mapping, dynamic shadows, full-screen post effects (HDR, AA, color correction…)
- Front End
  - 2D or 3D: Heads-up display (HUD), in-game menus, console, development tools, in-game GUI
  - Full-motion video or in-game cinematics system

Animation

Collision and Physics

- Collision detection
- "Rigid body kinematics and dynamics" system

Engine Architecture



Human Interface Devices

Manages and transforms the low-level raw data from the hardware

Provides high-level game controls and detection (chords, sequences, gestures…)

Audio

Needs lots of tuning, engines vary greatly in sophistication

Ex: XACT (Microsoft), SoundR!OT (EA), Scream (Sony)…

Multiplayer/Networking

Single-screen, Split-screen, Networked, Massively multiplayer online

Single-player is often special case of a multiplayer game: better to design multiplayer features at the beginning

Engine Architecture

Gameplay Systems: at the interface between game and engine

- Game's rules, objectives, and dynamic world elements
- Game object model
- Game objects updating
- Messaging and event handling
- Scripting language
- Level management and streaming
- Objectives and game flow management
- Artificial Intelligence

Game-Specific Subsystems: features of the game

- Mechanics of the player character, in-game camera systems, AI for NPCs, weapon systems, vehicles…

Assets Management

Game resources

- 3D model/mesh
- Material properties, texture, shaders…
- Animations, skeletal data
- Collision and physical properties
- Audio clips
- Particles system…

Usually created with external specialized content creation tools

- Ex. Maya, 3ds Max (Autodesk), Photoshop (Adobe), Soundforge…

Unreal Editor browser

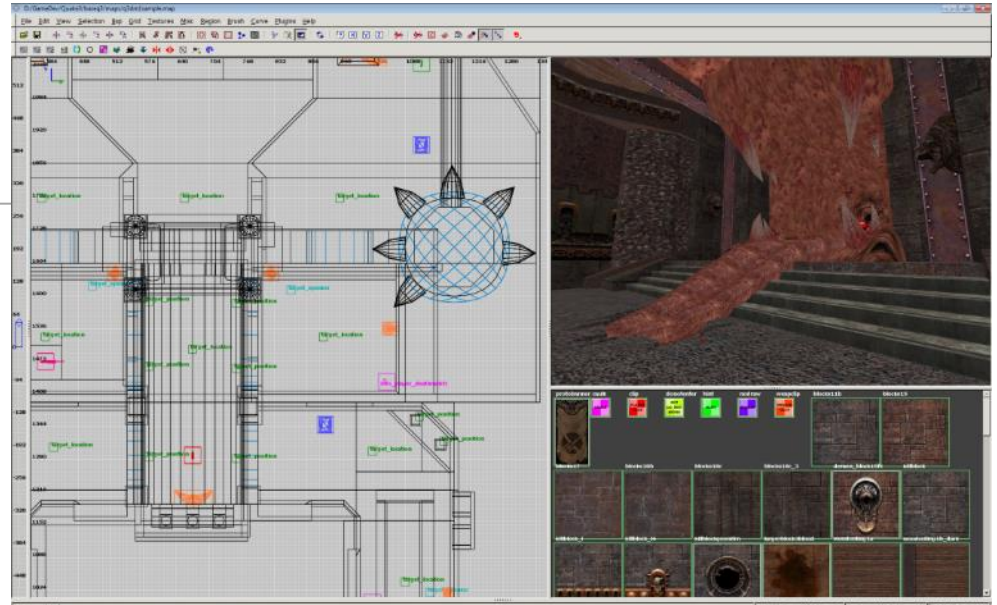
Assets Management

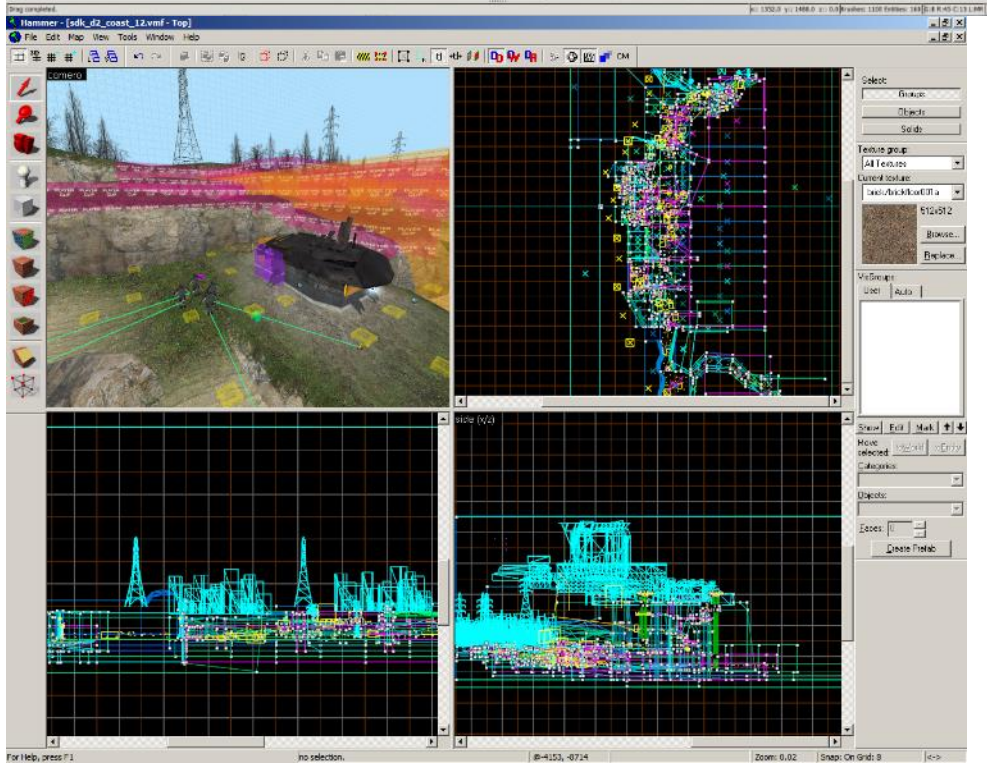
- Data formats of assets rarely ready for direct in-game use
  - In-memory model too complex
  - File format too slow to read at runtime, or proprietary
- Asset Conditioning Pipeline (ACP)
  - Data exported to a more accessible standardized or custom format, then further processed (ex. differently for each target platform)

Game World Editor

- GUI tool(s) to build the game world
  - Dedicated, with custom rendering engine
  - Integrated into a 3D geometry editor
  - Integrated into the engine
- Rapid iteration
  - Dynamic tweaking

GtkRadiant (Quake engine)





Hammer (Source engine)

Game World Editor

- Insertion and selection of game objects
  - Placement and alignment aids (special handles, assistance tools)
  - 3D or tree view (hierarchy)
  - Special object handling (lights, cameras, particles…)
- Visualization and navigation
  - 3D perspective view of the world and/or a 2D orthographic projection
  - View pane divided into sections
  - Camera control
- Levels/world chunks
  - Creation, saving, loading and management
  - Tools for authoring specialized static elements: terrain, water, background sprites…

Game Scripting

Provides high-level, relatively easy access to features of the engine to

- Develop a new game
- Mod an existing game
- Customize the functionalities of the engine's subsystems (callbacks)
- Create data structures consumed by the engine
- Create new game object types or components (inheritance, composition)
- Handle communication between objects…

