05 Game and Engine Architecture

Tvorba a dizajn počítačových hier
Návrh a vývoj počítačových hier

Michal Ferko

17. 10. 2024

Game development is software development

- Use of well-defined design patterns
- Relatively strict structure of a game and game engine
- Not only graphics and physics
- Although lots of concepts are borrowed/required from these fields

Game architecture

- All the parts a game (+engine) consists of
- How all the parts fit together
- Relying on established standards will make your life easier
  - You will create maintainable and reusable software pieces

- Avoiding standards might pay off in terms of efficiency
  - Engine custom-made for a specific game – e.g. Minecraft
  - Not a good idea for most games

CHANGE

The only constant in software development

Game architecture layers

- Architecture with lots of layers (think TCP/IP)
- Every subsystem can be put into one of these categories:
  - **Application** layer
    - Deals with the hardware and the operating system
    - This is usually handled by the game engine
  - **Game logic** layer
    - Manages your game state and how it changes over time
  - **Game view** layer
    - Presents the game state with graphics and sound
- Similar to the Model-View-Controller (MVC) design pattern
- Changes in hardware/OS should not affect the game logic or game view layers
  - Just like MVC

Game logic layer

- This is your game – all its mechanics
  - Without input systems, rendering & audio playback…
- Contains subsystems that manage the game world state
- Communicating state changes to other systems
  - Examples:
    - Playing a sound when you fire a gun
    - Playing an animation for the gun as it fires
    - Updating health of enemy hit by the shot
- Contains systems that enforce rules of your game world
  - Physics system
  - …

Game view layer

- Presents the game state to the player
- Translates input into game commands
- Not only drawing the game state on the screen
- Other views include:
  - AI agents get a “view” of the game state
  - A remote player gets a view of the game state
  - The state observed from game logic is the same
    - But they do different things

An example – racing game

- Game logic
  - Holds the data that describes cars and tracks
    - Car weight distribution, engine performance, tire performance, fuel efficiency, …
    - Track shape, surface properties, physics
  - **Input** – just like a real driver
    - Steering, acceleration, braking
  - **Output** – state changes and events
    - Car and wheel positions and orientations, damage stats, how much ammo is left
    - Events: car collisions, passing checkpoints, …

Racing Game – Human Game View

- Draw the scene, spawn particles, play audio, force feedback…
- Read the input devices
  - “Accelerator at 100%”
  - “Steer left”
  - Sends these commands back to the game logic
- What happens when you press SPACE (emergency brake)
  1. The view sends a message to game logic
  2. Game logic sets the emergencyBrakeOn to true
  3. Game logic notifies the view that state changed
  4. The view responds by playing a sound and spawning a dust particle effect on tires

Racing Game – AI Game View

- Receives the same game state and events as the human view
  - Track, weather, car positions and velocity
- Can react in response to events (such as “Go!”) by sending info to the game logic
  - Set accelerator to 100%
  - Steer left at 50%
- Commands remain the same!

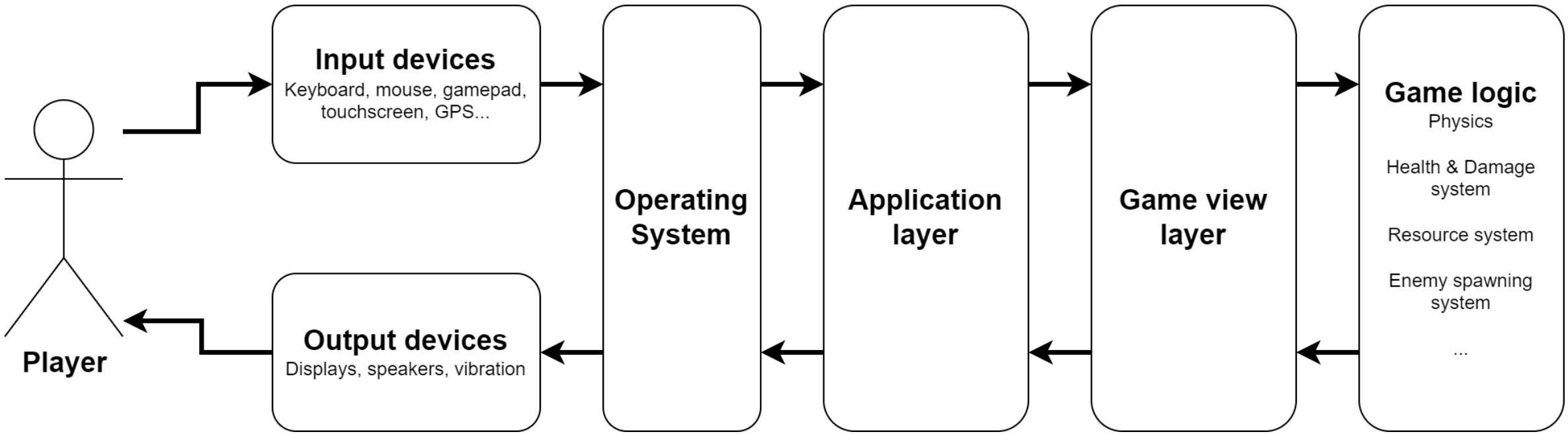
Game views

- Flexible – any number of human/AI/remote views
  - E.g. multiple human views on one device – split-screen multiplayer

