COMP4002/G54GAM Games

Architectures

Now what?

- Given a game design, how do we go about building it?
  - Highly complex interactive system
- We need to plan it otherwise it becomes a mess
  - Difficult to understand
  - Difficult to maintain
  - Difficult to extend
- “Games must be designed, but computers must be programmed”
  - Often mainly native / optimised vs HLL
    - C, C++
    - *Speed and efficiency*
  - However a game can (obviously) be written in any language

Games as Systems

- Game are *soft real-time interactive agent-based computer simulations*

- A Subset of the real world / an imaginary world
  - Modelled mathematically
    - Approximation, simplification
    - Numerical over analytical – can determine the next state of the system
- Simulation
  - Agent-based - Distinct entities interact autonomously
  - Temporal - The model of the world is dynamic, changes over time
  - Interactive - Respond to user interactions
- Deadline driven
  - The screen has to be updated 30 times per second
  - Soft - Missing a deadline is not catastrophic

Traditional Separation of Work

- Game Engine
  - Software, written by programmers
- Rules and Mechanics
  - Created and scripted by designers, with help from programmers
- User Interface
  - Coordinated with programmer, HCI specialist
- Content
  - Created by designers using tools

Game Engines - Systems

- A collection of sub-systems
- E.g. Physics
  - An example of a game *sub-system*
    - Specifies the space of possibilities for a game
    - Not the specific parameters of elements
  - Separates programmer from game designer
    - Programmer creates the engine
    - Designer fills in the parameters
- Defines physical attributes of the world
  - There can be a gravitational force
  - Objects can have friction
  - Light can reflect in the following ways
- Does not define precise values or effects
  - The direction or value of gravity
  - Friction coefficients for each object
  - Specific lighting for each material

Game Engines - Characteristics

- Broad, adaptable and extensible
  - Encodes all ***non-mutable*** design decisions
  - Parameters for all ***mutable*** design decisions
- Outlines gameplay *possibilities*
  - Cannot be built independent of game design
  - But only needs high level information
- Data driven design
  - Ideal: No code outside the engine
    - Everything else is data
  - Create game content using editors / authoring tools
  - Art, music in standard file formats
  - Object, level data in XML (e.g.)
  - Character behaviour specified via scripts

Rules and Mechanics

- Fills in the values for the system
  - Parameters
    - Gravity, hit points, power
  - Types of player abilities, mechanics
  - Types of world interactions
  - Types of obstacles, challenges
- Does not include specific challenges
  - Set of challenges that *could* exist
  - Contents of the *pallet* for the level editor

Interfaces

- Engine and mechanics
  - …describe what the player can do
  - …do not specify *how* it is done
- Interface specifies
  - How a player does things
    - Player -> computer
  - How a player receives feedback
    - Computer -> player
- Bad interfaces can ruin a game
  - Strive for metaphorical “invisible interface”
  - Flow / immersion


STATS
HP 170/170
AP 8/8
XP 10/1000
Strength 5
Perception 4
Endurance 5
Charisma 2
Intelligence 4
Agility 7
Luck 6
Agility affects your Small Guns and Sneak skills, and the number of Action Points available for V.A.T.S.
Status
S.P.E.C.I.A.L.
Skills
Perks
General
STATS
ITEMS
DATA
Pip-Boy
Model 3000
© 2007 Bethesda Softworks LLC. All Rights Reserved.

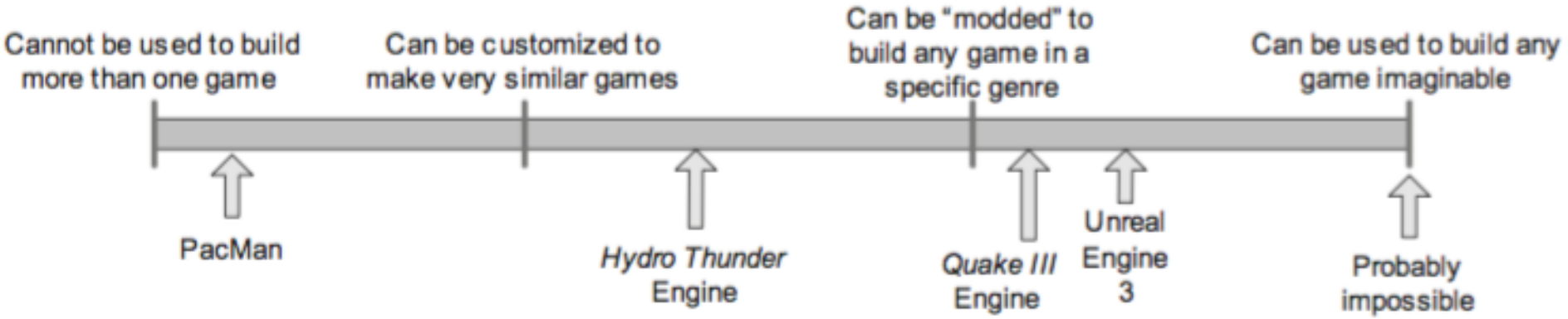
Content

- Content is *everything else*
- Gameplay content defines the actual game
  - Goals and victory conditions
  - Mission and quest logic
  - Levels
  - Interactive story choices
- Non-gameplay content that affects player experience
  - Graphics and cut-scenes
  - Sound effects and background music
  - Non-interactive story

Why the division?

- These components are not developed sequentially
  - Content may require changes to the game engine
  - Interface may change throughout
- A way of organising the design
  - Engine
    - Decisions to be made early on, hard-coded
    - *Unity,* Unreal, Frostbite, Source, id tech
  - Mechanics
    - Mutable design decisions
  - Interface
    - How to shape the user experience
  - Content
    - Specific gameplay and level-design

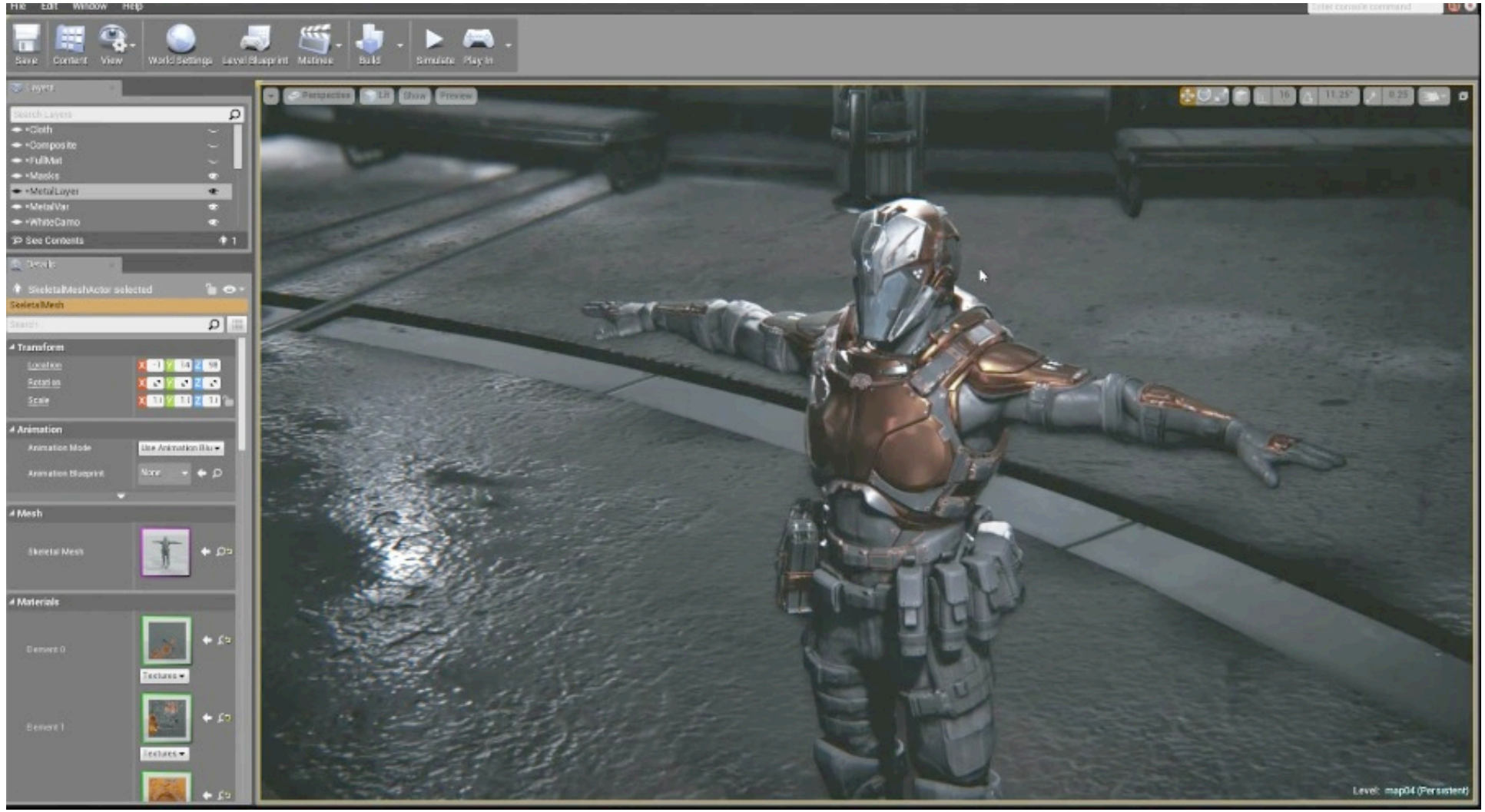
Why Game *Engines*?