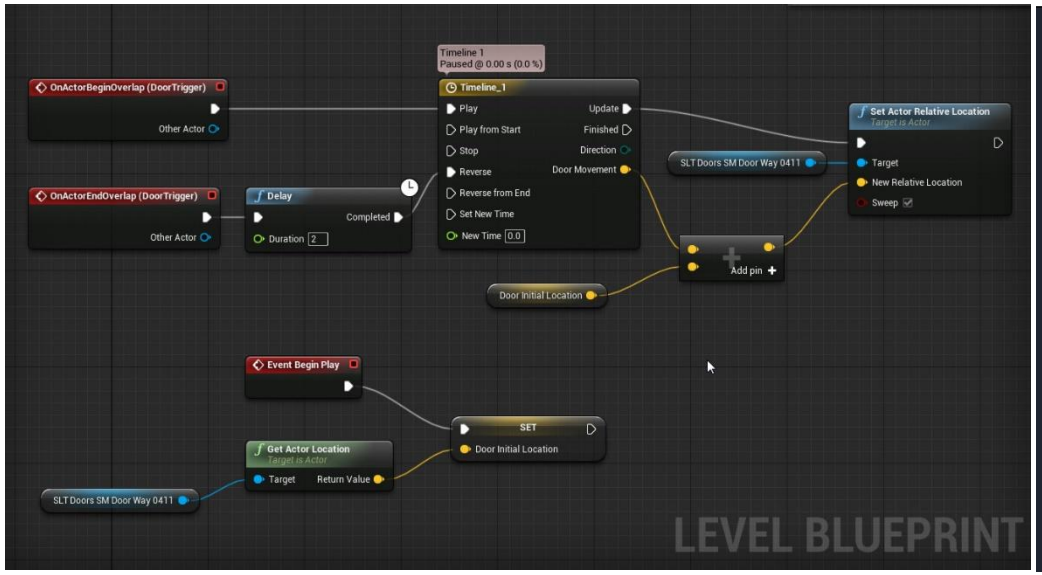
Benefits

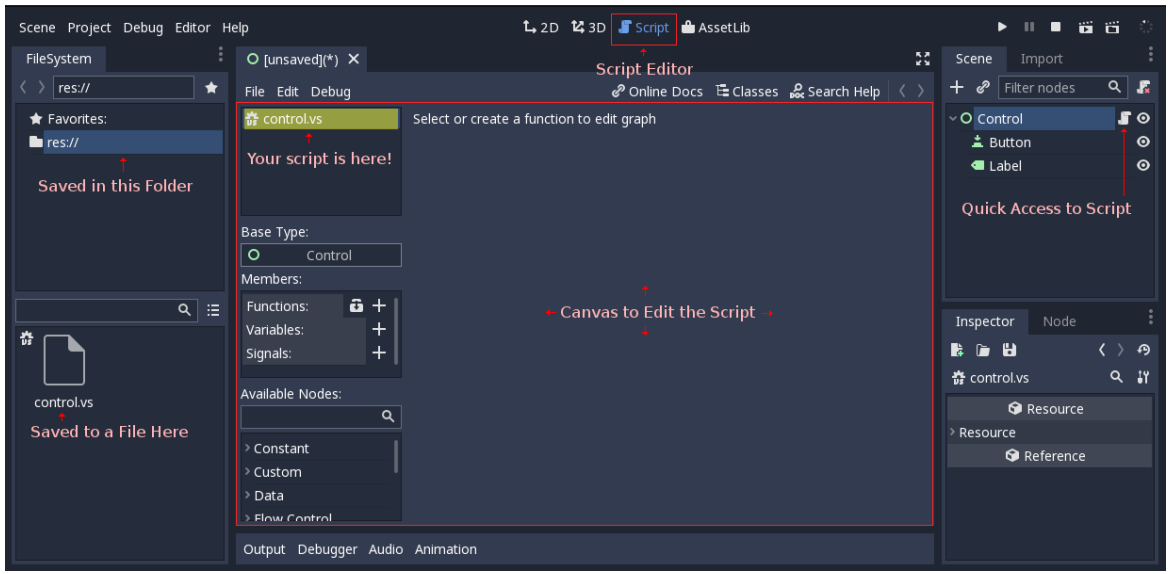
- Faster iteration than native language source code (sometimes no recompilation/relink, no game shut down and rerun)
- Customizable to suits the needs of a particular game
- Can make common tasks simple and less error-prone