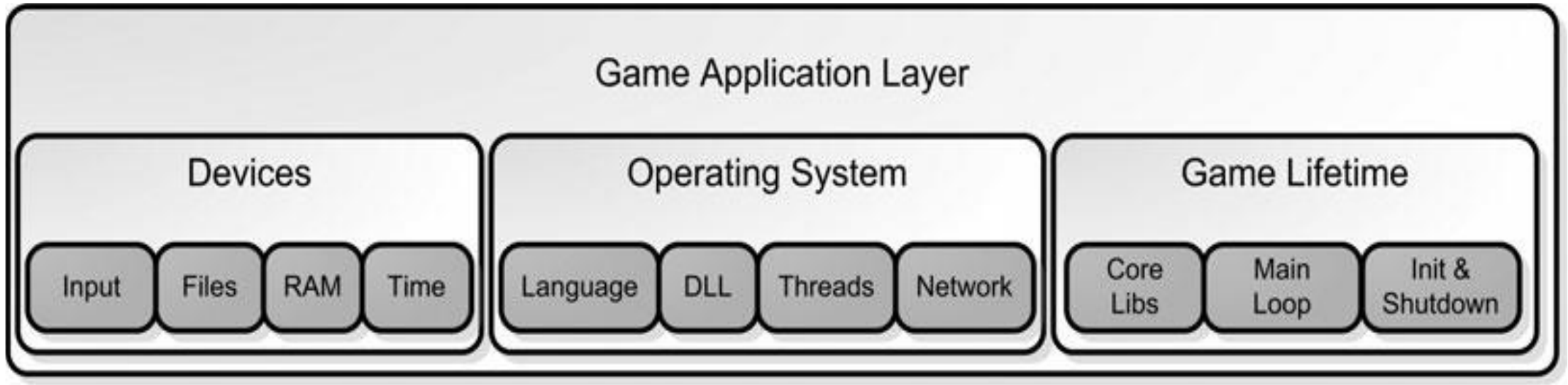
- A game view that just records game events into a buffer
  - You might replay them later
  - The game logic is kind of disabled during replay, since a view is sending all the events
  - With a little extra work, you can get a rewind feature working

Game views (2)

- Views can give advantages or disadvantages
  - 4:3 aspect ratio might give smaller field of view than 16:9 aspect ratio
  - AI might know more about game state than the player (see through walls)
- Game views are difficult to get used to
  - Unity does not offer a strict separation of views from logic implicitly
  - Using it explicitly will make more maintainable code
  - Even if you do not strictly separate, it’s good practice
    - You should know which layers are communicating


Input devices
Keyboard, mouse, gamepad, touchscreen, GPS...
Operating System
Application layer
Game view layer
Game logic
Physics
Health & Damage system
Resource system
Enemy spawning system
...
Player
Output devices
Displays, speakers, vibration

