INTRODUCTION TO GAME PROGRAMMING & GAME ENGINES

Guillaume Bouyer, Adrien Allard

v3.2

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

<table>
<tr><th colspan="2">Monday</th><th colspan="2">Tuesday</th><th colspan="5">Wednesday</th><th colspan="2">Thursday</th><th colspan="2">Friday</th></tr>
<tr><td>Am</td><td>Pm</td><td>Am</td><td>Pm</td><td colspan="4">Am</td><td>Pm</td><td>Am</td><td>Pm</td><td>Am</td><td>Pm</td></tr>
<tr><td>JIN Intro</td><td>Proj.</td><td colspan="2">Course Part. 1
+ Project Part. 1</td><td>TC2.b</td><td>TC2.d</td><td>TC2.c</td><td>TC2.a</td><td>Proj..</td><td colspan="2">Course Part. 2
+ Project Part. 2</td><td colspan="2">Adrien Allard
Amplitude Studios,
Talk + Project Part. 3</td></tr>
</table>

Homeworks: 1 (start of Monday), 2 (end of Tuesday), 3 (end of Thursday), 4 (end of Friday)

1. Prerequisites : Unity >= 2017,4 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
Office 111 @ ENSIIE

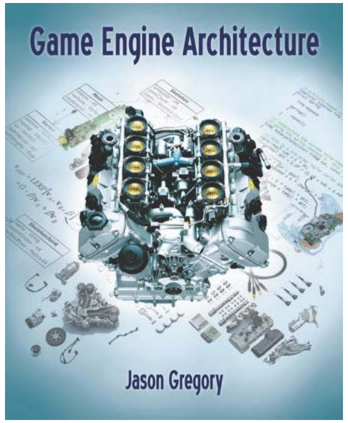
References

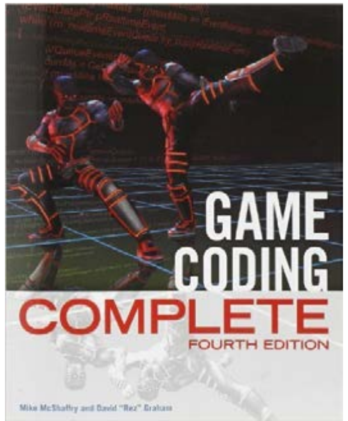
Game Engine Architecture, Jason Gregory, A K Peters/CRC Press, 2009 (http://www.gameenginebook.com/)

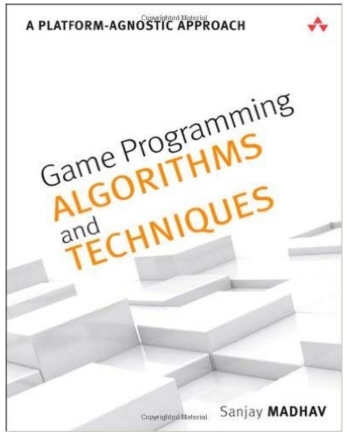
Game Coding Complete, 4th Edition, Mike McShaffry and David Graham, Course Technology, 2013

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

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







What if we programmed our own video game?

JIN PROJECT

PART 1:

THE BASICS

A VIDEO GAME?

What is a video game?

Player's point of view:

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

Raph Koster, A Theory of Fun for Game Design

Artistic & interactive content designed to entertain

Gameplay, game mechanics

- Rules of interactions between the entities
- Objectives, criteria for success and failure
- Player character's abilities
- Number and types of non-player entities in the virtual world
- Overall flow of the gaming experience

More crucial to define a game than its technology

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

- Dynamic game world based on approximated (=numerical) mathematical model
- Various game systems (AI, game logic, physics…) regularly update their state
- Soft: approximations (rendering, physics, audio…) are allowed

Interactive

- Must respond to unpredictable human input
- Must provide rendering of the simulation result by displaying graphics, sound…

Various technical components

- 3D graphics rendering system, collision detection system, audio system, network…

Our game

JIN PROJECT

- Artistic content
- Interactive content
- Gameplay, rules, genre
- Real-time simulation
- Humain input
- Graphics rendering, audio…
- Objects
- …

TEAM

Typical Game Team

Engineers (= programmers)

- Runtime programmers: engine and game
  - Single engine system: rendering, AI, physics…
  - Low level: memory, network…
  - Gameplay (3C : Character-Controls-Camera)
- Tools programmers: off-line tools for the team

=> Lead engineers (+ management), technical directors (high level),… chief technical officer (for the entire studio)

when an engineer meets artists

Artists Produce visual and audio content

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

=> Lead artists, art directors

Typical Game Team

Game designers Design the gameplay

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

Producers

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

Publishers

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

Our team

- Producer : me
- Game designer : contractor
- Artists : internet…
- Engineers : you
- Self-published

TECHNOLOGICAL REQUIREMENTS

BY GENRES

Technological differences

First-Person Shooters (FPS)

Rendering	high fidelity & large 3D virtual worlds (optimized for a particular type of environment)
3C	responsive camera & aiming mechanic, forgiving player character motion and collision model (“floaty”),
Animations	high-fidelity player’s virtual arms and weapons, high-fidelity non-player characters…
AI	non-player characters
Multiplayer	small-scale online capabilities (ex. 64), “death match” gameplay mode…
Gameworld	wide range of hand-held weaponry and pickable items, complex level design…

Third-Person games

~FPS (Rendering, AI, Multiplayer…)

3C	emphasis placed on the main character’s abilities and locomotion modes, 3rd-person “follow camera” focused on the player character + complex camera collision system
Animations	high-fidelity full-body player’s avatar
Gameworld	interesting locomotion modes: moving platforms, ladders, ropes…, puzzle-like environmental elements

Technological differences

Fighting games

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

Racing games

Rendering	usually focus all graphic detail on the vehicles, track, and immediate surroundings, various “tricks” to optimize rendering (distant background elements…)
3C	follow camera (3rd-person) or inside the cockpit (FPS)
Physics	realistic (tires, materials…)
AI	path finding for non-human-controlled vehicles…
Multiplayer	small-scale online capabilities, local split-screen, ranking…
Audio	realistic (tires, engines…)

Technological differences

Real-Time Strategy (RTS)

Rendering units relatively low-res, to support large numbers on-screen, height-field terrain

3C typically oblique top-down camera, restrictions allow to optimize the rendering, grid-layout system to aid align units and buildings, complex user interaction (single-click and area-based selection of units, menus or toolbars containing commands, equipment, unit types, building types…)

AI non-player characters

Multiplayer typically 2 players local or online, ranking

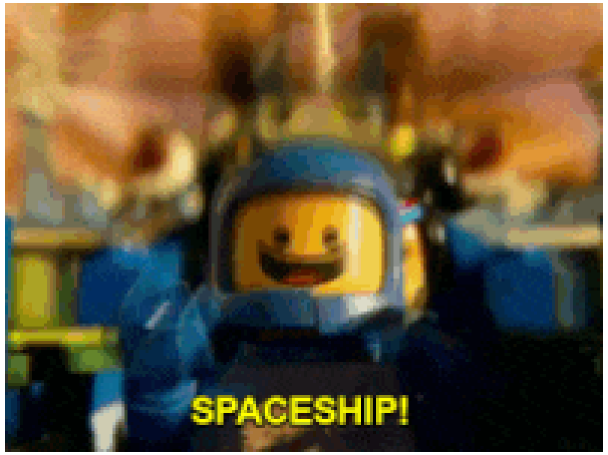
Massively Multiplayer Online Games (MMOG)

Rendering graphics fidelity almost always lower than non-massively multiplayer counterparts (huge world sizes and large numbers of users)

Network powerful battery of servers to maintain the authoritative state of the game world, manage users signing in and out of the game, provide inter-user chat or VoIP services, central server to handle the billing and micro-transactions

Our game?

SCHMUP !



Spaceships !
Bullets !
Shoot'em up !

Our game

SCHMUP !

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…)

GAME DESIGN

Maxence Voleau - *Game Designer @ Amplitude*

Player avatar

- 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

Fixed*

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

Energy

- 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

- 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

- 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

- + 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

DEVELOPMENT

Let's Go !

When development environment is not adapted…

DEVELOPMENT

SCHMUP !

A bit of organization before…

Usual Development Tools

- Editors and IDE
  - Specific engine editor
  - Microsoft Visual Studio
  - Hex editor (inspecting and modifying the contents of binary files)…
- Version Control
  - Central repository to share files
  - Multiple users can modify files collectively
  - History of changes for each file (track and revert)
  - Tagging of versions, branches
  - Source code and game assets
  - SVN, Git, Mercurial, Perforce…
- Difference & 3-way merge tools
- Build tools

Git

Register on a git hosting platform
- Github, gitlab, bitbucket, forge ensiie or tsp …
- Complete the necessary procedure for secure connections (ssh)

Install the git shell + graphical client
- Github desktop, Sourcetree …

Create the dev project

Initialize the git repository in the project folder with the "create“ function

Dev

Commit

Set the remote repository

Push

Goto 5

Coding practices

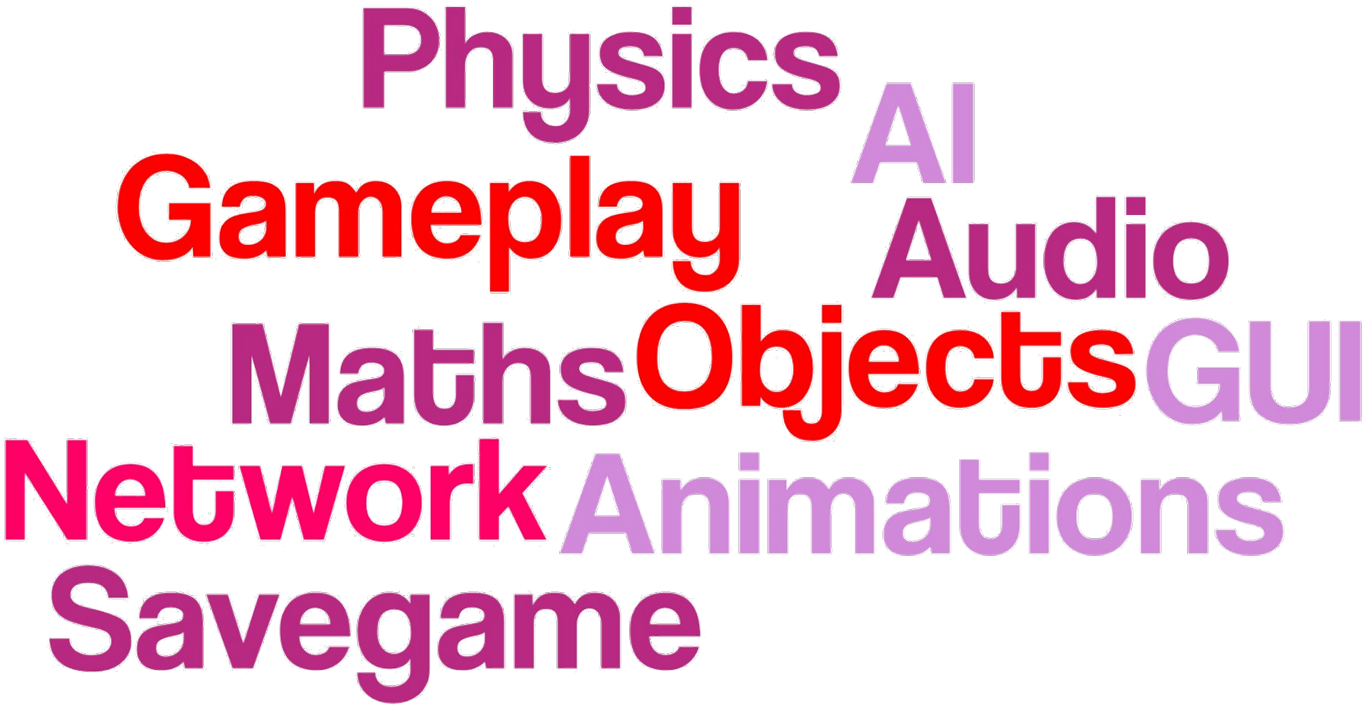
- Design patterns
  - *Gang of Four* book
  - gameprogrammingpatterns.com
  - Singleton, Iterator, Abstract Factory…
- Recommended coding standards
  - Clean, understandable and commented interfaces
  - Good names and prefixes
  - Consistency
  - Make common errors easier to see

Mandatory components

1. Humain input
2. Real-time simulation of game objects
3. Graphics rendering

What else ?