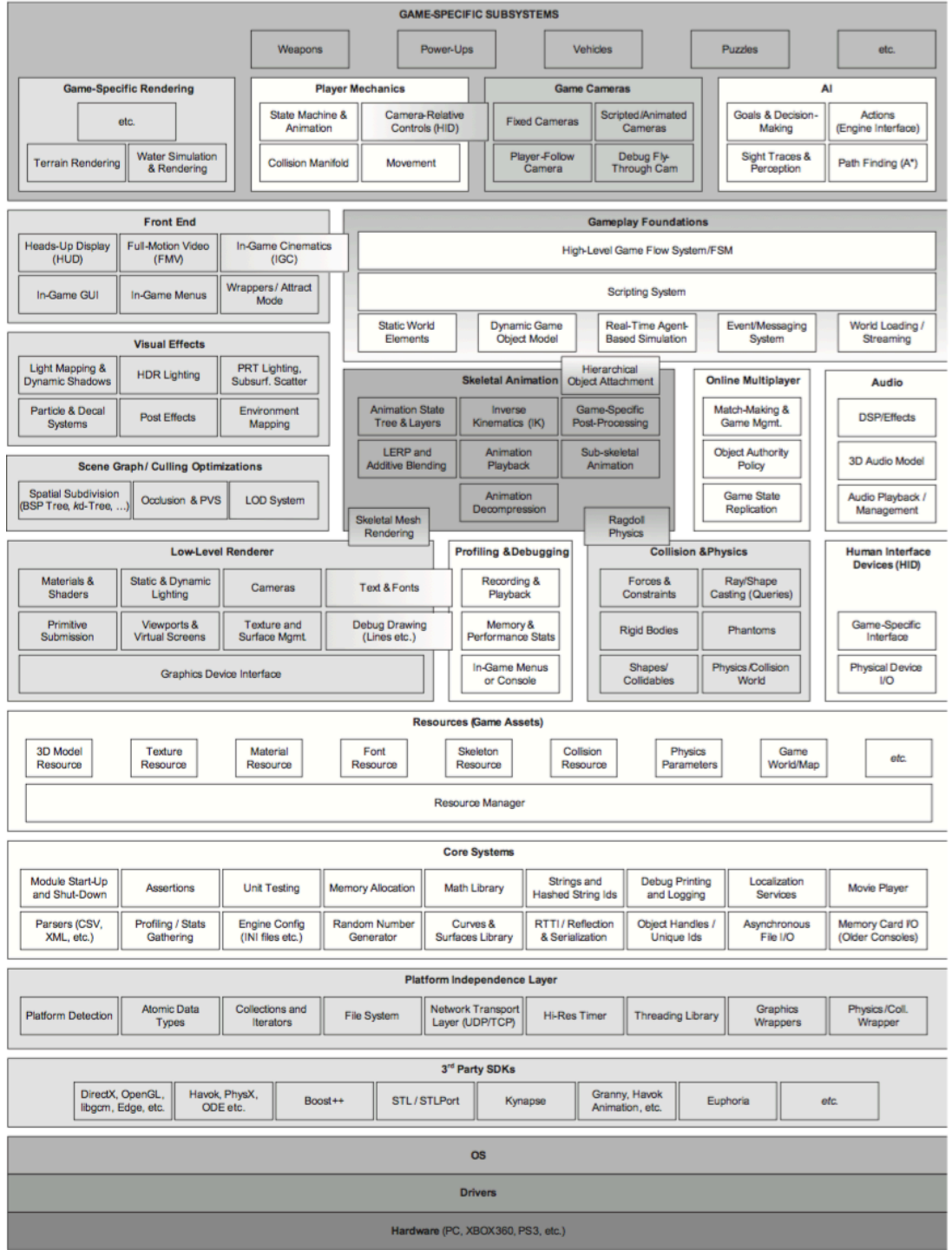
- Initially games were completely bespoke
  - Software specialised for individual application and hardware
- Now software aims to be fully featured, reusable and "kit" oriented
  - Robustness improves in the long term
  - Faster to develop new games
  - Abstraction over multiple platforms
  - Secondary revenue from licensing game engines
  - But do games made with the same engine all look alike?

Game Engines across Genres

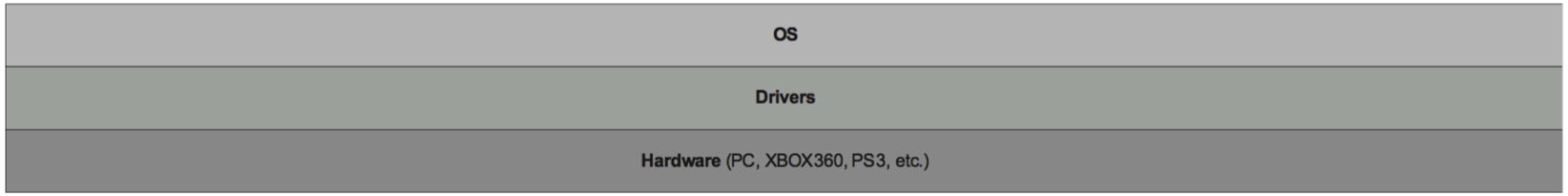
- Engines are typically *genre specific*
  - But with a large overlap – e.g. user input, visual rendering
- FPS
  - Efficient rendering, physics-based animation, AI of NPCs
- Platformers
  - Dynamic world, animated sprite characters, collision detection
- Fighting games
  - Large set of animations, accurate user input, character animation
- Racing games
  - Level of detail, rendering, rigid body physics and deformations
- RTS
  - Crowds, evolving environment, complex agent-based AI
- MMORPGS
  - Large scale network support (servers), data optimisation, persistent worlds


File Edit Window Help
Save
Content
View
World Settings
Level Blueprint
Matinee
Build
Simulate
Play In
Layers
Search Layers
Cloth
Composite
FullMat
Masks
MetalLayer
MetalVar
WhiteCamo
See Contents
Details
SkeletalMeshActor selected
SkeletalMesh
Transform
Location
Rotation
Scale
Animation
Animation Mode
Use Animation Bla
Animation Blueprint
None
Mesh
Skeletal Mesh
Materials
Element 0
Element 1
Textures
Perspective
Lit
Show
Preview
Level: map04 (Persistent)