Application layer

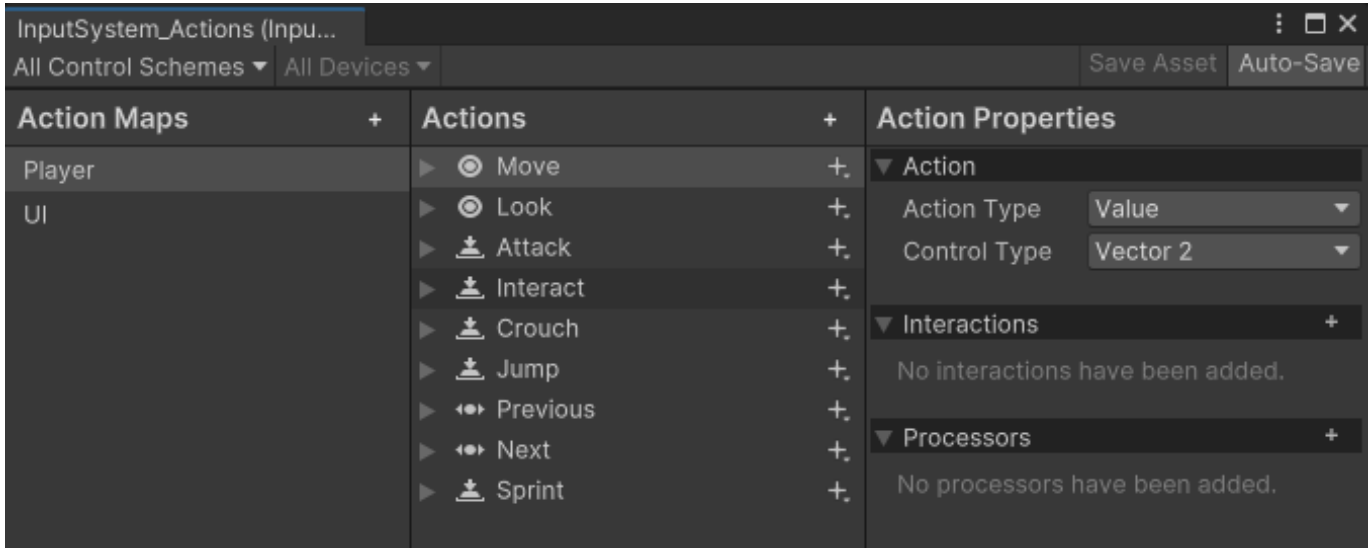


Reading input

- Layer between the OS and the rest of the game application layer
- The state is translated into game commands
- Should be configurable
- Game state should never change directly from reading user input
  - Not flexible
  - Lots of changes when controls change (keyboard vs. gamepad, mouse vs. touchscreen)

Reading input in Unity

- Old Input system
  - Direct ⇒ Input.GetKey()
  - Indirect ⇒ Input.GetAxis() / Input.GetButton()
    - can configure in the editor or at runtime
- New Unity Input System is indirect
  - Can configure named actions
  - Much more structured
  - Great for multiple sets of controls



File System and Resource Caching

- Reading and writing from disk and other storage media
- Managing resource files can be complicated
- One of the hidden systems is the **resource cache**
  - Commonly used assets are always in memory
  - Rarely used assets are in memory only when needed (end-game video)
  - The resource cache tries to “fool” the game into thinking that all the assets are available in memory
  - If all goes well, the cache can load files *before* they are needed
  - Cache misses might occur if it fails to load something in time
  - Solve by loading screens or small lags

Resource Loading in Unity

- SceneManager.LoadScene()
- SceneManager.LoadSceneAsync()
- Resources.Load/Addressables allows more fine-grained memory control
  - Can have a “weak reference” and load manually with Addressables
  - Can load by name from a folder with Resources.Load or Addressables
  - If you need this, use Addressables
  - Avoid Resources.Load as much as you can

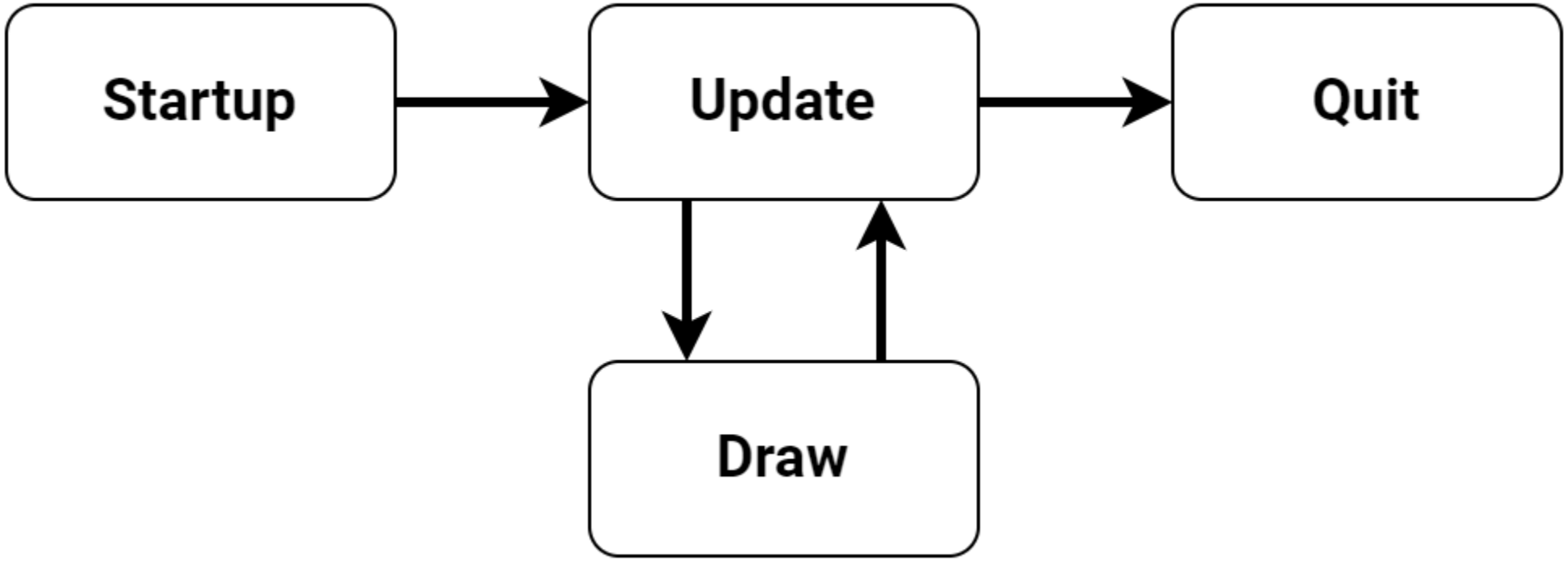
Resource Loading in Unity

- Instantiate(prefab) is NOT loading anything
- Everything that is referenced must already be in memory
- When a scene loads:
  1. Read scene and load all assets that it references – prefabs, textures, models…
  2. For each of those assets, load all assets that they reference (unless already loaded)
- These references are also used when building the game
  - Takes a list of all scenes and gathers all references recursively
  - Adds Addressables or Resources folder
  - Unreferenced assets are NOT packaged into the build

