INTRODUCTION TO GAME PROGRAMMING & GAME ENGINES

Guillaume Bouyer, Adrien Allard

v4.2

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 a video game (among others to succeed as well as possible in the team project)

Understand the theoretical and technical components of game engines

Operate a high-level but relatively closed game engine (Unity). Being able to create 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			SHS	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

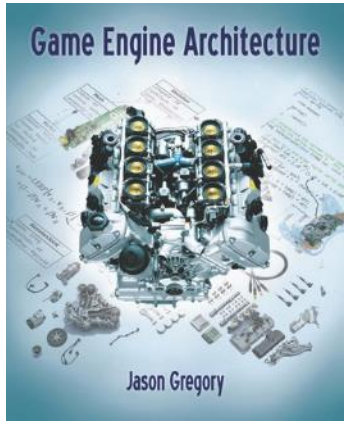
References

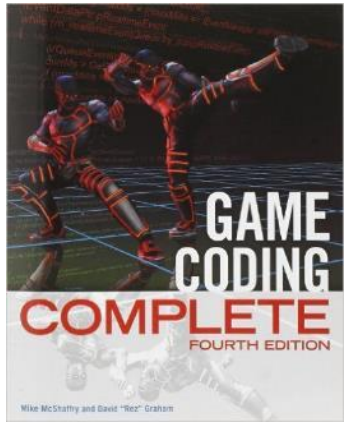
Game Engine Architecture, Jason Gregory, A K Peters/CRC Press, 2009-2015-2019 (http://www.gameenginebook.com/)

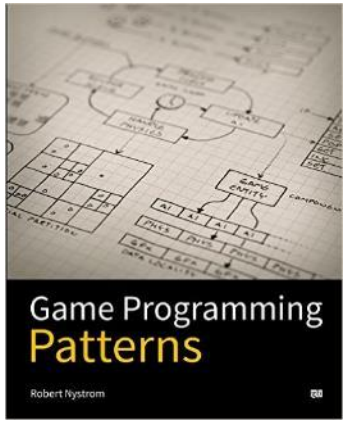
Game Coding Complete, 4th Edition, Mike McShaffry and David Graham, Course Technology, 2013

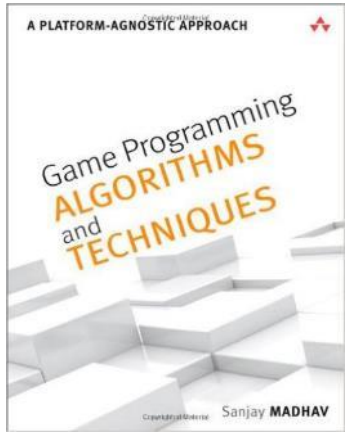
Game Programming Algorithms and Techniques, Sanjay Madhav, Addison-Wesley, 2013

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







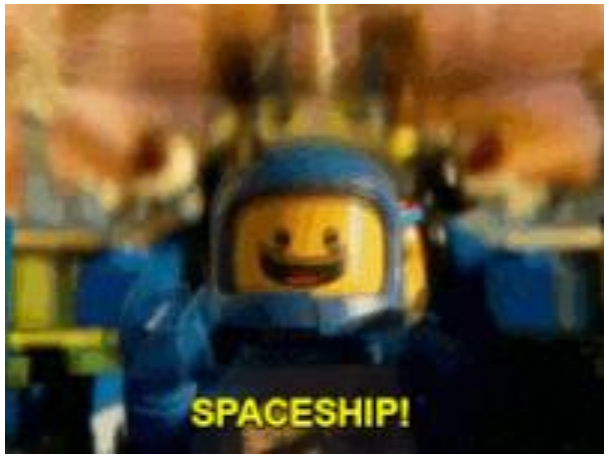


Contents

What if we programmed our own video game?

JIN PROJECT

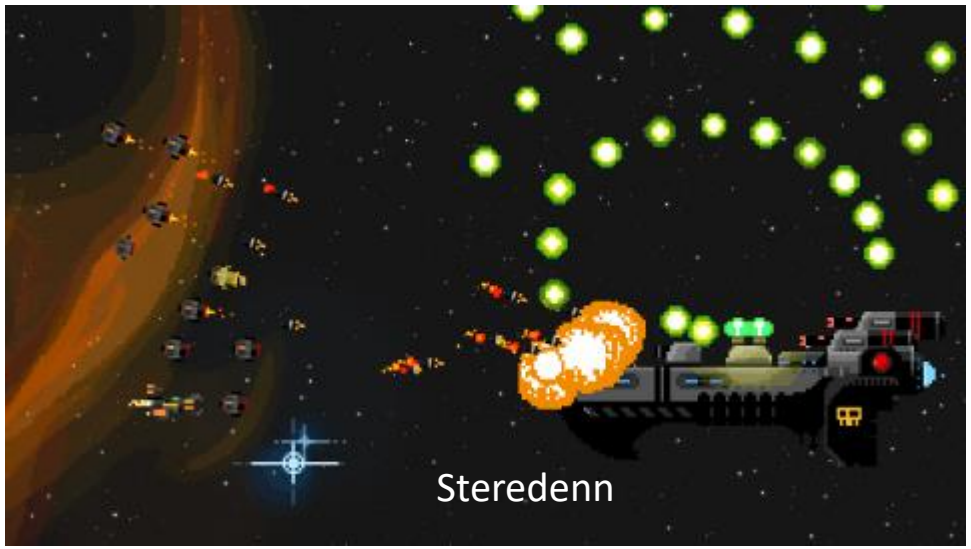
Our game?



SCHMUP !

Spaceships !

Bullets !



PART 1:

THE BASICS

A VIDEO GAME?

What is a video game?

Player's point of view:

"An interactive experience that provides the player with an increasingly challenging sequence of patterns which he learns and eventually masters"

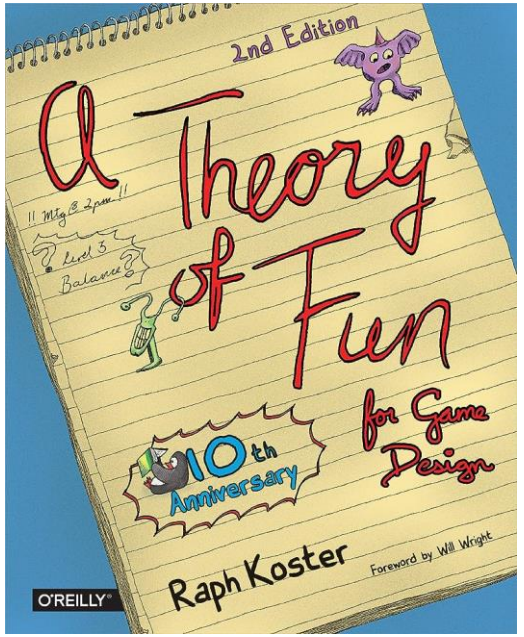
Raph Koster, A Theory of Fun for Game Design

Artistic & interactive content (assets, objects, world…)

Gameplay, game mechanics

- Player's abilities
- Non-player entities
- Rules of the world and interactions
- Objectives, criteria for success/failure
- Overall flow
- …

=> More significant than technology to define a game



What is a video game?

Developer's point of view:

"A soft real-time interactive agent-based computer simulation"

Jason Gregory, Game Engine Architecture

Agents: distinct entities or objects in the game world

Real-time Simulation

- Update objects to create a dynamic game world
- Approximated numerical mathematical models
- Various technical sub-systems (3D, AI, game logic, physics...)

Interactive

- Responds to unpredictable human input
- Provides visual and audio rendering of the simulation result

Our game

SCHMUP !

Artistic content

Interactive content

Gameplay, rules, abilities…

Objects

Real-time simulation

Human input

Graphics rendering, audio…

…

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)

More job descriptions

SNJV

Gaming Campus

Our team

SCHMUP !

Producer : me

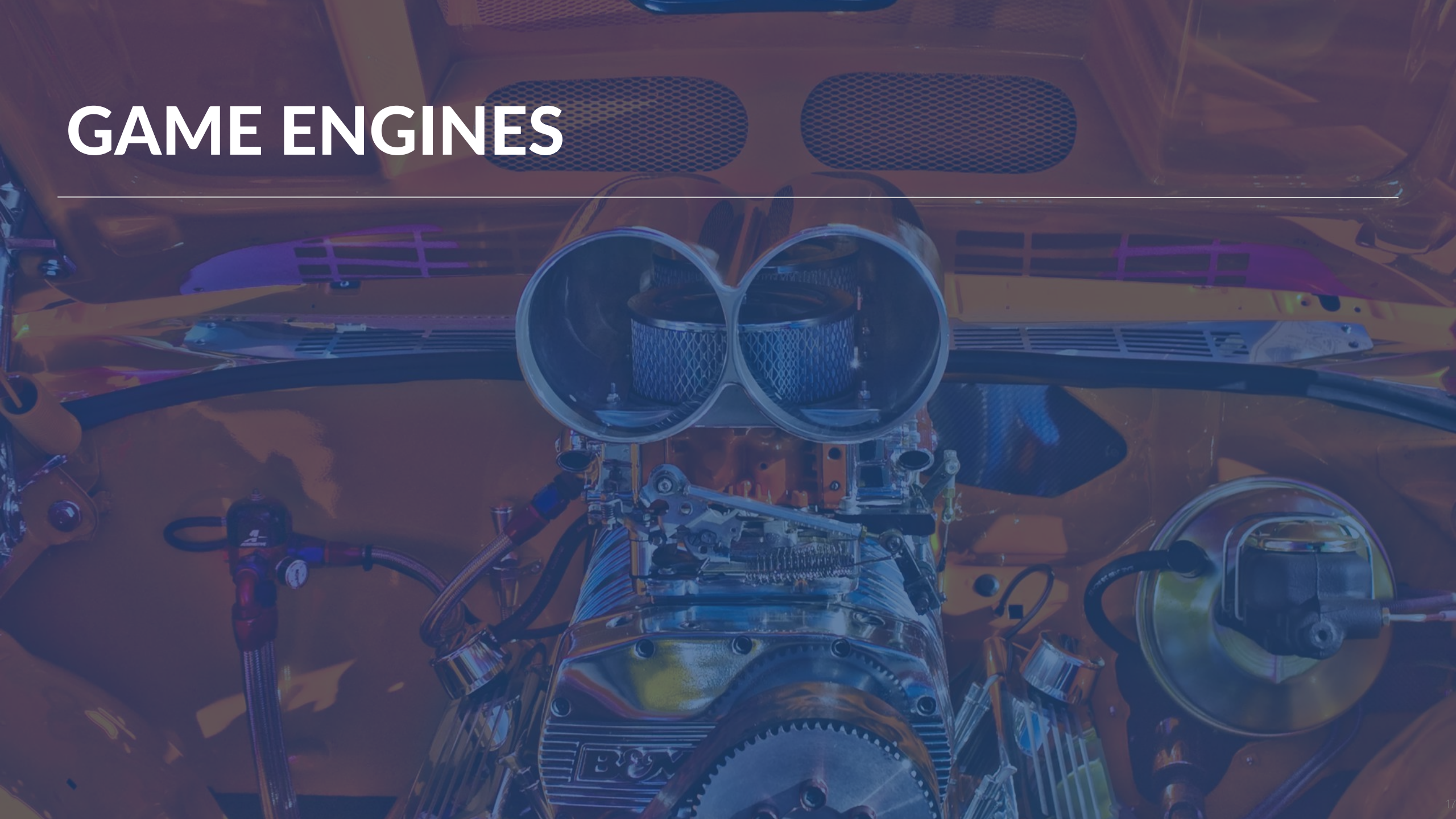
Game designer : contractor

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…

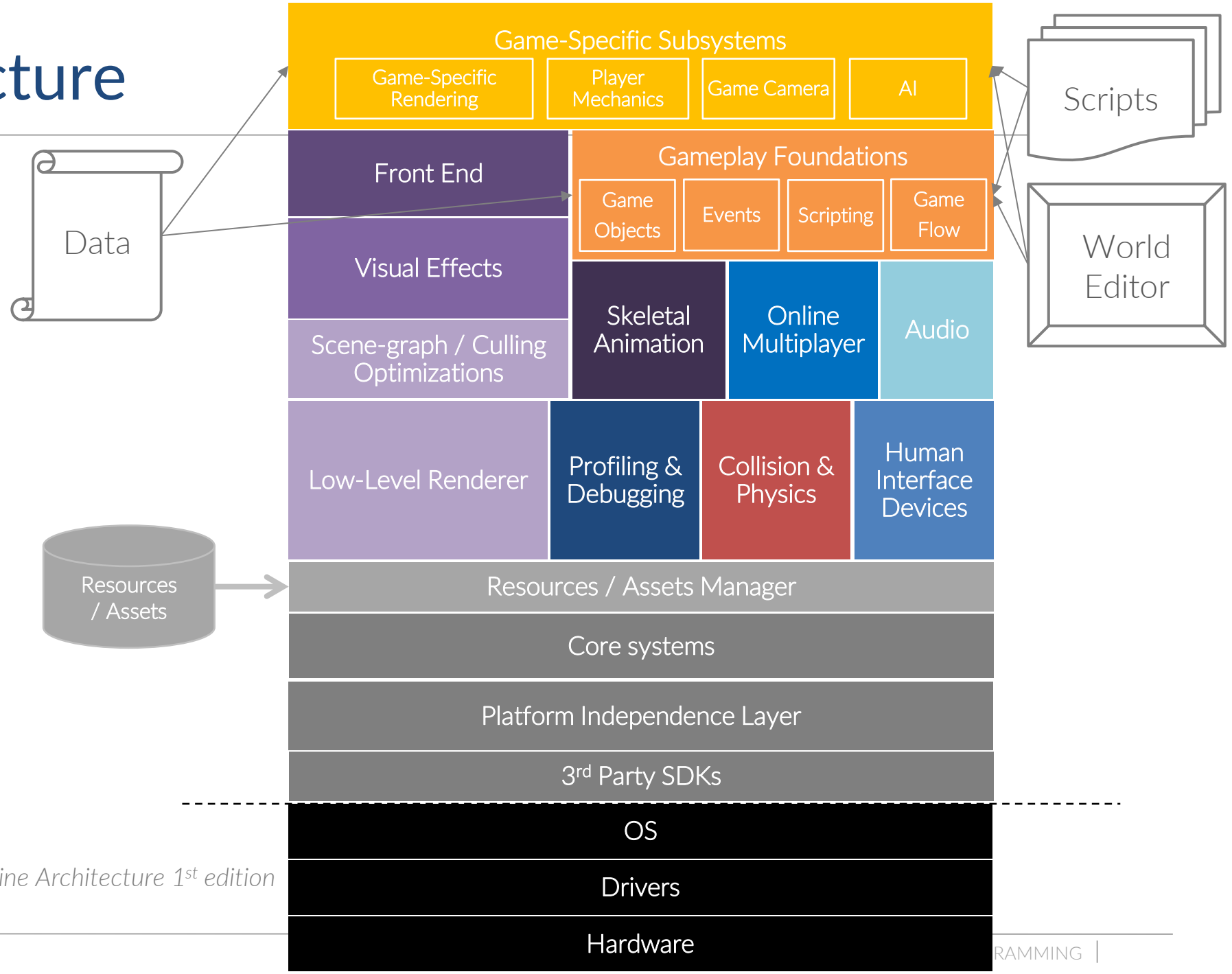
Open Source Engines

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

Proprietary in-House Engines

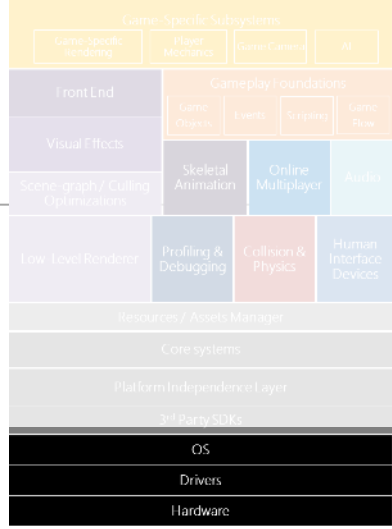
More at https://en.wikipedia.org/wiki/List_of_game_engines

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)

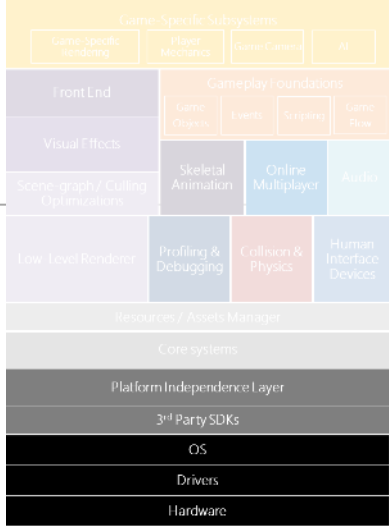
Runs all the time

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



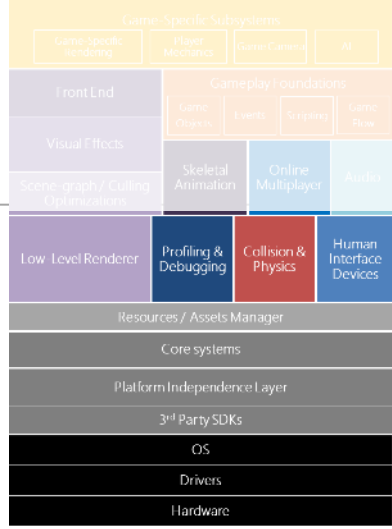
Third-Party SDKs and Middleware

- 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

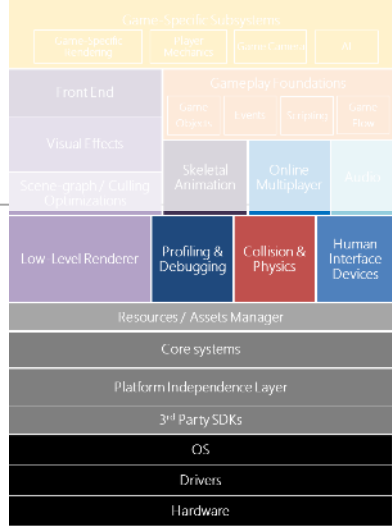
Resources/Assets Manager

- 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

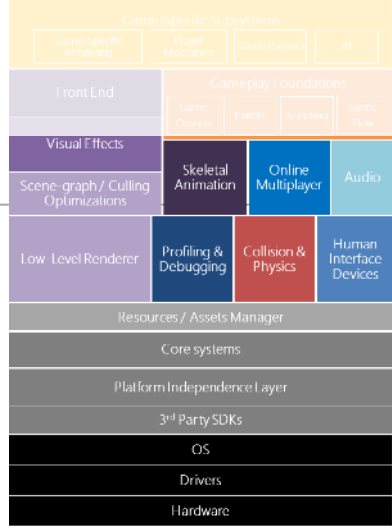
- Low-Level Renderer
- Scene Graph/Culling Optimizations
- Visual Effects
  - Particles, decal, light and environment mapping, dynamic shadows, full-screen post effects (HDR, AA, color correction…)
- Front End
  - 2D or 3D: Heads-up display (HUD), in-game menus, console, development tools, in-game GUI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