Examples: QuakeC, UnrealScript, LUA, Python, Pawn / Small / Small-C

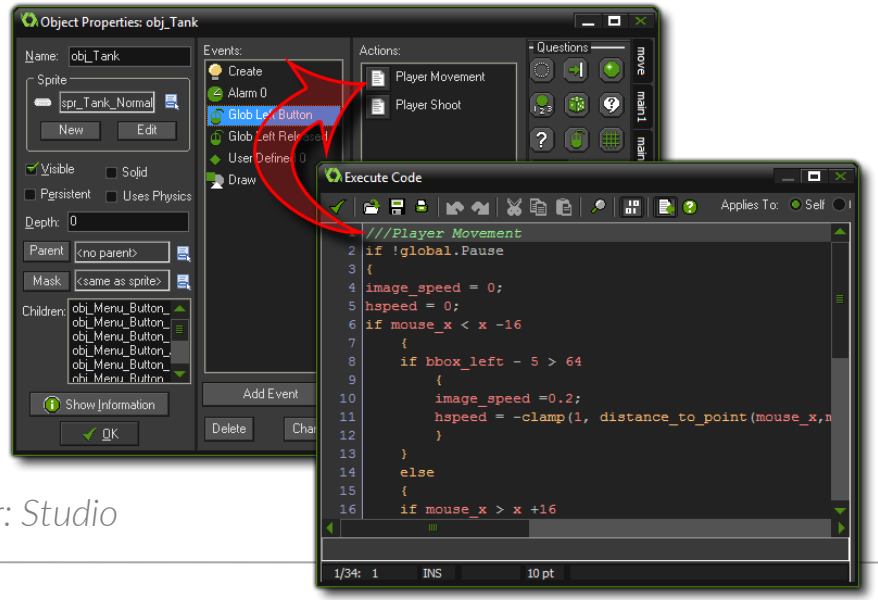
Visual Scripting Editors

Unreal Engine Blueprint

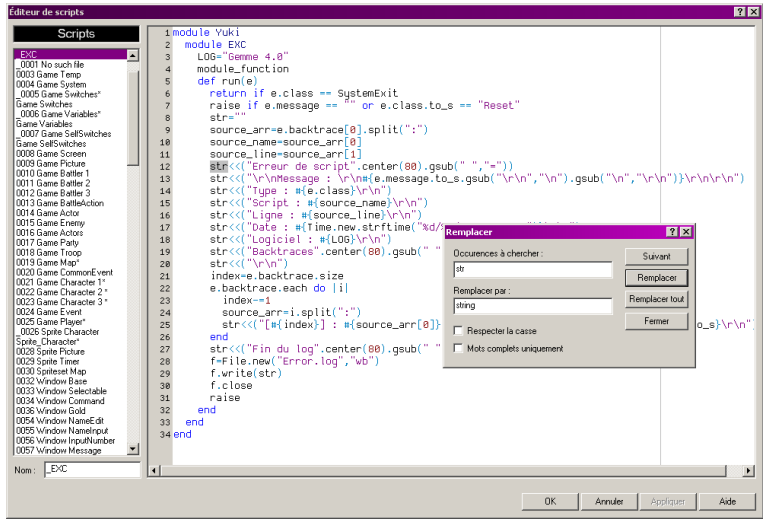




Godot Visual Scripting

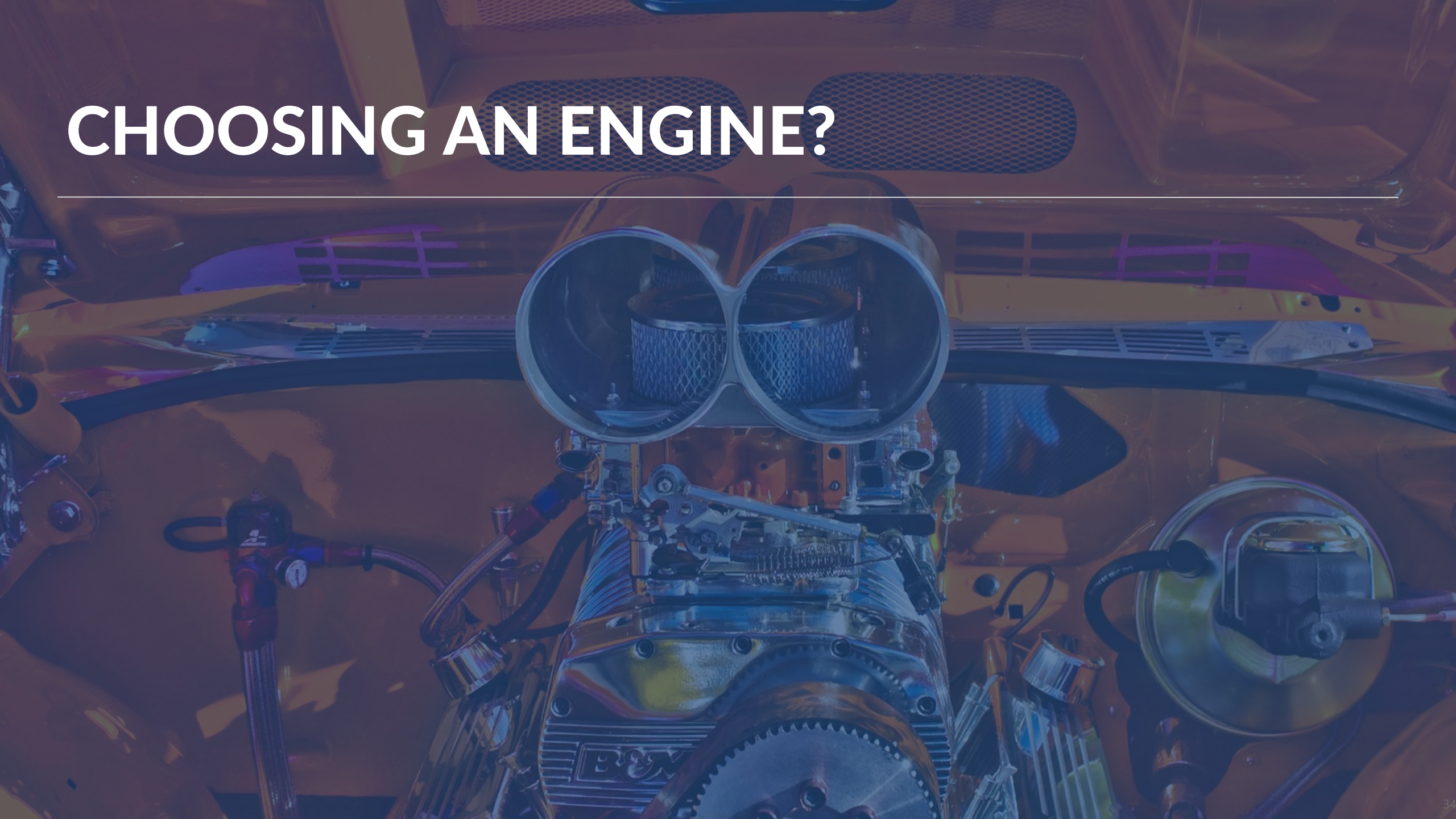


GameMaker: Studio



RPG Maker

CHOOSING AN ENGINE?



Questions

1. What's my timeframe?
2. How big is my team?
3. What's my budget?
4. Am I good at programming?
5. What genre is my game?
6. How big is my scope/what platform am I releasing on?

Source: blackshellmedia.com/2016/09/29/6-crucial-questions-ask-choosing-game-engine/

Choosing an engine for 1 person

1. Pick the game engine for you, not for your game
2. Apply the marketing filter
3. Performance is not a feature
4. Prefer a programming language you already know
5. Documentation
6. Maintenance
7. Support
8. Cost
9. Features

Source: "The Game Engine Dating Guide: How to Pick Up an Engine for Single Developers", Steffen Itterheim (no more available)

General/Optimal Trade-off

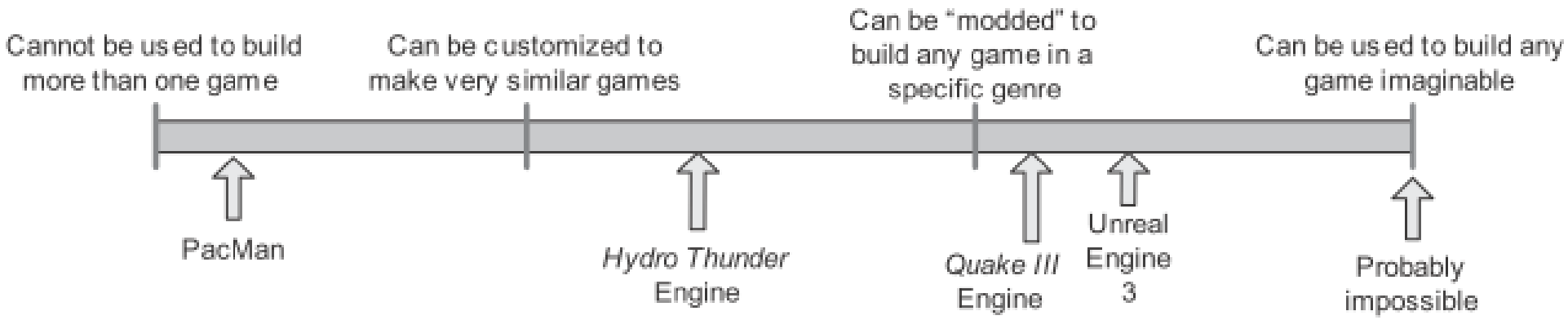
Initial game engines designed and tuned to run a particular game on a particular hardware platform

Now, **technological and content overlap** between games/genres + more powerful hardware

=> Possible to **reuse the same engine** across disparate genres and hardware platforms

But **more general** engine => **less optimal** for running a particular game

=> **Assumptions** about how the software will be used and about the target hardware



Reusability gamut (Source: J. Gregory, Game Engine Architecture 1st edition)

Technical differences by genre

First-Person Shooters (FPS)