Z ZOOM VIEW
HEALTH 100
SUIT 30
AMMO 6 30

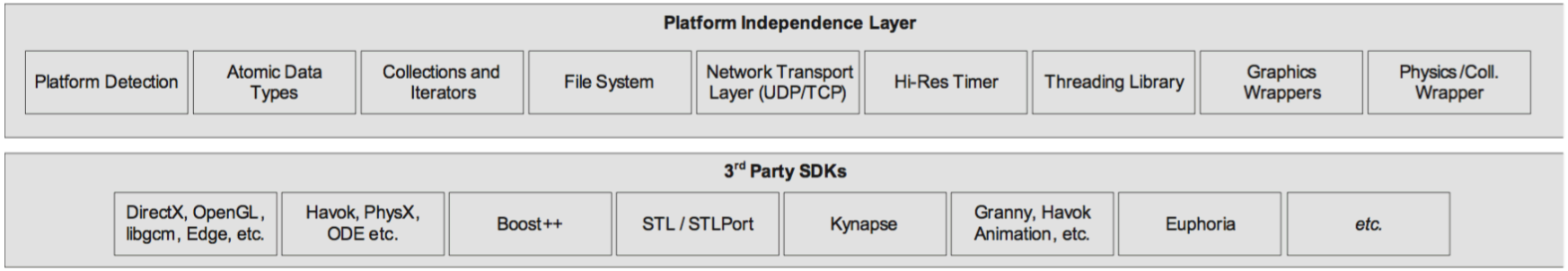

GAME-SPECIFIC SUBSYSTEMS
Weapons
Power-Ups
Vehicles
Puzzles
etc.
Game-Specific Rendering
etc.
Terrain Rendering
Water Simulation & Rendering
Player Mechanics
State Machine & Animation
Camera-Relative Controls (HID)
Collision Manifold
Movement
Game Cameras
Fixed Cameras
Scripted/Animated Cameras
Player-Follow Camera
Debug Fly-Through Cam
AI
Goals & Decision-Making
Actions (Engine Interface)
Sight Traces & Perception
Path Finding (A*)
Front End
Heads-Up Display (HUD)
Full-Motion Video (FMV)
In-Game Cinematics (IGC)
In-Game GUI
In-Game Menus
Wrappers / Attract Mode
Gameplay Foundations
High-Level Game Flow System/FSM
Scripting System
Static World Elements
Dynamic Game Object Model
Real-Time Agent-Based Simulation
Event/Messaging System
World Loading / Streaming
Visual Effects
Light Mapping & Dynamic Shadows
HDR Lighting
PRT Lighting, Subsurf. Scatter
Particle & Decal Systems
Post Effects
Environment Mapping
Skeletal Animation
Hierarchical Object Attachment
Animation State Tree & Layers
Inverse Kinematics (IK)
Game-Specific Post-Processing
LERP and Additive Blending
Animation Playback
Sub-skeletal Animation
Animation Decompression
Skeletal Mesh Rendering
Ragdoll Physics
Online Multiplayer
Match-Making & Game Mgmt.
Object Authority Policy
Game State Replication
Audio
DSP/Effects
3D Audio Model
Audio Playback / Management
Scene Graph / Culling Optimizations
Spatial Subdivision (BSP Tree, kd-Tree, ...)
Occlusion & PVS
LOD System
Low-Level Renderer
Materials & Shaders
Static & Dynamic Lighting
Cameras
Text & Fonts
Primitive Submission
Viewports & Virtual Screens
Texture and Surface Mgmt.
Debug Drawing (Lines etc.)
Graphics Device Interface
Profiling & Debugging
Recording & Playback
Memory & Performance Stats
In-Game Menus or Console
Collision & Physics
Forces & Constraints
Ray/Shape Casting (Queries)
Rigid Bodies
Phantoms
Shapes/ Collidables
Physics/Collision World
Human Interface Devices (HID)
Game-Specific Interface
Physical Device I/O
Resources (Game Assets)
3D Model Resource
Texture Resource
Material Resource
Font Resource
Skeleton Resource
Collision Resource
Physics Parameters
Game World/Map
etc.
Resource Manager
Core Systems
Module Start-Up and Shut-Down
Assertions
Unit Testing
Memory Allocation
Math Library
Strings and Hashed String Ids
Debug Printing and Logging
Localization Services
Movie Player
Parsers (CSV, XML, etc.)
Profiling / Stats Gathering
Engine Config (INI files etc.)
Random Number Generator
Curves & Surfaces Library
RTTI / Reflection & Serialization
Object Handles / Unique Ids
Asynchronous File I/O
Memory Card I/O (Older Consoles)
Platform Independence Layer
Platform Detection
Atomic Data Types
Collections and Iterators
File System
Network Transport Layer (UDP/TCP)
Hi-Res Timer
Threading Library
Graphics Wrappers
Physics/Coll. Wrapper
3rd Party SDKs
DirectX, OpenGL, libgcm, Edge, etc.
Havok, PhysX, ODE etc.
Boost++
STL / STLPort
Kynapse
Granny, Havok Animation, etc.
Euphoria
etc.
OS
Drivers
Hardware (PC, XBOX360, PS3, etc.)

OS/Hardware layer



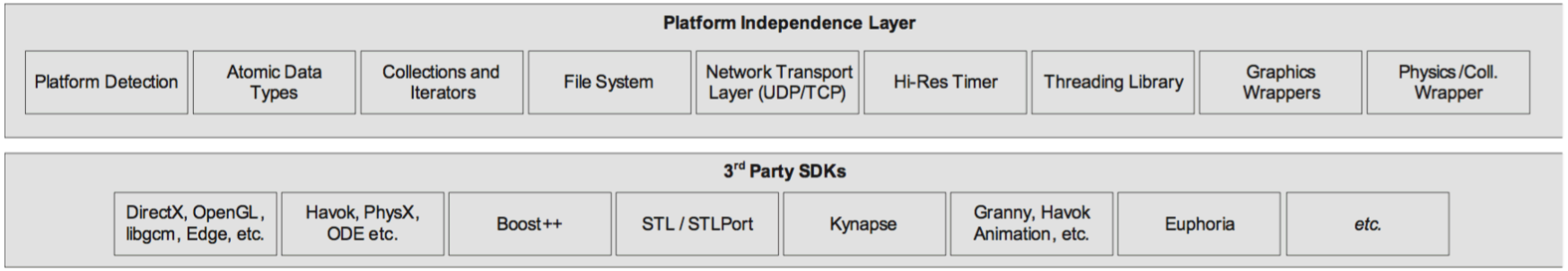
- Hardware
  - PC, PS2,3,4, Xbox,360,One, Nintendo…
  - Graphics card
  - Sound card
  - (Physics card?)
  - Input devices
    - Keyboard, mouse, game pad, controller, camera…
- Drivers
  - Low level interface to the above
  - Provided by hardware vendor
- OS
  - Time-sliced (Windows, Linux)
  - Console – thin library layer compiled into the game

3rd party SDKs



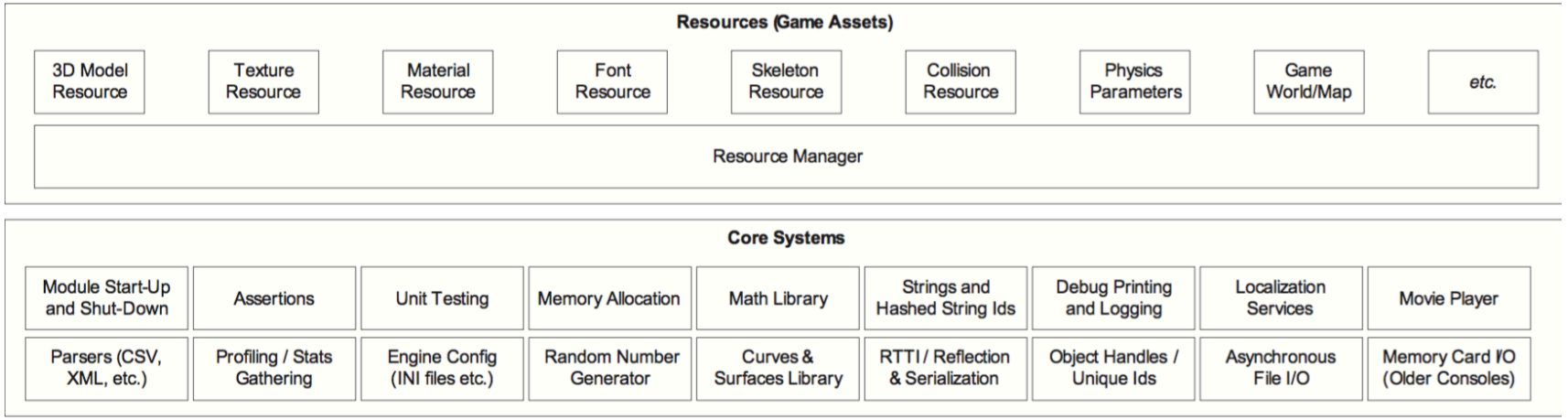
- Hardware interface libraries
  - OpenGL - Widely used, multiplatform
  - DirectX - Microsoft's 3D graphics SDK
  - Libgcm + Edge - PS3 graphics interface
- Physics
  - Havok, PhysX
- Data structures and algorithms
  - Boost, STL