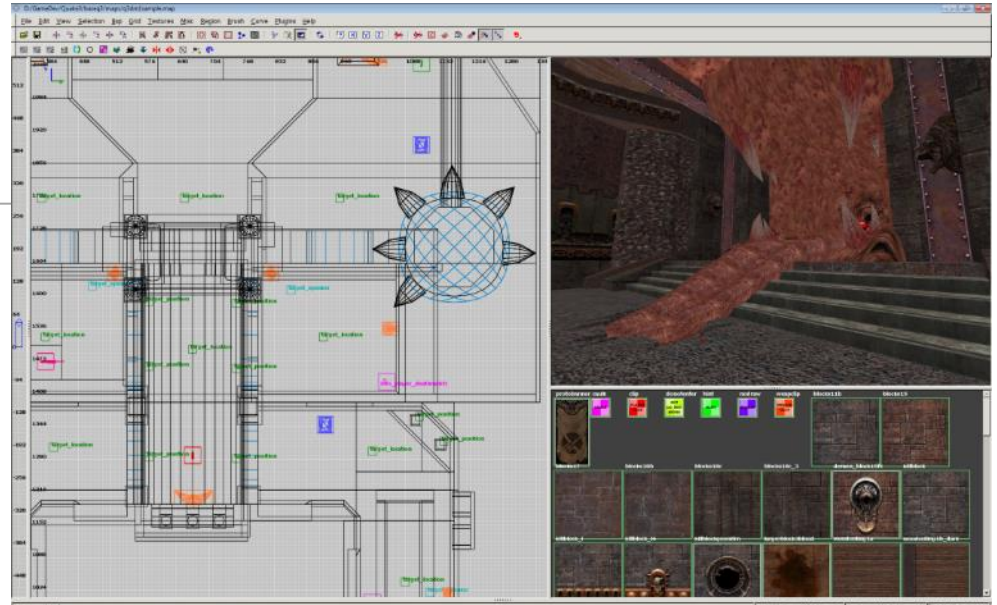
Assets Management

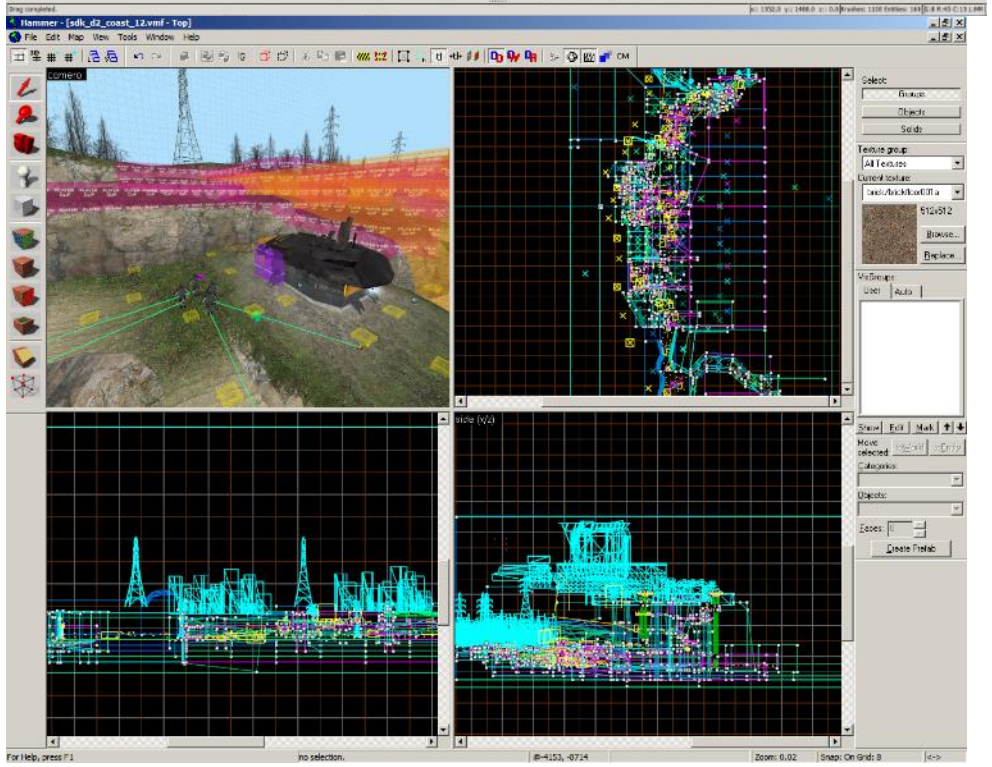
- Data formats of assets rarely ready for direct in-game use
  - In-memory model too complex
  - File format too slow to read at runtime, or proprietary
- Asset Conditioning Pipeline (ACP)
  - Data exported to a more accessible standardized or custom format, then further processed (ex. differently for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

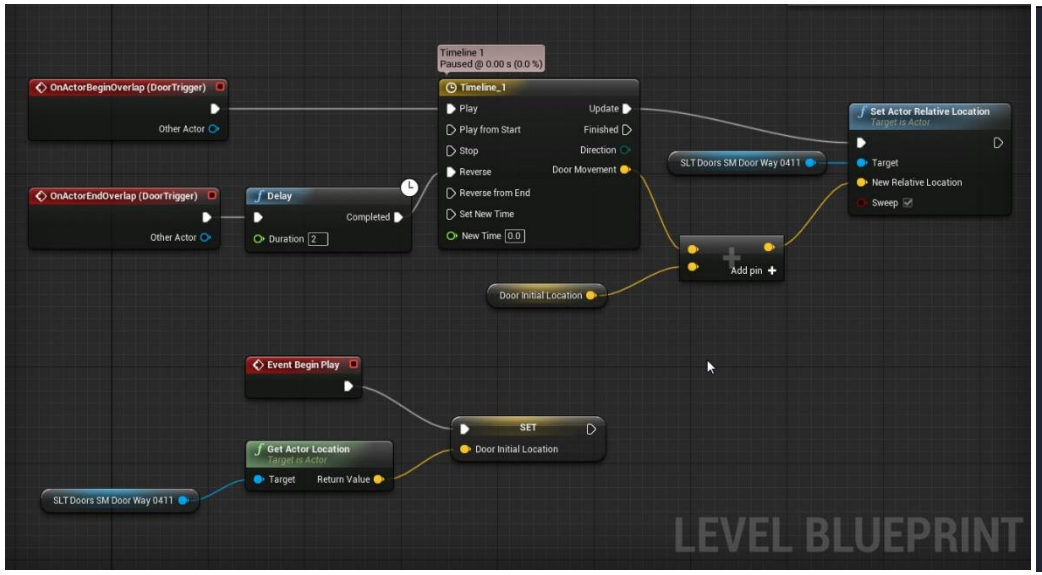
Benefits

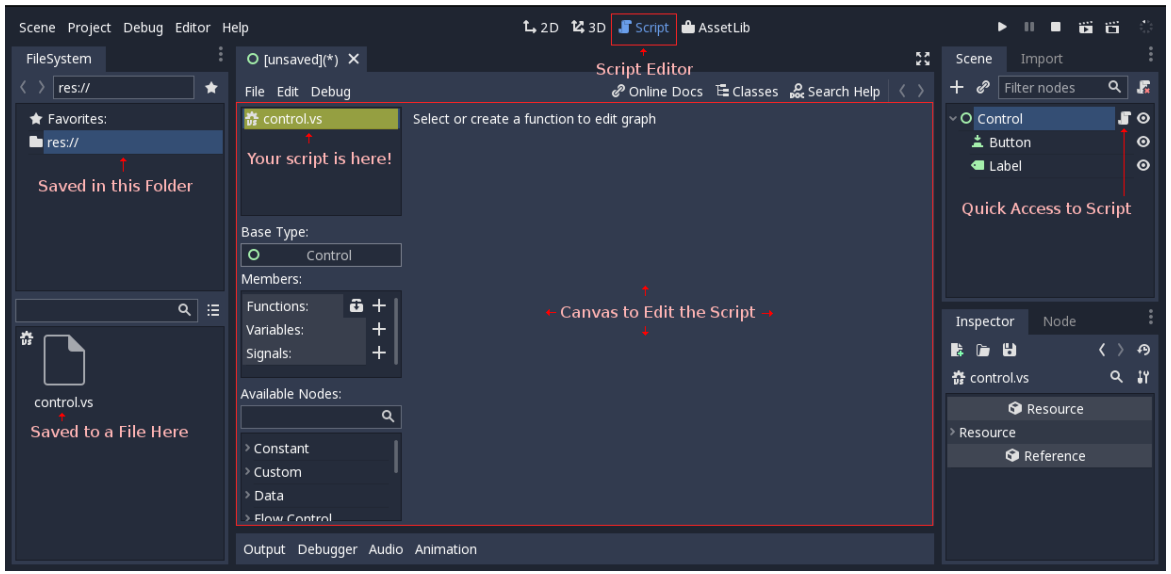
- Faster iteration than native language source code (sometimes no recompilation/relink, no game shut down and rerun)
- Customizable to suits the needs of a particular game
- Can make common tasks simple and less error-prone

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

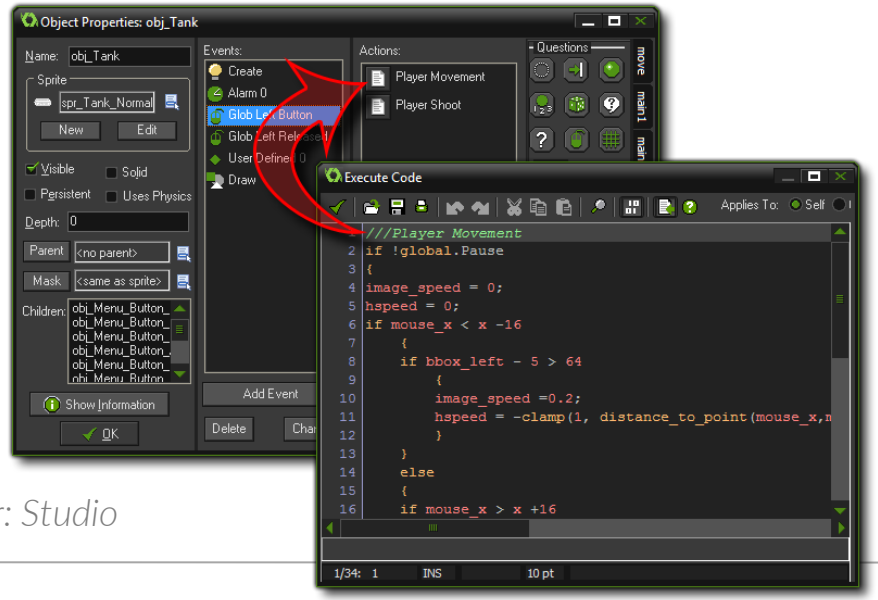
Visual Scripting Editors

Unreal Engine Blueprint

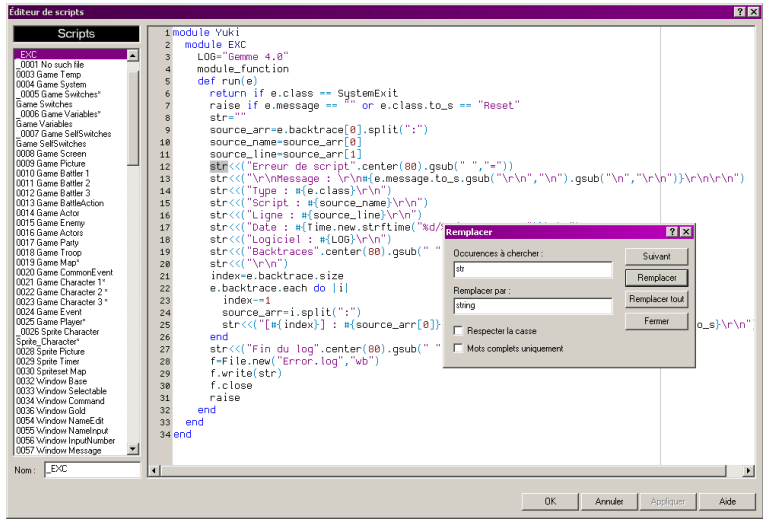




Godot Visual Scripting



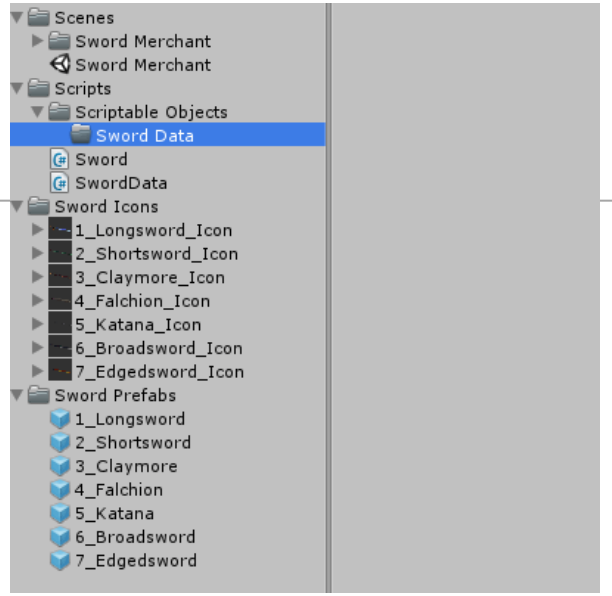
GameMaker: Studio



RPG Maker

Data-Driven

Source: raywenderlich.com



Data-driven engine permits designers and artists to

- Create content
- Control some parts of the behavior of the game
- Directly by data rather than programming

Benefits and risks

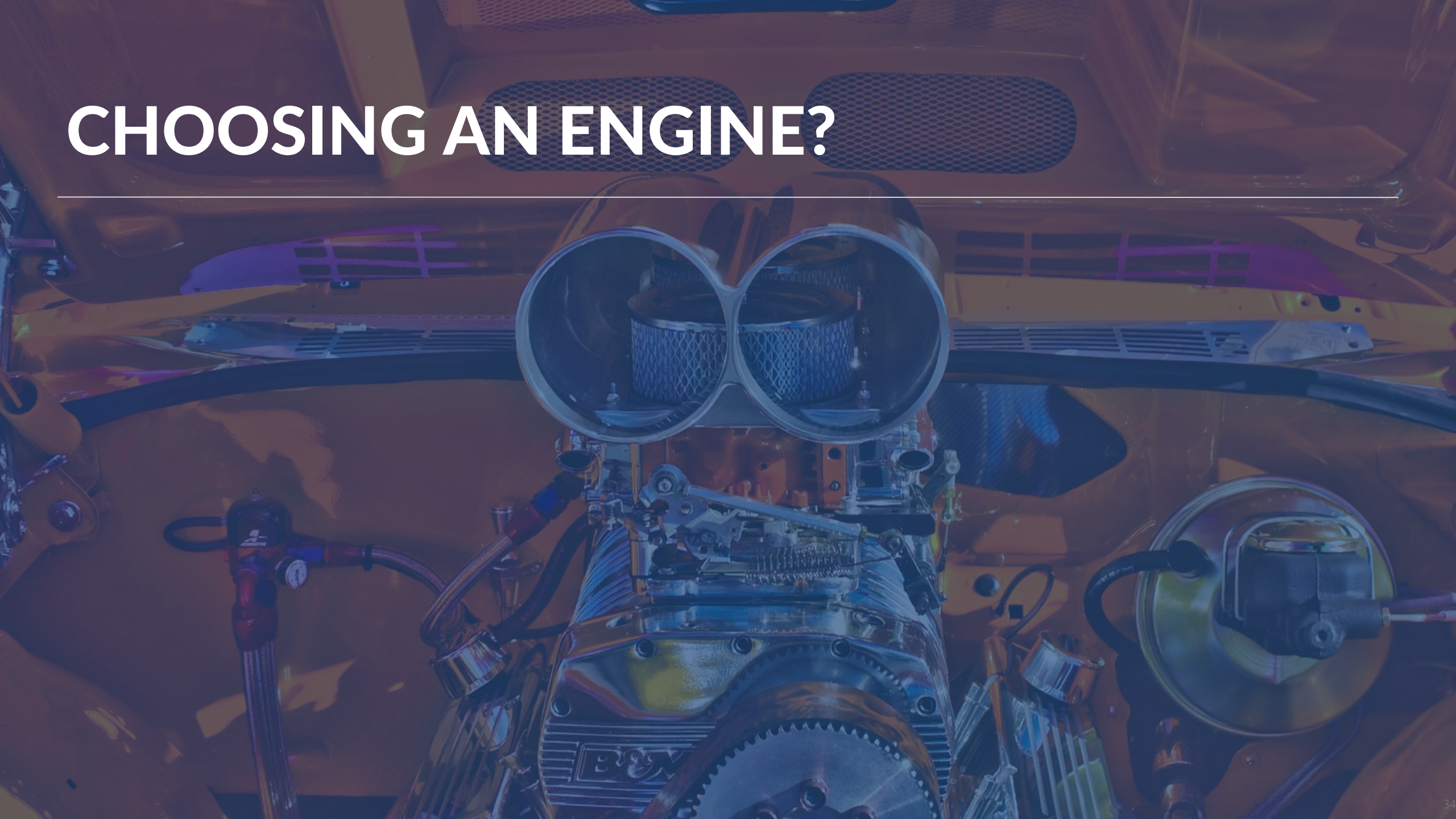
- Improved creation and iteration times
- Heavy cost to develop appropriate runtime code and robust and usable tools

ActorBlueprints.xml

```
<ActorBlueprint name="Orc Pawn" controlType="AI" description="The lowest ranking Orc solider" >
    <ActorVisibility sightRadius="6.2" />
    <SpriteData textureImage="Art/Actors/TestOrc.png" width="1.12" height="1.12" deadTexture="Art/Actions/DeadOne.png" />
    <Physics>
        <CollisionData collisionType="AI" >
            <AABB width="0.90" height="0.90" />
        </CollisionData>
        <Movement velocity="1.20" >
            <Marble additionalGCost="-0.45" movementSpeedCoefficient="1.05" />
            <Dirt additionalGCost="-0.10" movementSpeedCoefficient="1.02" />
            <Grass additionalGCost="-0.15" movementSpeedCoefficient="1.15" />
            <WoodBridge additionalGCost="-0.04" movementSpeedCoefficient="0.95" />
            <Wood additionalGCost="-0.04" movementSpeedCoefficient="0.95" />
            <Sand additionalGCost="0.35" movementSpeedCoefficient="0.90" />
            <Desert additionalGCost="2.0" movementSpeedCoefficient="0.75" />
            <Door additionalGCost="20.0" movementSpeedCoefficient="0.50" />
            <Water additionalGCost="500.0" movementSpeedCoefficient="0.25" />
        </Movement>
    </Physics>
    <Combat meleeWeightPreference="1.0" rangeWeightPreference="0.0" >
        <Melee baseDamage="2.5" baseAttackTimeCooldownSeconds="2.1" baseRange="1.0" />
    </Combat>
    <AIProfile looksForAmmoWhenFleeing="false" patrolsWhenIdle="false" >
        <HateLevel startValue="50" >
            <OnDamageReceived damageAmountMultiplier="2.50" />
        </HateLevel>
        <FearLevel startValue="0" >
        </FearLevel>
        <IndifferenceLevel startValue="0" >
        </IndifferenceLevel>
    </AIProfile>
    <Attributes>
        <Health maxHitPoints="45" startHitPoints="28" healthRegen="0.08" />
        <Mana maxManaPoints="10" startManaPoints="2" manaRegen="0.02" />
        <Strength baseStrengthValue="10" />
        <Agility baseAgilityValue="3" />
        <Intelligence baseIntelligenceValue="0.2" />
        <Armor baseArmorValue="4.5" />
    </Attributes>
    <Items dropsItemsOnDeath="false" >
        <Item name="MinorManaPotion" count="1" />
        <Item name="StoutShield" />
        <Item name="BroadSword" />
    </Items>
    <LevelData canGainExperience="false" experienceValueWhenKilled="90.0" nextLevelExperience="0" eachLevelIncreasesByPercent="0.00" >
    </LevelData>
</ActorBlueprint>
```

Source: paulallenrenton.com

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

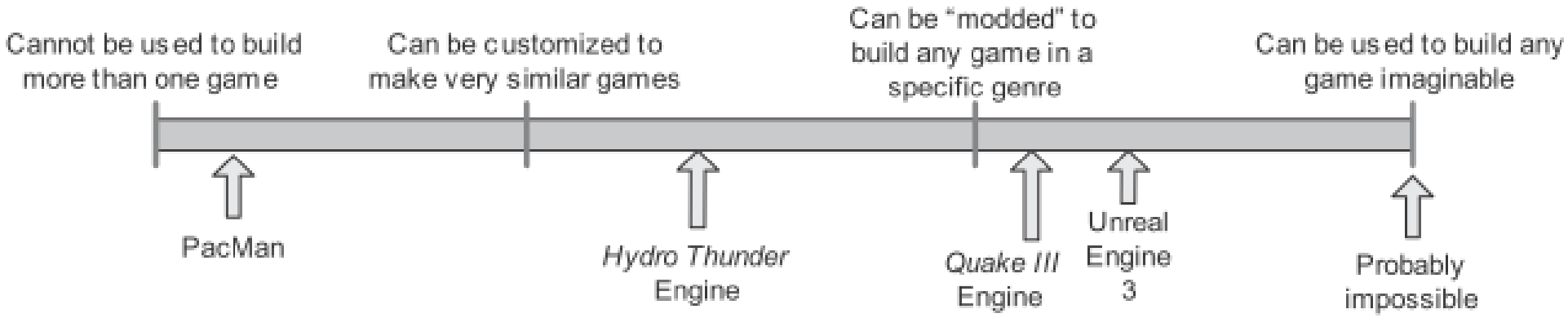
Initial game engines designed and tuned to run a particular game on a particular hardware platform

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

Technical differences by genre

Real-Time Strategy (RTS)

Rendering low-res but large number of units, height-field terrain

3C oblique top-down camera, can optimize rendering, grid-layout system for units and buildings, complex user interaction (keyboard/mouse, menus, equipment, unit types, building types…)

AI non-player characters

Multiplayer typically 2-4 players local or online, ranking

Starcraft 2

Massively Multiplayer Online Games (MMOG)

Rendering lower fidelity than offline counterparts

Gameworld huge and varied (zones)

Network data centers to maintain the state of the game world, manage users login, provide chat or VoIP services, central server to handle the billing and micro-transactions

World of Warcraft

Our game

SCHMUP !