Other components



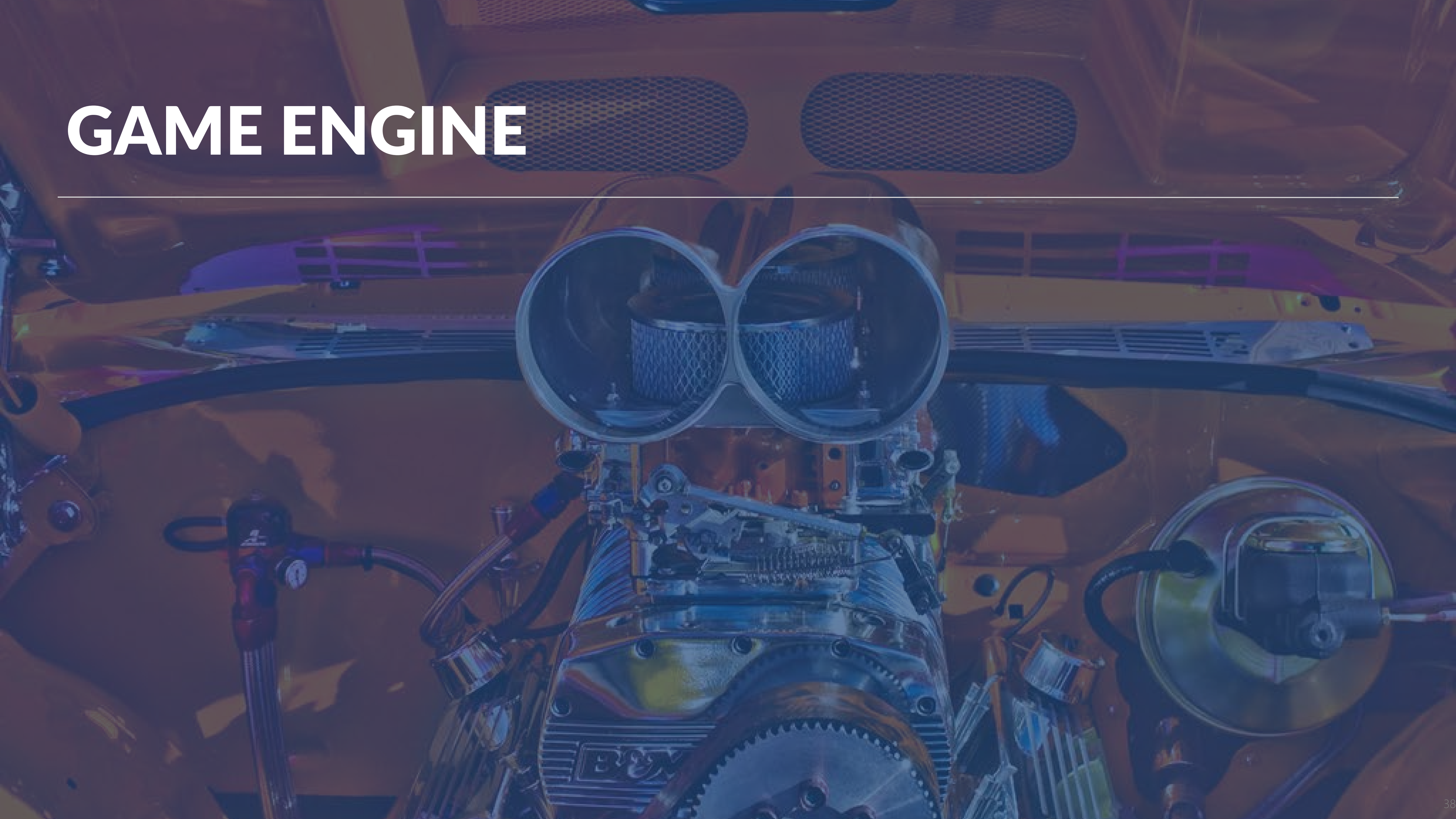
And it might be good to not redo everything for each game

What about not starting from scratch…

SCHMUP !

A little help ?

GAME ENGINE



Game Engine

Extensible set of software that can be used as the foundation for different games

Separates:

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

=> Create new games with new contents & "minimal" changes to reusable core software

=> Mod community

=> Engine licensing = secondary revenue stream

Trade-off's generality/optimality

Technological overlap between games/genres, especially within the same hardware platform

More and more powerful hardware

=> differences between genres are decreasing

=> possible to reuse the same engine technology across disparate genres, and even across disparate hardware platforms

But

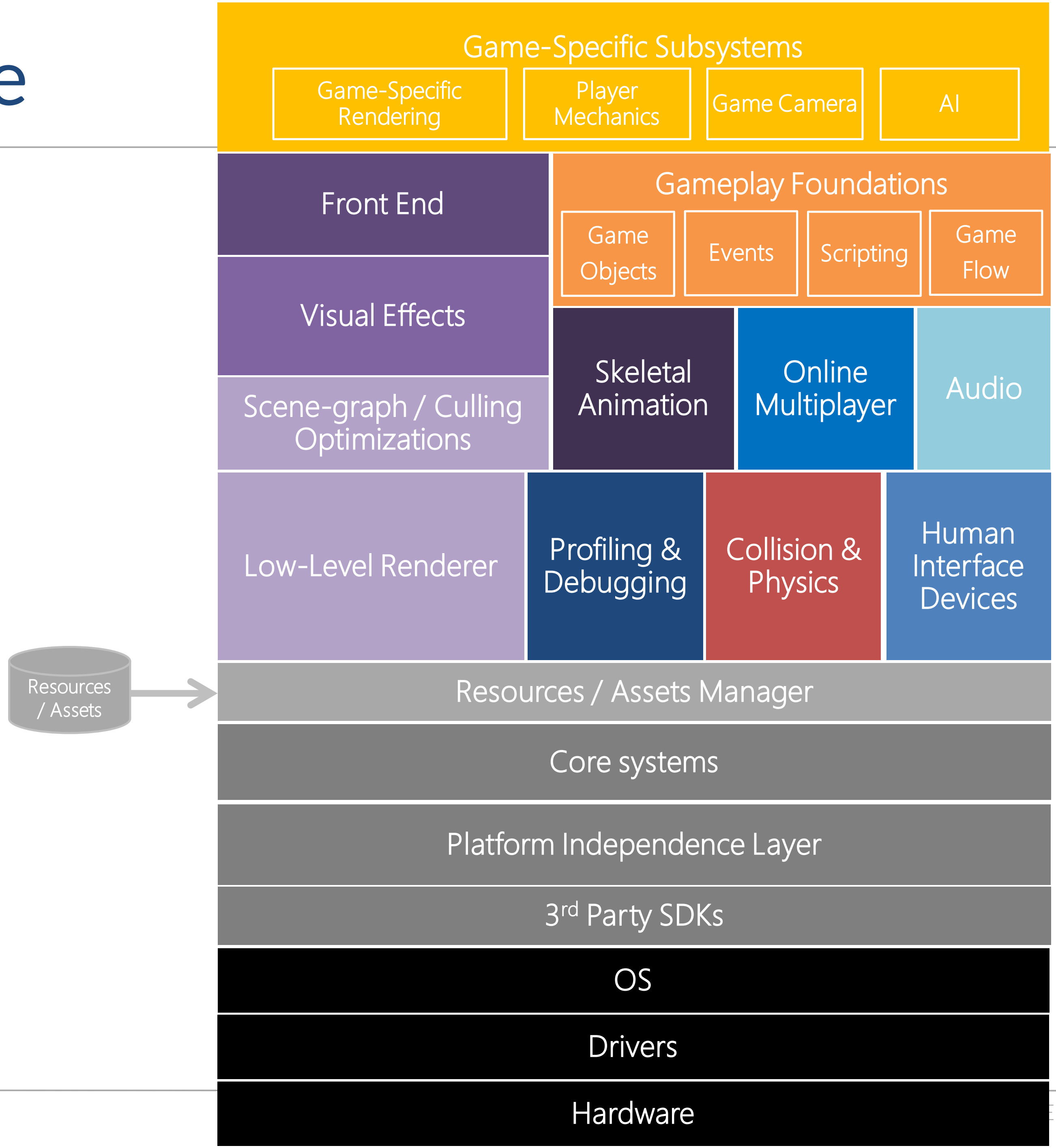
The more general-purpose a game engine or middleware component => the less optimal for running a particular game / particular platform

=> Assumptions about how the software will be used and/or about the target hardware on which it will run

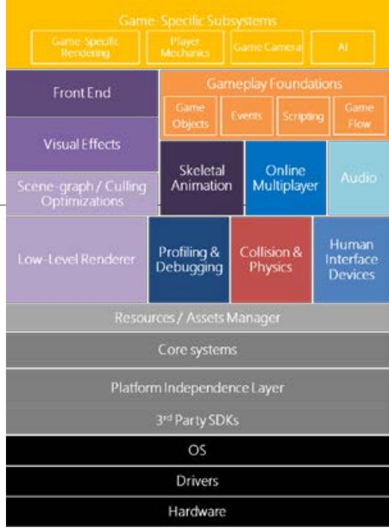
Game Engine Examples

- 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), Amazon Lumberyard
  - FarCry (2004), Crysis (2007), Crysis 2 (2011), Crysis 3 (2013), Evolve (2015)…
- Unity 3D
- Gamemaker, Construct 2, RPG Maker…
- Proprietary in-House Engines
- Open Source Engines
  - Ogre 3D, Panda3D, Yake, Crystal Space, Torque, Irrlicht…

Engine Architecture



Engine Architecture



Drivers

Manage hardware resources, shield the OS and upper engine layers from the communication details

Operating System (OS)

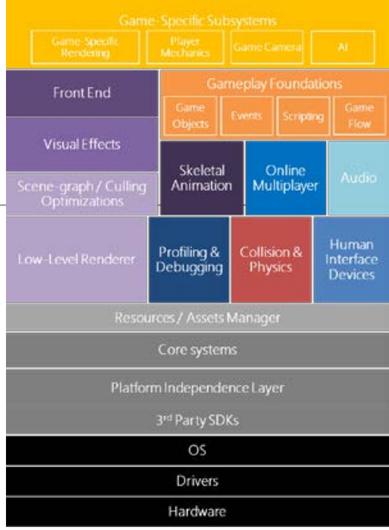
PC

- OS runs all the time
- Orchestrates the execution of multiple programs, including the game
- Pre-emptive multitasking: time-sliced approach to sharing the hardware

Console

- Previously a thin library layer compiled into the game executable: game "owns" the machine
- Now can interrupt the execution of the game, or take over resources, display online messages or dashboard, allow to pause the game…

Engine Architecture



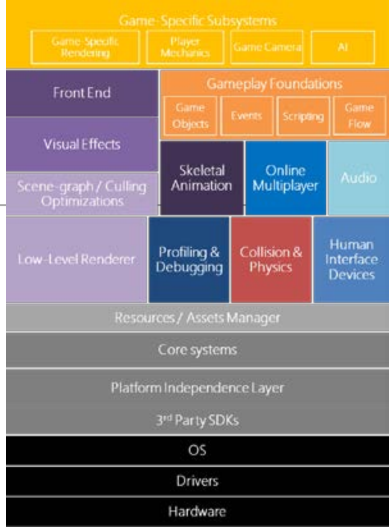
Third-Party SDKs and Middleware

- 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 or replace the most commonly used standard C library functions, OS calls, and other foundational APIs
- Shields the rest of the engine from the majority of knowledge of the underlying platform

Engine Architecture



Core Systems: useful software utilities

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

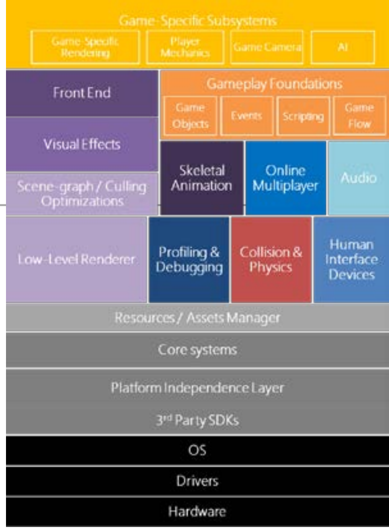
Resource Manager

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

Profiling and Debugging Tools

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

Engine Architecture



Rendering Engine

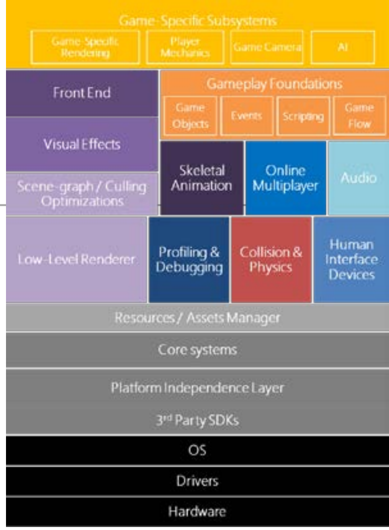
- Low-Level Renderer
- Scene Graph/Culling Optimizations
- Visual Effects
  - Particles, decal, light and environment mapping, dynamic shadows, full-screen post effects (HDR, AA, color correction…)
- Front End
  - 2D or 3D: Heads-up display (HUD), in-game menus, console, development tools, in-game GUI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

- Manages and transforms the low-level raw data from the hardware
- Provides high-level game controls and detection (chords, sequences, gestures…)

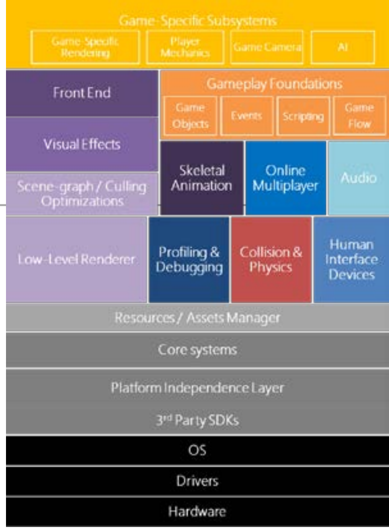
Audio

- 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
- Level management and streaming
- Messaging and event handling between objects
- Scripting language
- 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…

Data-Driven Engines

- Game team must efficiently produce very large amounts of contents
- Data-driven engine permits designers and artists to
  - Create content
  - Control (some parts of) the behavior of the game
  - Directly by data rather than exclusively by programming
- Benefits and risks
  - Improved creation and iteration times
  - Heavy cost to develop appropriate runtime code and robust and usable tools

Choosing an engine

Questions examples

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?