Platform Independence



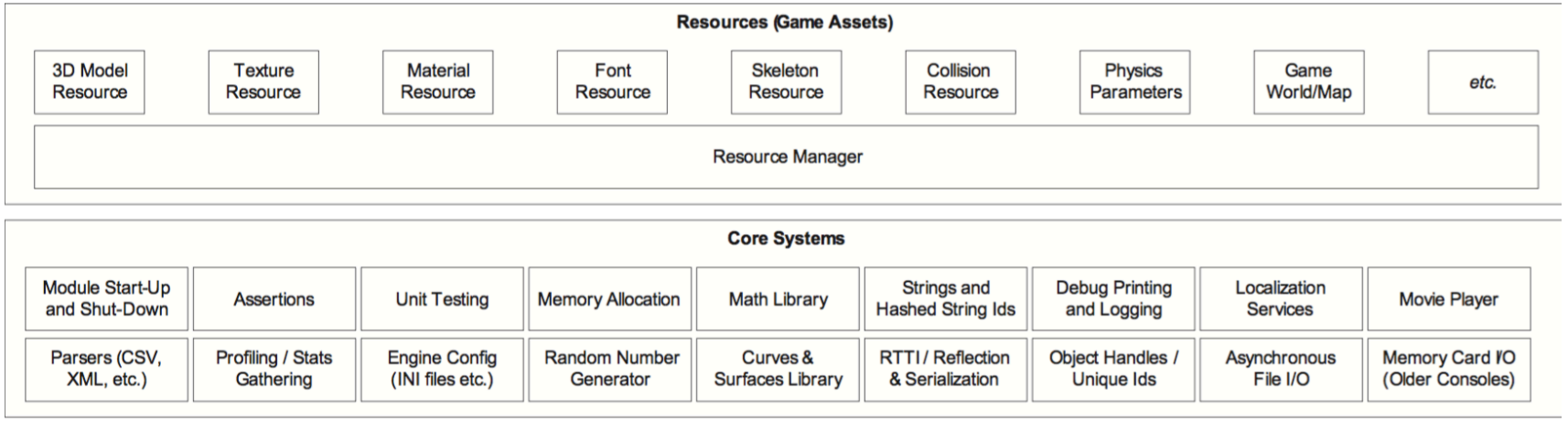
- Most game engines target multiple platforms
  - Notable exceptions are first-party studios
    - Naughty Dog / Sony
- Wrap / replace standard OS functions
  - Platform independent API
    - A lot of variation between platforms
  - Threading, Filesystem, Time, Network

Core Systems



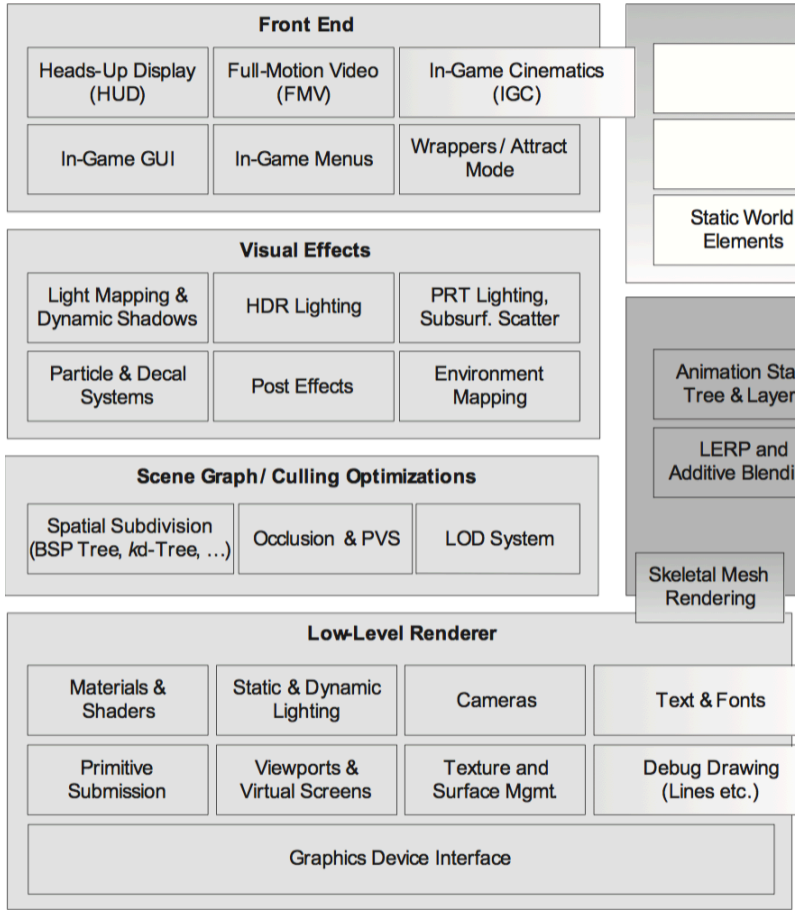
- Useful software utilities
  - Memory management
  - Localisation
  - Module start-up, shutdown
  - Configuration
  - RNG

Resources



- Unified interface for accessing assets
  - Abstraction over filesystem
- Packaged in .uasset, .pak, .pk3, .wad

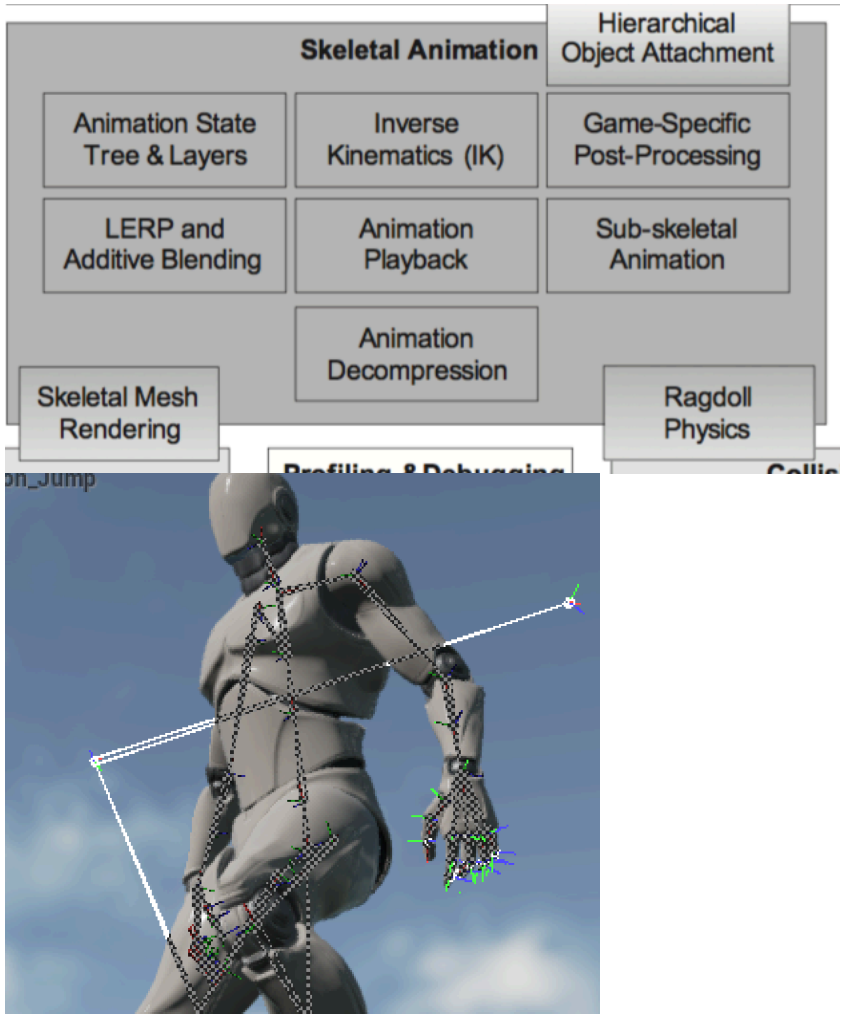
Graphics



- The visual representation of game data
  - HUD, GUI, 2d interfaces
  - Fancy visual effects
  - Decide what to draw
    - Frustum culling, spatial subdivision
  - Draw geometric primitives as quickly / richly as possible
  - Set up drawing context