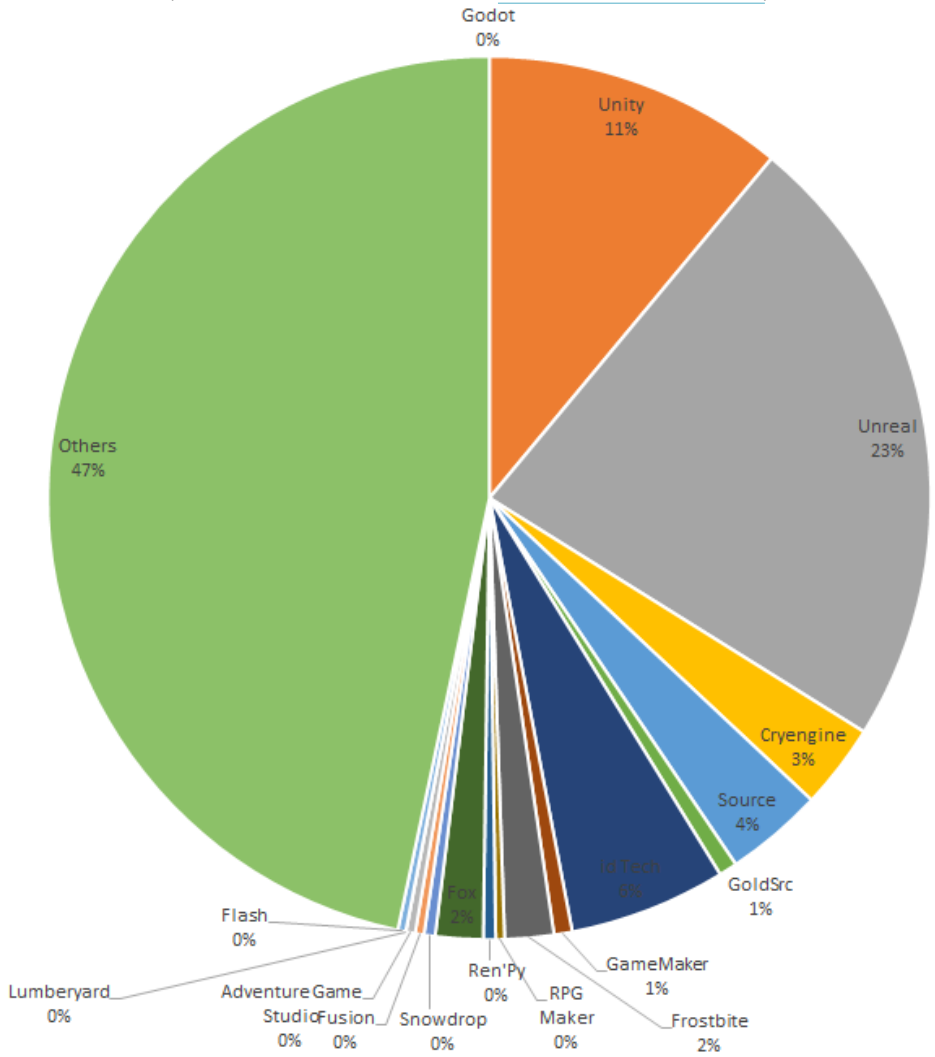
Read the "game design document" (see end of presentation)

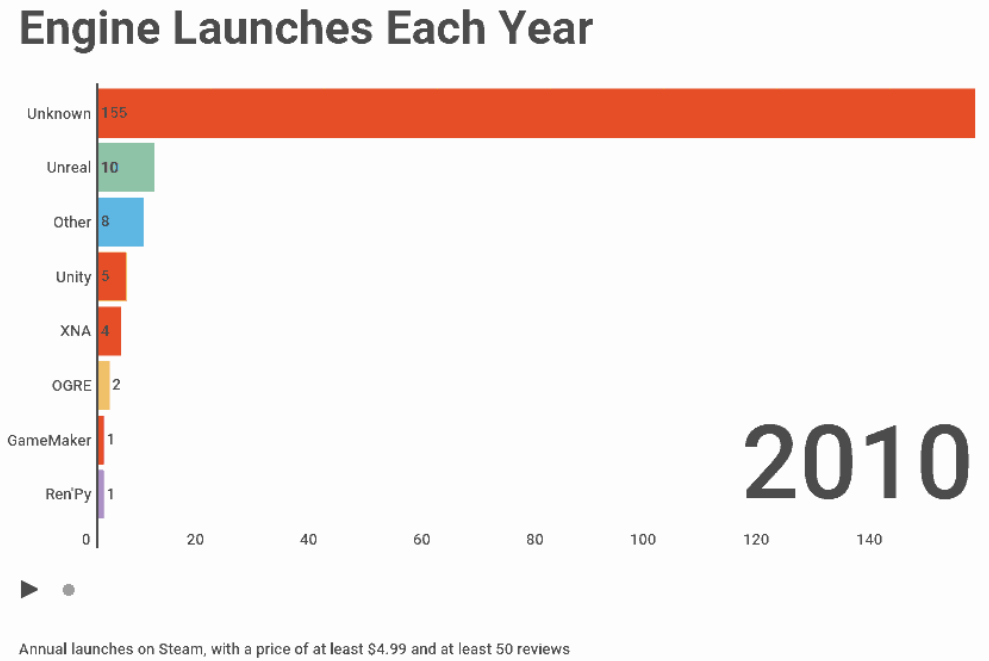
Rendering	2D, relatively low res, sprites
3C	fixed camera, very responsive user input system, accurate hit detection
Animations	simple animated sprites
AI	no, scripted/randomized levels
Multiplayer	no
Gameworld	relatively static backgrounds, numerous visible objects (enemies, bullets, particles…)

Classic Dominant Choice

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/
57 Programmer jobs (04/09/24)
Unity : 13+3 = 28%
Unreal : 6+9 = 26%
C++: 4+10 = 25%

Our game

Unity

Editor / Runtime

Rapid iteration

3D views

Scenes

Game Objects, Object-oriented

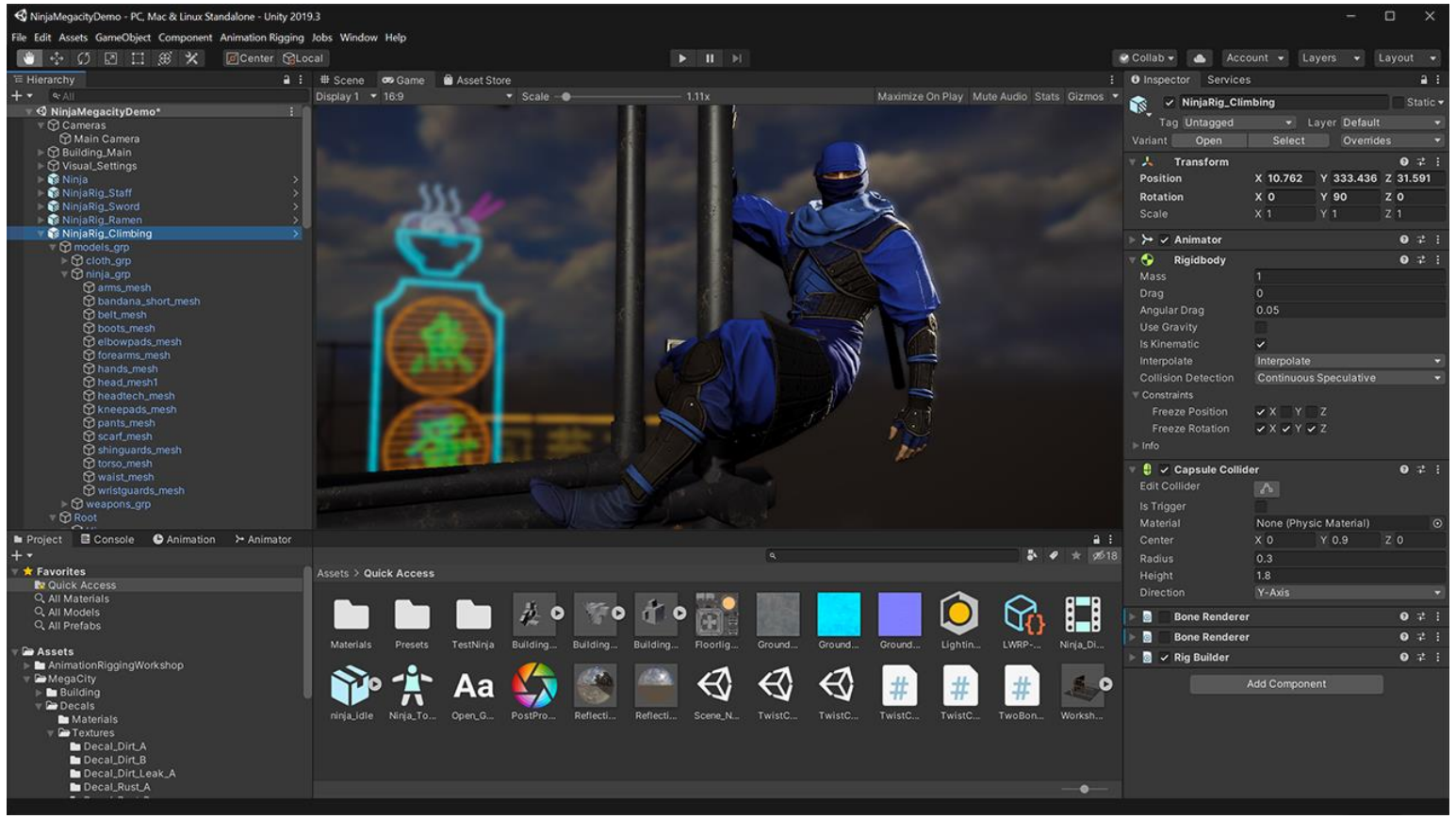
Hierarchy

Inspector

Assets, Management tools

Layers

Scripting system



PART 2

AGENT-BASED INTERACTIVE REAL-TIME SIMULATION

AGENTS / GAME OBJECTS

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…

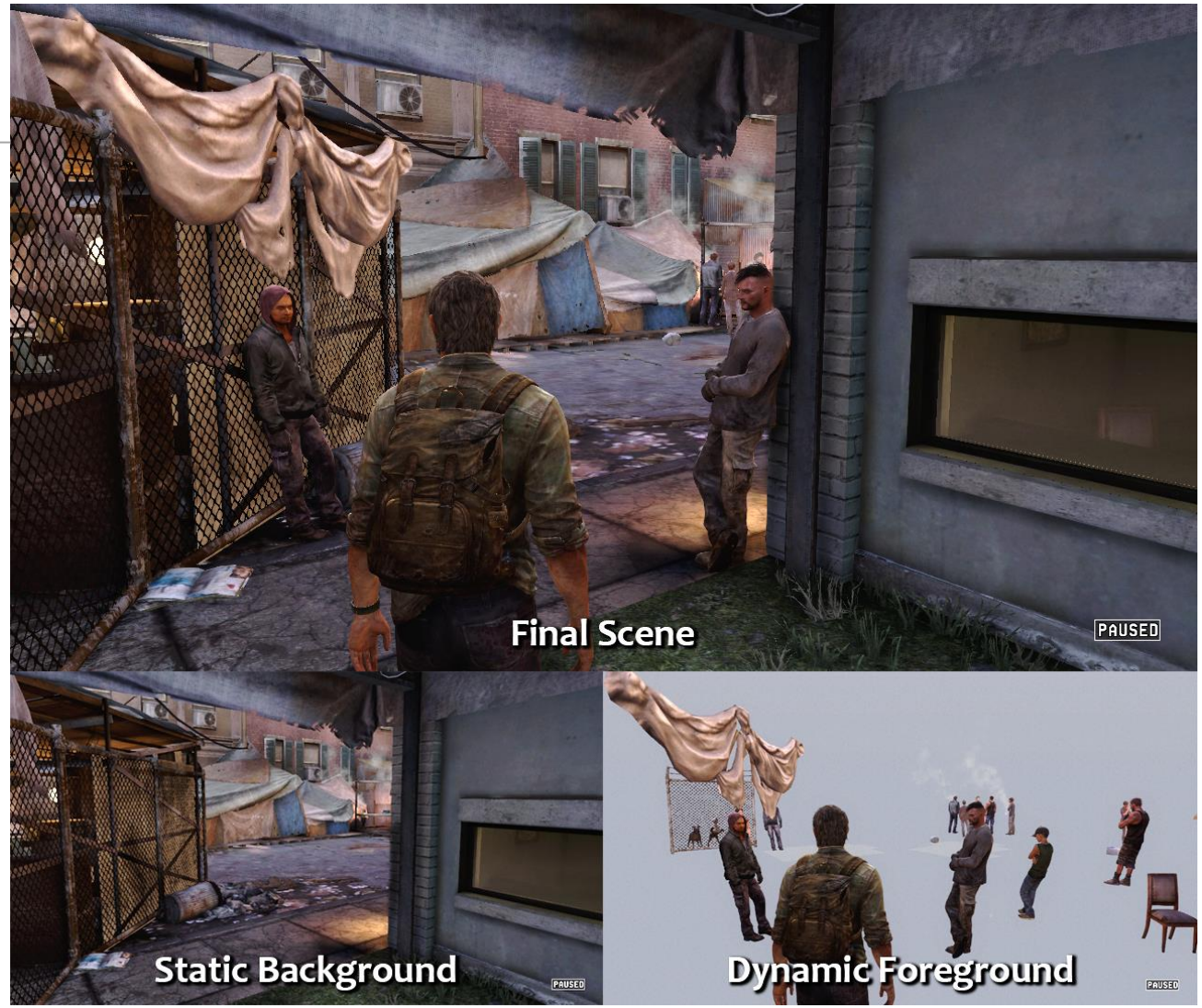
3D objects, data containers, spatial zones, invisible or special objects…

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…

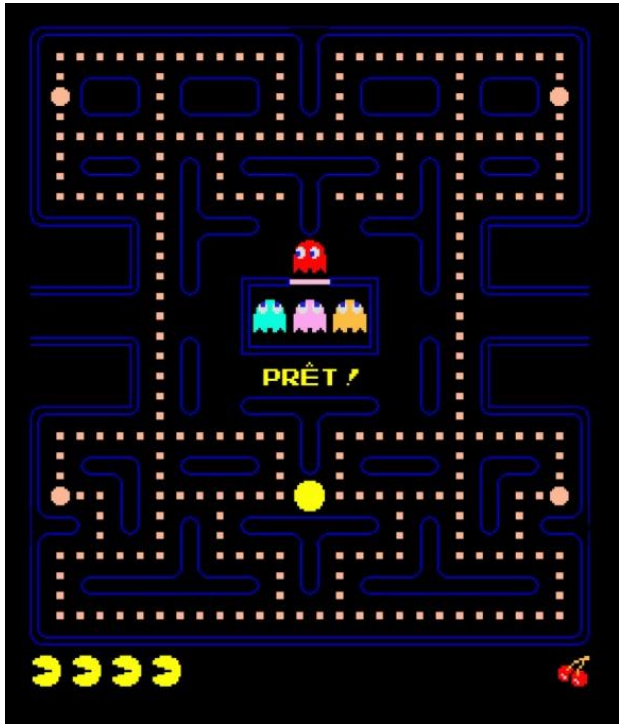
Behavior

- 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 Objects?

SCHMUP !

Read the "game design document" (end of presentation)

Assets?

SCHMUP !