Initialization, Main Loop, and Shutdown

- Games are simulations that have a life of their own
  - Player input is not required for the game to continue simulation
- The system controlling the game simulation is the **main loop** or **game loop**
- Usually has three stages
  1. Grab user input
  2. Update game logic
  3. Present the updated game state through all views
     - Rendering, playing sounds, sending state over the internet

Simple game loop



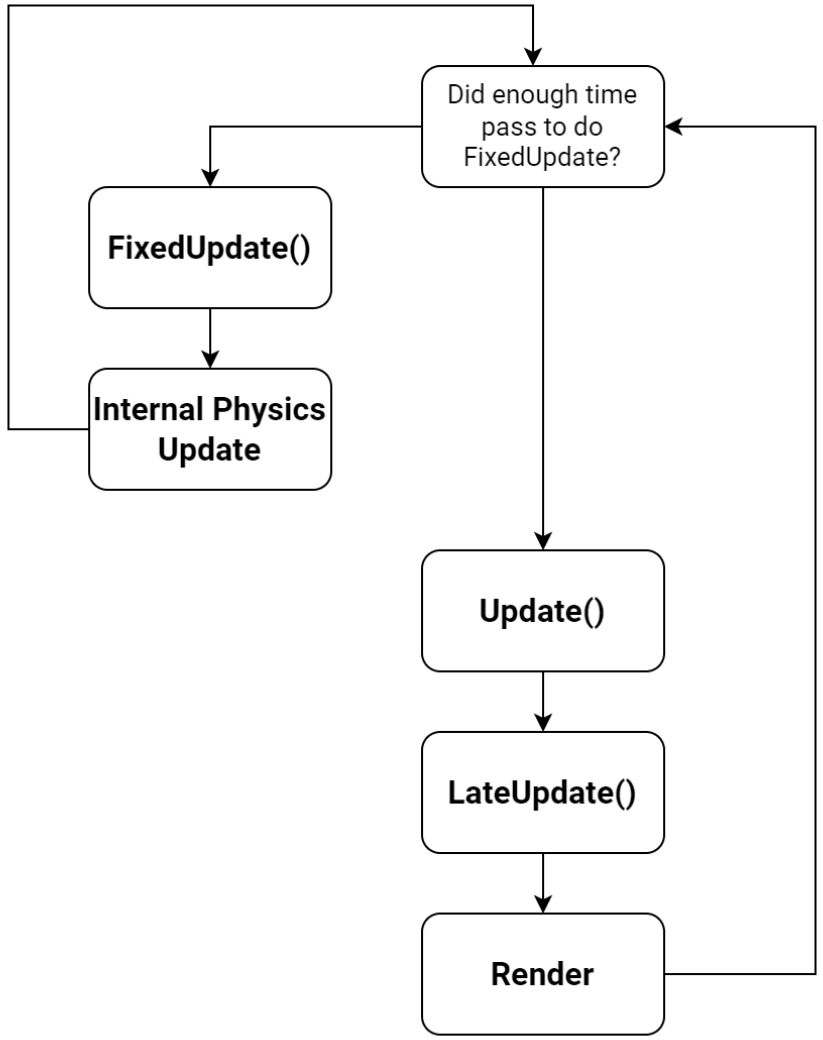
Game loop in Unity

- Game loop handled implicitly
- Querying user input is handled by the engine
- Game programmers are given two main callbacks
  - `Update()`
    - Updates as often as possible
    - Called once for each frame rendered (lower FPS => less updates)
  - `FixedUpdate()`
    - Updates in fixed intervals (by default every 20ms => 50 FixedUpdates per second)
    - Called once before each physics engine step
- Rendering is done by the engine implicitly – we only set what to render
  - Exception: Using low-level rendering access with the `GL` or `Graphics` class