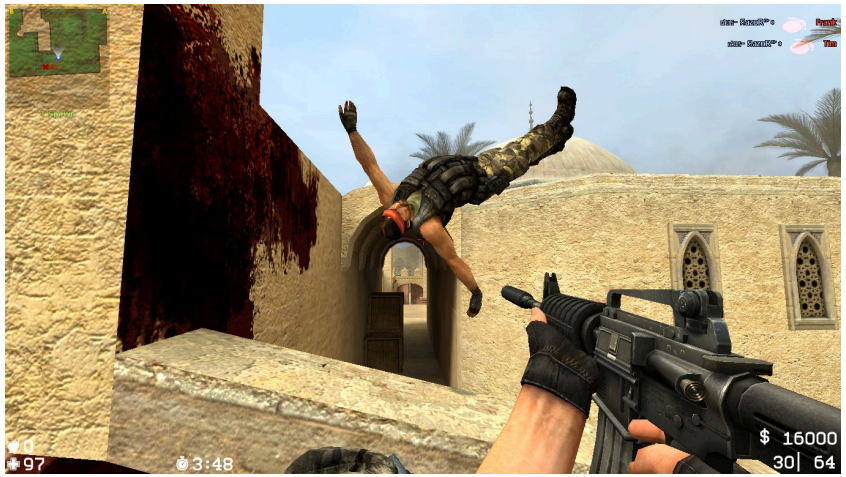
Skeletal Animation

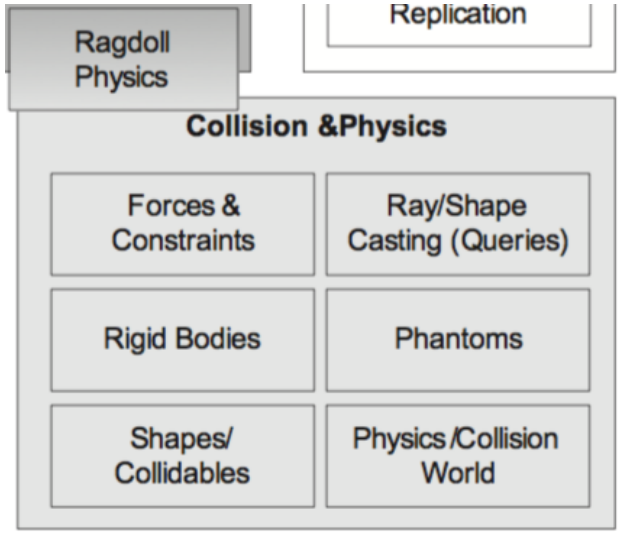
- Pose a 3D character mesh using a simple system of bones
  - ”Skin”
- Bones driven
  - By animation
  - By inverse kinematics
    - Figure out where the joints should be to put the foot *here*
- Blend animations together
  - Shoot while jumping



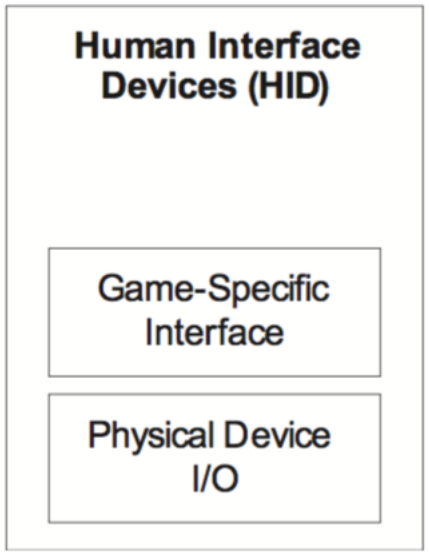
Collisions and Physics

- Handles the simulation of the world
  - Physical behaviours
    - Gravity, laws of motion
    - Kinematics, dynamics
  - Collisions
  - Ragdoll characters
  - Object breaking and destruction
- Physics is increasingly tightly integrated into gameplay
  - Physics-based animation
  - Interaction with objects using physics
  - But limited in simple games
- Physics is hard
  - Off-the-shelf sub-systems – Havok, ODE, PhysX





HID – Human Interface Devices

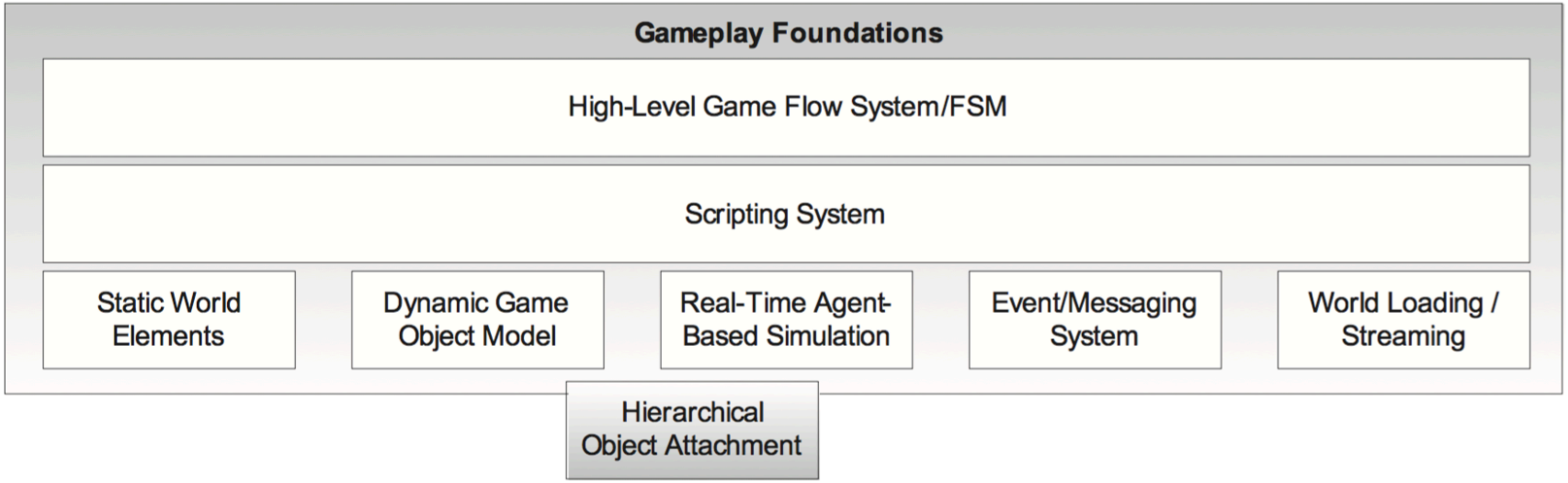


- Abstraction between
  - Low level controller
    - PS4, Xbox controllers
    - Keyboard, mouse
  - High level game actions
    - Jump, fire, move
- Player I/O
  - Includes haptic feedback

Networking

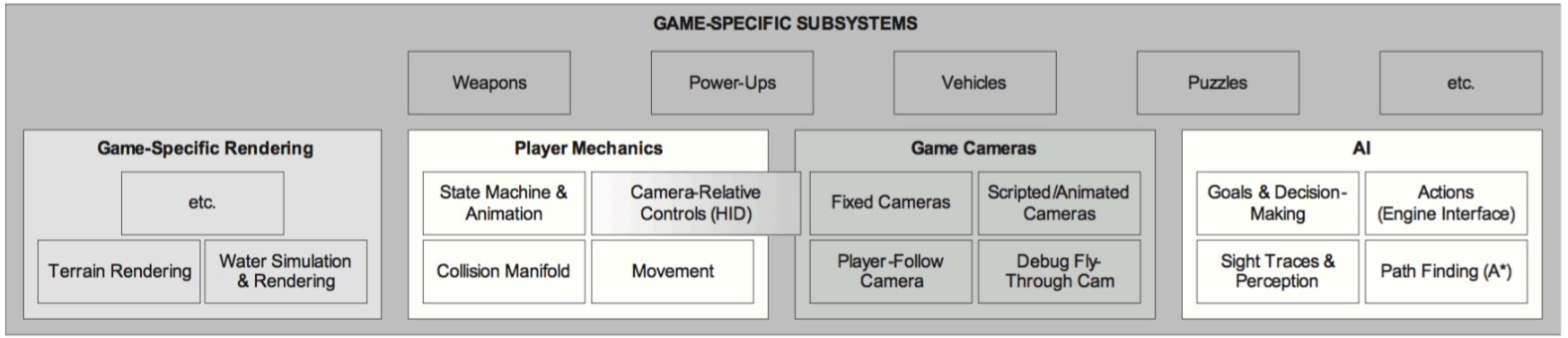
- To allow multiple players to play together within a share world
  - Networked multiplayer
  - Massively multiplayer
- Transfer data between multiple machines
  - How to share data?
  - What data?
  - How to enforce consistency?
- Has a profound impact on the design of a game engine
  - On world modeling, rendering, animation, input, objects