"Runtime" vs. "Tool-side" Game Objects

Model in the world editor != concrete implementation

Must be flexible to easily define new game object types (data-driven or programmed)

A single tool-side game object type can be implemented as

- A single instance of a class
- A collection of interconnected instances of classes
- A collection of loosely coupled objects
- Or even a unique id, with state data stored in tables

(Not necessarily object-oriented language)

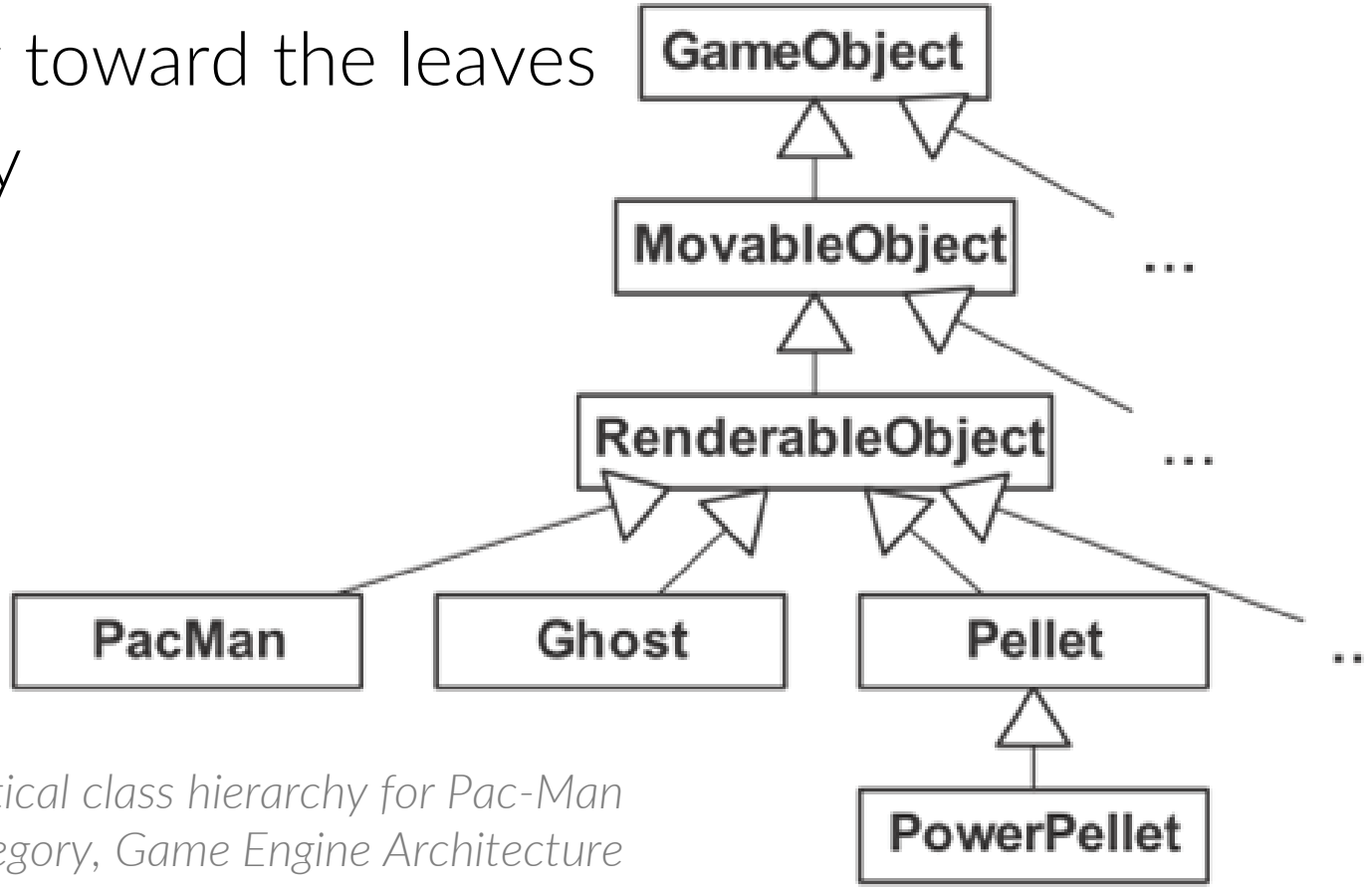
Class Hierarchies

Provides a taxonomy of game objects

Common, generic functionality at the root

Increasingly specific functionality toward the leaves

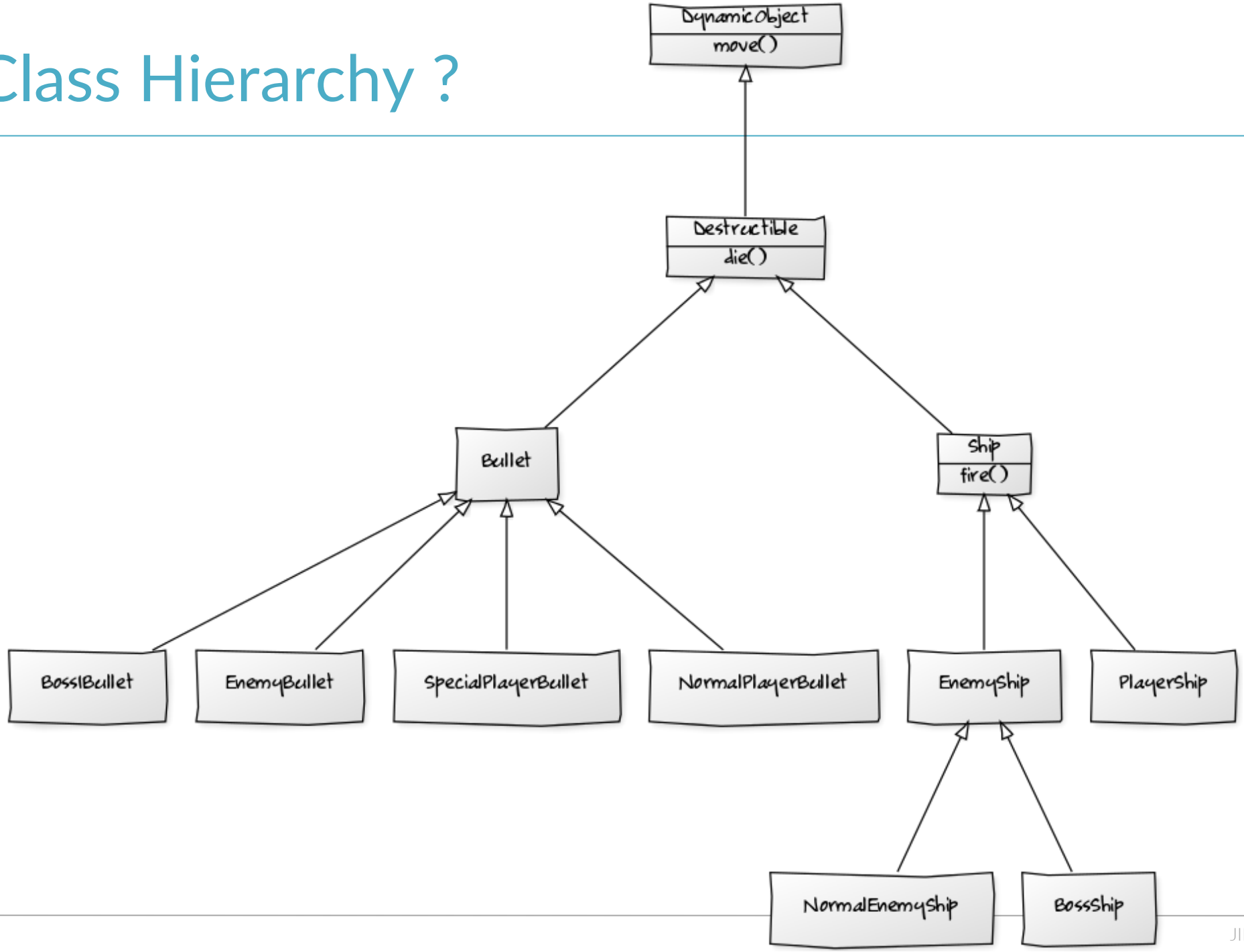
Tendency to monolithic hierarchy



A hypothetical class hierarchy for Pac-Man
Source: J. Gregory, Game Engine Architecture

Class Hierarchy ?

SCHMUP !



Problems with Deep Hierarchies

Understanding, maintaining, and modifying classes

- Need to understand all parents

Inability to describe multidimensional taxonomies

- A single axis/criteria at each level, “hack” the hierarchy to add unanticipated objects

Multiple inheritance: the deadly diamond

- Most game studios prohibit/limit the use of multiple inheritance in their hierarchies
- => Mix-in classes

The bubble-up effect

- Factorization vs. Duplication of code

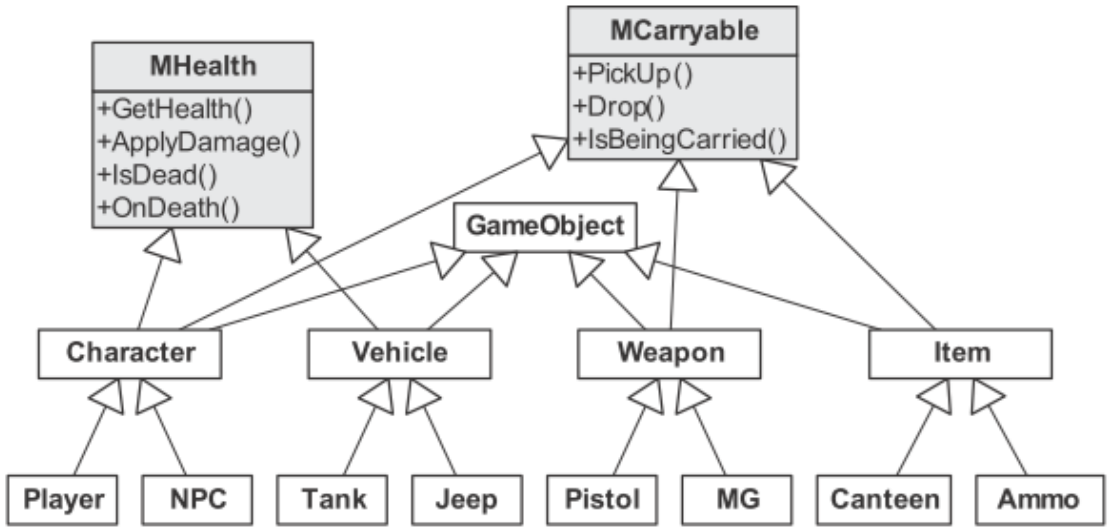


Figure 14.6. A class hierarchy with mix-in classes. The MHealth mix-in class adds the notion of health and the ability to be killed to any class that inherits it. The MCarryable mix-in class allows an object that inherits it to be carried by a Character.

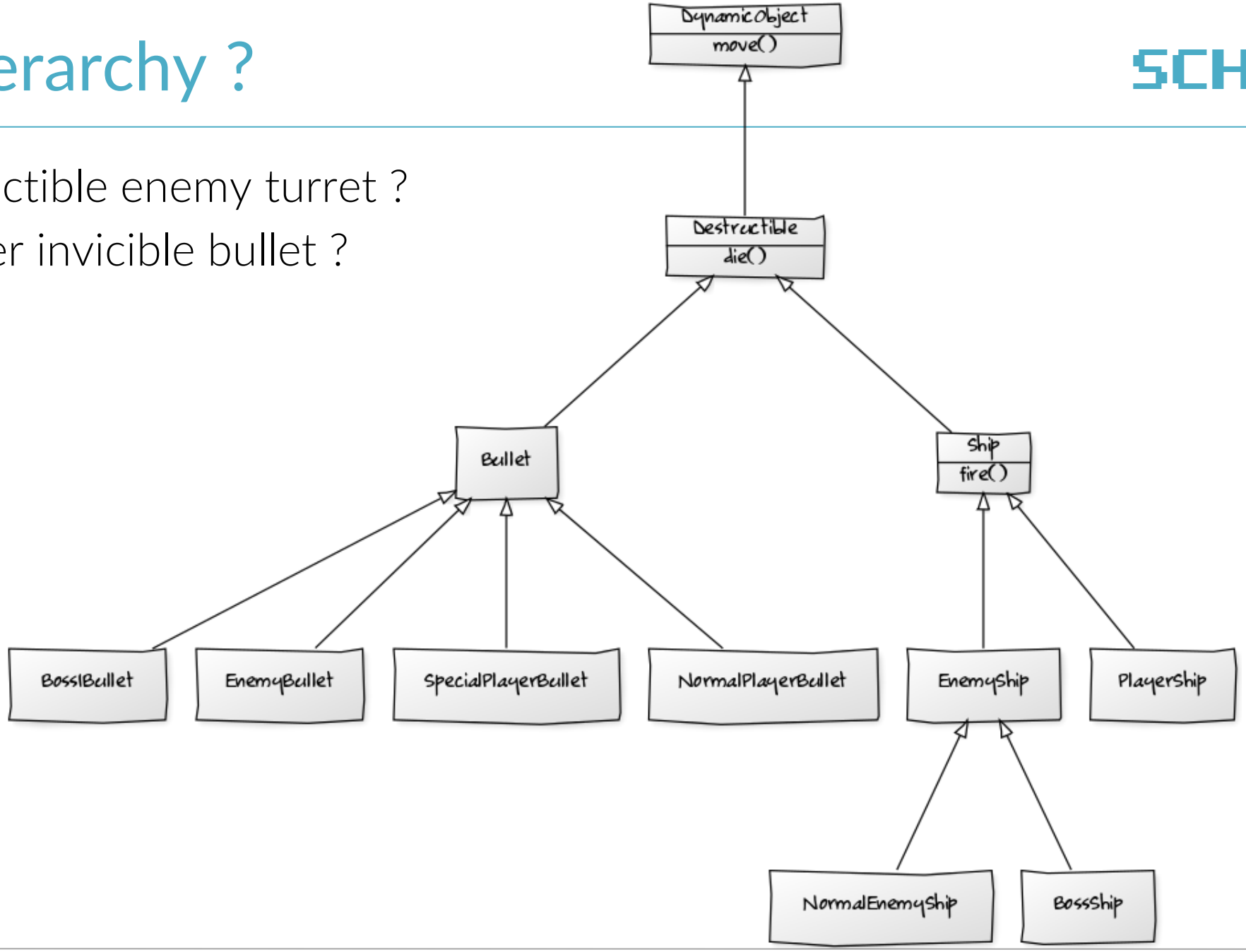
Source: J. Gregory, Game Engine Architecture

Class Hierarchy ?

SCHMUP !

Static destructible enemy turret ?

Special player invicible bullet ?



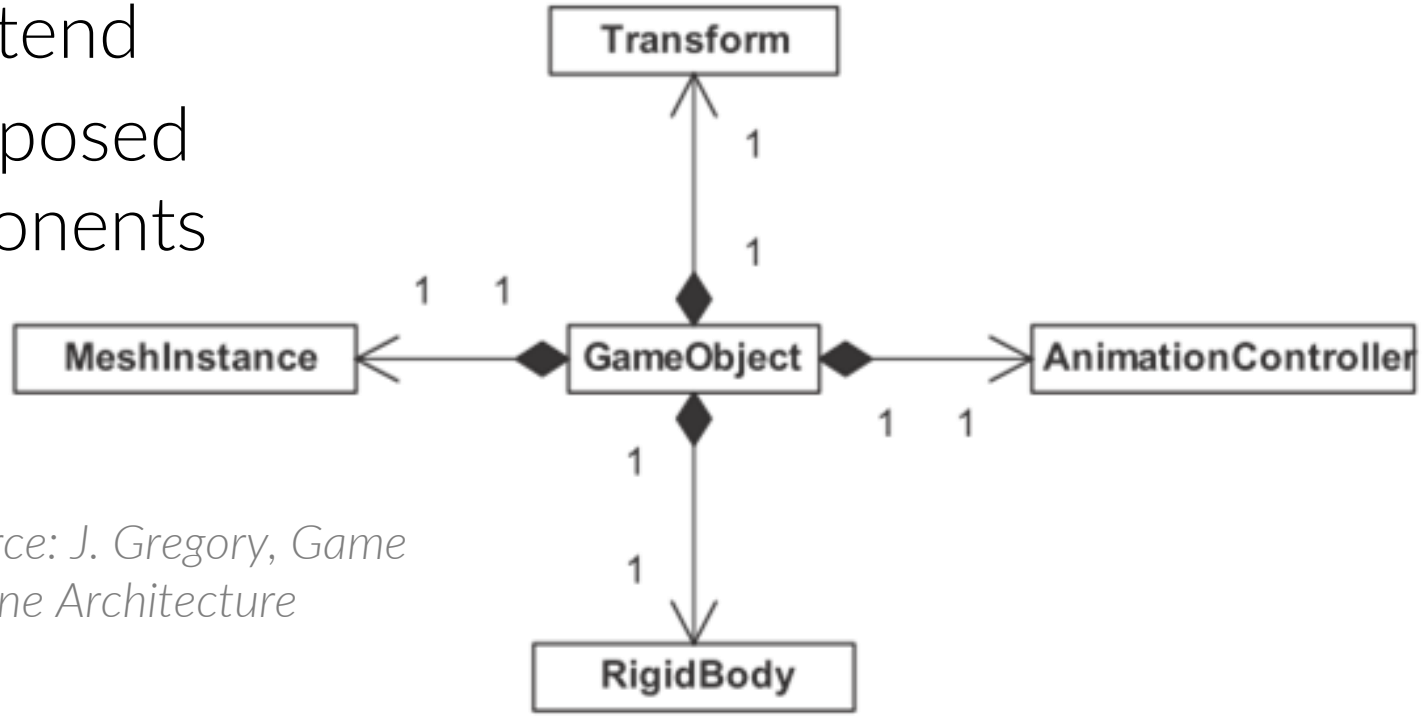
Components

Divide object into dedicated and loosely coupled classes

Each component provides a single well-defined and independent service

Functionalities easier to understand, test, maintain, reuse, refactor and extend

Ex: root GameObject class composed of pointers to all possible components



Source: J. Gregory, Game Engine Architecture

Generic Components

Arbitrary number of instances of each type of component

ex. linked list in the root GameObject

Permits to create new types of components without modifying the game object class

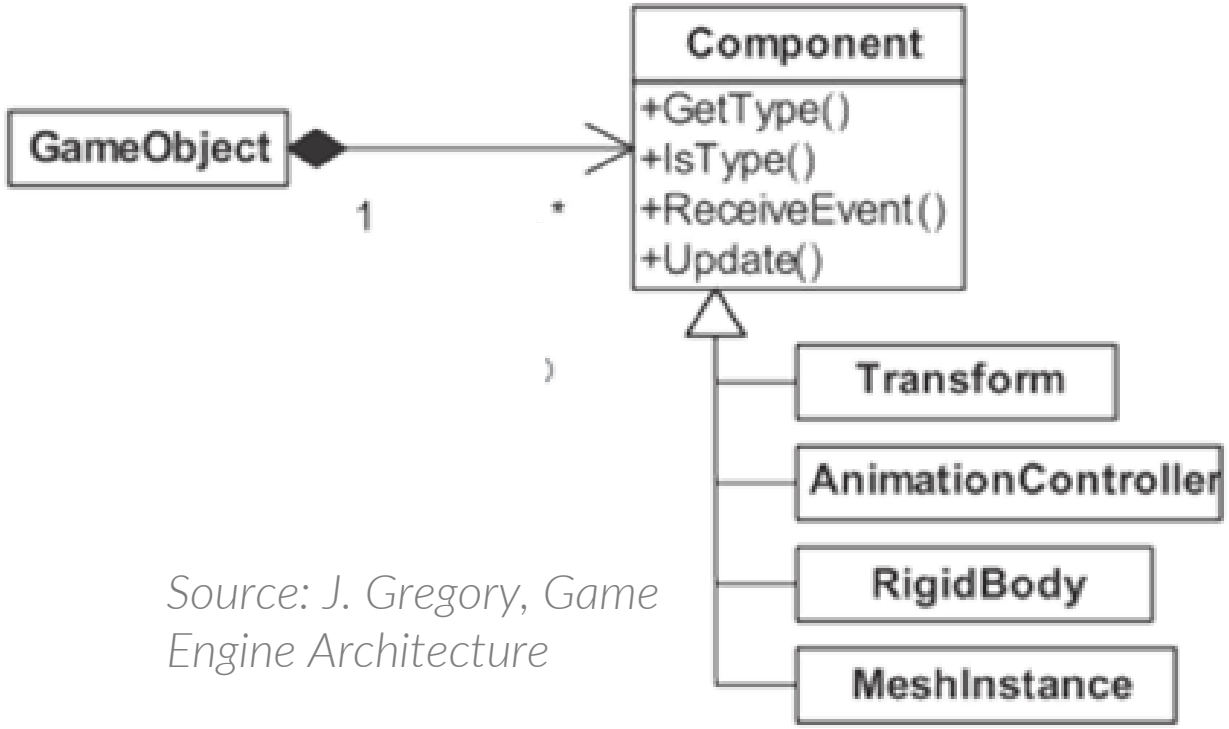
Can iterate polymorphic operations

ex. update

No assumptions about what other components exist within a particular game object

Components can have a hierarchy

ex. Input -> PlayerInput & AIInput



Source: J. Gregory, Game Engine Architecture

Unity

Generic components :

transform, renderer, collider…

Behaviours, Monobehaviours

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)