Rendering	high fidelity & large 3D worlds (often specific environment)
3C	responsive camera & aiming, forgiving character motion and collision (“floaty”)
Animations	high-fidelity player’s arms and weapons, high-fidelity NPC
AI	non-player characters
Multiplayer	small-scale online capabilities (ex. 64), various game mode, match making…
Gameworld	complex level design, wide range of items (weapons, pickable…)

Battlefield 1

Overwatch

Third-Person games

Rendering, AI, Multiplayer… = FPS

3C	emphasis on character’s abilities and locomotion, 3rd-person “follow camera”, complex camera collision system
Animations	high-fidelity full-body player’s avatar, high-fidelity NPC
Gameworld	complex level design with various locomotion modes: platforms, ladders, ropes, vehicles…, puzzle-like elements

Tomb Raider

Technical differences by genre

Fighting games

Rendering	high-definition character (skin, sweat…), physics-based cloth and hair simulations
3C	robust user input system, complex button and joystick combinations, accurate hit detection
Animations	rich and high-fidelity fighting characters animations
AI	non-player characters
Multiplayer	typically 2 players local or online, ranking
Gameworld	relatively static backgrounds (crowds)

Street Fighter 5

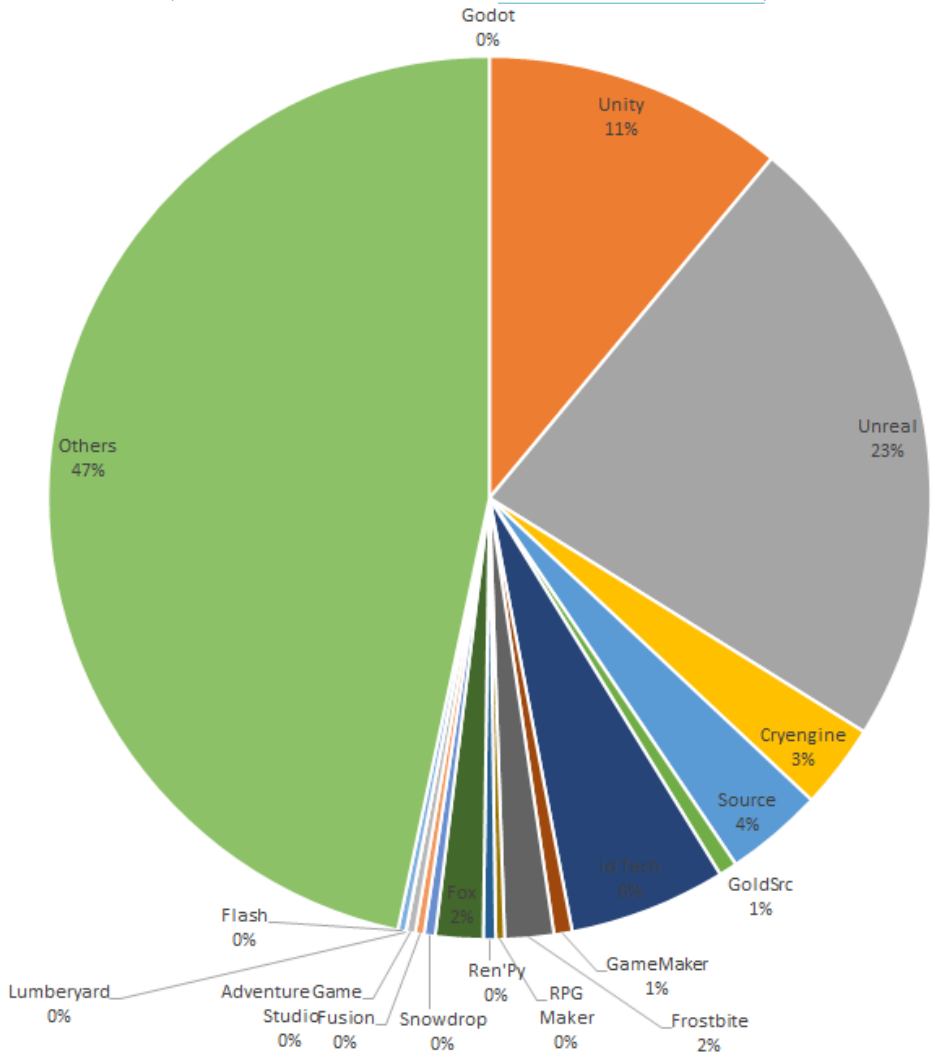
Racing games

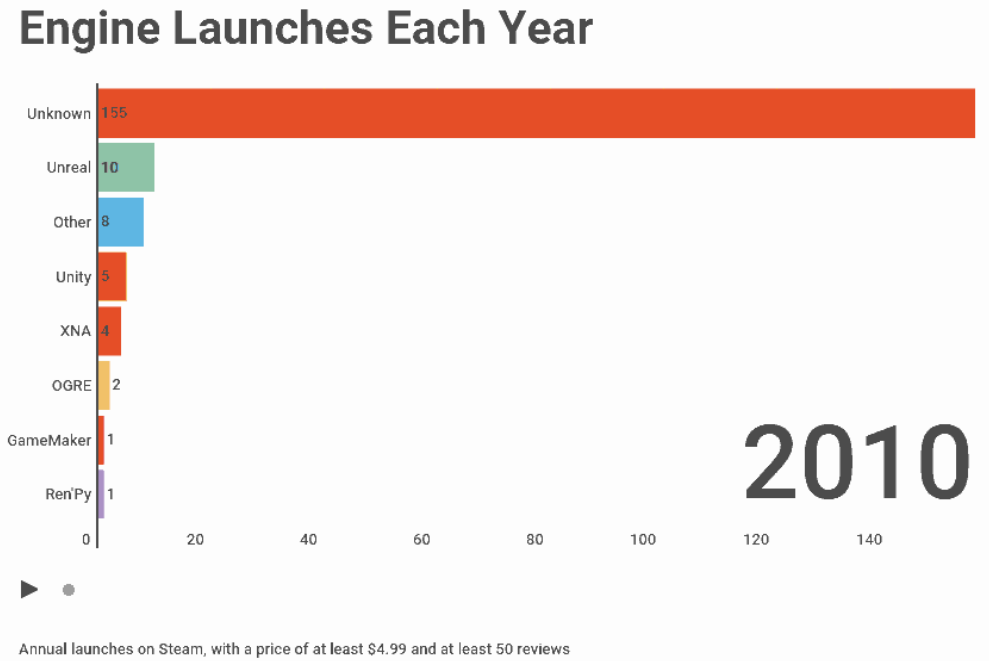
Rendering	high-fidelity vehicles, track, and surroundings, can optimize rendering (distant background elements…)
3C	various cameras: 3rd-person, 1st-person…
Physics	realistic (tires, materials, collisions…)
AI	path finding for non-player vehicles, driver assistance
Multiplayer	small-scale online capabilities, local split-screen, ranking…
Audio	high-fidelity (tires, engines, collisions…)