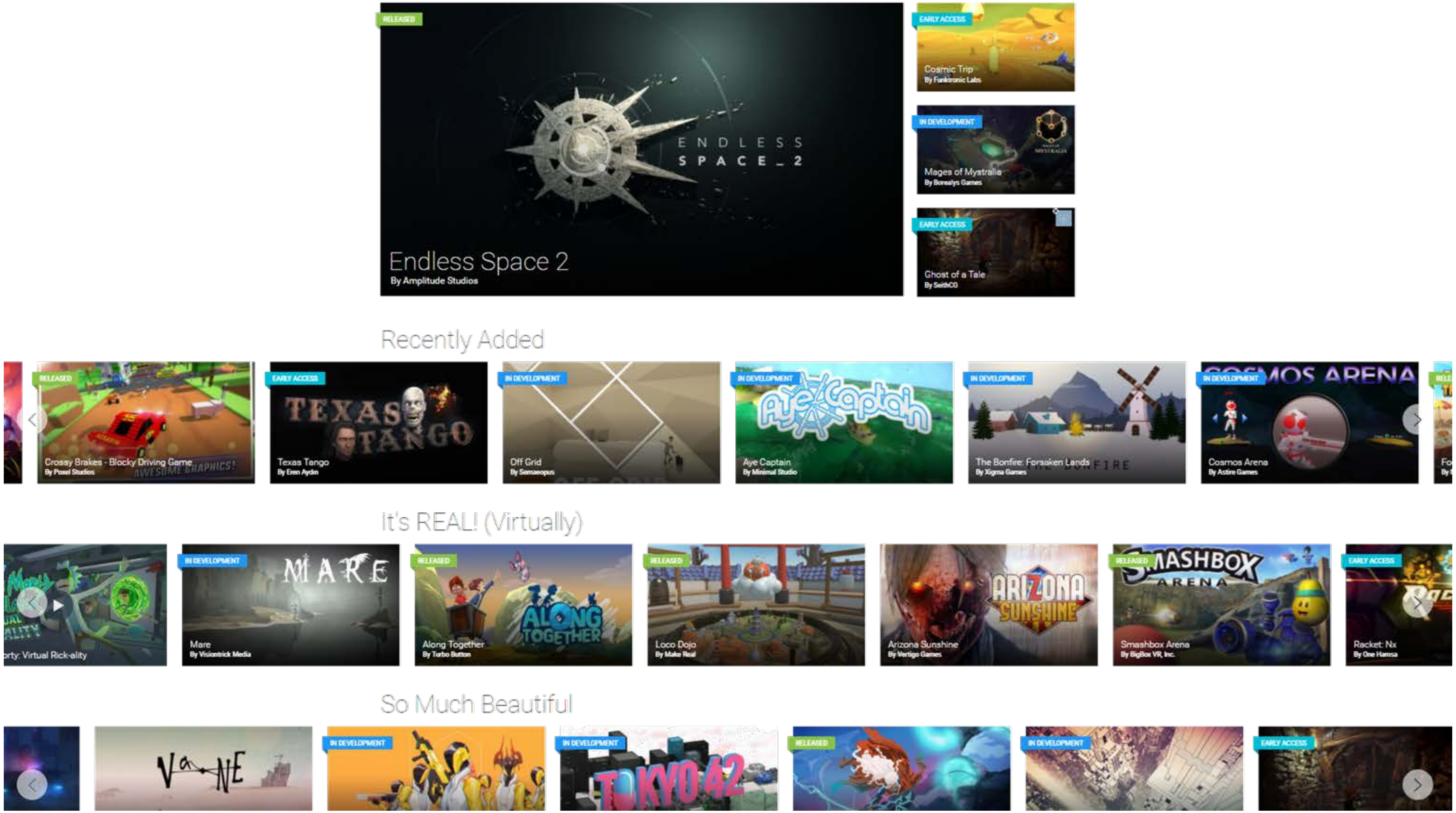
[blackshellmedia.com/2016/09/29/6-crucial-questions-ask-choosing-game-engine/]

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

[http://www.learn-cocos2d.com/2012/05/the-game-engine-dating-guide-how-to-find-the-right-engine-for-your-game]

Unity


RELEASED
Endless Space 2
By Amplitude Studios
EARLY ACCESS
Cosmic Trip
By Funktronic Labs
IN DEVELOPMENT
Mages of Mystralia
By Borealys Games
EARLY ACCESS
Ghost of a Tale
By SeithCG
Recently Added
Crossy Brakes - Blocky Driving Game
By Pixel Studios
EARLY ACCESS
Texas Tango
By Eren Aydın
IN DEVELOPMENT
Off Grid
By Semaeopus
IN DEVELOPMENT
Aye Captain
By Minimal Studio
IN DEVELOPMENT
The Bonfire: Forsaken Lands
By Xigma Games
IN DEVELOPMENT
Cosmos Arena
By Astire Games
It's REAL! (Virtually)
rty: Virtual Rick-ality
IN DEVELOPMENT
Mare
By Visiontrick Media
RELEASED
Along Together
By Turbo Button
RELEASED
Loco Dojo
By Make Real
Arizona Sunshine
By Vertigo Games
RELEASED
Smashbox Arena
By BigBox VR, Inc.
EARLY ACCESS
Racket: Nx
By One Hamsa
So Much Beautiful
IN DEVELOPMENT
IN DEVELOPMENT
RELEASED
IN DEVELOPMENT
EARLY ACCESS

GAME WORLD, GAME OBJECTS & EDITION TOOLS

Game Objects

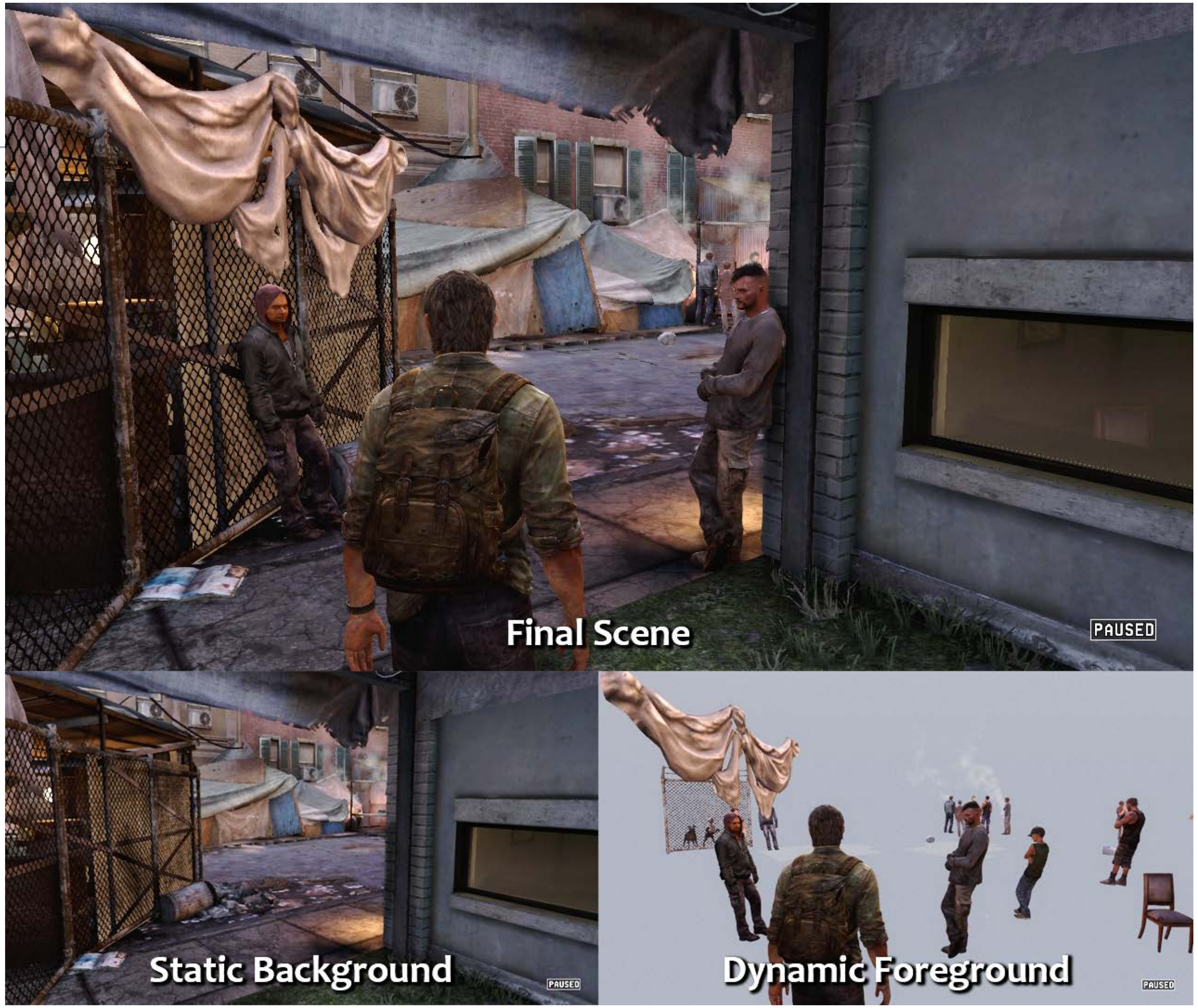
Actors, agents, entities…

Components of the game world

- Player & non-player characters
- Environment
  - Terrain, building, road, bridge, trees…
- Locomotion modes
  - Vehicles, platforms, ropes, graspable edges…
- Scenery and ambiance objects
  - Background, furniture, particle emitters, lights…
- Items
  - Weaponry, armor, collectible objects, floating power-ups and health packs…
- Invisible utilitarian data
  - Collision information, volumetric regions to detect events or delineate areas, AI navigation mesh, splines to define the paths of objects…

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

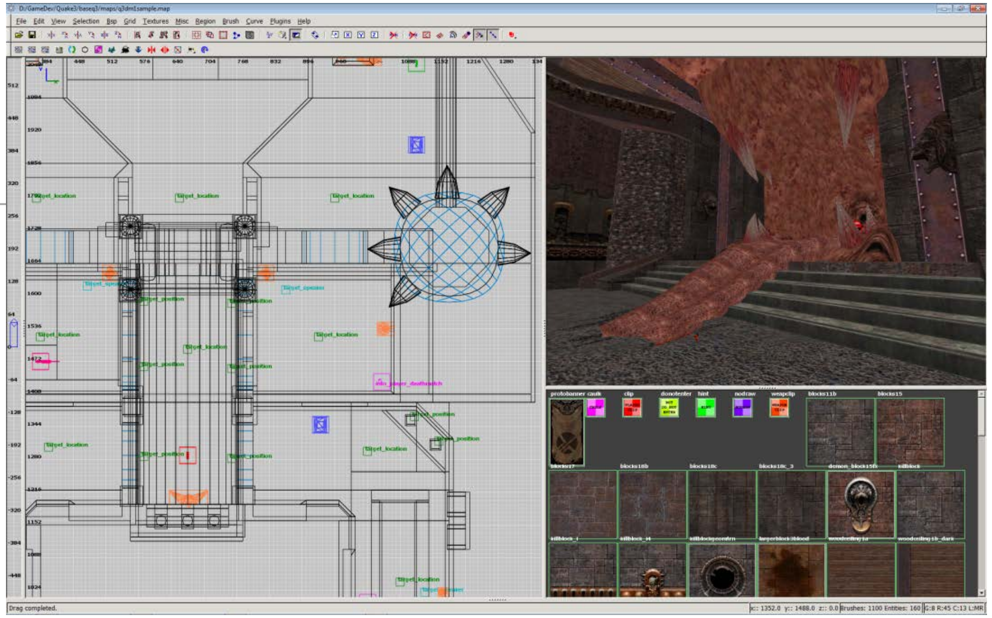
Dynamic vs. Static Objects



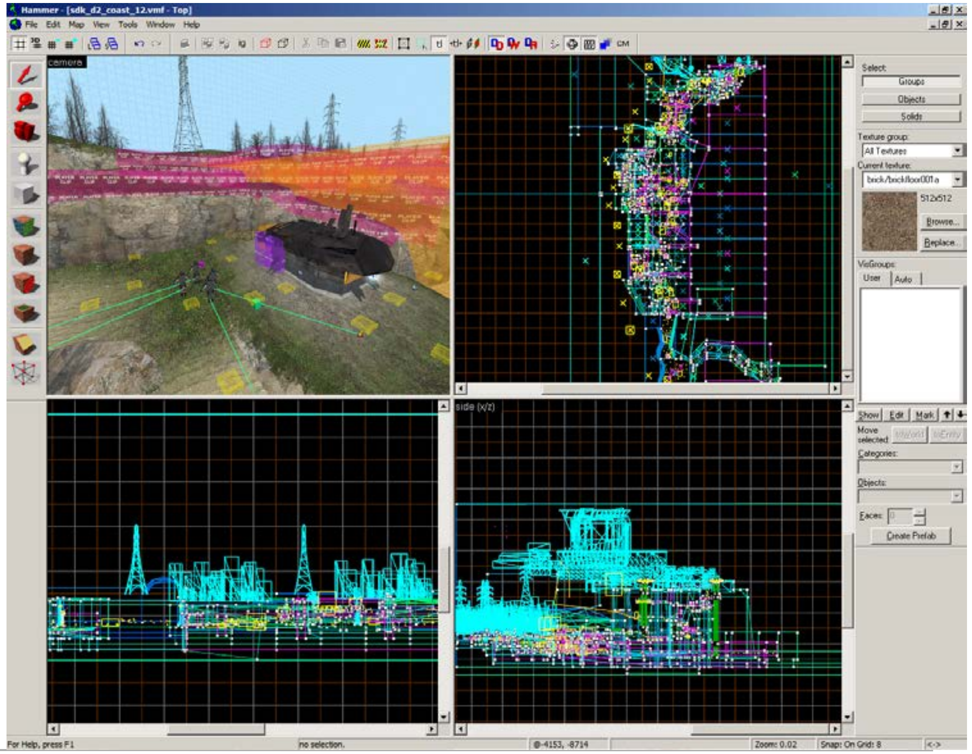
- "Dynamic" objects
  - Evolving state
  - Main support of the gameplay
  - Usually more CPU expensive
- "Static" objects
  - Stable state
  - No critical interaction with gameplay (event if layout can plays a crucial role)
  - Possible optimizations (static triangle mesh, precomputed lighting…)
- Dynamic/Static ratio
  - Distinction often blurry
    - *Ex. waterfalls, destructible elements*
  - High ratio => perception of a more "alive" and interactive game world
  - Most games consist of a limited number of dynamic elements within a relatively large static background area (hardware dependent)

Game World Editor

- GUI tool (or suite of tools) 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)

World Edition

- Insertion and selection of game objects
  - Placement and alignment aids (position, orientation, and scale via special handles, assistance tools for densely populated worlds)
  - 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
- Level creation, saving, loading and management

Game Objects Edition

Game objects usually have an object-oriented appearance

- Types/Instances
- Attributes/Values
  - Current state of the object (locations, orientations, parameters…)
- Behavior
  - How the state will change over time and in response to events
- 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: Pacman ?