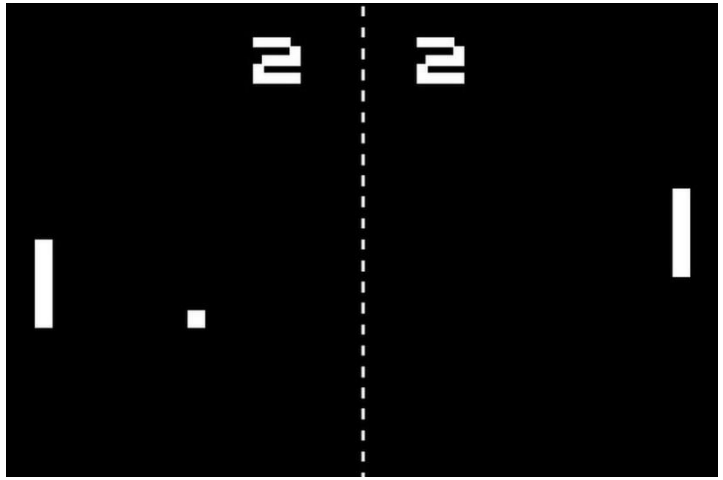
Higher-level systems (e.g. AI): 1 or 2 times/second (can be async. with rendering)

⇒ 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
}
```

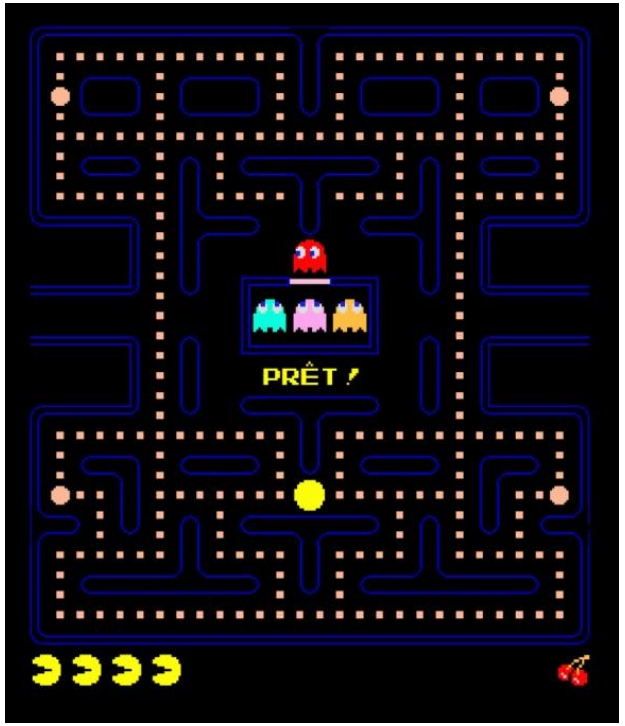
Theoretical Example: Pong

```
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();
}
```

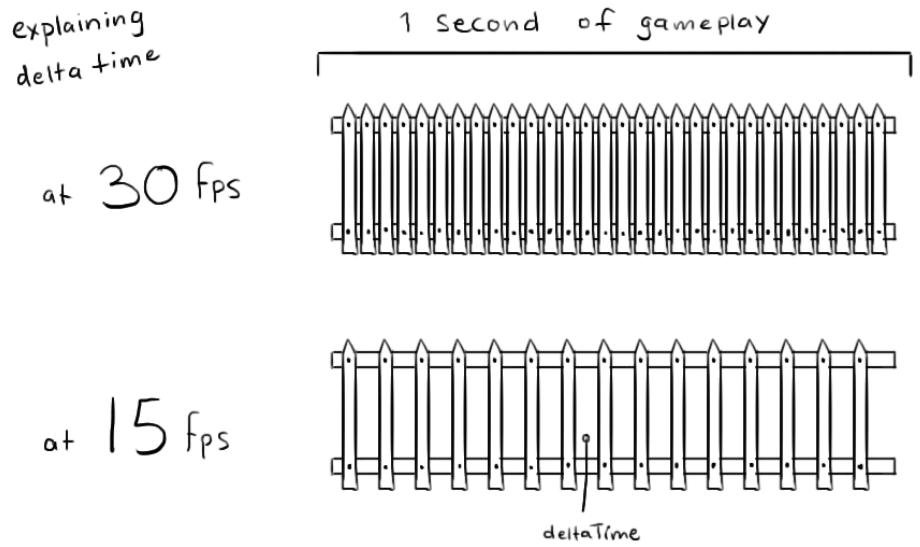
Frame rate

- 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



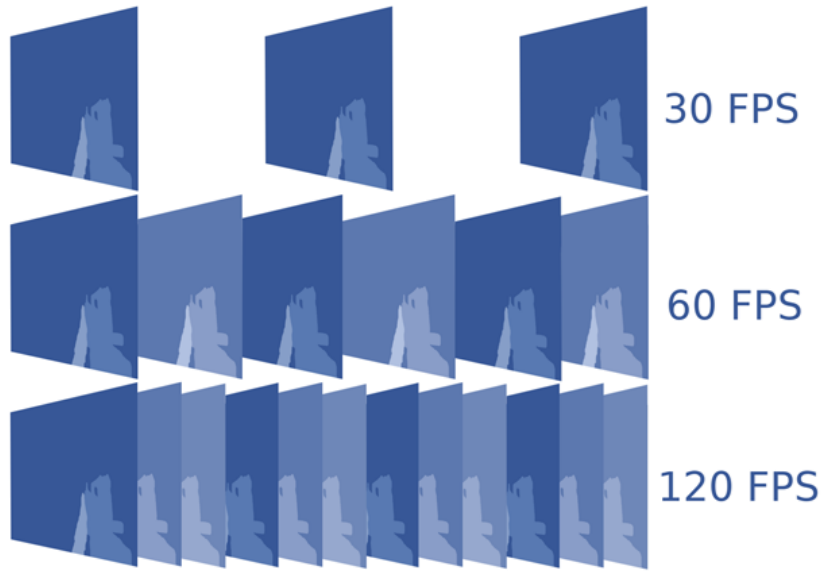
Frame rate

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

Breakpoints issue

Game loop stops running but not CPU nor clock

=> A false measured frame time

Simple solution:

Compare $\boldsymbol{\Delta t}$ to predefined upper limit and set $\boldsymbol{\Delta t}$ to an artificial target frame rate

Frame Rate Governing

Attempt to guarantee frames' duration (rather than guess)

Frame limiting: delay rendering if update is complete before a fixed target frame rate

```
while (true){
    double start = getCurrentTime();
    processInput();
    update();
    render();
    sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

Frame drop: skip a rendering if an update is too long

Frame Rate Governing

OK if game's frame rate is close to target frame rate on average

- "Variable frame rate" mode during development
- Switch on rate governing when game close to consistent frame rate

Consistent frame rate is important for

- Physics
- Graphics
- Record and playback
- Power consumption

Further readings

- http://gameprogrammingpatterns.com/game-loop.html
- http://gafferongames.com/game-physics/fix-your-timestep/
- http://www.brandonfoltz.com/downloads/tutorials/The_Game_Loop_and_Frame_Rate_Management.pdf
- http://higherorderfun.com/blog/2010/08/17/understanding-the-game-main-loop/