Gameplay / Game State



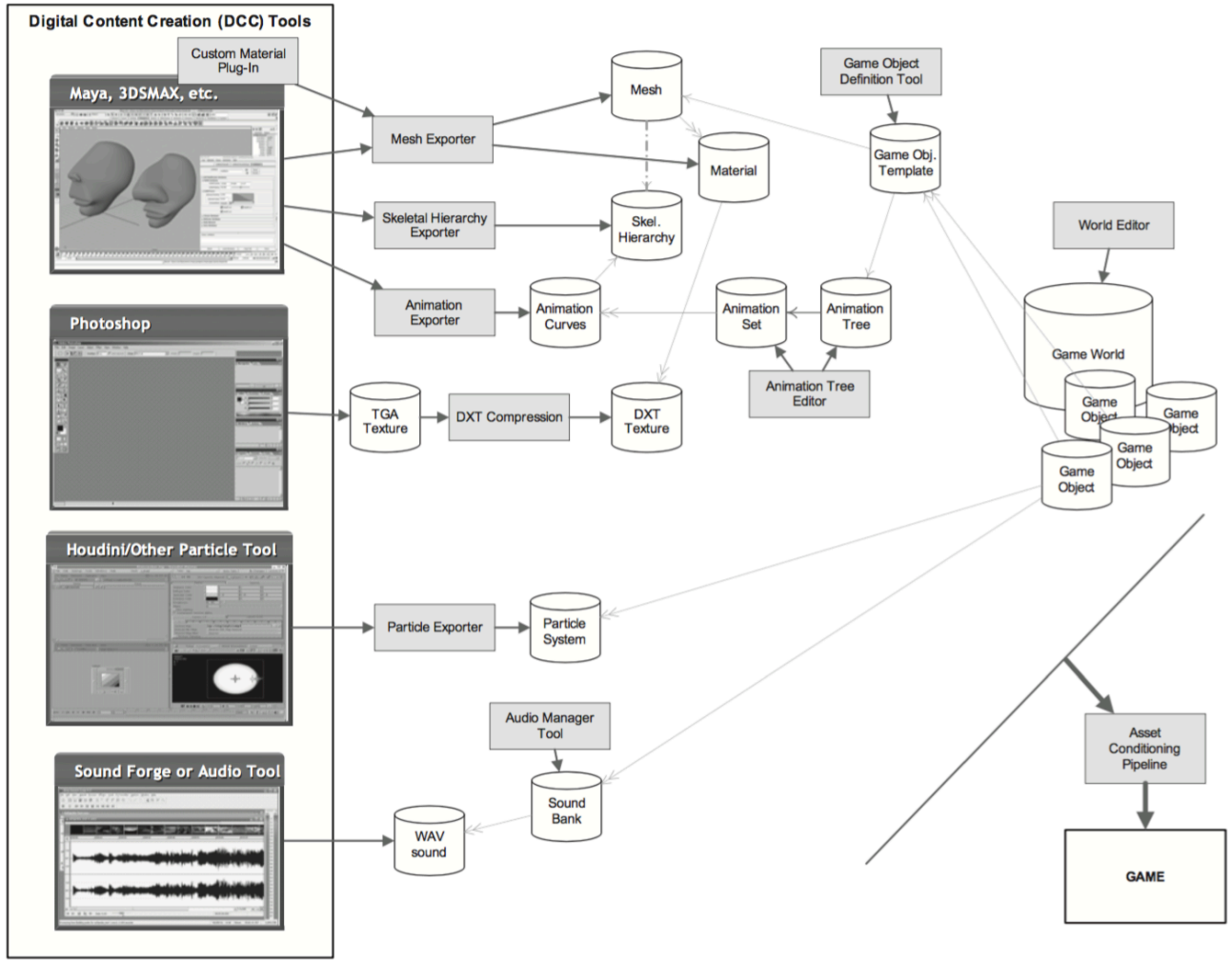
- A game-object model
  - The “state” of the game
    - How is the simulation driven (onTick)
  - What are the core game objects?
    - Actors, created / destroyed
- Pathfinding (navmesh), scripting (blueprint), events
- Levels and game modes
- Promotes “data driven design”

Game Specific



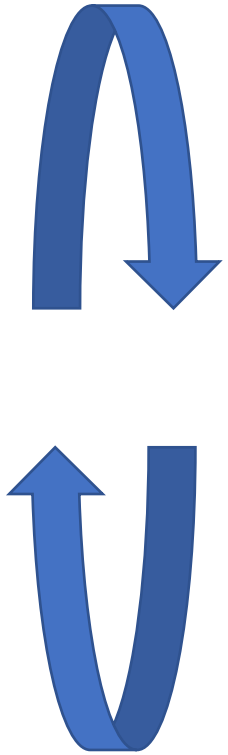
- The line between engine / game
  - Core mechanics
  - Game objects
  - AI Behaviours
  - Implementation of rules
  - …

Asset Pipeline



How do we put it all together?

- User interface
  - Configuration and selection
  - Help
  - HID / HUD / UI
- Game Logic
  - Loading
  - Script
  - Physics Engine
  - Artificial Intelligence
  - Events
  - Collisions
  - Network communication
- Outputs
  - Graphics renderer
  - Sound and music



Player Action

...

System Outcome

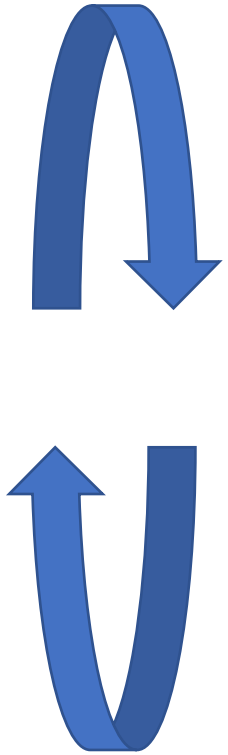
Meaningful play in a game emerges from the relationship between player action and system outcome

Games as Systems

- Game are *soft real-time interactive agent-based computer simulations*
- A Subset of the real world / an imaginary world
  - Modelled mathematically
    - Approximation, simplification
    - Numerical over analytical – can determine the next state of the system
- Simulation
  - Agent-based - Distinct entities interact autonomously
  - Temporal - The model of the world is dynamic, changes over time
  - Interactive - Respond to user interactions
- Deadline driven
  - The screen has to be updated 30 times per second
  - Soft - Missing a deadline is not catastrophic

How do we put it all together?

- User interface
  - Configuration and selection
  - Help
  - HID / HUD / UI
- Game Logic
  - Loading
  - Script
  - Physics Engine
  - Artificial Intelligence
  - Events
  - Collisions
  - Network communication
- Outputs
  - Graphics renderer
  - Sound and music



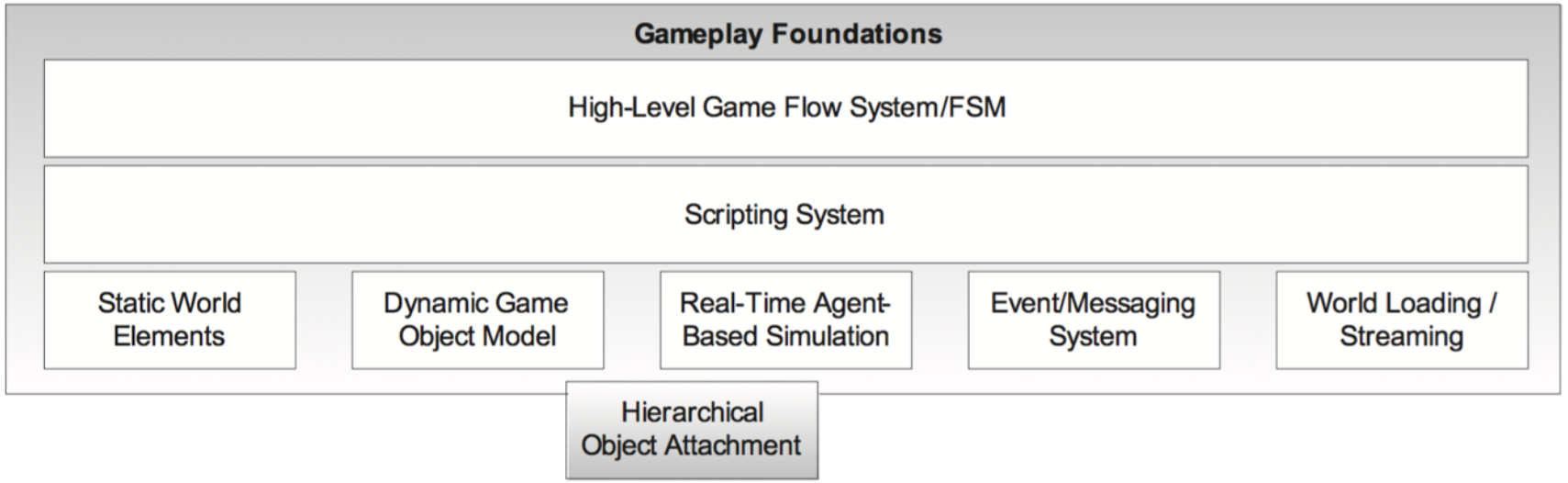
Player Action

...