Game Objects?

Game Objects Edition

- States of game objects (values of their attributes) edited in a property grid
  - Atomic data types
  - Key-value pairs
  - Arrays
  - Structures
  - Strings…
- Behavior usually controlled with
  - Data-driven configuration parameters
  - Scripting language

Assets

All the ressources for the game

- 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…

Need management tools

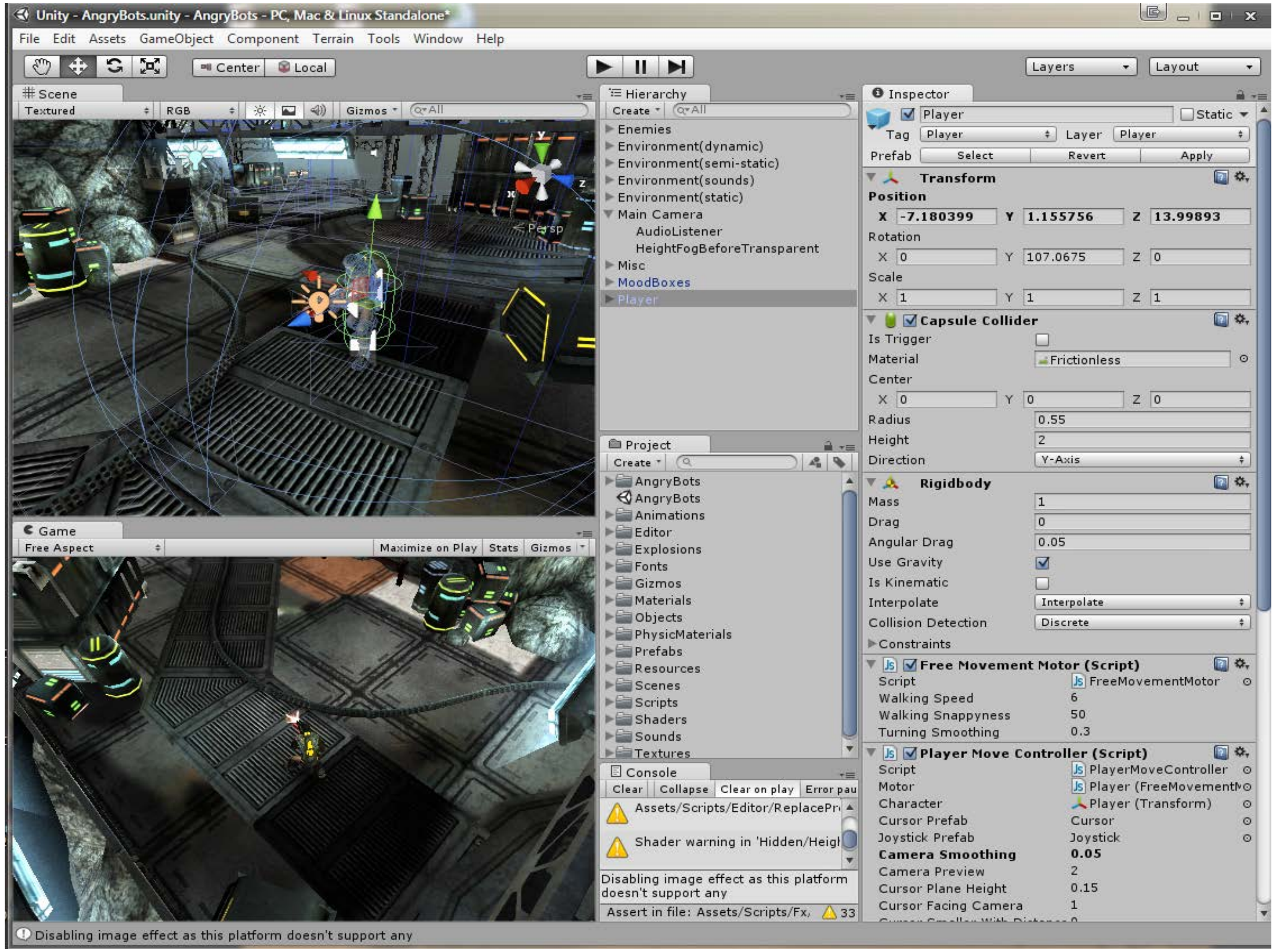
Unreal Editor browser

Assets Tools

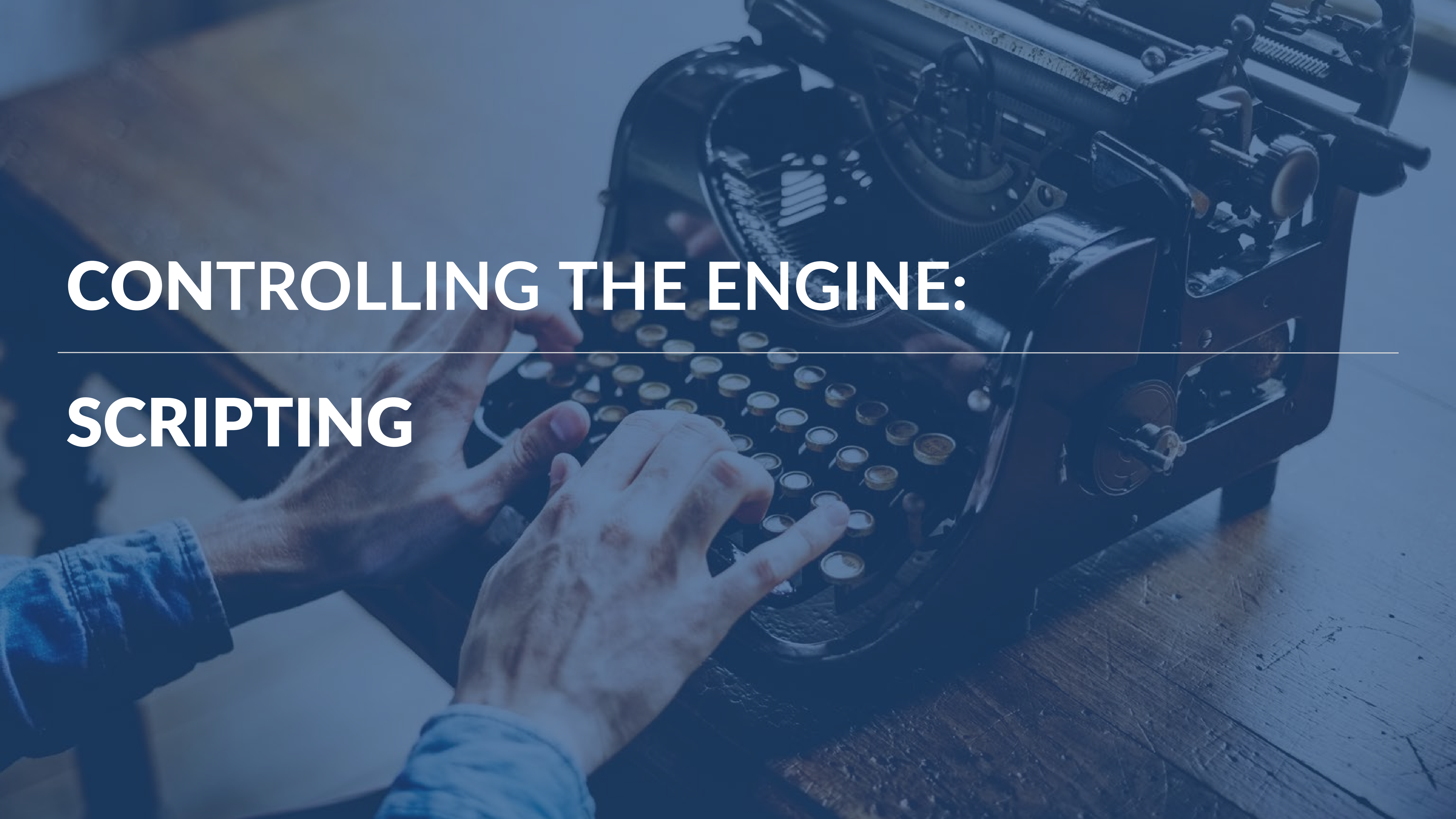
- Data formats of created assets rarely suitable for direct use in-game
  - In-memory model too much complex
  - File format too slow to read at runtime, and sometimes proprietary
- => Asset Conditioning Pipeline (ACP)
  - Data exported to a more accessible standardized or custom format, then further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

Unity

- Editor / Runtime
- Rapid iteration
- 3D views
- Scenes
- Game Objects, Object-oriented
- Hierarchy
- Inspector
- Assets,
  Management tools
- Layers…



Assets?



CONTROLLING THE ENGINE:

SCRIPTING

Game Scripting Languages

- High-level, relatively easy-to-use programming languages
- Provides convenient access to commonly used features of the engine
- Used by programmers and non-programmers to
  - Develop a new game
  - Customize (“mod”) an existing game
  - Extend or customize the hard-coded functionality of the engine’s game object model and other subsystems
  - Create and populate data structures that are later consumed by the engine (data definition languages)
- *Examples*
  - *QuakeC, UnrealScript, LUA, Python, Pawn / Small / Small-C*

Scripts Benefits

- Rapid iteration: faster to see in-game effects of changes than native language source code
  - No recompilation/relink
  - Sometimes no game shut down and rerun
- Convenience and ease of use, often customized
  - Suits the needs of a particular game
  - Can make common tasks simple and less error-prone

Scripts Execution

- Usually interpreted and executed in an embedded virtual machine at runtime within the context of the engine
- Flexibility
  - Game engine manages scripts' execution
- Portability
  - Platform-independent byte code treated like data by the engine and loaded into memory just like any asset
- Lightweight
  - Simple virtual machines, small memory footprints
- Callbacks
  - User-supplied function called by the engine to customize some functionalities

In Practice

- New game object types or components
  - Inheritance: deriving a scripted class from a native class
  - Composition/aggregation: attaching an instance of a scripted class to a native game object
- Communication between objects
- Entirely scripted game object model
  - Native engine code called only when requires the services of lower-level engine components
- Entirely scripted game
  - Native engine code called to access high-speed features of the engine

Unity

Scripting system

- http://docs.unity3d.com/ScriptReference/
- http://unity3d.com/learn/tutorials/modules/beginner/scripting

PART 2

INTERACTIVE REAL-TIME SIMULATION

GAME LOOP

The 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 servicing 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 (not necessarily synch. with rendering)

⇒ Solution: a single “game 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

```
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 !

TIME MANAGEMENT

Frame rate

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

Frame rate

- Number of game loop renderings / second (Hz or FramePerSecond)
- Describes how rapidly the sequence of still 3D frames is presented to the viewer

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

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

Frame rate

Depends on the complexity of calculating each frame and the power of the underlying platform

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

Ex: move x meters per frame

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

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 = CPU last powered on or reset
  - Counts the units of elapsed CPU cycles (or some multiple thereof)
  - Converted into units of 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 engine and debug flythrough 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

- Ex: move (x * elapsed time) meters per frame

```
double lastTime = getCurrentTime(); //CPU's high resolution 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 upcoming frame $(k + 1)$

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

Undeterminism

- Basic physics will have different behavior based on the frame rate (numeric integration / rounding error)
- Online multiplayer will not function properly with variable simulation 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 softening the effects of momentary performance spikes
  - Long averaging interval => less responsive to varying frame rate + less spikes impact

Breakpoints issue

Game loop stops running but not CPU nor real-time clock

=> A measured frame time of several seconds or even minutes

Simple solution: compare Δt to predefined upper limit and set Δt 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

- Works when game's frame rate is reasonably close to target frame rate on average
  - "Variable frame rate" mode during development
  - Switch on frame-rate governing when the game is close to consistent frame rate
- Consistent frame rate is important for
  - Physics
  - Graphics
  - Record and playback
  - Power consumption

Callback-Driven Frameworks

Game loop exists but is largely empty

=>Write callback functions to complete it

Ex: Ogre3D

```
while (true){
    for (each frameListener)
        frameListener.frameStarted();
    renderCurrentScene();
    for (each frameListener)
        frameListener.frameEnded();
    finalizeSceneAndSwapBuffers();
}
[cf.Ogre::Root::renderOneFrame() in OgreRoot.cpp]
```

Callback-Driven Frameworks

Derive a class from `Ogre::FrameListener`

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

called before and after the rendering of the main 3D scene

```
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.
    }
};
```

Unity

- Callback-driven framework
  - Game parts already implemented: game loop, rendering…
  - customizable functions called during the game loop (`Start()`, `Update()`…)