Forza Horizon 4

Reality: Market Fragmentation

Market share of engine for notable Steam games (source 2018 : reddit shorturl.at/ruF25)





https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown

https://emploi.afjv.com/

29 Programmer jobs (11/09/2025)

Unity : 8+3 = 38%

Unreal 3+3 = 20%

C++ : 1+8 = 31%

Our game

Game Objects: Components of the Game World

Agents, actors, entities…

- Player & non-player characters
- Environment:
  - Terrain, building, road, bridge, trees…
- Locomotion tools:
  - Vehicles, platforms, ropes, graspable edges…
- Scenery and ambiance elements:
  - Background, furniture, particle emitters, lights…
- Player items:
  - Weaponry, armor, collectible objects, floating power-ups and health packs…
- Invisible utility data:
  - Collision information, volumetric areas (events or logic), navigation mesh, paths of objects…

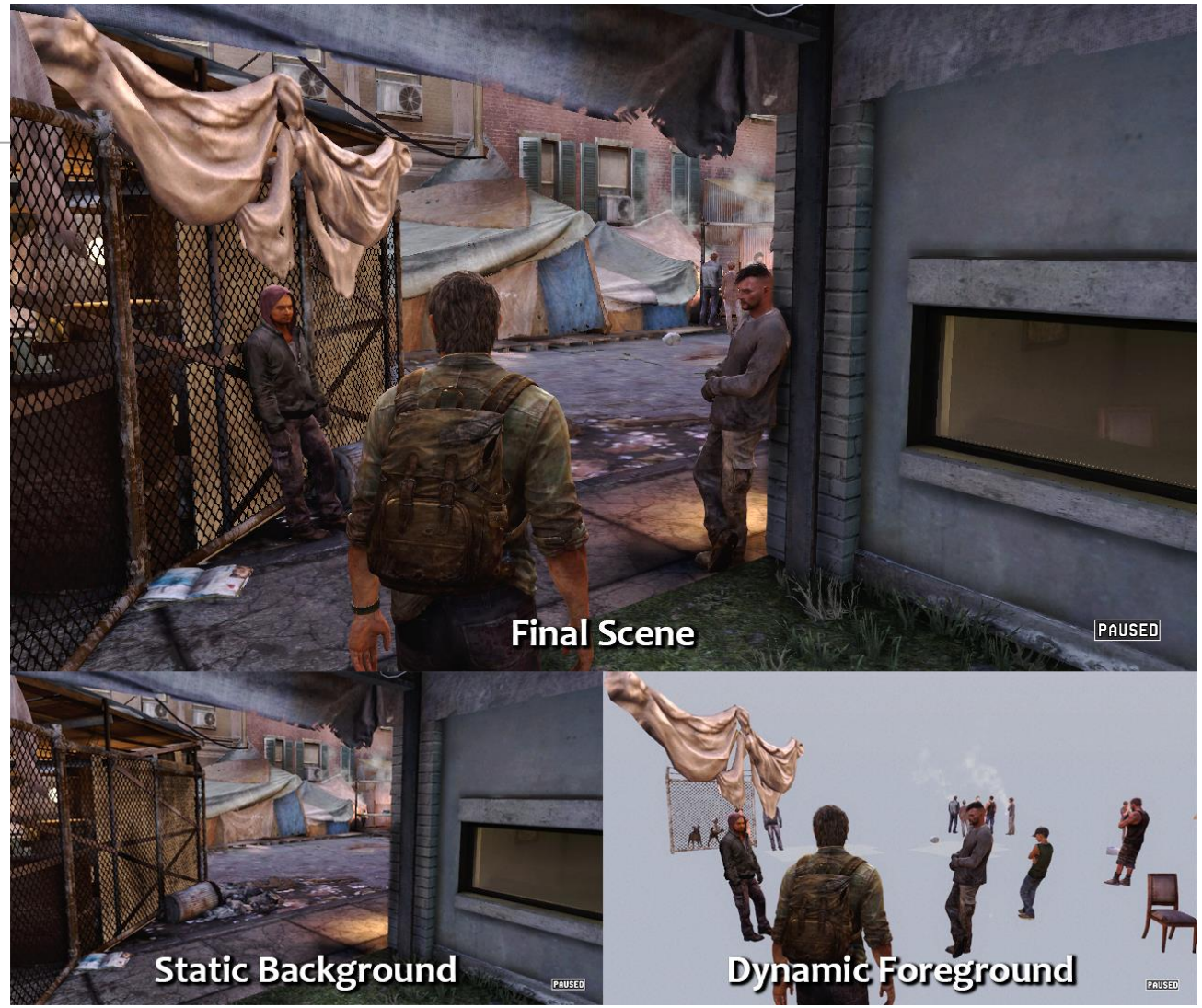
3D objects, data containers, spatial zones, invisible or special objects…

Dynamic vs. Static Objects

"Dynamic" objects
- Evolving state
- Main support of the gameplay
- Usually more CPU expensive

"Static" objects
- Stable state
- No critical interaction with gameplay (but layout can play a role)
- Possible optimizations (eg. baked lighting)

Dynamic/Static ratio
- High ratio => perception of a more “alive” and interactive game world
- Distinction often blurry (eg. waterfalls, destructible elements)
- In general, limited number of dynamic elements in a large static background area



Uncharted

Game Objects: Types and Properties

Object-oriented logic

Types/Instances

Attributes/Values

- Current state of the object (locations, orientations, parameters…)
- Atomic data types, Key-value pairs, Arrays, Structures, Strings…

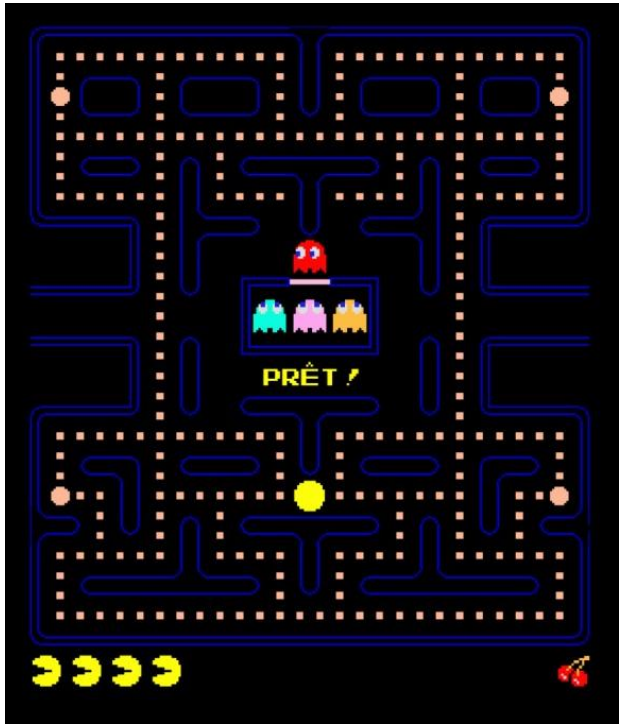
Behavior

- How the state will change over time and in response to events
- usually controlled with Data-driven configuration parameters or Scripting language

Different types of objects have different attributes and different behaviors

All instances of a type have the same attributes and behaviors, but different values

Ex: Types and instances for Pacman ?