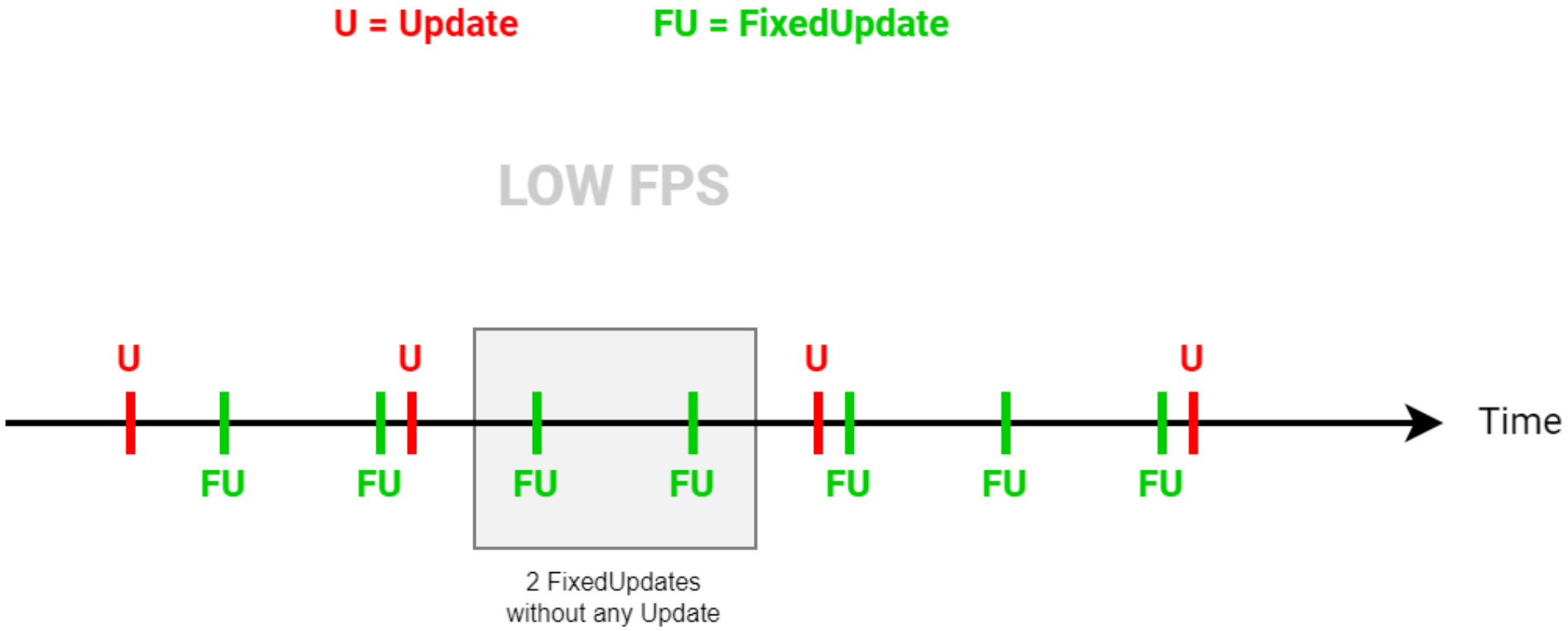
How Unity game loop looks (simplified)

```
float timer = 0;
while (true)
{
    while (timer > fixedTimeStep)
    {
        FixedUpdate();
        PhysicsUpdate();
        timer -= fixedTimeStep;
    }
    Update();
    LateUpdate();
    Render();
    timer += deltaTime;
}
```