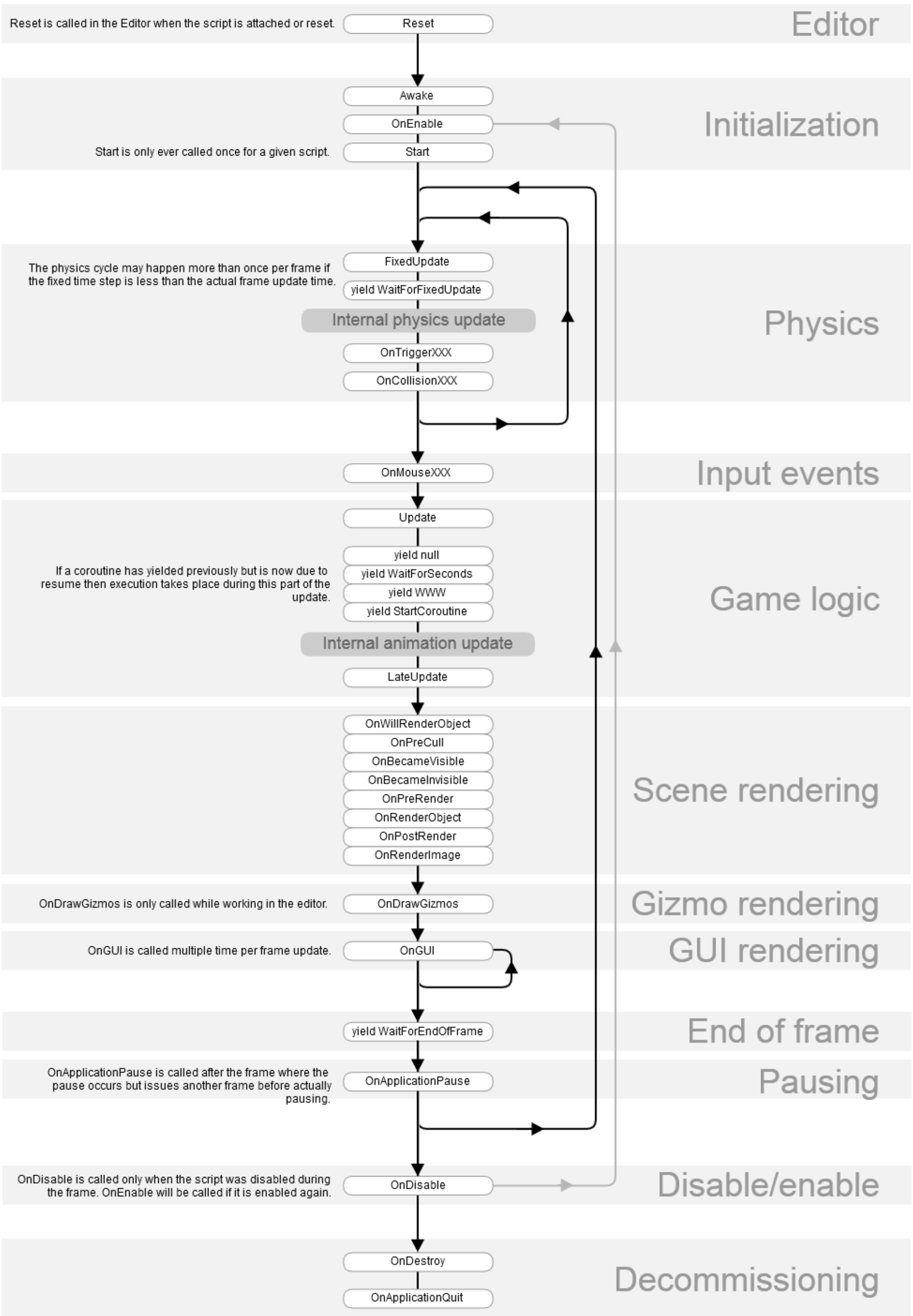
Unity

https://unity3d.com/learn/tutorials/modules/beginner/scripting/update-and-fixedupdate

http://www.codeproject.com/Tips/761922/Unity-and-Csharp-Game-Loop-Awake-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

Further Readings

http://gameprogrammingpatterns.com/game-loop.html

http://www.koonsolo.com/news/dewitters-gameloop/

http://gafferongames.com/game-physics/fix-your-timestep/

http://obviam.net/index.php/the-android-game-loop/

http://entropyinteractive.com/2011/02/game-engine-design-the-game-loop/

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/

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

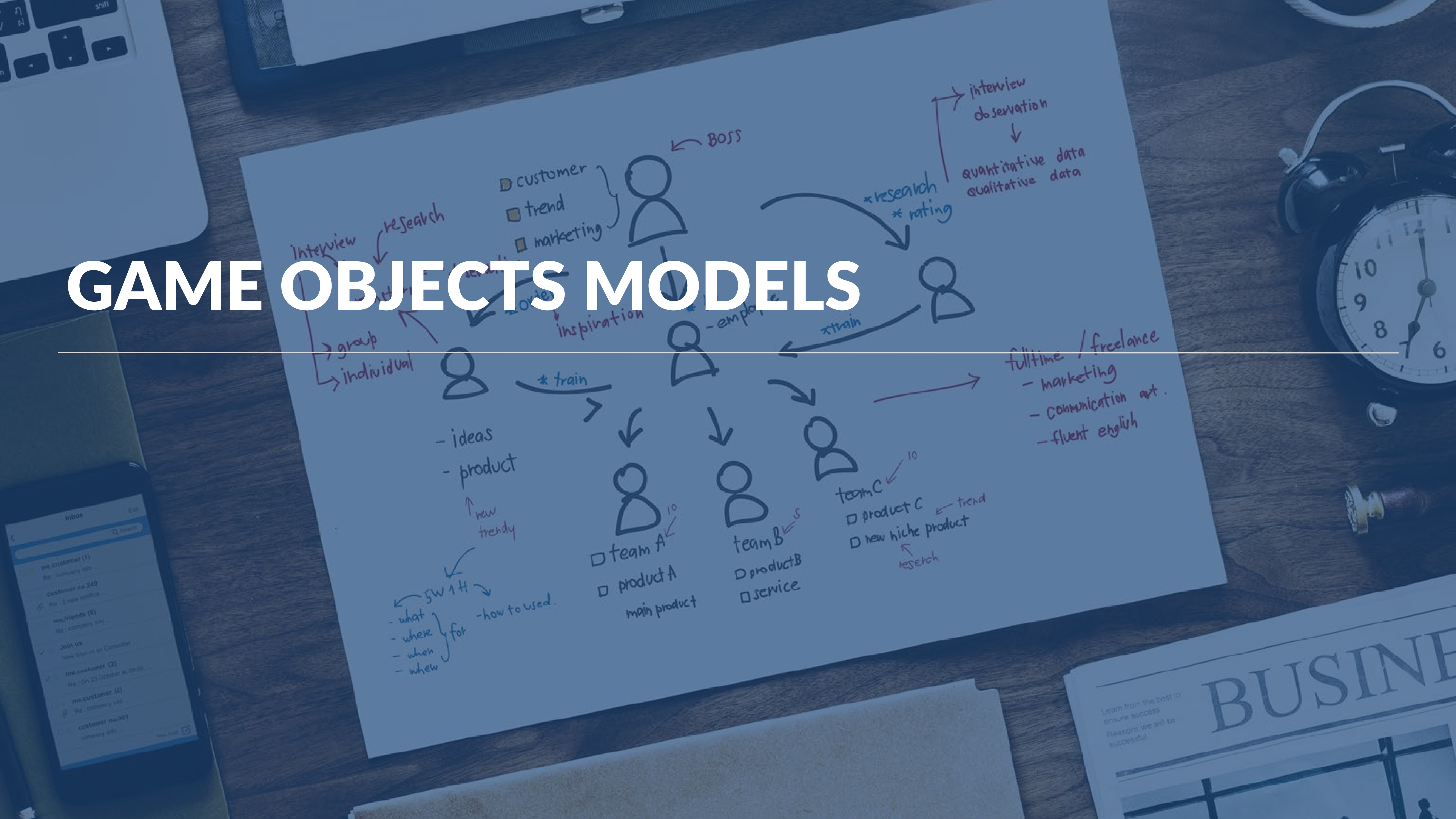
- NB: Try to keep inputs/events handling separated from the game logic

Unity

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

SCHMUP !

GAME OBJECTS MODELS



“Runtime Game Object Model”

- Concrete implementation of the collection of tool-side game objects available in the world editor (type, attributes/values, behaviors)
- Must be flexible enough to easily define new game object types (data-driven or programmed)
- A single tool-side game object type might be implemented at runtime 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

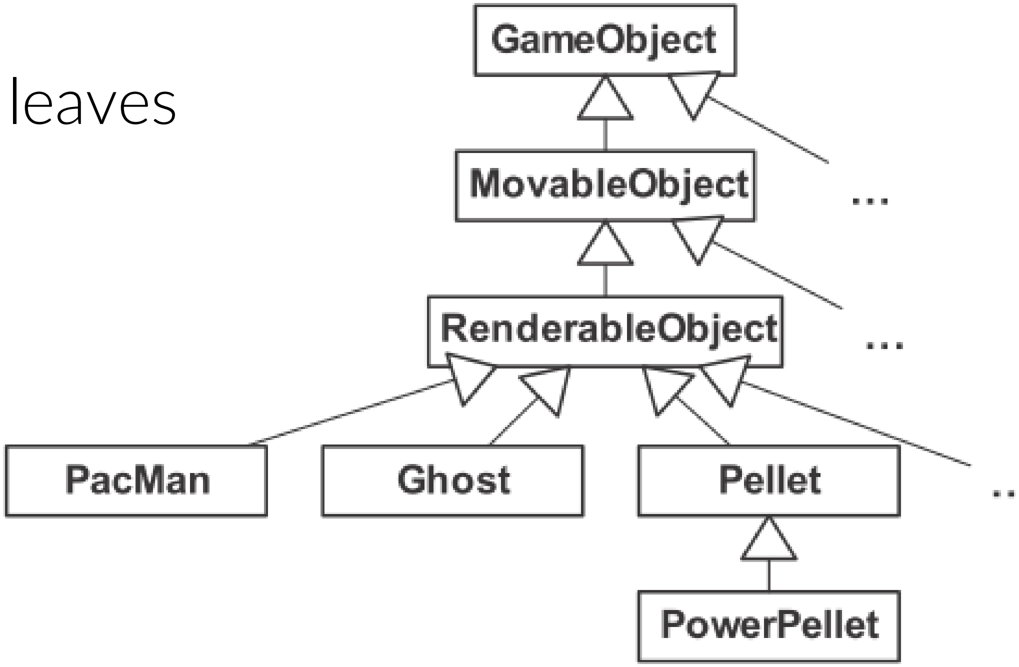


Figure 14.2. A hypothetical class hierarchy for the game Pac-Man.

SCHMUP !

Problems with Deep Hierarchies

Understanding, maintaining, and modifying classes

Need to understand all parents (ex: assumptions about virtual functions)

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 class hierarchies

=> Mix-in classes

Stand-alone classes with no base class

Multiple inheritance limited to 1 grand-parent: and any number of mix-in classes.

The bubble-up effect

Factorization vs. Duplication of code

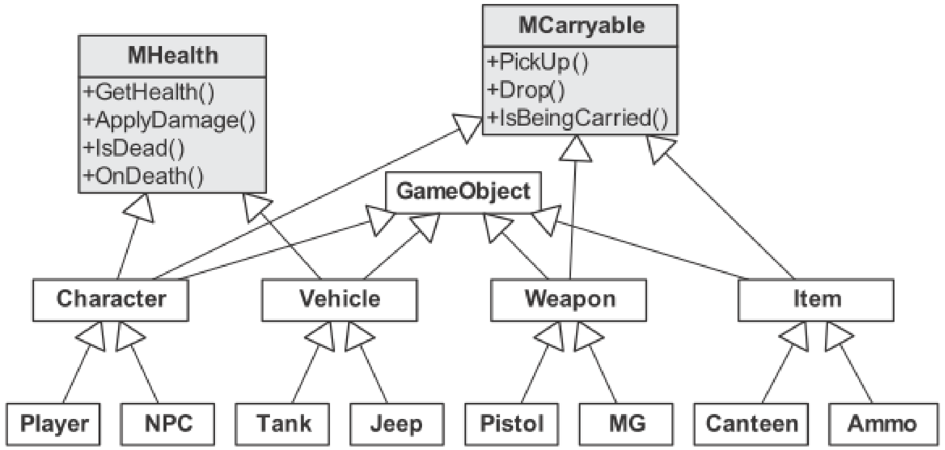


Figure 14.6. A class hierarchy with mix-in classes. The MHealth mix-in class adds the notion of health and the ability to be killed to any class that inherits it. The MCarryable mix-in class allows an object that inherits it to be carried by a Character.

SCHMUP !

Static destructible enemy turret ?

Special player invicible bullet ?

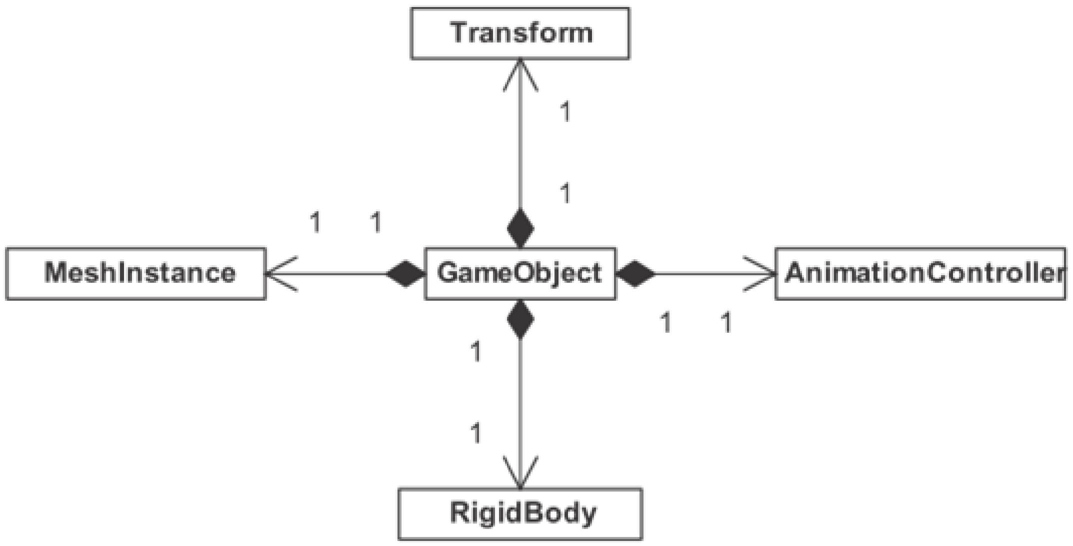
Components

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



Generic Components

Arbitrary number of instances of each type of component

ex. linked list in the root GameObject

Can iterate polymorphic operations

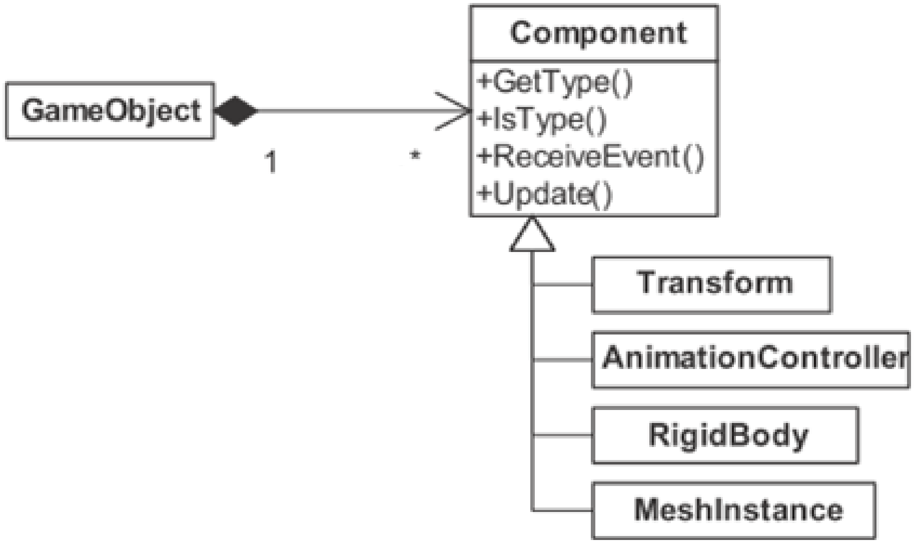
ex. update

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

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

Components can also have a hierarchy

ex. Input -> PlayerInput & AIInput



Unity

Generic components :

transform, renderer, collider…

Behaviours, Monobehaviours

SCHMUP !