GAME LOOP

PRINCIPLES

Game Loop

Game composed of many interacting subsystems

I/O, rendering, animation, collision detection, rigid body dynamics simulation (optional), multiplayer networking (optional), audio, game objects model…

Subsystems require periodic update with various rates

Rendering and Animation: 30 or 60 Hz

Dynamics simulation: higher rates (e.g. 120 Hz)

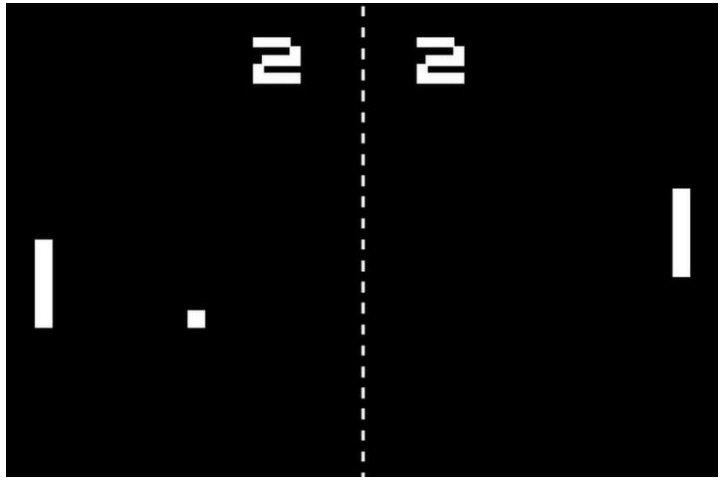
Higher-level systems (e.g. AI): 1 or 2 times/second (can be async. with rendering)

⇒ Solution: a single loop to update everything

```
while (true) {                  //(need something to quit…)
    processInput();             //but don't wait for input
    updateGameState();          //one step of the game simulation
    renderGame();               //generate outputs
}
```

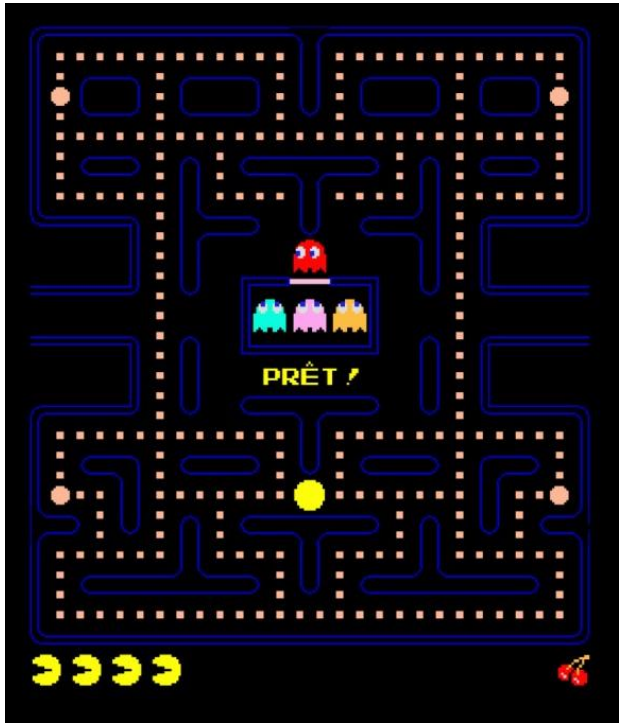
Theoretical Example: Pong

```
initGame()
while (true) {          // game loop
    readHumanInterfaceDevices();
    if (quitButtonPressed())
        break;      // exit the game loop
    movePaddles();
    moveBall();
    collideAndBounceBall();
    if (ballImpactedSide(LEFT_PLAYER)){
        incrementScore(RIGHT_PLAYER);
        resetBall();
    }
    else if (ballImpactedSide(RIGHT_PLAYER)) {
        incrementScore(LEFT_PLAYER);
        resetBall();
    }
    renderGame();
}
```



Theoretical Example: PacMan

```
while (player.lives > 0){
    // Process Inputs
    JoystickData j = grabRawDataFromJoystick();
    // Update Game World
    player.move(j);
    for (Ghost g in world){
        if (collision(player, g))
            killPlayerOrGhost(player, g);
        else
            g.move(player.position);
    }
    // Pac-Man eats any pellets
    ...
    // Generate Outputs
    renderGame();
}
```



Our game loop (theory)?

SCHMUP !

```
initGame();
while (getHealth(player)>0) {      // game loop
    readHumanInterfaceDevices();
    if (quitButtonPressed())
        break;                     // exit the game loop
    moveAndShootShip();            // up/down/left/right/shoot based on inputs
    movePlayerBullets();
    moveAndShootAllEnemies();
    moveEnemiesBullets();
    foreach (enemyBullet) {
       if (collide(player, enemyBullet)){
                    decreaseHealth(player);
       }
    }
    //... same for player's bullets
    updateScore()
    renderGame();                  // draw entire content
}
```

GAME LOOP

TIME MANAGEMENT

Frame rate

```
while (true) {
    processInput();
    updateGameState();
    renderGame();
}
```

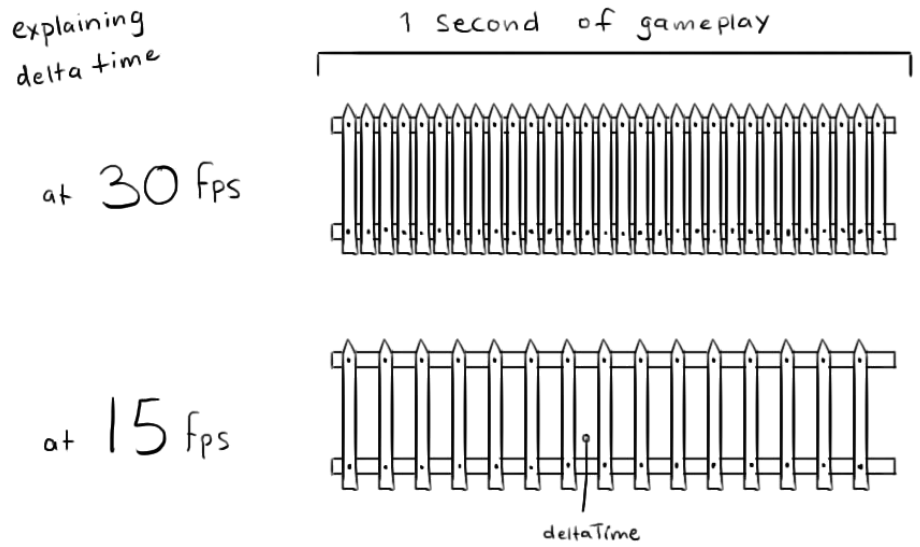
Frame rate

- Number of game loop renderings/second (Hz or FramePerSecond)
- Describes the speed at which the sequence of images is displayed

Frame time, Time delta, Delta time, Frame period…

- Amount of time between 2 successive frames (seconds)
- Amount of time to get inputs, update game state and render image
- Ex: f = 60 FPS -> dt = 16,6 ms/frame…