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();
}
```

Phased updates

Game objects/subsystems can depend on one another => Order

=> Subsystems update within the main game loop

=> Call of game objects updates

- May be updated multiple times during the frame if it depends on intermediate results of calculations
- Not all game objects require all update phases
- Cost of iteration

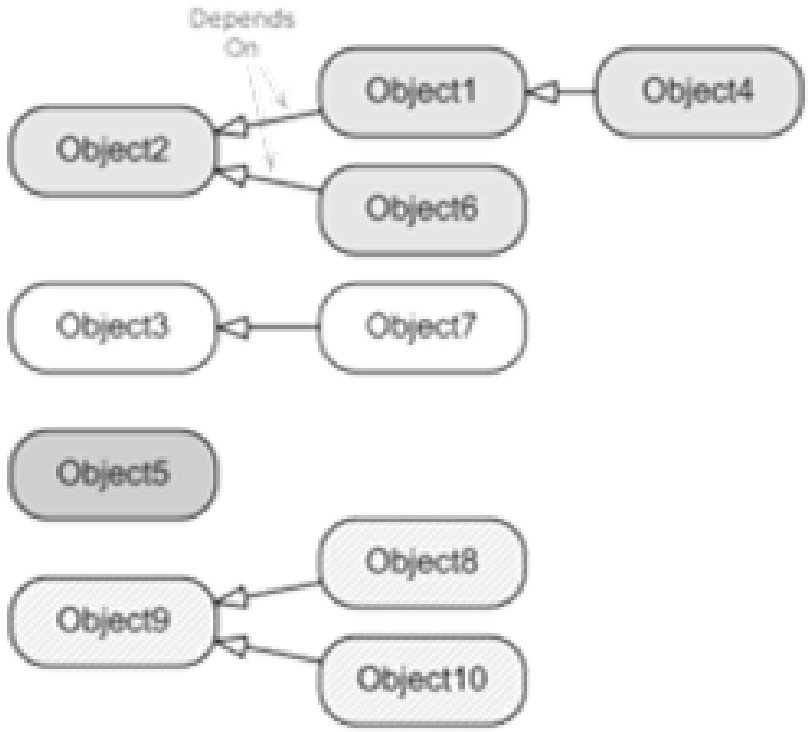
Bucketed updates

Inter-object dependencies can lead to conflicting rules governing the order of updates

=> Collect objects into N independent groups

For each bucket, run complete update of the game objects and the engine systems, then all update phases

Repeat for each bucket



Object State Inconsistencies - One-Frame-Off Lag

Objects are not updated from $t1$ to $t2$ instantaneously and in parallel but sequentially

The states are consistent before and after the update loop, but may be **inconsistent during the loop**

Problem when game objects query one another for state information: previous state or new state?

=> Object state caching + Time-stamping

- Caches previous consistent state vector $Si(t1)$ while calculating new $Si(t2)$ rather than overwriting it during update
- Allows any object to query the available $Si(t1)$ of any other object without regard to update order
- Can linearly interpolate between previous and next states to approximate the state of an object at any moment

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/

Unity

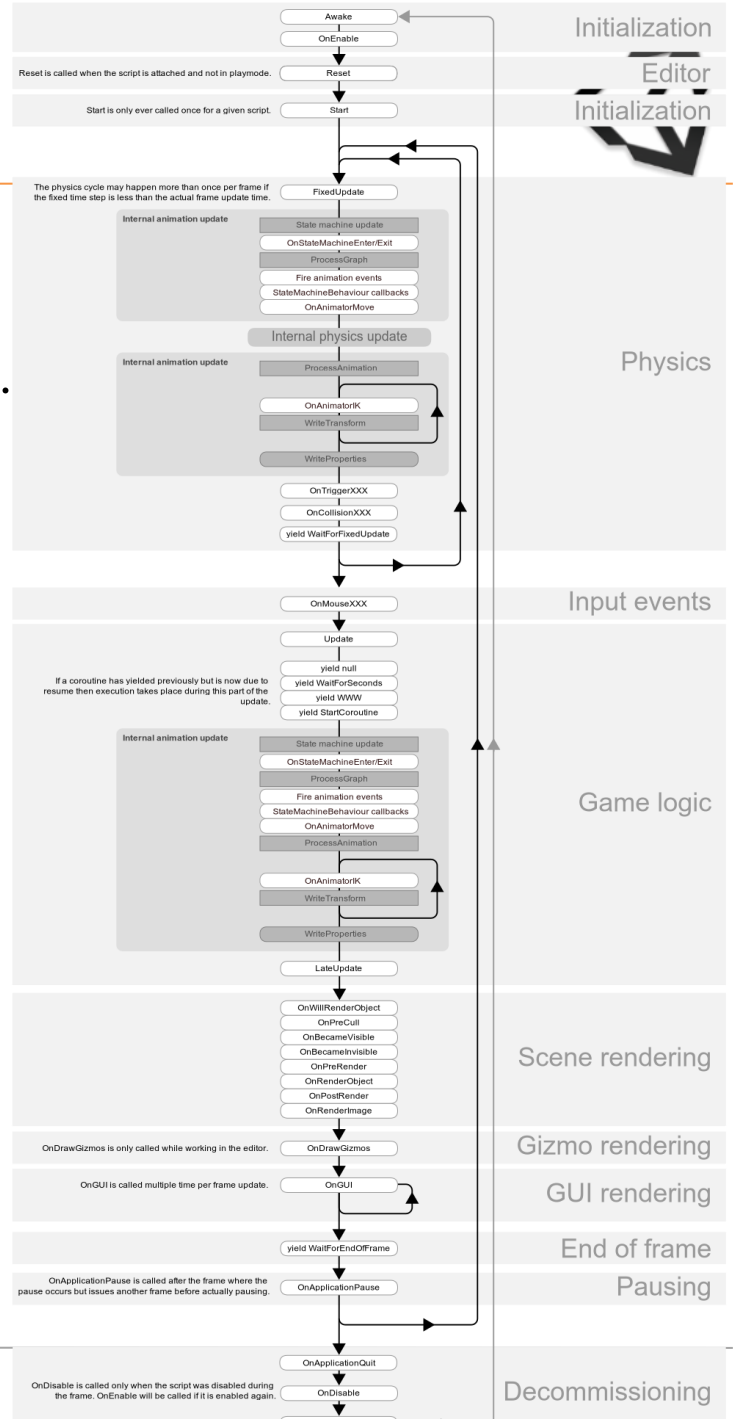
Callback-driven framework

Game parts already implemented: game loop, rendering…

customizable functions called during the game loop (`Start()`, `Update()`…)

http://docs.unity3d.com/Manual/ExecutionOrder.html

http://docs.unity3d.com/Manual/class-ScriptExecution.html



Unity

Time

- http://docs.unity3d.com/ScriptReference/Time.html
- “Game time“
- timeScale
- deltaTime
- timeSinceLevelLoad
- captureFramerate
- maximumDeltaTime…

http://docs.unity3d.com/Manual/class-TimeManager.html

INPUTS

Inputs

Collect and store all information from the outside world

- Player: mouse, keyboard, touch, controller…
- Network message queues (multiplayer…)
- Saved replay information
- Others: camera, gps…

Process input but doesn't wait for it

NB: Try to keep inputs/events handling separated from the game logic

Unity

- Input
  - http://docs.unity3d.com/ScriptReference/Input.html
- Input Manager
  - Custom axis and buttons, dead zone, gravity, sensitivity, key binding…
- Time
- New Input System (2021)

Inputs ?

SCHMUP !

Read the "game design document" (end of this course)

Component **Input Manager**

In game :

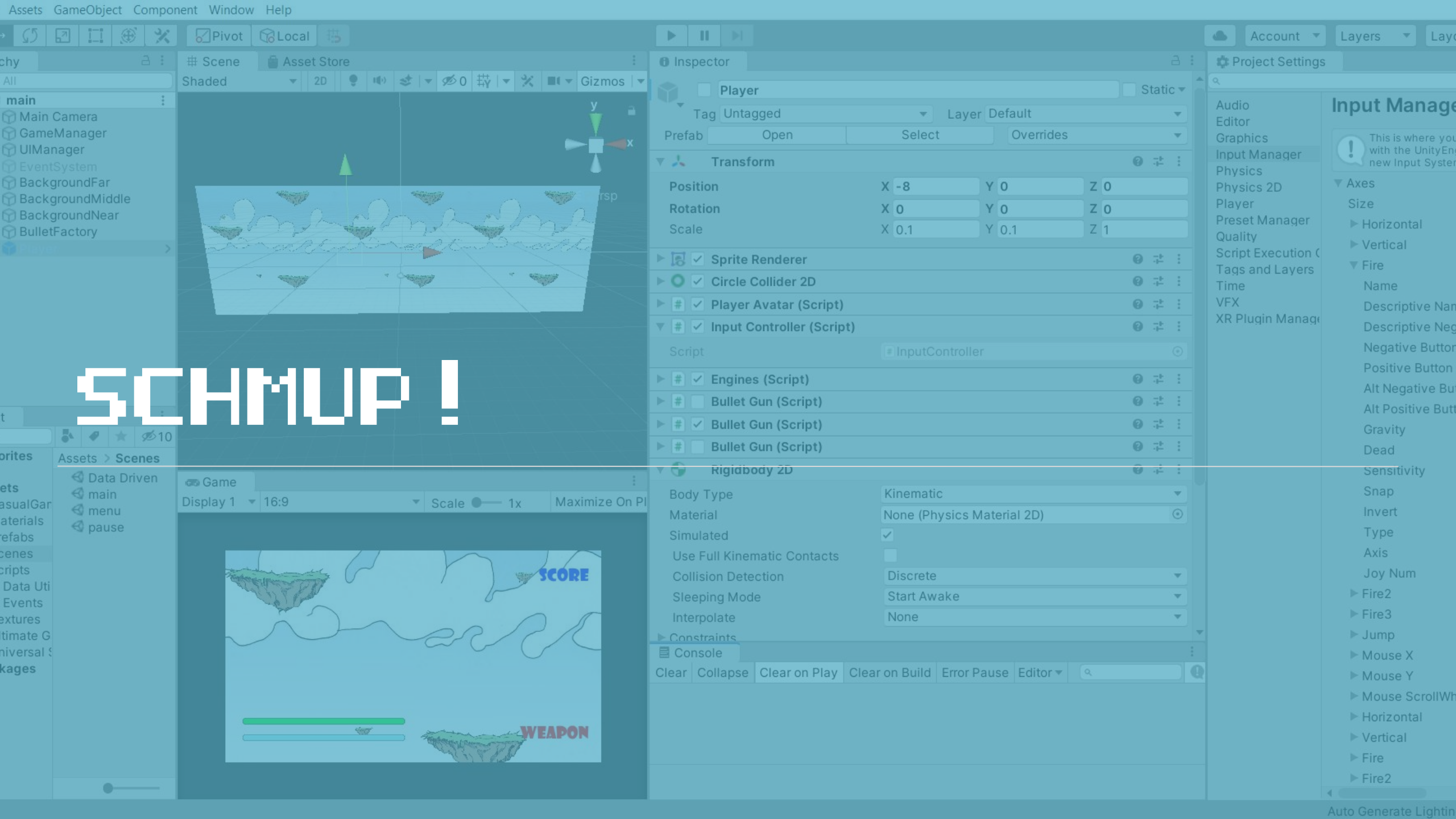
- Player Input only (no network or replay)
- 4 Direction arrows
- 1 Fire key
- 1 Change of shoot key
- 1 Pause/Esc key

In Launch Menu

- Launch
- Quit

In Pause Menu

- Resume
- Restart
- Quit


SCHMUP !
SCORE
WEAPON

SCHMUP !

SETUP

Usual Development Tools

- Game Engine Interface
- Various editors and/or IDE
  - Code: specific engine editor, Visual Studio…
  - Files: json, xml, hex (binary files)…
- Version Control
  - SVN, Git, Mercurial, Perforce…
- Difference & 3-way merge tools
- Build tools

Git

SCHMUP !

1. Register on a git hosting platform

   Github, gitlab, bitbucket, forge ensiie or tsp …

   Complete the necessary procedure for secure connections (ssh)
2. Install the git shell + graphical client

   Github desktop, Sourcetree …
3. Create the dev project
4. Initialize the git repository in the project folder with the "create“ function
5. Dev
6. Commit
7. Set the remote repository
8. Push
9. Goto 5

Coding practices

- Design patterns
  - *Gang of Four* book
  - gameprogrammingpatterns.com
  - Singleton, Iterator, Abstract Factory…
- Recommended coding standards
  - Clean, understandable and commented interfaces
  - Good names and prefixes
  - Consistency
  - Make common errors easier to see

Project Setup

SCHMUP!

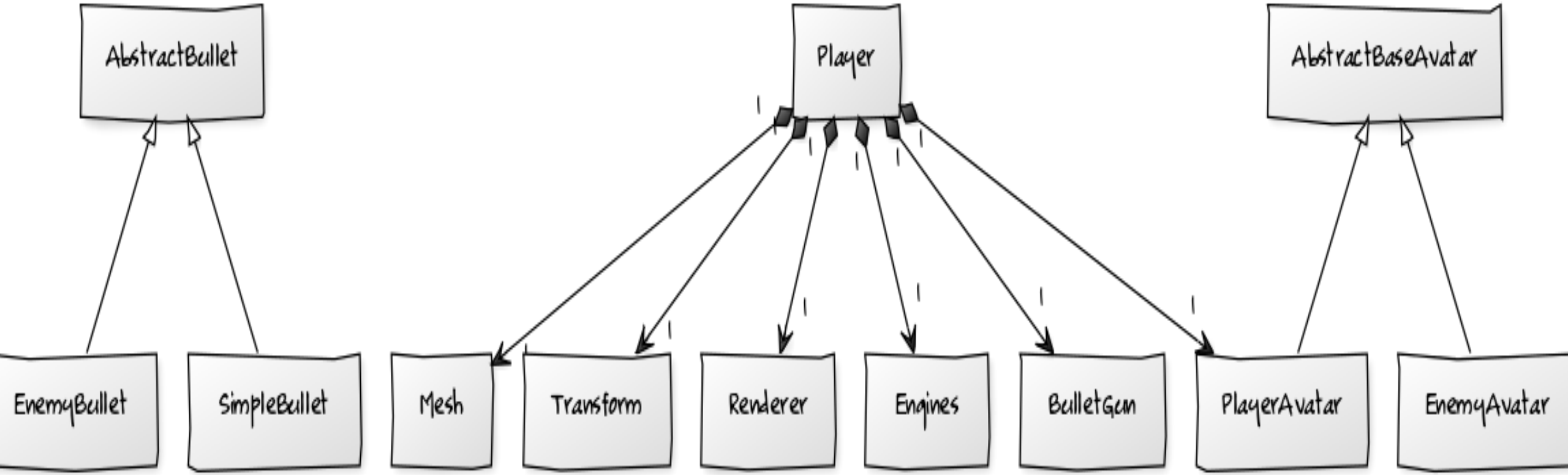
Design the game objects architecture

Gather and organize the assets

Build the game world and set up the objects


SCHMUP !
PLAYER & ENEMIES = MOVE & SHOOT
SCORE
WEAPON

SCHMUP !


AbstractBullet
Player
AbstractBaseAvatar
EnemyBullet
SimpleBullet
Mesh
Transform
Renderer
Engines
BulletGun
PlayerAvatar
EnemyAvatar

Player (resp. Enemy)

SCHMUP !

Prefab composed of

- Sprite renderer
- Collider + RigidBody2D
- PlayerAvatar <- BaseAvatar
  - maxSpeed
  - health
  - energy
  - ...
- Engines
- InputController
- BulletGun(s)

Inputs

SCHMUP !

- Input Manager + Input class Unity
- InputController.cs
  - gathers all user inputs
  - know the other components of the player
    - can get/set their attributs and call their methods

Movements

SCHMUP !

InputController component

change the speed of the engines based on dedicated axis (ex. horizontal/vertical)

Engines component

calculate new position based on position, speed, time et maxspeed

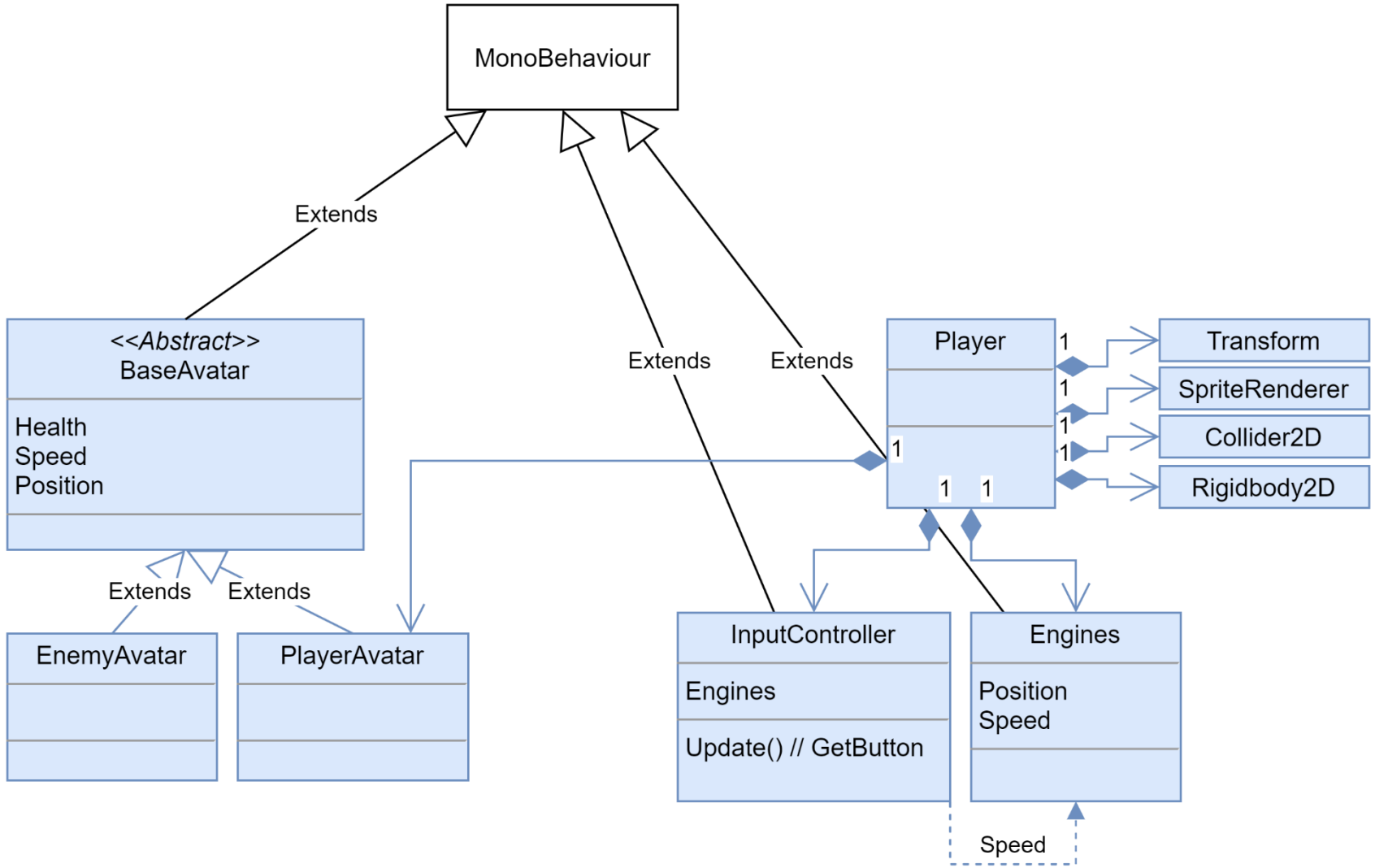
For the enemies : same component with input replaced by a "AI" controlling the speed

Shoot

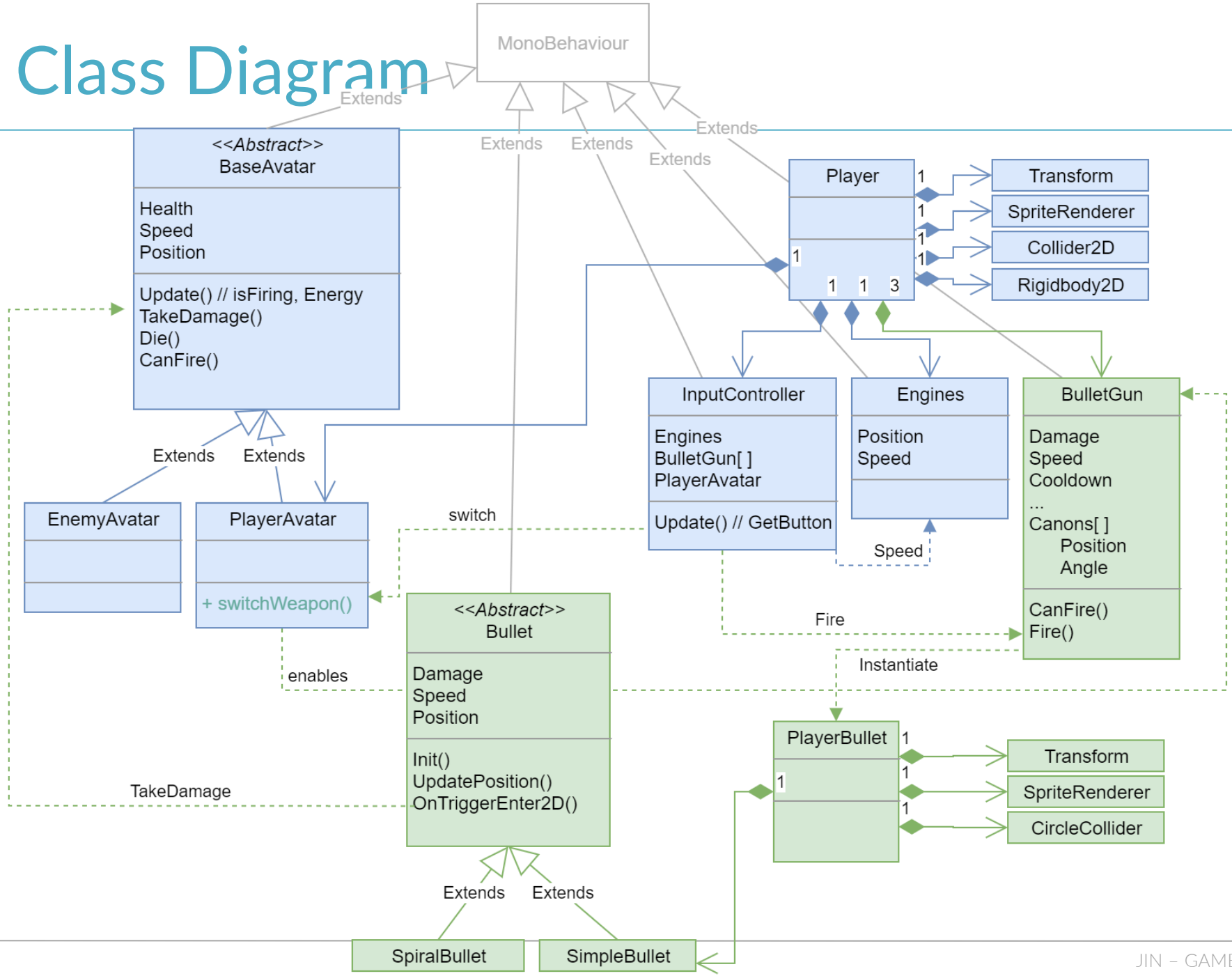
SCHMUP !

- PlayerBullet object
  - Sprite
  - Bullet component
    - damage and speed
    - update position
    - collision test
      - damages to the avatar
- BulletGun component
  - damage and speed
  - fire()

Ex. of Class Diagram



Ex. of Class Diagram



PART 3

LOW-LEVEL ASPECTS

DEBUGGING & PROFILING

Errors

Player errors

Inform and continue

Ex: impossible action, bad input information

Creator errors

Inform and stop

Handle the problem

Ex: bad asset

Programmer errors (bugs)

Fix

Error return codes

Exceptions

- Performance costs

Assertions

- Checks an expression (i.e. assumptions): if false stops the program

Logging and Tracing

- "printf debugging"
- Formatted output
  - Ex. custom `OutputDebugString()`
- Level of verbosity, channels, filters
- Log files
- Crash Reports
  - Gather useful information: level, player location, animation state, running scripts, stack trace, memory allocators states…
  - E-mail

Debug Facilities

- Debug cameras
- Pause and slow motion
- Cheats
  - Displacement, invincibility, infinite characteristics…
  - Might be in the final game
- Screen shots and movie capture
- Debug drawing API
  - Visualization: math calculations…
  - Lines, shapes, points, 3D text…
- In-game menus
  - Configure subsystems options at runtime
  - Call engine functions
- In-game console
  - Command-line interface to the engine
  - Hard-coded commands, rich interface or scripts

Profiling: “90/10 rule”

90% of software running time is caused by 10% of the code

=> Optimizing 10% of the code can potentially save 90% of execution time

Measure the execution time

Time spent in each function, nb of function calls, call graph,

% of the function’s time spent calling each descendant,

% of the overall running time for each function…

Ex. 3rd party profilers

Vtune (Intel), Rational Quantify (IBM)

Memory-Tracking

Stats