GAME OBJECTS & UPDATE

A Part of the Game Loop

- 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 rather than continuous
- 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 **`virtual void Update(float deltatime)`** on each object once per frame of the main loop
- Custom implementations of **`Update(dt)`** for each game object type 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

`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

Performance constraints of low-level engine systems

- Large quantity of data and large number of calculations every frame as quickly as possible

=> Benefits from batched updating

- Minimal duplication of computations: global calculations done once and reused for many game objects rather than being redone for each object
  - Ex: collisions depend on multiple objects by nature
- Reduced reallocation of resources: allocated once per frame and reused for all objects
- Maximal cache coherency: per-object data arranged in a single 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 on its behalf
- Ex: the game object controls how it is rendered by manipulating the properties of the mesh instance object, but does not control the rendering of the mesh instance directly

Batched Updates

```
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 Object's Update

```
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

Phased updates

Game objects/engine subsystems can depend on one another: updates order is crucial

=> Engine subsystem updates in the proper order within the main game loop

=> Update the states of the game objects at the right time during the game loop

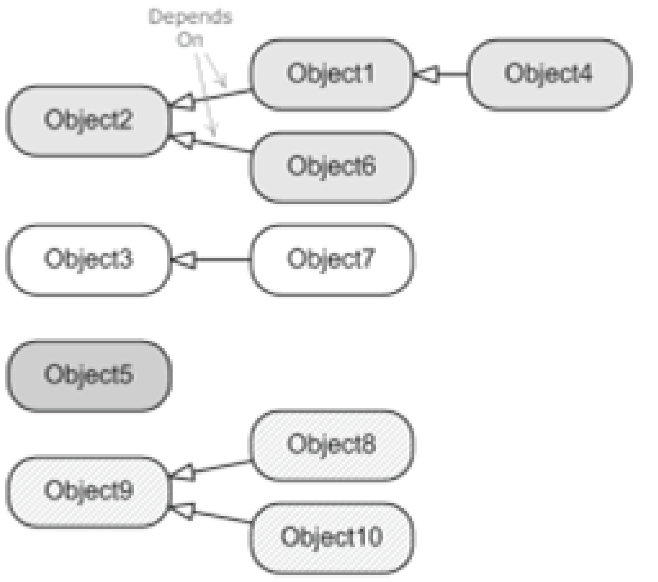
- May be updated multiple times during the frame if it depends on intermediate results of calculations
- Not all game objects require all update phases
- To minimize the cost of iteration, can maintain multiple linked lists of game objects (one for each update phase)

Bucketed updates

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

=> 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 and One-Frame-Off Lag

- Objects are not updated from $t1$ to $t2$ instantaneously and in parallel but 1-by-1
- The states are consistent before and after the update loop, but may be inconsistent during it
- 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

Unity

Update pattern : custom **`Update()`** functions associated to game objects

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

Goto Dev...

Project Setup

SCHMUP !

- Gather and organize the assets
- Build the game world and set up the objects

Player (resp. Enemy)

- Prefab composed of
  - Sprite renderer
  - Collider + RigidBody2D
  - PlayerAvatar <- BaseAvatar
    - maxSpeed
    - health
    - energy
    - …
  - Engines
  - InputController
  - BulletGun(s)

Inputs

- 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

InputController component

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

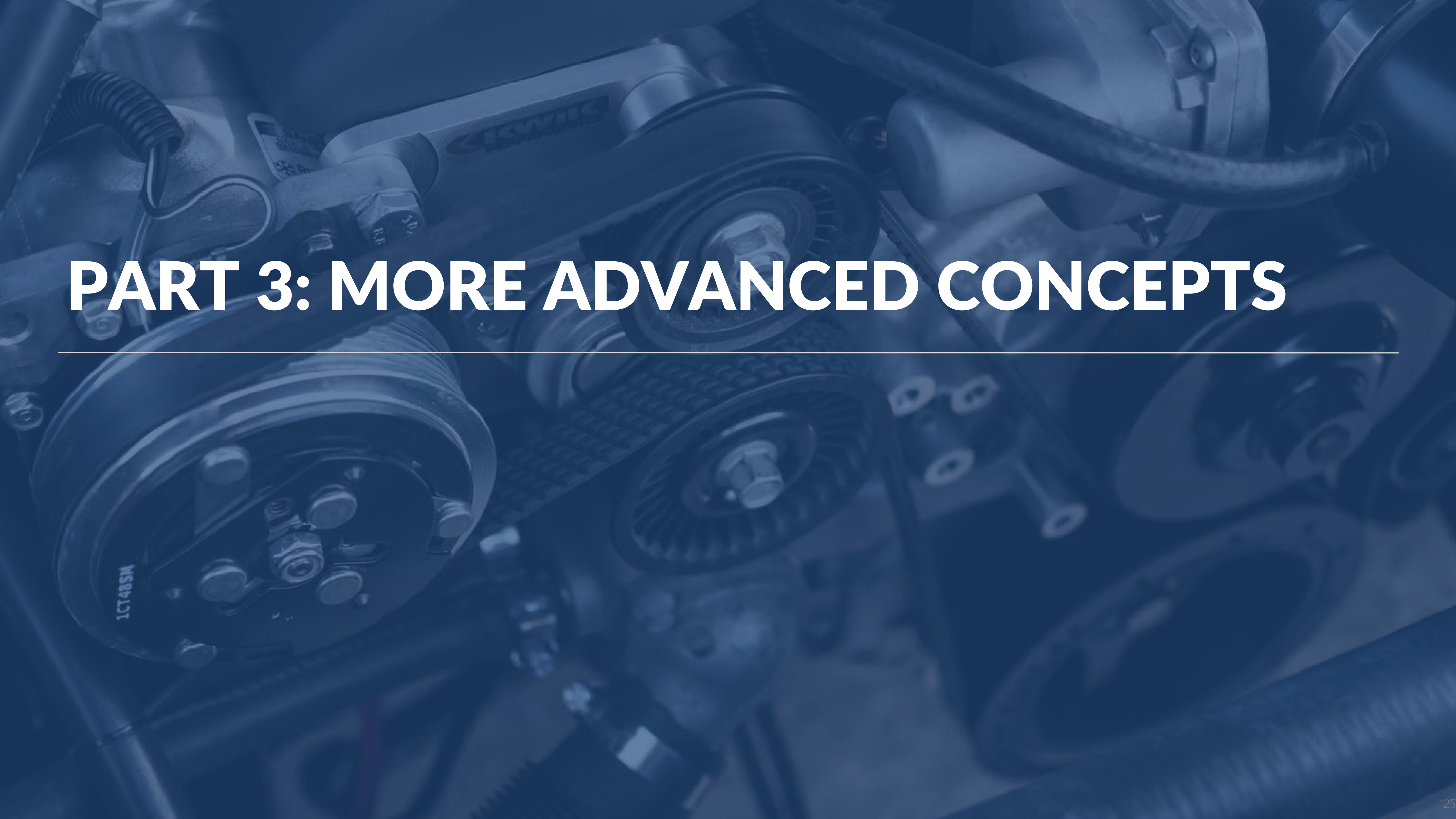
Engines component

- 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

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



PART 3: MORE ADVANCED CONCEPTS

DEBUGGING & PROFILING

Errors

Error return codes

- `bool`, impossible value, `enum`…
- Pb for transmission threw the call stack

Exceptions

- Avoid on consoles: limited memory and processing bandwidth

Assertions

- Checks an expression (i.e. assumptions): if false stops the program
- Performance cost
- Only for fatal errors,
  never for user errors

```
#if ASSERTIONS_ENABLED
// check the expression and fail if it is false
#define ASSERT(expr) \
   if (expr) { } \
   else { \
    reportAssertionFailure(#expr, __FILE__, __LINE__); \
    debugBreak(); }
#else
#define ASSERT(expr)  // evaluates to nothing
#endif
```

Logging and Tracing

- "printf debugging"
- Formatted output
  - Ex. custom `OutputDebugString()`
- Level of verbosity, channels, filters
- Mirroring output to log files
  - Cost: flush buffers after every output
- Crash Reports
  - Gather useful information: level, player location, animation state, running scripts, stack trace, memory allocators states…
  - Top-level exception handler
  - E-mail

Debug Facilities

- Debug cameras
- Pause and slow motion
  - Game's logical clock != real-time clock
- 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 arbitrary functions
- In-game console
  - Command-line interface to the engine's features
  - Hard-coded commands, rich interface or scripts

Profiling

"90/10 rule“

- 90% of the time spent running any software is accounted for by only 10% of the code
- => Optimizing 10% of the code can potentially realize 90% of all the gains in execution speed

=>Measure the execution time

- How much time is spent in each function, how many times each function is called, call graph, percentage of the function’s time spent calling each of its descendants, percentage of the overall running time accounted for by each individual function…

Ex. 3rd party profilers

- *Vtune (Intel), Rational Quantify (IBM)*

Memory-Tracking

Stats

Leak = out-of-memory

- Memory allocated but not freed

Corruption = data written on wrong memory location

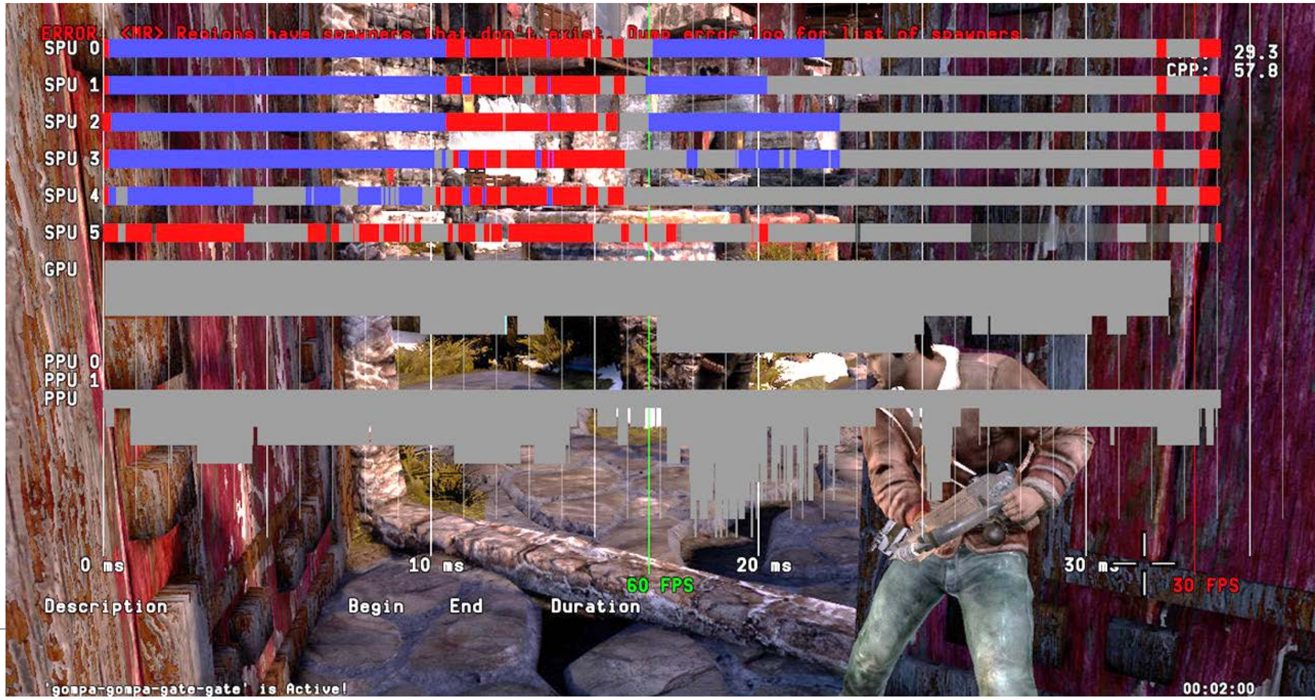
- Other data overwritten
- Right location not updated