Source: Tim Hengeveld



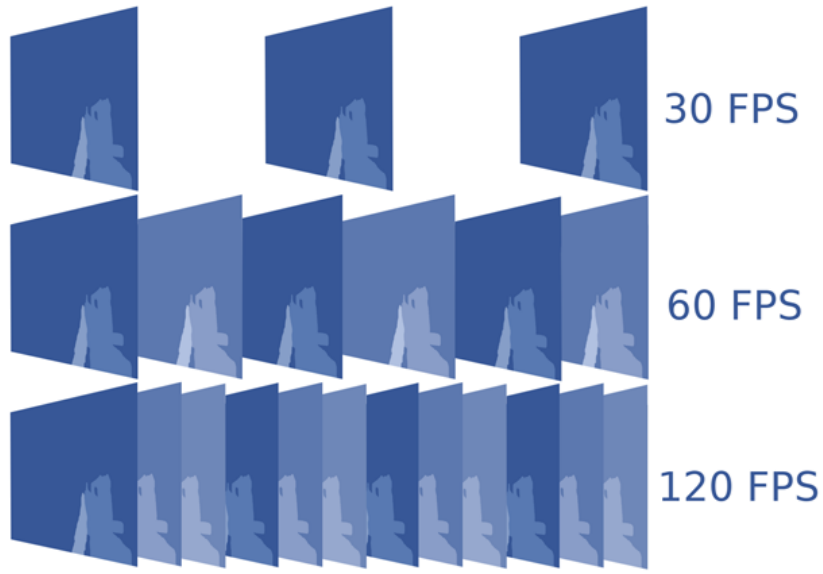
Frame rate

Depends on the complexity of calculating each frame and the power of the hardware

=> Basic game loop will run the game at inconsistent speeds depending on the hardware or the situation

Ex: move x meters per frame

=> Need to track time and adapt the loop architecture to control the rate of the game



Source: PCMag

Real Time

Amount of time elapsed in the real world

Insufficient resolution of OS function for querying the system time

Ex. `time()` in C

=> Use high-resolution timer hardware register on CPU

Origin = last power on or CPU reset

Counts the units of elapsed CPU cycles (or some multiple thereof)

Converted into seconds by multiplying by the frequency

Ex: 3 GHz CPU, incremented 3 billion times / s -> 0.333 ns resolution

~~Wrapping problem !~~

Caution with multicore CPU: 1 timer / core

Game Logic Time

- Amount of time elapsed in the game world
- What happens during 1 frame (or "tick") of the game loop
- Independent from real time and rendering time
  - Pause -> stop updating the game temporarily (!= breakpoint)
  - Slow-motion -> updating the game more slowly than the real-time clock
  - Rewind…
- Useful for debug
  - Ex: freeze the action but not the rendering and debug camera (different clock)
  - Single-stepping the game clock by 1 target frame interval (e.g., 1/30 of a second) with a button while the game is in a paused state

Use delta time in update

Most game engines

Update takes into account the amount of elapsed game time since last frame

1. Basic Game Loop

move 1 meter/frame

30 fps => 30m/s
10 fps => 10m/s

2. With delta time

move (30*dt) meter/frame

30 fps => (30*0,033)*30 = 30m/s
10 fps => (30*0,1)*10 = 30m/s

Use delta time in update

Most game engines

Update takes into account the amount of elapsed game time since last frame

```
double lastTime = getCurrentTime(); //CPU's timer
while (true){
    double current = getCurrentTime();
    double elapsed = current - lastTime; //last frame duration
    processInput();
    update(elapsed);
    render();
    lastTime = current;
}
```

Use delta time in update

Consistent rate on different hardware

Faster machines = smoother gameplay

Measured value Δt for frame k is an estimation of the duration of the next frame $(k + 1)$

=> Subject to “frame-rate spike” (sudden change of time frame)

Undeterminism

Physics will behave differently depending on the frame rate (numerical integration / rounding error)

Online multiplayer will not work properly with variable frame rates

Running average

Game loops tend to have at least some frame-to-frame coherency

=> Use an average of the frame-time on a small number of frames as an estimate of Δt

Allows the game to adapt to varying frame rate, and mitigates the effects of momentary performance peaks

- Long averaging interval => less responsive to varying frame rate + less spikes impact

GAME LOOP

GAME OBJECTS

Game Objects Updating

- Game loop updates the states of all game objects dynamically, maybe in a particular order
  - Dependencies between the objects
  - Dependencies on various engine subsystems
  - Interdependencies between those engine subsystems themselves
- Linkage to low-level engine systems: ensure that every game object has access to the services it depends on
  - Rendering, particles, audio, animation, collisions, physics…

Game Objects Updating

Game object's notion of time is discrete

Game object's state describes its configuration at one specific instant in time

- Defined as the values of all its attributes

Game object updating:

- Process of determining the state of each object at the current time $Si(t)$ given its state at a previous time $Si(t - \Delta t)$
- Once all object states have been updated, the current time t becomes the new previous time

Simplistic Approach

Iterate over a collection of active game objects

- Often stored in a singleton manager class (“GameWorld”, “GameObject Manager”…)
- A linked list or array of pointers, smart pointers, or handles

Call a custom implementations of **`Update(dt)`** on each object once per frame of the main loop to advance its state

```
while (true){
    PollJoypad();
    float dt = g_gameClock.CalculateDeltaTime();
    for (each gameObject) {
        gameObject.Update(dt); // updates all engine subsystems
    }
    g_renderingEngine.SwapBuffers();
}
```

Simplistic Approach

Each **`Update()`** function updates directly all the engine subsystems concerned by the object (rendering, animation, physics…)

```
virtual void Tank::Update(float dt){
    // Update the state of the tank itself.
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();
    // Now update low-level engine subsystems on behalf
    // of this tank. (NOT a good idea)
    m_pAnimationComponent->Update(dt);
    m_pCollisionComponent->Update(dt);
    m_pPhysicsComponent->Update(dt);
    m_pAudioComponent->Update(dt);
    m_pRenderingComponent->draw();
}
```

Batched Updates

- Low-level engine systems benefit from batched updating
  - Global calculations done once and reused for many game objects rather than being redone for each object
  - Minimal duplication of computations
    - Ex: collisions depend on multiple objects by nature
  - Reduced reallocation of resources: once per frame and reused
  - Maximal cache coherency: data arranged in a contiguous region of RAM
- Each engine subsystem is updated by the main game loop rather than each object's `Update()`
  - A game object can require a particular engine subsystem to allocate some state information
  - Ex: game object control the properties of the mesh instance, but not directly the rendering

Batched Updates: Example

Game Object's Update

```
virtual void Tank::Update(float dt){
    // Update the state of the tank itself.
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();
    // Control the properties of the various engine
    // subsystem components, but do NOT update
    // them here...
    if (justExploded) {
        m_pAnimationComponent->PlayAnimation("explode");
    }
    if (isVisible) {
        m_pCollisionComponent->Activate();
        m_pRenderingComponent->Show();
    }
    else {
        m_pCollisionComponent->Deactivate();
        m_pRenderingComponent->Hide();
    }
    // etc.
}
```

Game Loop

```
while (true){
    PollJoypad();
    float dt = g_gameClock.CalculateDeltaTime();
    for (each gameObject) {
        gameObject.Update(dt);
    }
    g_animationEngine.Update(dt);
    g_physicsEngine.Simulate(dt);
    g_collisionEngine.DetectResolveCollisions(dt);
    g_audioEngine.Update(dt);
    g_renderingEngine.RenderFrameAndSwapBuffers();
}
```

GAME LOOP

IN PRACTICE

Callback-Driven Frameworks

Game loop exists but is largely empty

=>Write callback functions to complete it

Ex: Game Loop Ogre3D

Cf. **`Ogre::Root::renderOneFrame()`** in **`OgreRoot.cpp`**

```
while (true){
    for (each frameListener)
        frameListener.frameStarted();
    renderCurrentScene();
    for (each frameListener)
        frameListener.frameEnded();
    finalizeSceneAndSwapBuffers();
}
```