System Outcome

Meaningful play in a game emerges from the relationship between player action and system outcome

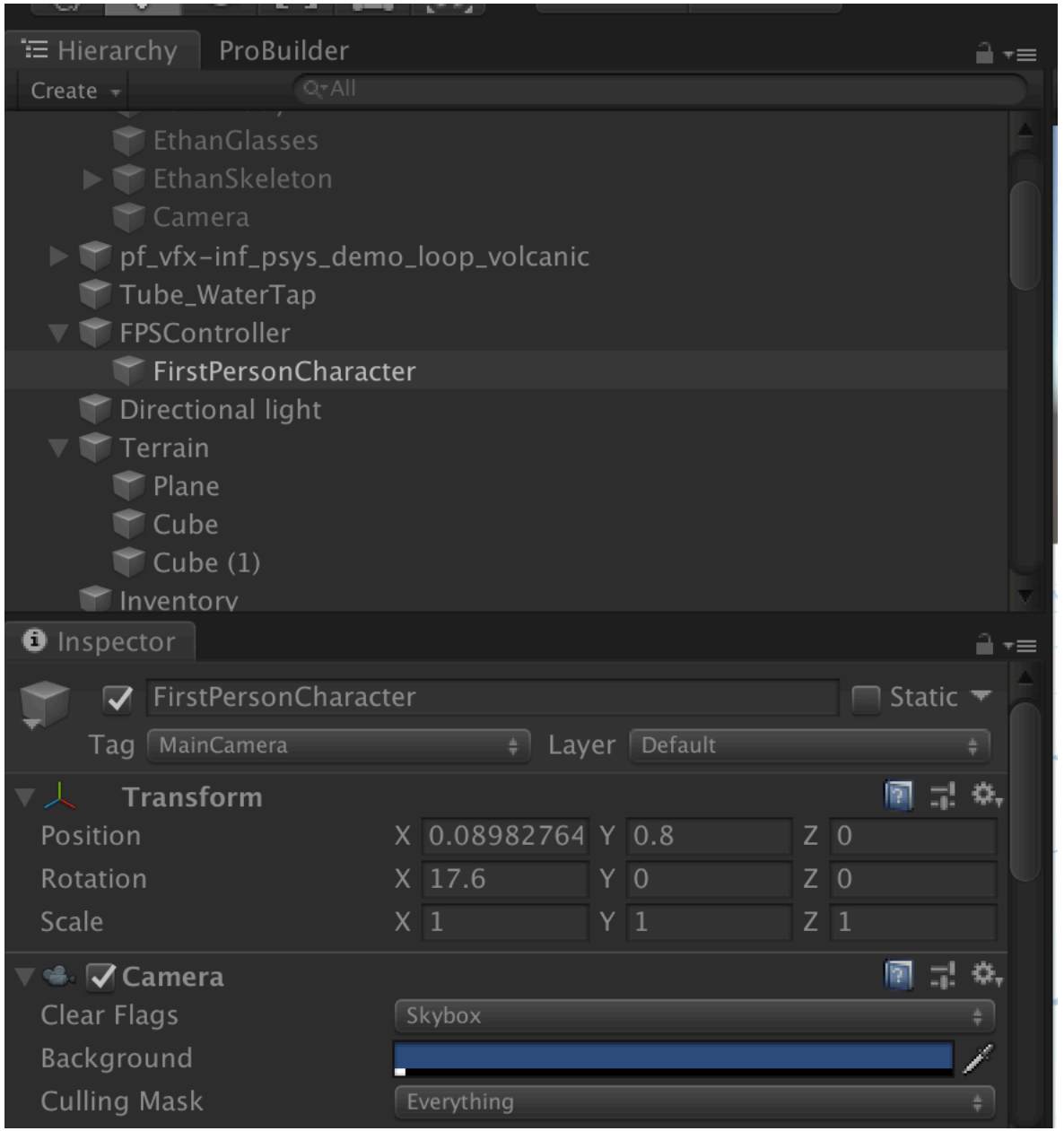
Gameplay / Game State



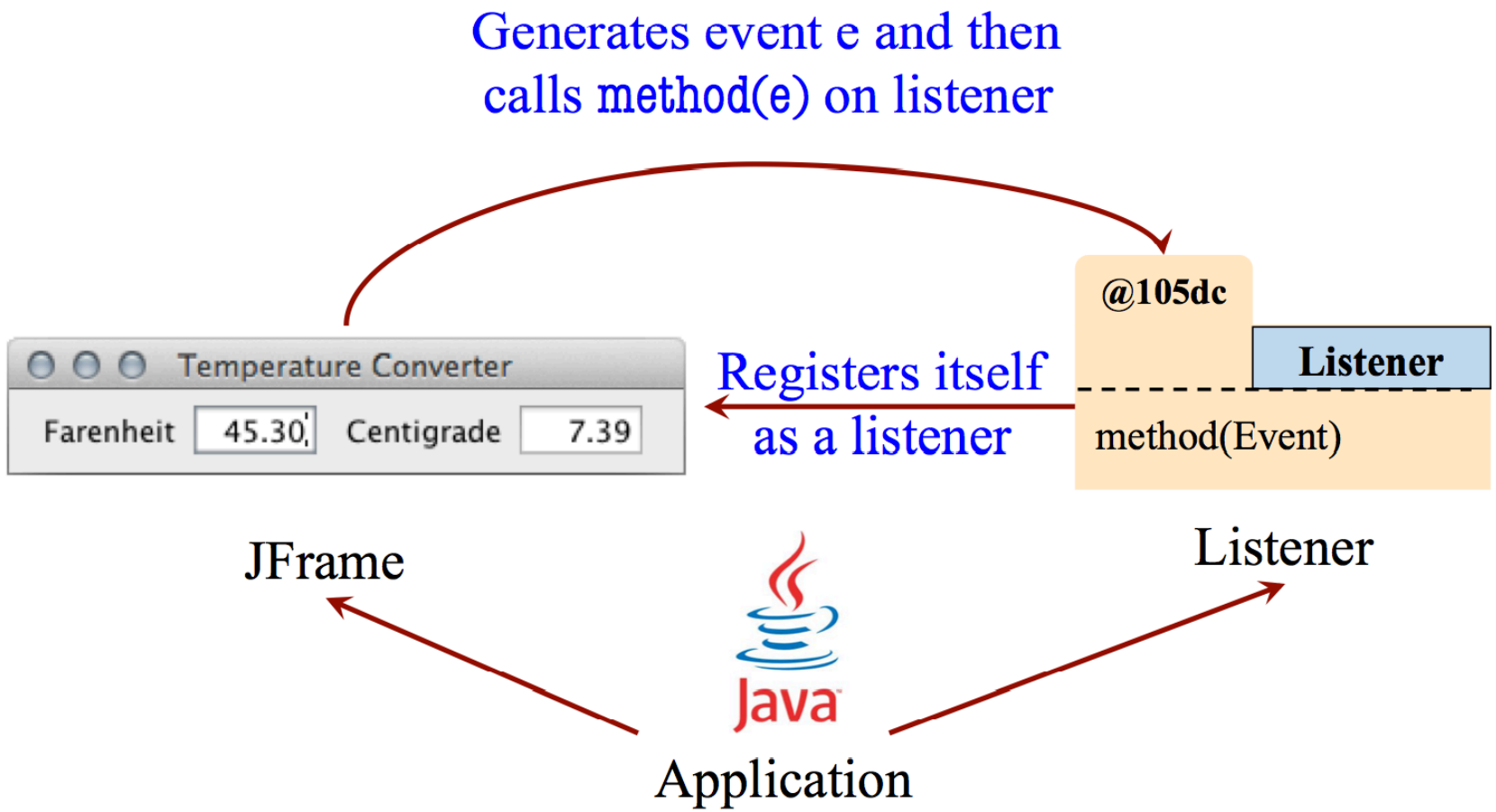
- A game-object model
  - The “state” of the game
    - How is the simulation driven (onTick)
  - What are the core game objects?
    - Actors, created / destroyed
- Pathfinding (navmesh), scripting, events (onTriggerEnter)
- Levels and game modes
- Promotes “data driven design”

How do we put it all together?

- Game State
  - Changing position, orientation, velocity of all dynamic entities
  - Behaviour and intentions of AI controlled characters
  - Dynamic, and static attributes of all gameplay entities
    - Scores, health, powerups, damage levels
- All sub-systems in the game are interested in some ongoing aspect of the game state.
  - Renderer, Physics, Networking, and Sound systems need to “know’ about objects / actors
    - Renderer needs to draw an object at some location
    - Physics needs to check whether A should be allowed to move to B
    - Many systems need to know when a new entity comes into or goes out of existence
    - AI system knows when player is about to be attacked – sound system should play ominous music when this happens


Hierarchy
ProBuilder
Create
All
EthanGlasses
EthanSkeleton
Camera
pf_vfx-inf_psys_demo_loop_volcanic
Tube_WaterTap
FPSController
FirstPersonCharacter
Directional light
Terrain
Plane
Cube
Cube (1)
Inventory
Inspector
FirstPersonCharacter
Static
Tag
MainCamera
Layer
Default
Transform
Position
X 0.08982764
Y 0.8
Z 0
Rotation
X 17.6
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Camera
Clear Flags
Skybox
Background
Culling Mask
Everything