Main cause = pointers

Custom or 3rd-party tools

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

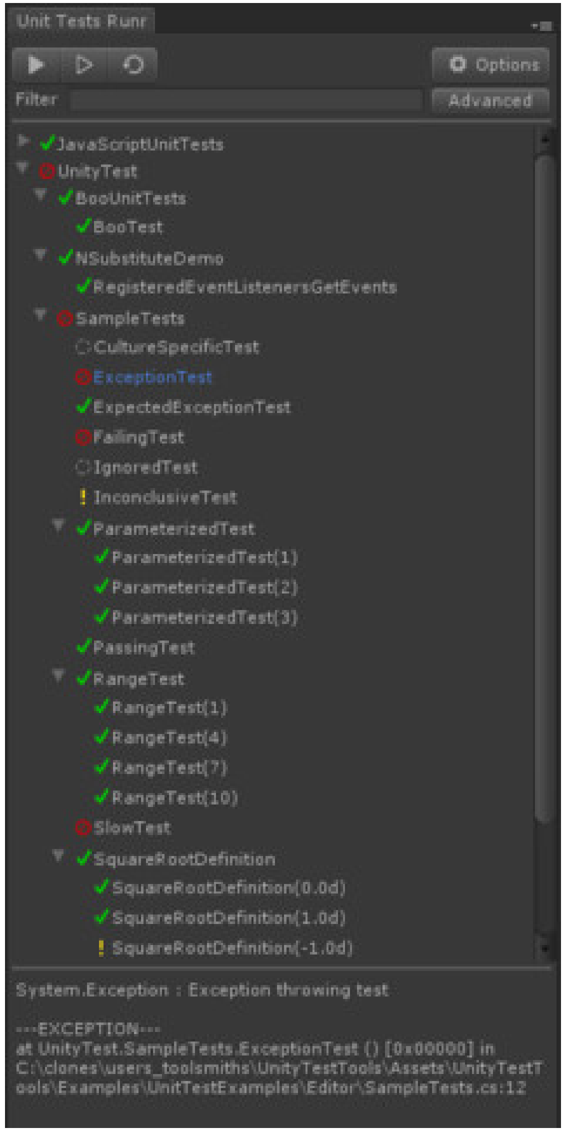
Uncharted 4

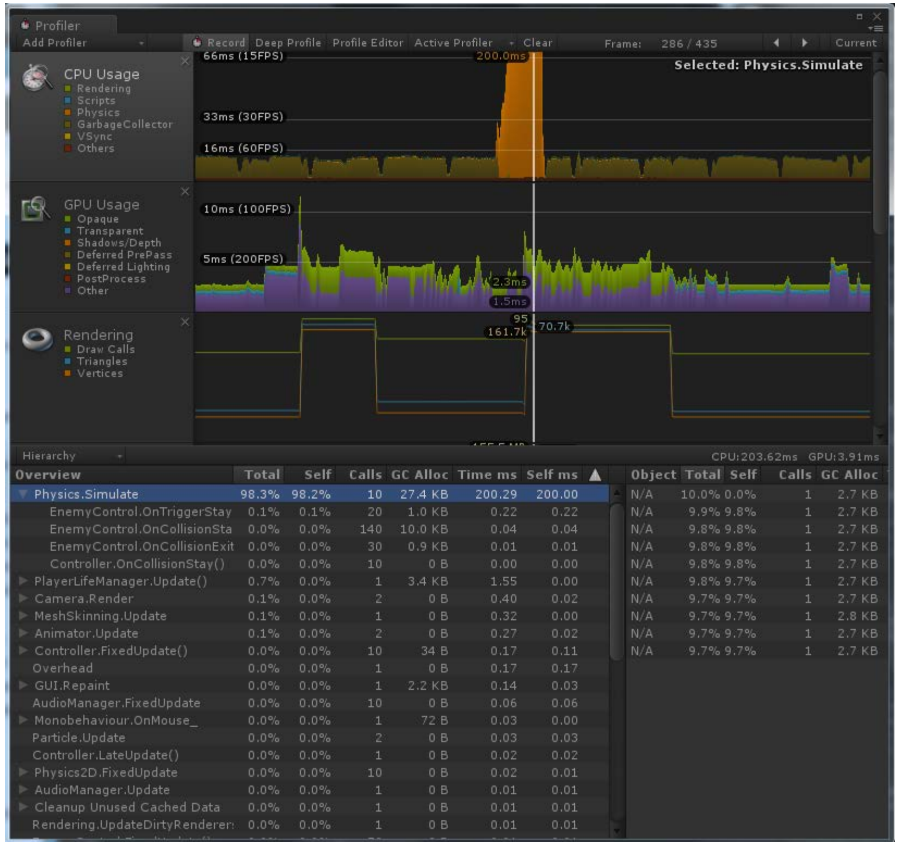


Uncharted 2

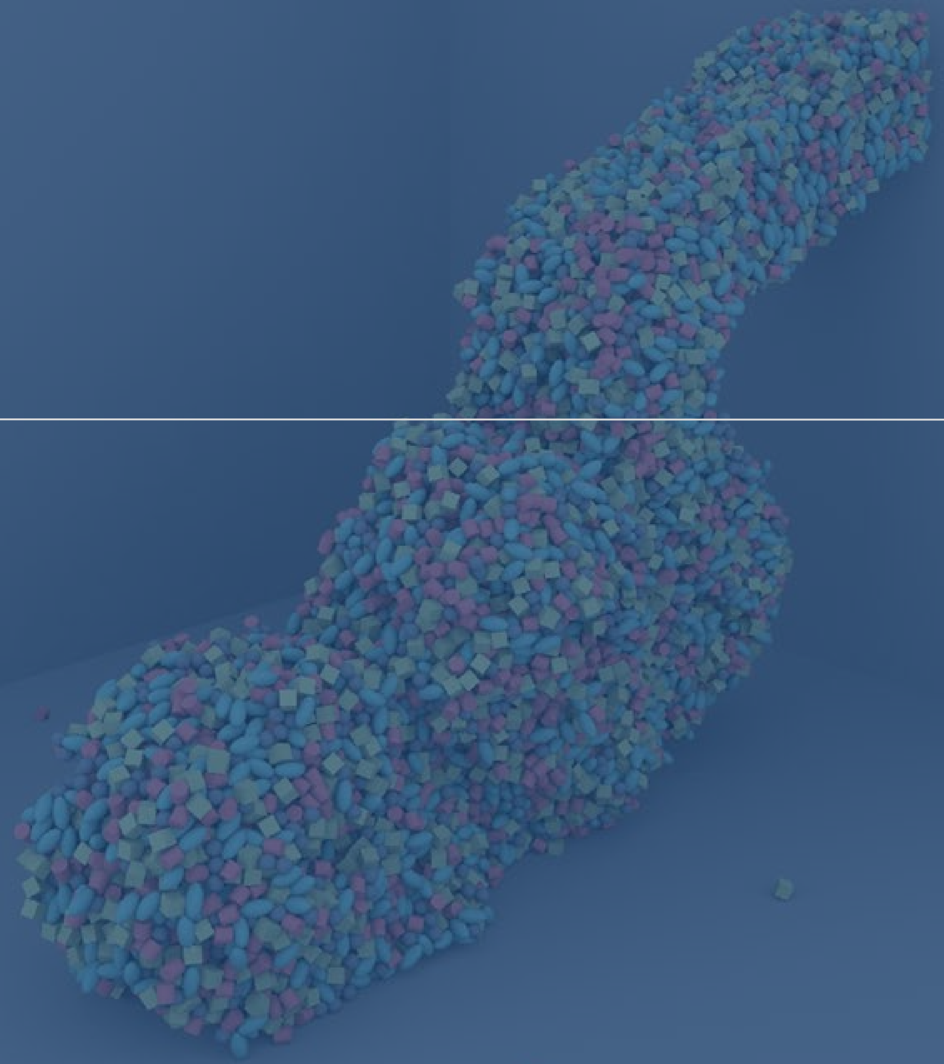
Unity

- IDE
- Console
  - print(), Debug.Log(), Debug.Draw()
- Debugger
- Profiler
- Unit tests
- Assert system (>=5.1)
- 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
- Emergent behaviors: 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

Basis

- Simulating the motions of game objects over time
- Classical (newtonian) mechanics
- Rigid bodies: solid and undeformable
- Ensure conformity to constraints: ex. non-penetration (collision response), joints…

Equations of motion for linear dynamics

- $\boldsymbol{v}(t) = {}^{d\boldsymbol{r}(t)}/_{dt}$ $\quad \boldsymbol{a}(t) = {}^{d\boldsymbol{v}(t)}/_{dt}$ $\quad \boldsymbol{F}(t) = {}^{d(m\boldsymbol{v}(t))}/_{dt} = m\boldsymbol{a}(t)$
- Solving = finding $\boldsymbol{v}(\boldsymbol{t})$ and $\boldsymbol{r}(\boldsymbol{t})$ given knowledge of the net force $\boldsymbol{F}(\boldsymbol{t})$ and the previous position and velocity
- Analytical solutions almost impossible in games

Numerical integration: not exact but stable

- Time-stepped: finding $\boldsymbol{r}$, $\boldsymbol{v}$ et $\boldsymbol{F}$ for $\boldsymbol{t2} = \boldsymbol{F(t1)}$
- Explicit euler
- $\boldsymbol{r}(t2) = \boldsymbol{r}(t1) + \boldsymbol{v}(t1).\Delta t \;\; \boldsymbol{v}(t2) = \boldsymbol{v}(t1) + {}^{\boldsymbol{F}(t1)}/_{m}.\Delta t = {}^{\boldsymbol{r}(t2)-\boldsymbol{r}(t1)}/_{\Delta t}$

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

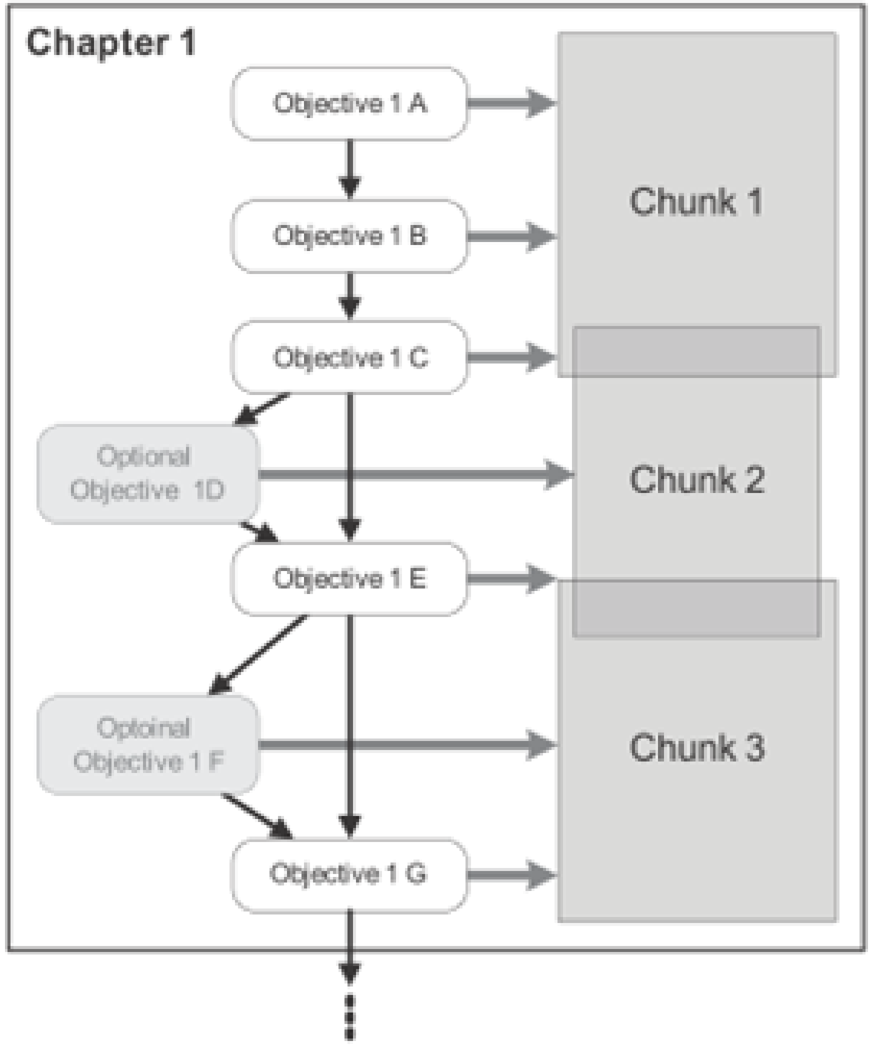
- Linear progression
- Star topology
  - Central hub area
  - Access other areas at random from the hub (sometimes locked)
- Graph-like topology
  - Areas connected to one another in arbitrary ways
- Illusion of a vast, open world

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



Flow & Finite State Machines

- List of states and transitions triggered by conditions
- Each state
  - Represents a single player objective or encounter
  - Is associated with a particular location within the virtual game world
- When the player completes a task
  - The state machine advances to the next state
  - The player is presented with a new set of goals
- When the player fails to complete a task
  - The state machine advances to the corresponding state
  - Ex : send the player back to the beginning of the current state, send back 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

- Principle: store all objects in data file
  - Type
    - String, hashed string id, other unique type id…
  - Initial state (values of attributes)
    - Different formats
- Sol. 1: Binary image of each object
  - Trivial spawning
  - Problematic storing
  - Problematic for changes
    - suitable for stable data structures: mesh data, collision geometry…

Chunks Data

Sol. 2: Serialized Game Object Descriptions

- Writing/reading stream of data that contains enough detail to permit the original object to be reconstructed later
- Supported natively by some programming languages
- Stored in a more-convenient and more-portable format (ex: XML or proprietary)
- Slow to parse
- Size problem
- **`serializeIO()`** customized functions vs reflection and generic serialization system