Source: http://wiki.ogre3d.org/

Callback-Driven Frameworks

Derive a class from **`Ogre::FrameListener`**

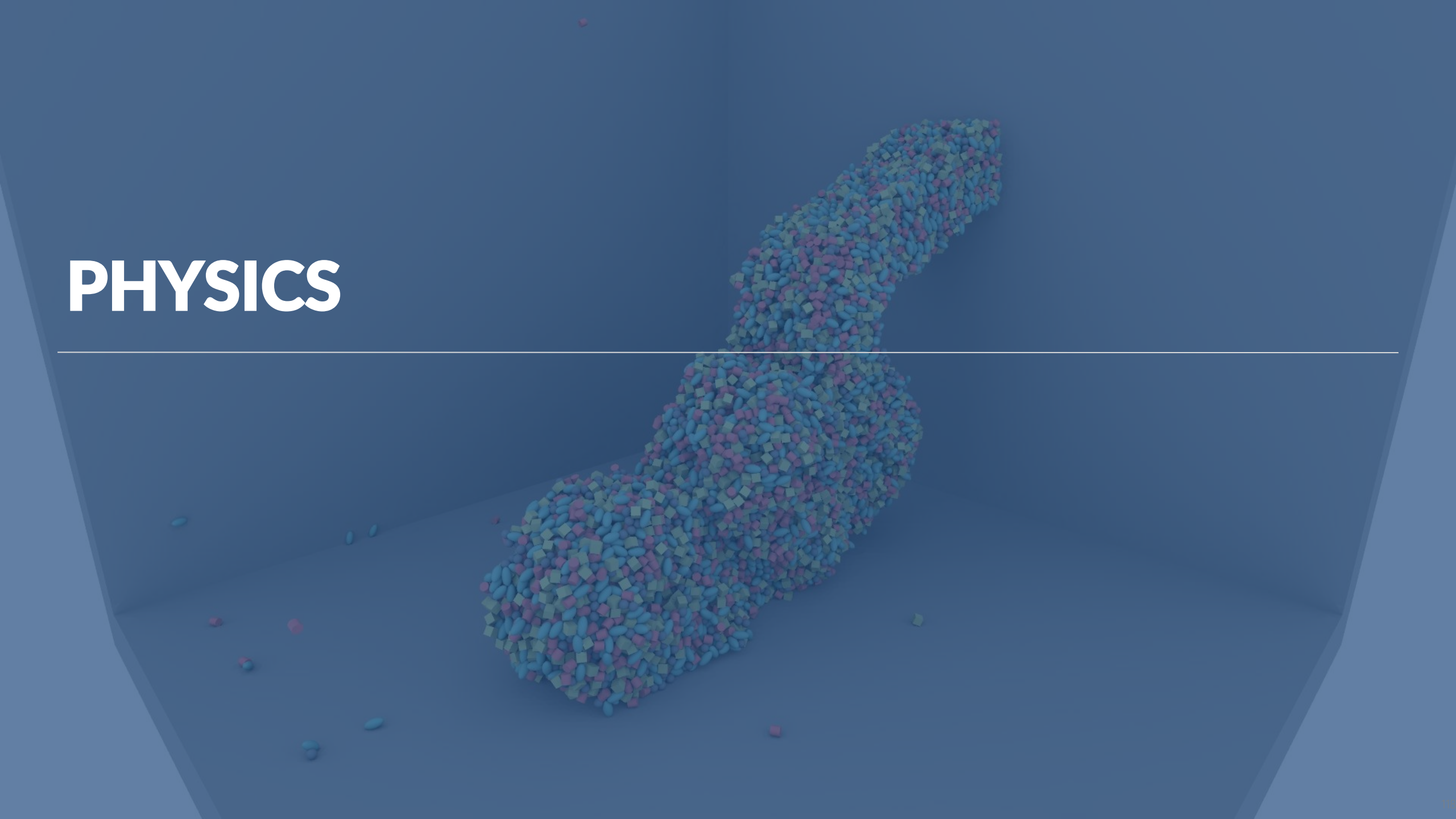
Override **`frameStarted()`** and **`frameEnded()`**

Resp. called before and after the rendering

```
class GameFrameListener : public Ogre::FrameListener {
public:
    virtual void frameStarted(const FrameEvent& event) {
        // Do things that must happen before the 3D scene is rendered
        // (i.e., service all game engine subsystems).
        pollJoypad(event);
        updatePlayerControls(event);
        updateDynamicsSimulation(event);
        resolveCollisions(event);
        updateCamera(event);
        // etc.
    }
    virtual void frameEnded(const FrameEvent& event) {
        // Do things that must happen after the 3D scene has been rendered.
        drawHud(event);
        // etc.
    }
};
```

Source:
http://wiki.ogre3d.org/

PHYSICS



Physics in a game

Detect collisions between dynamic objects and static world geometry

Rigid body dynamics

- gravity, other forces…

Ray and shape casts

- line of sight, bullet impacts…

Trigger volumes

- objects enter, leave, or inside pre-defined regions

Destructible structures

Characters picking up rigid objects

Spring-mass systems

Complex machines (cranes, moving platform puzzles…), Traps (such as an avalanche of boulders)

Vehicles

Rag doll character deaths

Hair, cloth, water surface, dangling props simulations

Audio propagation

…

Integrating and Using Physics

- Not necessarily fun
  - Chaotic behavior can disturb the experience
  - Depends on many factors (interactions, genre…)
- Unpredictability
- Difficult tuning and control
- Unexpected features
  - Ex: rocket-launcher jump trick in FPS
- Additional work for engineers and artists

Collision + Rigid Body Dynamics

- The physics system drives the collision system
  - Dynamic rigid body associated with a collidable object
- Collision library
  - Geometric (simple) shapes intersection tester
  - Casts of ray, shapes, phantoms
  - Layers
- Rigid Body Dynamics
  - Simulate the motions of game objects over time
  - Classical (newtonian) mechanics
  - Solid and undeformable objects
  - Ensure conformity to constraints: ex. non-penetration (collision response), joints…

Rigid Body Dynamics

Equations of motion for linear dynamics

$\boldsymbol{v}(t) = {}^{dp(t)}/_{dt}$ $\boldsymbol{a}(t) = {}^{dv(t)}/_{dt}$ $\boldsymbol{F}(t) = {}^{d(m\boldsymbol{v}(t))}/_{dt} = m\boldsymbol{a}(t)$

Solving $v(t)$ and $p(t)$ given force $F(t)$ and previous pos. and velocity

Analytical solutions almost impossible in games

Numerical integration not exact but stable

Time-stepped: finding p, v et F for $t2 = F(t1)$

Explicit euler

$p(t2) = p(t1) + p(t1).\Delta t$

$\boldsymbol{v}(t2) = \boldsymbol{v}(t1) + {}^{\boldsymbol{F}(t1)}/_{m}.\Delta t = {}^{p(t2)-p(t1)}/_{\Delta t}$

Unity

- Nvidia PhysX
- 2D, 3D
- Components
  - Collider: shape, center, scale…
  - Rigidbody: gravity, kinematics, static…
- Events/Callbacks
  - OnCollisionEnter()…
  - OnTriggerEnter()…
- Physics class
  - Raycast, spherecast, forces, velocity…
- Physics manager
  - Collision layers…