Unity’s game loop simplified



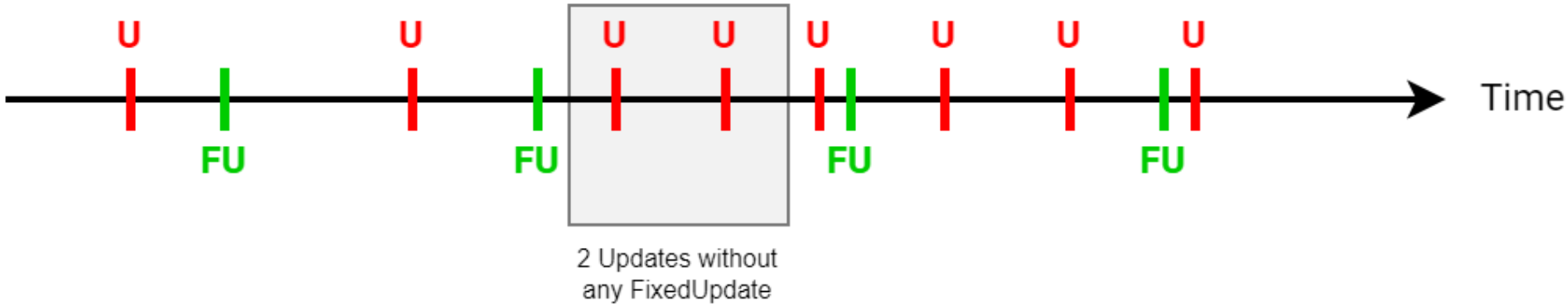
Unity Update & FixedUpdate



Unity Update & FixedUpdate

U = Update FU = FixedUpdate

HIGH FPS



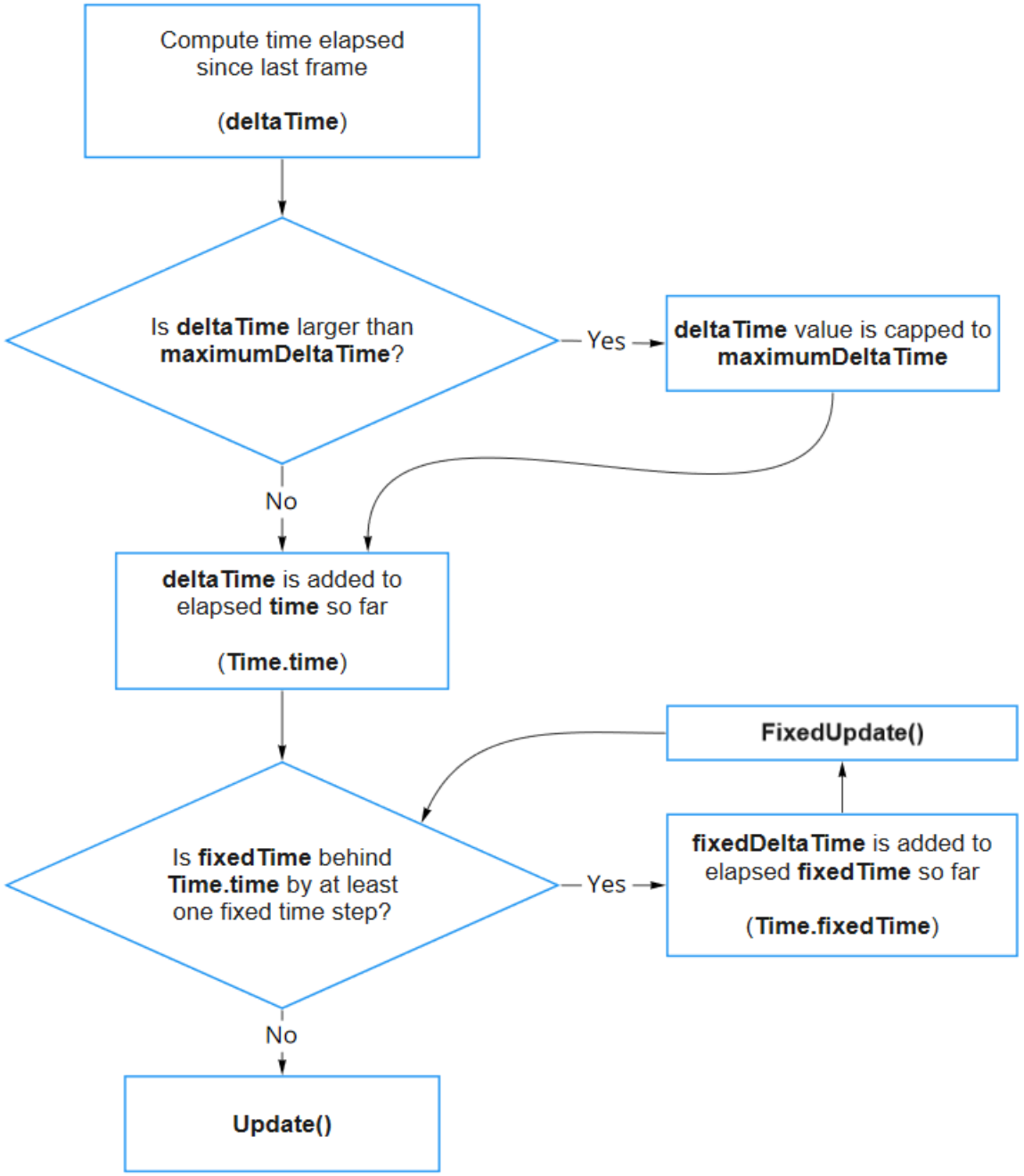
Unity Execution Order

- You need to know what happens and in which order!

- **This is the most important graph for understanding Unity**

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

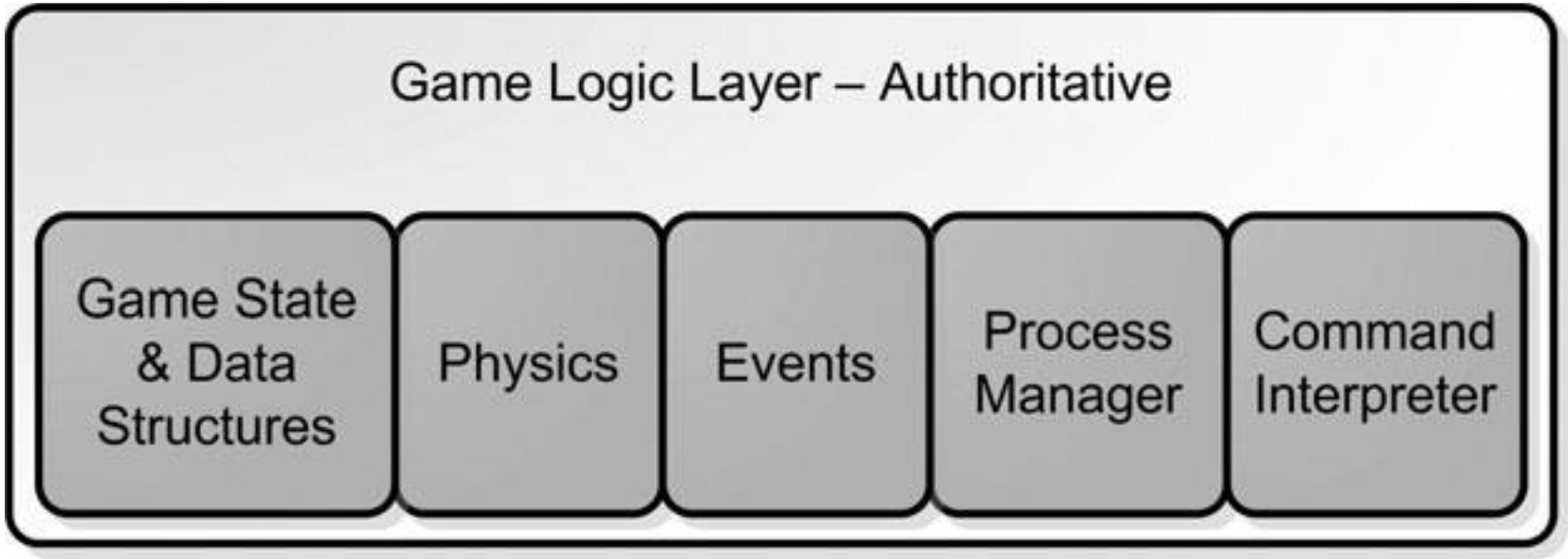
- From https://docs.unity3d.com/Manual/TimeFrameManagement.html