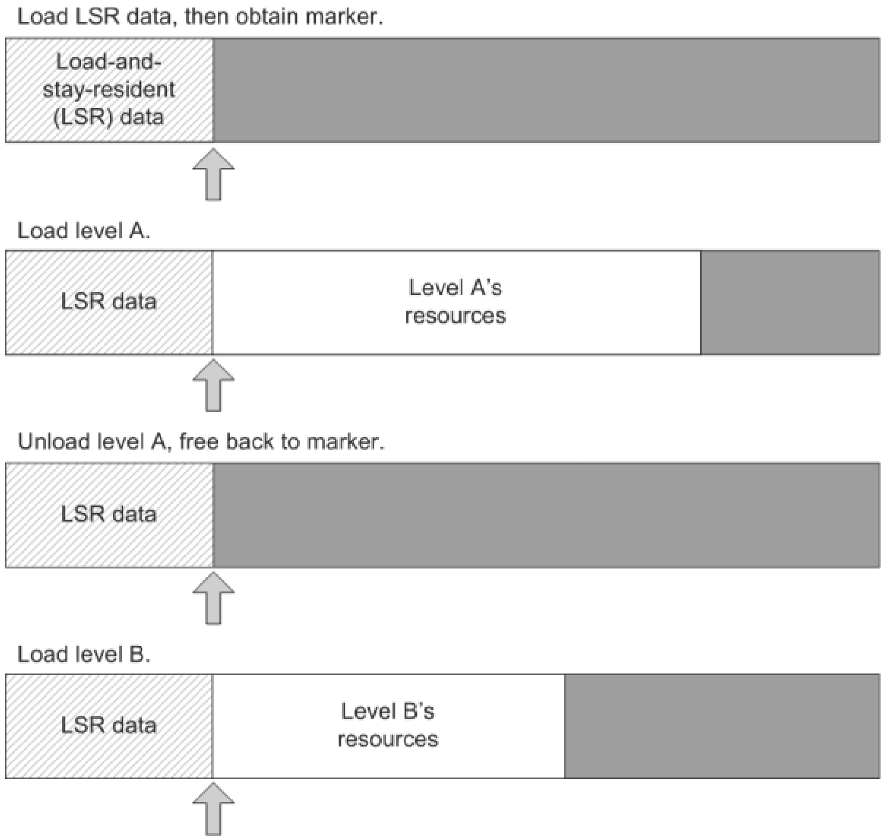
Level Loading

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: required across all game levels
  - Level loaded on top of LSR data
  - When complete, memory freed by resetting the stack pointer
- 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



Streaming

Main goals

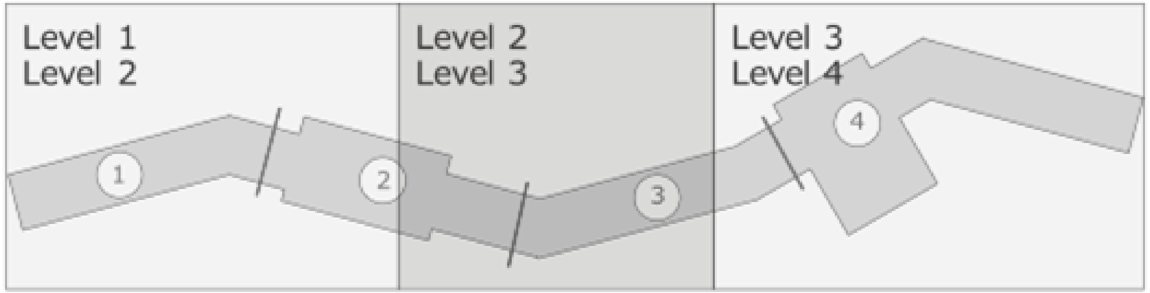
- Load data while the player is engaged in regular gameplay tasks
- Manage the memory without fragmentation while permitting data to be loaded and unloaded as needed as the player progresses

Divide memory in *n* parts

- Restrictions on the size of each chunk

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?



Object Spawning

- Off-line memory allocation
  - = Disallowing dynamic memory allocation during gameplay after world chunks loading: no game objects can be created or destroyed
  - Game's memory usage patterns highly predictable
  - Limits game design, have to predict the total needed number of game objects of each type
- Dynamic memory management
  - Can be slow
  - Can cause memory fragmentation, leading to premature out-of-memory conditions (because game objects have various sizes)
  - No global pool allocator (different types and sizes of objects)
  - No stack-based allocator (deallocation order)
  - Fragmentation-prone heap allocator examples
    - One memory pool per object type
    - Small memory allocators
    - Memory relocation

Spawners

Lightweight, data-only representation of a 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
- Loosely coupled to the engine's implementation
- 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

- ~~Application~~ (<5.3)
- UnityEngine.SceneManagement (>=5.3)
- SceneManager
  - LoadScene()
  - GetActiveScene()
- Object.DontDestroyOnLoad

Object Spawning

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

SCHMUP !

GAME OBJECTS COMMUNICATION:

EVENTS SYSTEM

Components Communication

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

Events and Communication

- Games are inherently event-driven
- Event = any interesting change in the state of the game or its environment
  - *Ex: player pressing a button, explosion, player sighted by an enemy, item picked up…*

Explicit Function Calls vs. Event System

- Coupled vs loosely coupled components
- Examples
  - Player hits monster
    - -> monster's health component
      - -> Monster death event -> ...
    - -> monster's animation
    - -> UI (damages)
    - -> sound
    - ...
  - Achievement system triggered by different aspects of gameplay
  - Game objects creation and destruction

Explicit Function Calls

- Call a (virtual) method on the object to notify that an event has occurred
- Inflexibility:
  - Requires to know all the game objects / components involved
  - May require that all game objects inherit from a common base class which declare the virtual functions for all possible events

```
for (each object o in explosion) o.OnExplosion()
```

  - Incompatible with data-driven additions
  - Every object know about every possible event

Explicit Function Calls

- In practice
  - Lots of includes to know the public methods of every other systems implied
  - Whole recompilation
  - Hard to find bugs
  - Hard to modify

Event System

- Global to the game: manages all communications
- 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 pattern

- Sender knows a list of receivers (but is not coupled to them) and the event to notify, and calls their notification virtual handler
- Receivers register to the sender and handle the notification
- Synchronous communication: direct call of the notification => wait for the return (avoid for UI)

Events as Objects

- Informing objects about an event is equivalent to sending a message or command (command pattern)
- Event type
  - Hierarchy possible
  - *Ex: explosion, friend injured, player spotted, item picked up…*
- 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…*

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

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 handling at a later time, 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
  - Extreme flexibility and data-driven nature
  - Name conflicts and typos -> user tools (database, user interface, documentation…)
  - High memory requirements and comparing costs -> hashed string ids

Common Types of Events

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.

Event Arguments

Data members of a class hierarchy of events

Collection of variants

- Static, dynamically sized array or linked list of tagged unions

Collection of key-value pairs

- Avoid indexed collection order dependency
- Unique id
- Closed or open hash table, array, linked list, or binary search tree

```
struct Variant {
    enum Type {
        TYPE_INTEGER,
        TYPE_FLOAT,
        TYPE_BOOL,
        TYPE_STRING_ID,
        TYPE_COUNT // nb of unique types
    };
    Type    m_type;
    union {
        I32   m_asInteger;
        F32   m_asFloat;
        bool  m_asBool;
        U32   m_asStringId;
    };
};
```

Key	Value	
	Type	
"event"	stringid	"explosion"
"radius"	float	10.3
"damage"	int	25
"grenade"	bool	true

Events Sending

- Each event is linked to a dynamic list of listeners
  - List of delegates = basically function pointers that can be coupled with an object pointer and used as a callback
- Two ways to send events
  - By queue: event in line with others to be processed by the event manager in a global `EventUpdate()`
    - Match and call each subscribed listener delegate function with events
    - 2 queues to handle new resulting events
  - By trigger: the event will be sent immediately
    - Almost like calling each delegate function directly

Event Handlers

- Single native virtual function or script function capable of handling all types of events
  - Usually switch statement or cascaded if/else

```
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

- Relationship graphs between game objects:
  - Transformation hierarchy (weapon, vehicles…)
  - Composition
  - Game logic (team…)
- Forwarding events within a graph of objects = Chains of Responsibility pattern
  - Can be applied to results of queries
  - Can lead to deep call stacks
    - Handler functions have to be fully re-entrant: called recursively without side-effects

Event Queuing

- Control over when events are handled (cf. game loop)
- Ability to post events into the future: later in the same frame, next frame, or some number of seconds after it was sent (clock + timestamp)
- Assign priorities when identical time
- Increase event system complexity
- Events and arguments deep copied and dynamically allocated
- Hard to debug
- Can be used for periodic updating

Data-Driven Event Systems: Scripts

Define how a particular kind of game object will respond to a particular kind of event, define new types of events, send events, and receive and handle events in arbitrary ways

Risks/Benefits

- Less powerful
- Easier-to-use and less error-prone
- Ability to easily search and replace within the source code
- Freedom of choice for editing tools

Data-Driven Event Systems: GUI

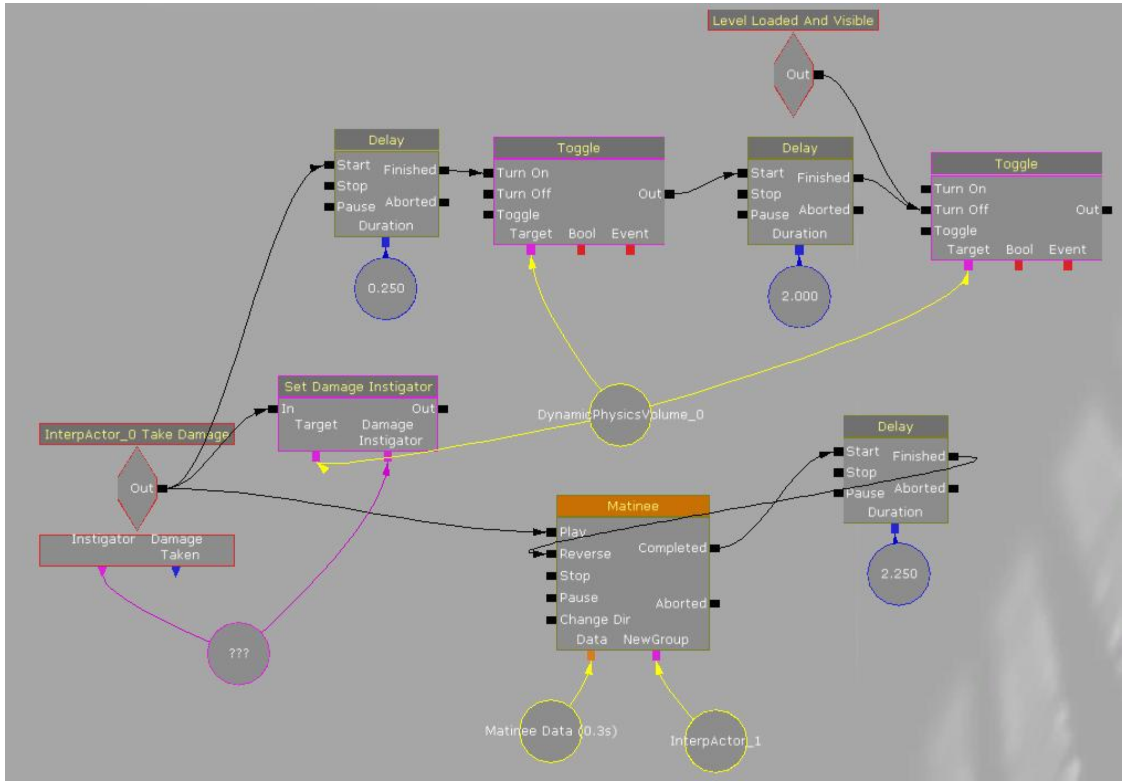
Possibility to configure how individual objects or classes of objects respond to certain events

More or less sophisticated

- List of all possible events that an object might receive ; control if, and how, the object responds
- Streams of data (bool, float, vector…) between objects with i/o ports, nodes, operators

Risks/Benefits

- Ease of use, gradual learning curve with the potential for in-tool help and tool tips to guide the user
- Error-checking
- High cost to develop, debug, and maintain
- Additional complexity, which can lead to bugs
- Designers are sometimes limited in what they can do with the tool



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

Event declaration

Delegate = basically a function pointer

Custom delegate and event

```
public delegate void NewEvent(int eventId);
public event NewEvent OnMyEvent;
```

Basic EventHandler delegate and event (no data)

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

Generic EventHandler delegate, EventArgs and event

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e) where TEventArgs :
EventArgs
public class MessageReceivedEventArgs : EventArgs {. . .}
public event EventHandler<MessageReceivedEventArgs> OnMessageReceived;
```

Unity / C# events

Event Receivers

Subscription/Unsubscription

```
OnMyEvent += MyCustomEventHandler;
OnCleanup -= MyCleanupEventHandler;
```

Handler:

```
void MyCustomEventHandler(int eventID){ … }
void MyCleanupEventHandler(object sender, EventArgs e)
{ … }
```

Unity / C# events

Event Raising

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

if (OnCleanup != null){
    OnCleanup(this);
}

if (OnMessageReceived != null){
    OnMessageReceived(this, new
     MessageReceivedEventArgs(aMessage));
}
```

Shoot events

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

- Each subsystem must be configured and initialized in a specific order (implicitly defined by interdependencies between subsystems)
- Shut-down typically in the reverse order
- Lack of C++ static initialization order
  - Major subsystems usually implemented as singleton (“managers”)
  - Simple approach: explicit start-up and shut-down functions for each singleton manager class

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
- 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 System

- File names and paths manipulation
  - More complex than strings:
    - Optional volume specifier - sequence of path components - reserved separator character
  - Differences across OS
- Directory scanning + File I/O
  - Synchronous or asynchronous
- Often engine-specific API
  - Independent from platform
  - Simplified & Extended

Off-Line Resource Manager

- Resource database
  - Multiple types of resources
    - Embedded, binary files, text, XML, relational db, GUI…
  - Metadata on how resources should be processed
  - Create, delete, inspect, modify, move
  - Cross-references, referential integrity
  - Revision control
    - Problem: copy of large assets size
  - Search and query

Off-Line Resource Manager

Asset Conditioning Pipeline

- Exporters to raw data
  - Custom plug-in for each DCC tool/format
- Resource compilers
  - Process the data to make it game-ready
- Resource linkers
  - Combine multiple resource files to a single package
- Build dependencies
- Optimization for specific platforms

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

http://www.gameenginebook.com/

gameprogrammingpatterns.com/

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

Game Engine Gems 1 and 2, 2010-2011

www.gamasutra.com

Game Developer Conference

- http://www.gdconf.com/
- https://www.youtube.com/channel/UC0JB7TSe49lg56u6qH8y_MQ
- www.gamasutra.com/features/gdcarchive/

gamedevs.org/

gamemechanicexplorer.com/#

www.redblobgames.com/

pixelnest.io/tutorials/gamedev-resources/

…