Uncharted 4

Leak = out-of-memory

- Memory allocated but not freed

Corruption = data written on wrong memory location

- Other data overwritten
- Right location not updated

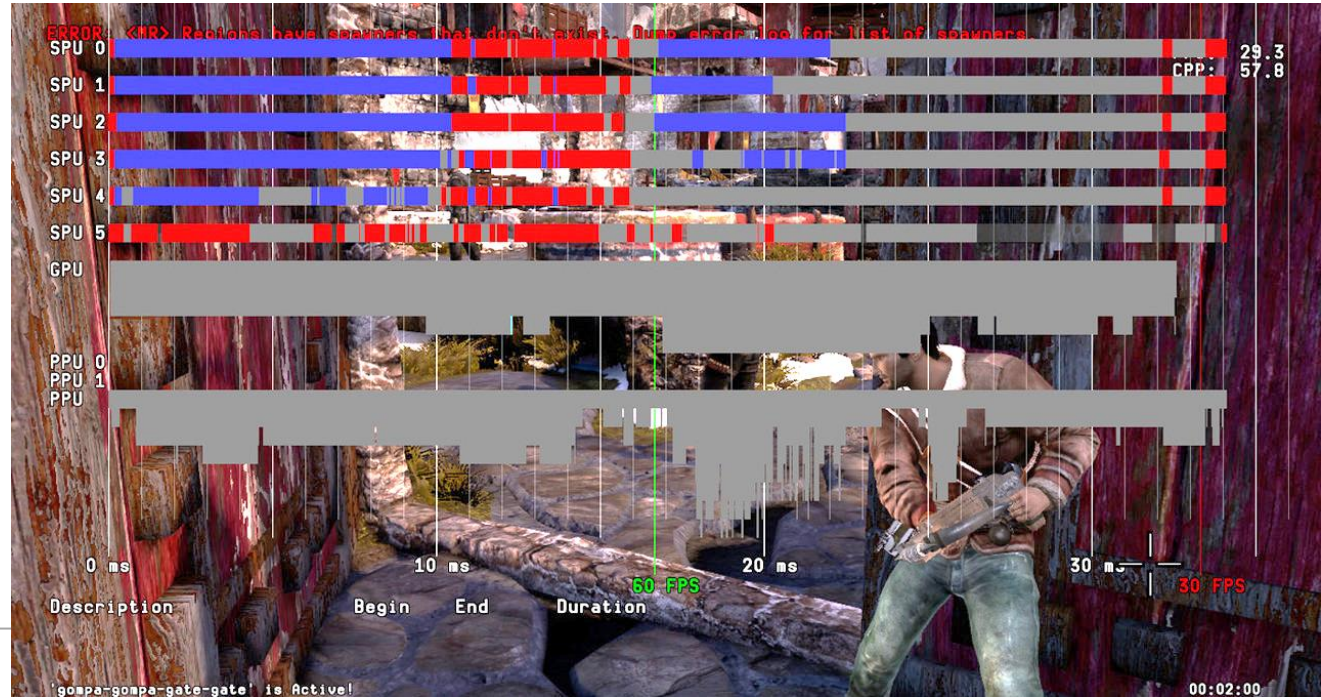
Main cause = pointers

Appropriate and contextual display

Custom or 3rd-party tools

- *Rational Purify (IBM),*
  Bounds Checker (CompuWare)

Uncharted 2



Unity

IDE

Console

print(), Debug.Log(), Debug.Draw()

Debugger

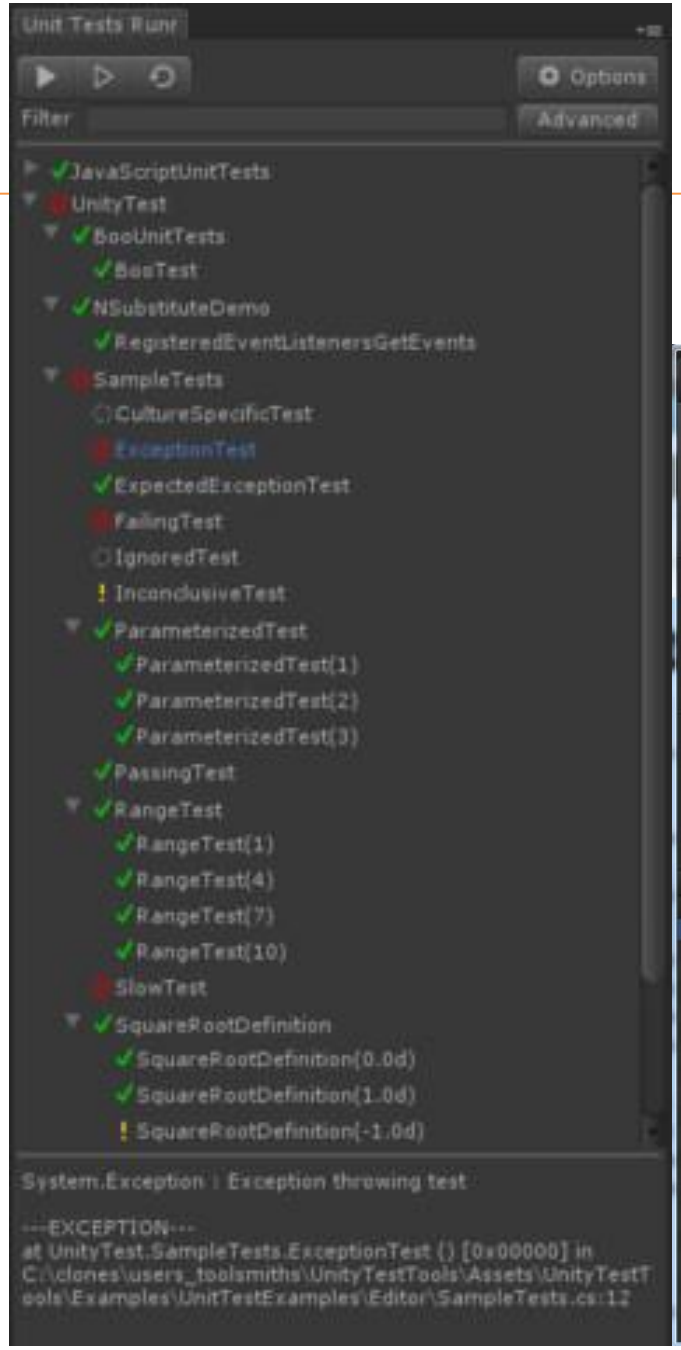
Profiler

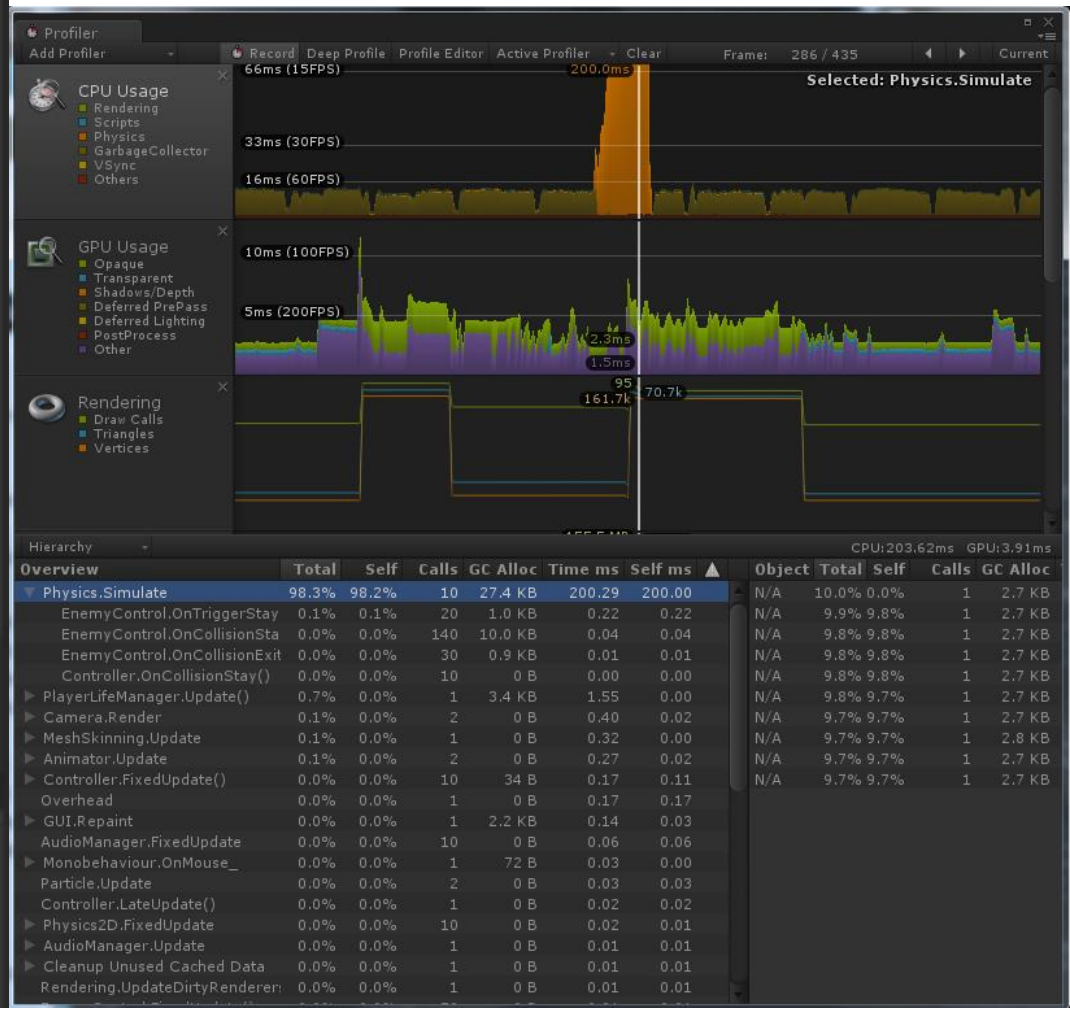
Unit tests

UnityEngine.Assertions

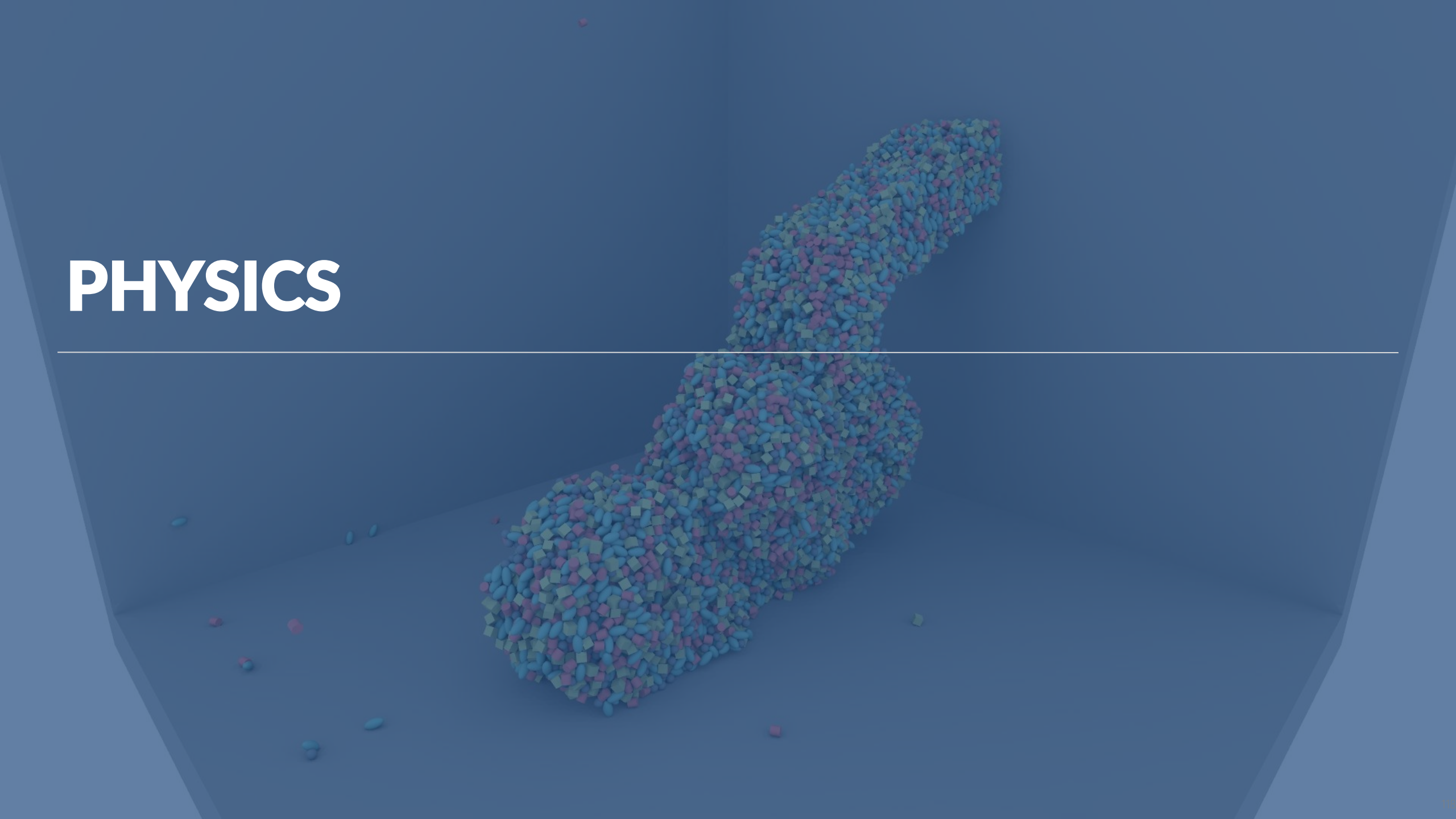
Version control

(integrated or external)





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…

GAME WORLD & FLOW MANAGEMENT

World Chunks

"Levels, scenes, maps, stages, areas"...

Game decomposed into discrete playable regions

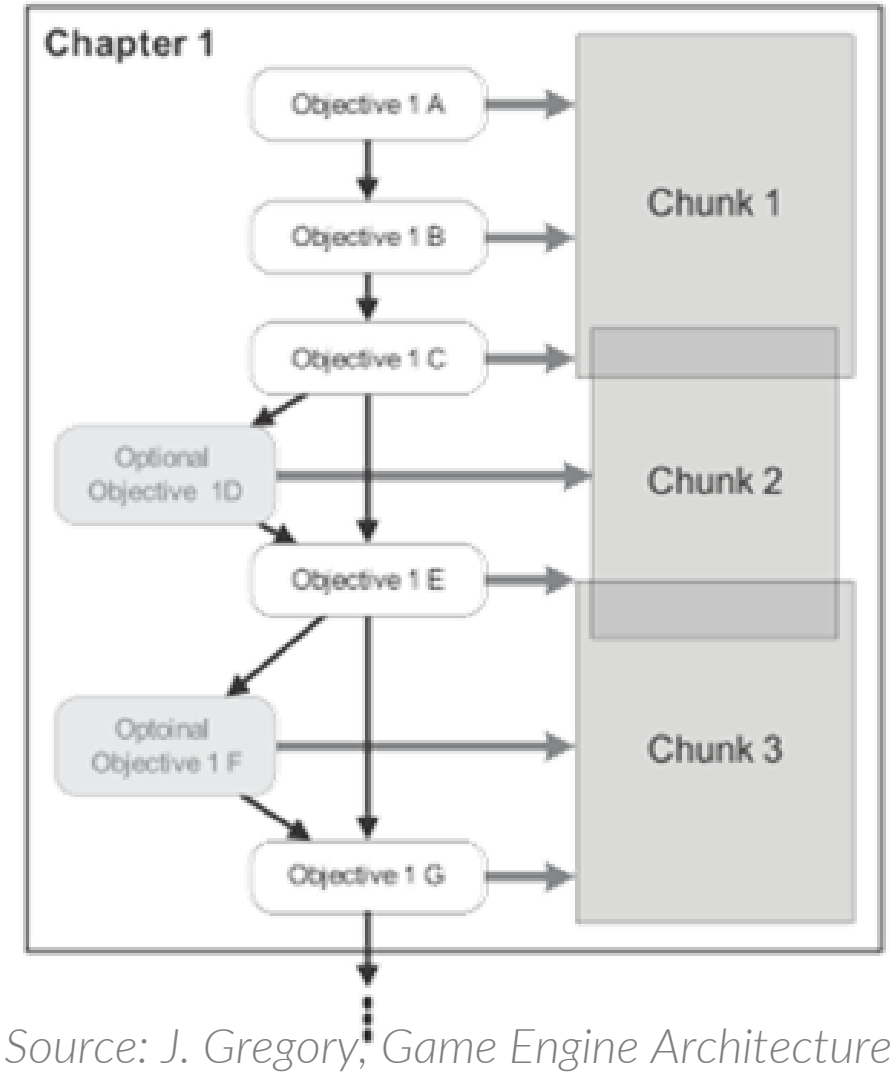
- Linear progression
- Star topology
  - Central hub area
  - Access other areas at random from the hub (sometimes locked)
- Graph-like topology
  - Areas connected to one another in arbitrary ways
- Illusion of a vast, open world

Benefits

- Memory usage : usually only 1 loaded at a time
- Control the overall flow of the game
- Division-of-labor

High-Level Game Flow

- Sequence, tree, or graph of player objectives
  - Definition of success/failure conditions and consequences
  - Can include various in-game movies
  - Ex: tasks, stages, levels, waves…
- Loose coupling chunks / objectives
  - Flexibility of design
  - Objectives grouped into sections of gameplay ("chapters", "acts")



Source: J. Gregory, Game Engine Architecture

Flow & Finite State Machines

- List of states and transitions triggered by conditions
- Each state links a single player objective or encounter and a particular location
- When the player completes a task, the state machine advances to the next state = new goals
- When the player fails to complete a task, the state machine advances to the corresponding state
  - Ex : send back to the beginning of the current state or to the main menu
- Rq: FSM can also be used for handling game objects' states

Loading and Streaming System

Manage the loading of game world chunks and other assets from disk into memory

Manage the spawning and destruction of game objects during the game = classes instantiation

=> File I/O

=> Allocation and deallocation of memory

Chunks Data

- Binary image of each object
  - Trivial spawning
  - Problematic storing
  - Problematic for changes
  - Suitable for stable data structures: mesh data, collision geometry…
- Serialized Game Object Descriptions
  - Writing/reading stream of data that contains enough detail to permit the original object to be reconstructed later
  - Stored in a more-convenient and more-portable format (ex: XML or proprietary)
  - Slow parsing
  - Customized functions vs reflection and generic serialization system

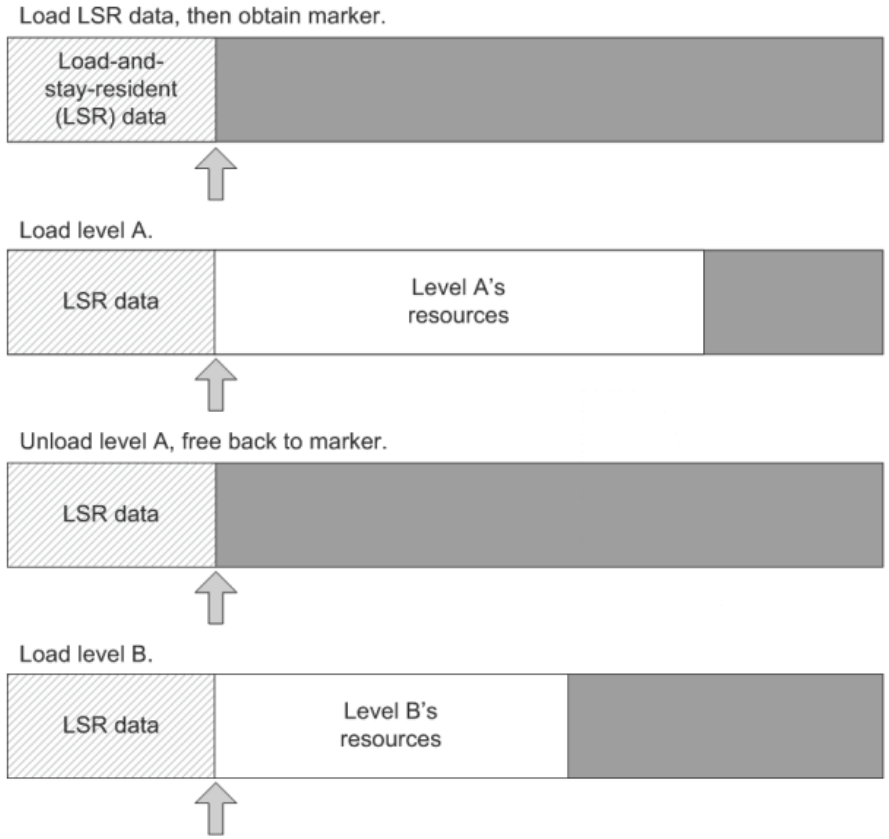
Level Loading

Simple level loading: allow one game world chunk loaded at a time

- Static or simply animated 2D loading screen
- Stack-based allocator
  - Load-and-stay-resident (LSR) data, levels loaded on top
- No way to implement a vast, contiguous, seamless world
- No game world in memory during loading

Air Locks

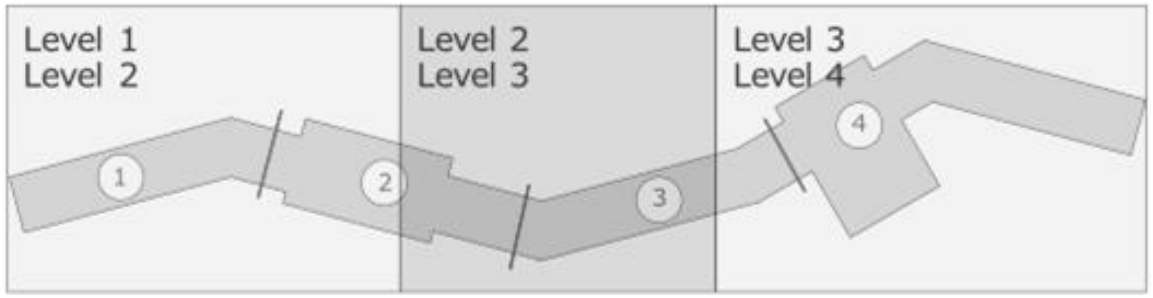
- Large block for a full world chunk
- Small block for a tiny one
- Full chunk can be unloaded and replaced when the player is in the air lock and kept busy



Source: J. Gregory, Game Engine Architecture

Streaming

- Main goals
  - Load and unload data as needed as the player progresses
  - Manage the memory without fragmentation
- Divide every game asset into equally-sized blocks of data
  - Use a pool-based memory allocation system to load and unload resource data as needed and avoid memory fragmentation
  - Which resources to load?



Source: J. Gregory, Game Engine Architecture

Object Spawning

- Off-line memory allocation
  - No game objects can be created or destroyed after world chunks loading
  - Game's memory usage highly predictable
  - Limits game design
- Dynamic memory management
  - Can be slow
  - Can cause memory fragmentation, leading to out-of-memory conditions
  - Need efficient heap allocators

Spawners

- Lightweight, data-driven representation of an object to create it at runtime
  - Id of type => instantiate appropriate class or classes
  - Table of key-value pairs => initialize attributes
- Benefits
  - Simple data management
  - Flexible approach
  - Can be used for other objects, ex: important points (POI for AI characters, coordinate axes for animations synchronization, location for particle or audio effect)
  - Configurable time of spawning

Saved Games

Similar to the world level loading system: saved file store the current state of the game objects

No duplicate copy of any information that can be determined by reading the world level data (static geometry, object without impact on gameplay)

Emphasis on compression

Check points = specific save points

- Some data are always exactly the same and needn't be stored
- Store only the name of the last check point reached, some information about the current state of the player character (health, number of lives remaining, inventory, weapons, ammo…)
- Or start the player off in a known state at each check point

Save anywhere

- Current locations and internal states of every game object whose state is relevant to gameplay
- Omit irrelevant details (ex. Animations)

Unity

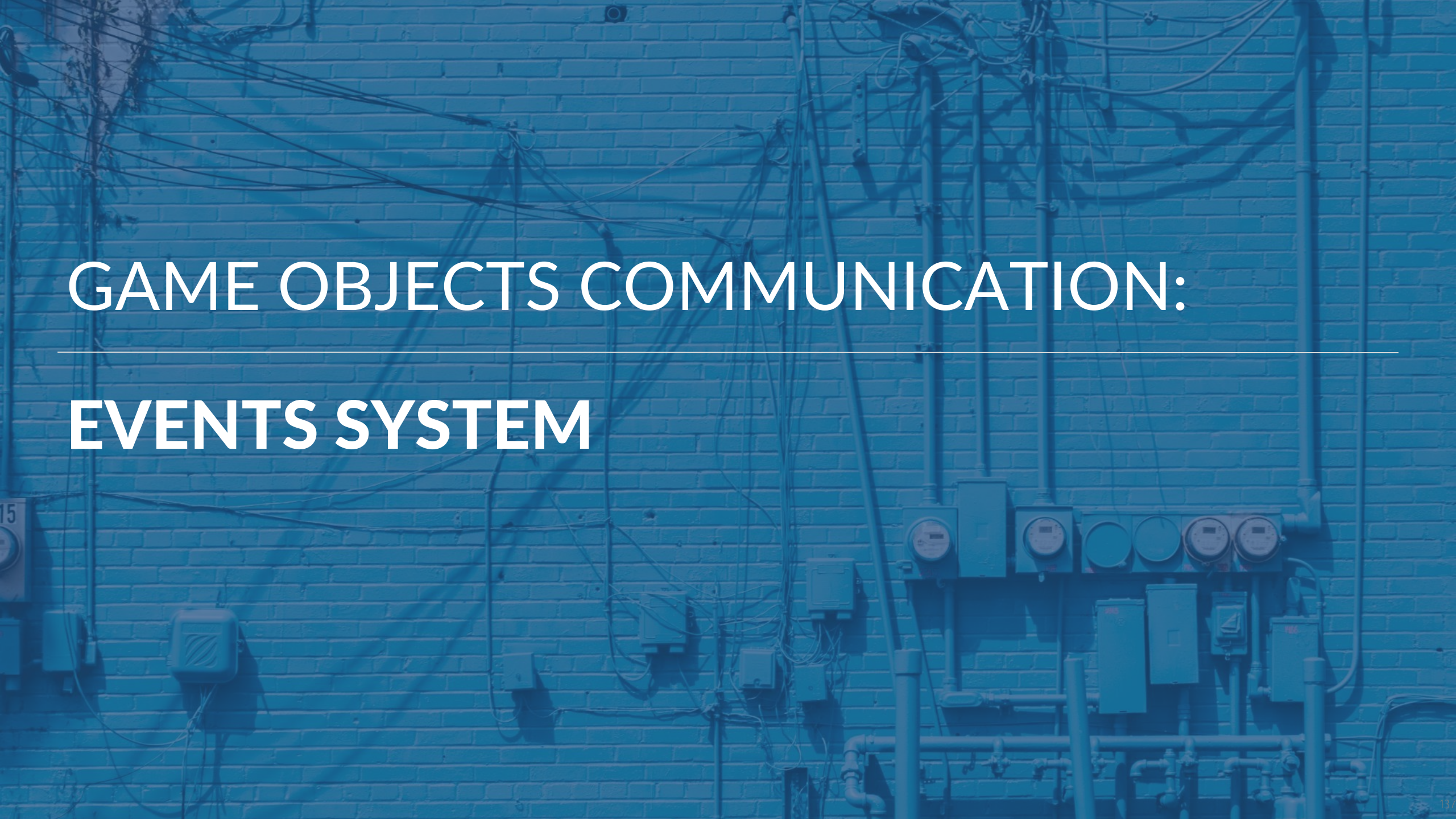
Level Flow

- UnityEngine.SceneManagement
- SceneManager
  - LoadScene()
  - GetActiveScene()
- Object.DontDestroyOnLoad

Object Spawning

- Prefab + Instantiate()
- Destroy()
- Serialization C#
- Resources.Load/Unload

Scenes ?



GAME OBJECTS COMMUNICATION:

EVENTS SYSTEM

Components Communication

- Direct references between some components
  - Simple and fast
  - Coupling
- Shared state in the container object
  - More complex container
  - Possible unused information
  - Communication implicit and order-dependent
- Messages/Events
  - Components can send and receive to/from container
  - Container can broadcast

Events and Communication

Games are inherently event-driven

Event = any interesting change in the state of the game or its environment

Ex. 1: Player hits monster

-> monster's health component

-> Monster death event -> ...

-> monster's animation

-> UI (damages)

-> sound

...

Ex. 2: Achievement system triggered by different aspects of gameplay

Event System

- Global management of all communications in the game
- Engine subsystems or game objects register their interest in particular kinds of events
- Notified when the event occurs
- Handle and respond to the event
  - Different types of game objects will respond in different ways
  - = crucial aspect of their behavior
- Cf. Observer and Command patterns

Event as Objects

Event type

- Hierarchy possible
- *Ex: explosion, friend injured, player spotted, item picked up…*