Other Application Layer Code

- System clock
- String handling
- System libraries
- Threads and thread synchronization
- Network communication
- Initialization
- Shutdown
- (Scripting language)

Game logic


Game Logic Layer – Authoritative
Game State & Data Structures
Physics
Events
Process Manager
Command Interpreter

Game logic

- Defines the game universe
- What things (entities) are there
- How they interact
- Defines how game state can be changed by external stimulus

Game state and data structures

- A game needs to store its game objects in a container
  - Must be able to traverse quickly to change game state
  - Should be flexible as to what data it will hold
  - Special game data (hitpoints, inventory, …) are stored in some custom data structure
- In Unity: Upgraded scene graph hierarchy
  - Retrieve objects either by pointing directly to them, or special functions
  - Static functions part of GameObject:
    - `GameObject.Find(), GameObject.FindWithTag(), GameObject.FindObjectsWithTag()`
  - Implicitly traverses a tree or a hash table to efficiently find the objects
  - Special game data stored as serialized variables of script components

Scene Graph Hierarchy in Unity

- Hierarchy stored in the **`Transform`** component of the game object
- Each Game Object can have a parent
  - `Transform.parent`
- Each Game Object can have any number of children
  - `Transform.GetChild(i), Transform.childCount`
- Good practices with the graph hierarchy
  - Never rely on the parent
    - A parent should be concerned about its children, not the other way around
  - Use `GetChild()` only to iterate through all children
  - Never hard-code `GetChild(3)`, use references to objects instead

Game state and data structures

- Easy to confuse game logic representation with the visual representation of objects
  - Amount of damage a weapon deals is stored in game logic
  - The weapon’s model, textures, icons are only relevant to the game view
- Another example:
  - Skeletal mesh object that is used for skinning the character when rendering
  - It seems that it has something to do with the character’s weight
  - Skeletal mesh – view, weight – logic (probably for physics calculations)

Entity-Component-System (ECS)

- Design pattern used to store and manipulate game state
- Favors composition over inheritance
- Entities consist of Components
- Each component has a single responsibility
- Systems run in background and handle simulation
- One system serves only one purpose
  - Physics System, Player Damage System
- Components register in one or more systems
- Entities are affected indirectly (through their components)

ECS in current Unity

- It’s not really ECS, but similar
  - Systems are not in one place
- Entity = GameObject – name, tag, active/inactive, layer…
- Component = Component (MonoBehaviour)
- System
  - Configurable but not directly visible
  - Can create own systems
  - Individual components register with their respective systems
  - **Split across multiple components**

Example of systems

- Collider and Rigidbody register with the Physics System
- MeshRenderer, Camera, Light register with the Rendering System
- Scripts register with the Scripting System and Event System
- You create other systems through MonoBehaviours – Damage System, Items & Inventory…
  - Health, Weapon and Explosive are all part of the damage system

True ECS in Unity

- Unity offers true ECS – it’s called DOTS (data-oriented technology stack)
- Game Objects and Components are different
- You don’t create MonoBehaviours, but create Systems instead

- Has been fully released a few months ago
- More complex and more difficult to work with
- Can provide performance improvements **if done right**

Physics and Collision

- Rules of your physical game universe
  - No need for over-complicated physics to make a game fun
- If our game is completely abstract, players tolerate unrealistic physics
- If the game simulates the real world, small errors might cause players to be very disturbed
- Usually, rigid body dynamics with dynamic objects being mostly convex
- Few exceptions:
  - Ragdoll physics
  - Fluid simulation
  - …

Game Events

- When the game state changes, a lot of other systems need to react accordingly
- Game logic is responsible for generating these events and passing them further
- Subsystems register with the Event Manager to listen to events that they react to
- Clean and efficient separation of unrelated code with a simple abstraction and the use of a unified event interface
- The Observer pattern

Process Manager

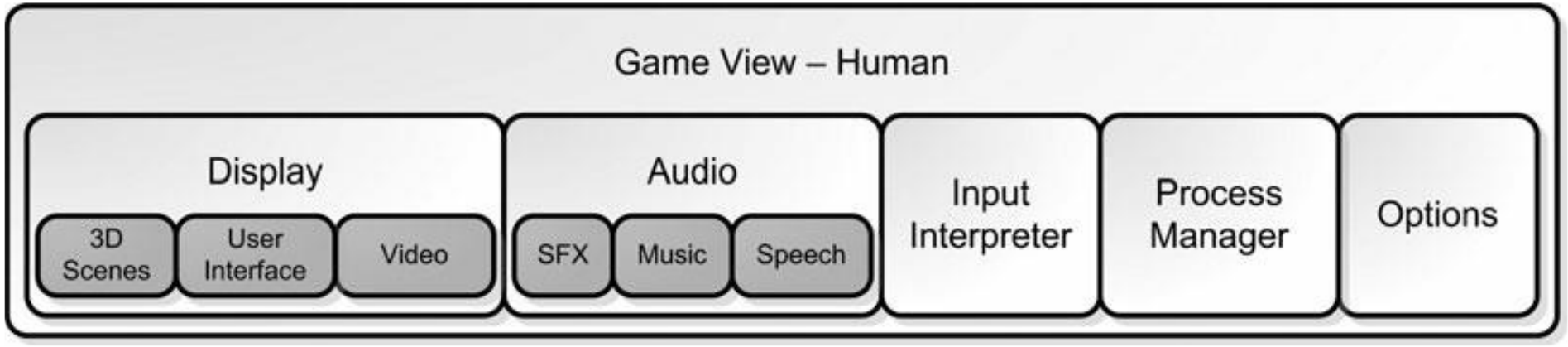
- The game logic is composed of small parts of code that need to be executed periodically for the game to work correctly
- Multiple processes that are completely independent (but we might provide dependencies)
- Mostly script executions (if there’s a scripting language)
  - Otherwise, it’s just one class per operation to keep things simple
- Chaining processes together (throwing a grenade, explode on collision)
- Unity handles this by calling the Update() (or other event) functions on all script components attached to currently enabled game objects in the scene
  - You may sometimes come up with code that needs a certain order of execution, **AVOID IT as much as you can**

Command Interpreter

- Provides a good separation of the game view and game logic
- Need to interpret commands sent by AI or human players
- Unified interface, no mixing of unrelated code
- Provides more efficient debugging
  - You can send separate commands while debugging
  - This is usually what consoles are for
  - Not the hardware, the Counter-Strike console
  - The Elder Scrolls also had a console
  - As well as many others

Game view - Human

- Views are a collection of systems that communicate with game logic to present the game to an observer
- Observers can be human, AI,…
- The view responds to game events as well as controller input
- Works as a translator component that outputs game commands on one side and the presentation of the game on the other side



Graphics Display

- Renders the objects in the scene
- Renders the user interface (HUD)
- Must draw the scene as fast as possible
- Lots of problems
  - Which objects to draw
  - How to handle complex transforms
  - How to handle complex visual effects
  - Handles animation interpolation
  - Lighting conditions
  - Post-processing
  - Level-of-detail
  - …

Rendering

- 3D Meshes
- Cameras
- Lights
- Post-processing

Graphics Display (2)

- For very complex 3D scenes, need a lot of pre-computation
  - Light maps
  - Potentially visible sets (PVS)
  - Light Probes
- **The artist must be aware of the game engine capabilities**
- Lots of constraints
- Unity tries to handle all of this
- You might reduce performance of the game without even knowing it

Audio

- Playing sounds
- Three main areas
  - **Sound effects** – simple, when an event occurs, play some audio
  - **Music** – a little harder when done right
    - Tone of music adjusting according to game situation (Elder Scrolls, Halo, …)
  - **Speech** – very tricky
    - Lip-sync and storage of lots of sounds are the main problems
- Take into account 3D audio
  - 3D positions of listener and sources

Audio in Unity

- AudioSource and AudioListener components
- AudioMixer, groups, filters, effects
- Can use other audio engine – FMOD, Wwise…

User Interface Presentation

- Every UI must be very specific and adjusted to the game
- Re-using components is possible, but only to a certain extent
- Sometimes you need a completely new GUI component to fit the game
  - A compass, special inventory, …

Unity UI

- Uses separate rendering from 2D and 3D
  - All components rendered on a Canvas
  - Objects use 2D sprites for rendering
- Has numerous tools for anchoring, scaling for different resolutions, events…
- Can do screen-space, camera-space and world-space UI
  - Can mix UI rendering and standard rendering

- Alternative: UI Toolkit
  - https://docs.unity3d.com/Manual/UI-system-compare.html

Options

- Configuring the view
- Resolution, aspect ratio
- Controls
- Sound effects
- Graphics quality and performance

- In Unity: see Project Settings

References

- McShaffry, M. & Graham, D. (2013). *Game Coding Complete*. Boston, Mass: Course Technology PTR. 4th ed.
  - Chapter 2 – What's in a game?
- http://gameprogrammingpatterns.com/
- http://entropyinteractive.com/2011/02/game-engine-design-the-game-loop/