```
struct Event {
    const U32 MAX_ARGS = 8;
    EventType m_type;
    U32 m_numArgs;
    EventArg m_aArgs[MAX_ARGS];
};
```

Event arguments = data about the event

- Timestamp
- Linked list, dynamically allocated array, various data types…
- *Ex: how much damages, which friend, where spotted, how much bonus…*

Event as Objects: Benefits

- Single event handler function
  - Any number of different event types can be represented by an instance of a single class
  - Need one virtual function to handle all types of events
    - *ex.* `virtual void onEvent(event& event)`
- Persistence
  - Can be stored in a queue for later handling, copied and broadcast to multiple receivers…
- Blind event forwarding
  - Don’t have to “know” anything about the event to send it

Event Types

Global **`enum`**: 1 integer by event

- Simple and efficient (integers)
- Knowledge of all events is centralized
- Hard-coded and Order-dependent
- **`#include`** in every system => global recompilation
- OK for small demos and prototypes

```
enum EventType {
    Event_Object_Moved,
    Event_Object_Created,
    Event_Object Destroyed,
    Event_Guard_Picked_Nose,
    // ...
};
```

GUIDs (globally unique identifiers) for each event + name

Strings

- Flexibility and data-driven nature
- Name conflicts and typos -> user tools (database, user interface, documentation…)
- High memory requirements and comparing costs -> hashed string ids

Ex. of Common Events Types

ActorMove	A game object has moved.
ActorCollision	A collision has occurred.
AICharacterState	Character has changed states.
PlayerState	Player has changed states.
PlayerDeath	Player is dead.
GameOver	Player death animation is over.
ActorCreated	A new game object is created.
ActorDestroy	A game object is destroyed.
Map/Mission Events	
PreLoadLevel	A new level is about to be loaded.
LoadedLevel	A new level is finished loading.
EnterTriggerVolume	A character entered a trigger volume.
ExitTriggerVolume	A character exited a trigger volume.
PlayerTeleported	The player has been teleported.

Game Startup Events	
GraphicsStarted	The graphics system is ready.
PhysicsStarted	The physics system is ready.
EventSystemStarted	The event system is ready.
SoundSystemStarted	The sound system is ready.
ResourceCacheStarted	The resource system is ready.
NetworkStarted	The network system is ready.
HumanViewAttached	A human view has been attached.
GameLogicStarted	The game logic system is ready.
GamePaused	The game is paused.
GameResumedResumed	The game is resumed.
PreSave	The game is about to be saved.
PostSave	The game has been saved.

Animation and Sound Events	
AnimationStarted	An animation has begun.
AnimationLooped	An animation has looped.
AnimationEnded	An animation has ended.
SoundEffectStarted	A new sound effect has started.
SoundEffectLooped	A sound effect has looped back to the beginning.
SoundEffectEnded	A sound effect has completed.
VideoStarted	A cinematic has started.
VideoEnded	A cinematic has ended.

Events Sending

- Each event is linked to a dynamic list of listeners
  - List of delegates = function pointers that can be coupled with an object pointer and used as a callback
- Send methods
  - By trigger: the event will be sent immediately
  - By queue: events in line and processed globally by the event manager
    - Ability to post events into the future, to assign priorities…

Event Handlers

Single function capable of handling all types of events

```
virtual void SomeObject::OnEvent(Event& event){
  switch (event.GetType()) {
    case EVENT_ATTACK: RespondToAttack(event.GetAttackInfo()); break;
    case EVENT_HEALTH_PACK: AddHealth(event.GetHealthPack().GetHealth()); break;
        //...
    default: break; // Unrecognized event
  }
}
```

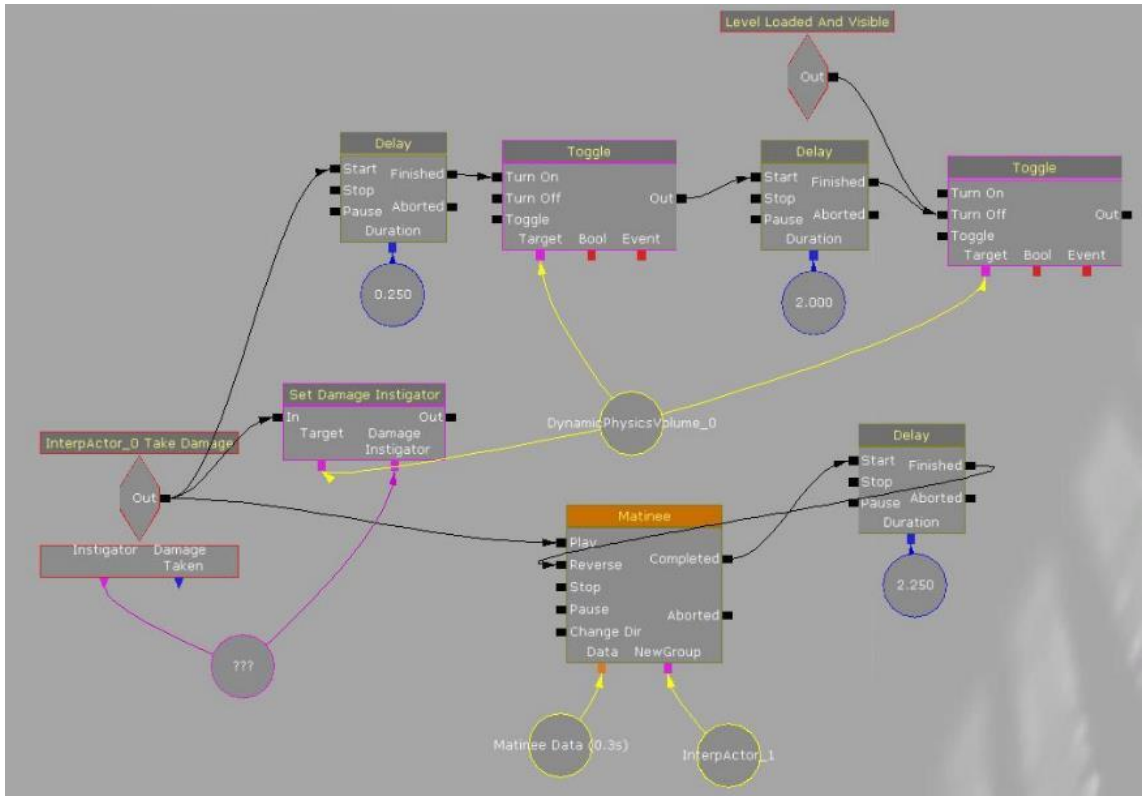
Suite of handler functions for each type of event

Event Forwarding within a graph of objects

= Chains of Responsibility pattern

Data-Driven Event Systems: GUI

- Possibility to configure how objects respond to certain events
- Risks/Benefits
  - Ease of use, gradual learning curve, in-tool help and tool tips
  - Error-checking
  - High cost to develop, debug, and maintain
  - Additional complexity, which can lead to bugs
  - Designers sometimes limited



Kismet (Unreal Engine)

Unity

Scripted Events/Messages

- http://docs.unity3d.com/ScriptReference/MonoBehaviour.html
- http://docs.unity3d.com/Manual/ExecutionOrder.html
- http://wiki.unity3d.com/index.php?title=Event_Execution_Order
- http://www.richardfine.co.uk/2012/10/unity3d-monobehaviour-lifecycle/

Unity / C# Events: Method 1

Sender: Simple custom **delegate** (= function pointer) and its associated **event**

Delegate and Event declaration	`public delegate void NewEvent(int eventId);` `public event NewEvent OnMyEvent;`

Event raising	`if (OnMyEvent != null){`
	`  OnMyEvent(i);`
	`}`

Receiver: Subscription and simple handler

Subscription/ Unsub.	`private void OnEnable(){`
	`  OnMyEvent += MyCustomEventHandler;`
	`}`
		`private void OnDisable(){`
		`  OnMyEvent -= MyCustomEventHandler;`
		`}`

Handler	`void MyCustomEventHandler(int eventID){ … }`	

Unity / C# Events: Method 2

Sender: `Event` based on .NET `System.EventHandler delegate` (no data except sender)

Event declaration	`//public delegate void EventHandler(object sender, EventArgs e)` `public event EventHandler OnCleanup;`

Event raising	`if (OnCleanup != null){`
	`  OnCleanup(this); //no data except sender`
	`}`

Receiver: Same subscription and generic handler

Subscription	`private void OnEnable(){`
	`  OnCleanup += MyCleanupEventHandler;`
	`}`

Handler	`void MyCleanupEventHandler(object sender, EventArgs e){ … }`

Unity / C# Events: Method 3 (1)

Custom class specializing **`EventArgs`**

`Event` based on generic .NET **`System.EventHandler`**

`delegate` (data)

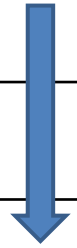
Event Class	`public class MessageReceivedEventArgs : EventArgs {`
	`  private string myMessage;`
	`  public MessageReceivedEventArgs(string m){myMessage = m;}`
	`}`

Unity / C# Events: Method 3 (2)

Sender:

Event declaration	`//public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e) where TEventArgs : EventArgs` `public event EventHandler<MessageReceivedEventArgs> OnMessageReceived;`

Event raising	`if (OnMessageReceived != null){`
	`  OnMessageReceived(this, new MessageReceivedEventArgs("message"));`
	`}`



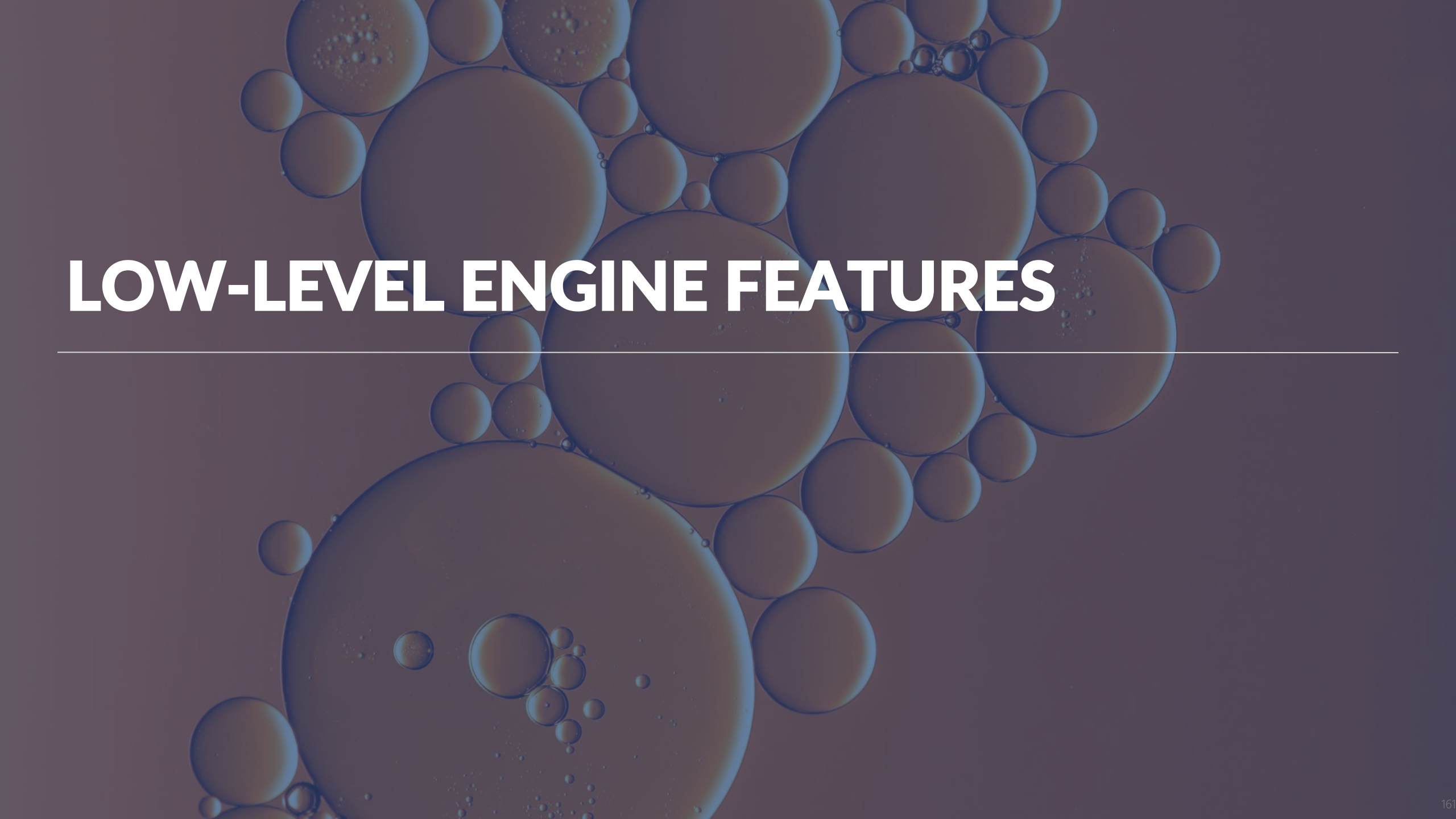
Receiver:

Subscription	`private void OnEnable(){`
	`  OnMessageReceived += MyMessageEventHandler;`
	`}`

Handler	`void MyMessageEventHandler(object sender, MessageReceivedEventArgs e){` `… }`

Events?

SCHMUP!



LOW-LEVEL ENGINE FEATURES

Engine Configuration

- Load and save configuration options
  - Text files: INI, XML
  - Compressed binary files: for memory cards
  - Windows registry
  - Command line
  - Environment variables
  - Online user profiles
- Per-user options
  - Slots, folders, registry…

Subsystem Start-Up and Shut-Down

- Major subsystems usually implemented as singleton ("managers")
  - Start-up and shut-down functions
- Each subsystem must be configured and initialized in a specific order defined by their interdependencies
- Shut-down typically in the reverse order

Memory Management

- Dynamic allocation is slow
- Fragmentation can occur
  - Allocations may fail even when there are enough free bytes
  - Allocated memory blocks must always be contiguous

=> Avoid heap allocations

=> Favor Pool/stack allocators

Cache coherency

- Processors have a high-speed memory cache
  - If the requested data already exists in the cache => loaded directly in registers => much faster than reading from RAM
- Practical solutions to avoid cache misses
  - Organize data in contiguous blocks as small as possible and access them sequentially
  - Keep high-performance code as small as possible
  - Avoid calling functions from within a performance-critical section of code or place it as close as possible

Containers

Types

- Array, dynamic array, linked list, stack (lifo), queue (fifo), double-ended queue, priority queue…
- Tree, binary search tree, binary heap
- Dictionary, hash table, set
- Graph, directed acyclic graph

Operations

- Insert, remove, sequential access, random access, find, sort

Iterators

Custom classes vs. 3rd party SDK

- Control, optimization, customization, no external dependies vs.
- Rich set of features, robustness, generic algorithms

Strings

Natural for objects and assets unique identifiers

Expensive at runtime: comparison, copy…

=> profiling

Storing

- Array of chars
- String class
- Hashed string ids (without collision): hashing at runtime or preprocessed

Localization concerns

File names and paths manipulation are complex

CONCLUSION

Takeaways

- Design, architecture, data structures…
- Deepen in search of solutions
  - Theory and Practice
  - Google: "Game programming/dev" rather than "unity"
  - Focus on a problem and solve it completely
- Test and compare other engines

Further readings

gameenginebook.com/

gameprogrammingpatterns.com/

Game Programming Gems 1 (2002) to 8 (2010), Charles River Media

Game Engine Gems 1 and 2, 2010-2011

gamedeveloper.com/ (ex. GamaSutra)

Game Developer Conference

- gdconf.com/
- youtube.com/channel/UC0JB7TSe49lg56u6qH8y_MQ
- gamasutra.com/features/gdcarchive/

gamedevs.org/ (list of various technical articles)

gamemechanicexplorer.com/#

redblobgames.com/

pixelnest.io/tutorials/gamedev-resources/

...

PROJECT GAME DESIGN

Maxence Voleau - *Game Designer @ Amplitude*

Player avatar

SCHMUP !

Simple movement

- Move up / down / left / right using directional arrows and ZSQD (for any keyboard) *
- Moving at constant speed. No slowdown when changing direction. Control must be fluid. *
- Shoot using space bar *

Advanced movement

- Dodging at a distance < d pixels gives invulnerability for x seconds if double tap in one direction (two inputs of the same input in less than y seconds)

Shoot system

SCHMUP !

- Tab to change the type of shooting among 3
  - continuous and straight *
  - continuous and in both diagonals, at 45 °
  - continuous and spiral
- The projectiles touch only the objects of the opposite camp *
- Shooting begins when the button is pressed, and ends when released *

Camera

SCHMUP!

Fixed*

Avatar placed in a band representing 10% of the screen to the left.

Energy

SCHMUP !

- Each moment spent shooting consumes energy *
  - depending on the type of fire: energy and delta variable time
- The energy recharges x per second as long as the ship does not shoot *
- If energy drops to zero, mandatory reload to 100% and reload slowed by 25% *
- Using dodge consumes energy

Enemies

SCHMUP !

2 types:

- Straight move and regular intervals shots *
- Zigzag move and regular intervals shots

Speed and shot interval are random between two bounds

Each enemy has little life: need a hit and an explosion at minimum, at best a + x score at each death

Game and levels structure

SCHMUP !

2 types of victory / defeat conditions

- Life and finite wave to beat
- No life just gaining score by killing and losing score if hit + combo system if killed without being hit

Main menu then level selection screen, a level is a series of waves of enemies

Collectibles

SCHMUP !

+ energy (current or max depending on the context)

+ life or + combo depending on the condition of victory

Unlock a new shooting type (3x this collectible to unlock the next if avatar progression constraint)

Generation of random collectibles, controlled by the evolution of the game

Go back to engine choice
Go back to objects list
Go back to game loop
Go back to inputs
Go back to objects model