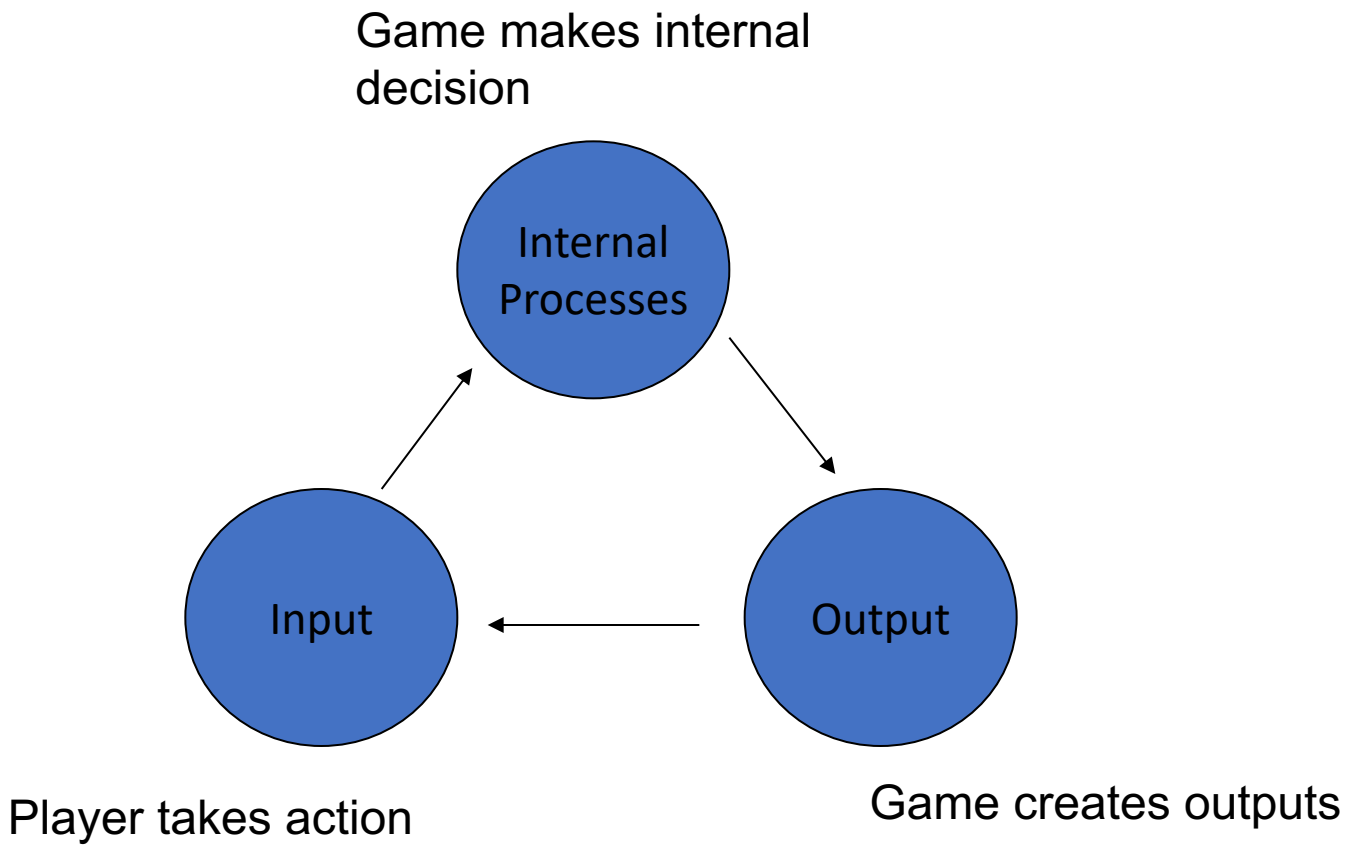
Event Driven Design?



Event Driven Design

- Not entirely useful for our *game engine*
  - Fundamentally everything receives events all the time
  - But entirely useful for higher level scripting mechanics
- Program only reacts to user input
  - Nothing changes if the user does nothing
  - Desired behaviour for productivity apps
    - Word etc.
    - Selective rectangle invalidation / redrawing of part of the screen
- Games generally continue without input
  - *Simulation*
    - Characters animate
    - Clock timers
    - Enemy AI
    - Physics simulation

Interactive System



Time and “The Game Loop”

- The “heart beat” of a game
  - Master loop that services all subsystems
- Performs a series of tasks every **frame**
  - Game state changes over time
  - Each rendered frame is a snapshot of the evolving game state
    - Determine the current state as different from the previous state
  - Illusion of motion is obtained by a high frequency rendering loop
    - E.g. 30 frames per second
- Run as fast as we can
  - A smooth game-play experience

The Game Loop (Simplified)

start game

while(user doesn't exit)

{

get user input

get network messages

simulate game world

for each object, update it – where should it be?

move objects <- when an object is allowed to move

resolve collisions <- which objects have collided - are they allowed to be where they are?

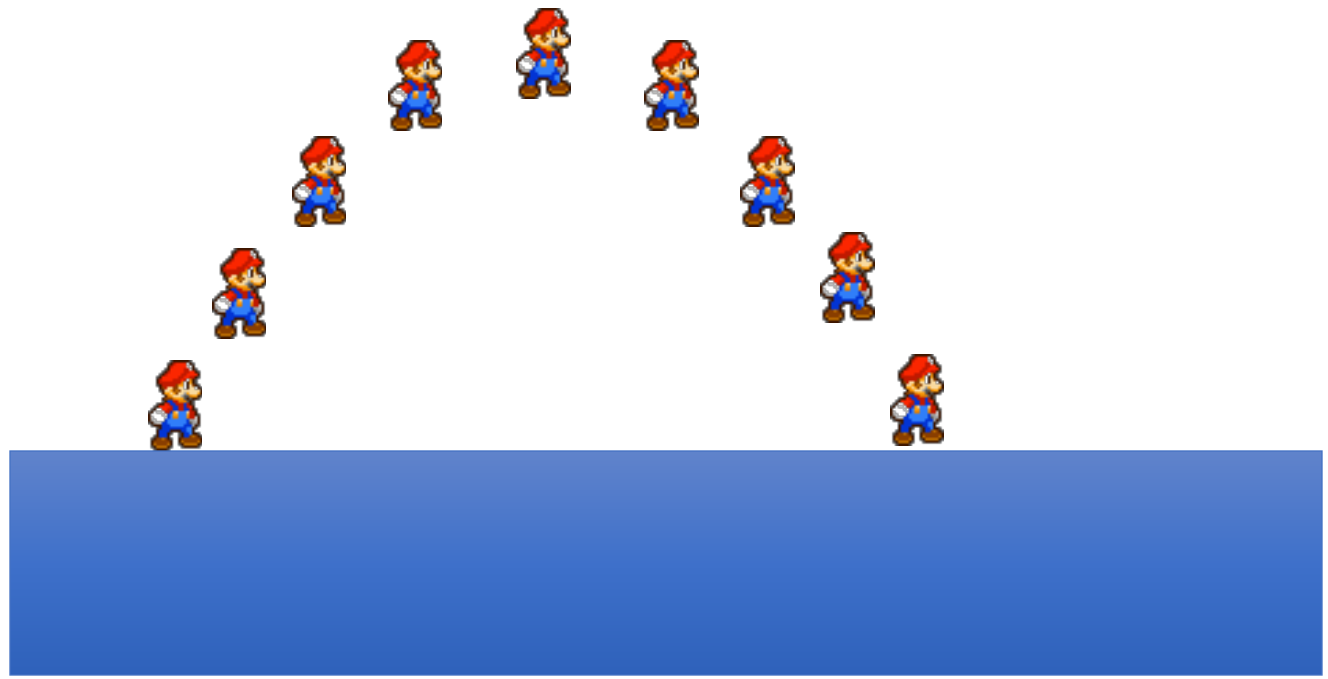
draw graphics

for each object, draw it

play sounds

}

exit




EVGA
GTX 580
1200W
Silent Pro Gold


TURBO
H.DISK
POWER
188
MHz
TURBO
RESET

The Game Loop – Naïve approach

```
start game
while( user doesn't exit )
{
    get user input
    get network messages
    simulate game world
    resolve collisions
    move objects
    draw graphics
    play sounds
}
exit
```

- Movement is CPU dependent
  - Pixels per frame

The Game Loop – Elapsed Time

start game

while(user doesn't exit)

{

how much time has elapsed?

get user input

get network messages

simulate game world(**elapsed time)**

resolve collisions

move objects

draw graphics

play sounds

}

exit

- Simple numeric integration
  - The position of each game object is a function of its position and the first time derivative at the current time t
- $\Delta x = v\,\Delta t$
- $x2 = x1 + \Delta x$
- $x2 = x1 + v\Delta t$
- Issue
  - Determine a suitable value for Δt (elapsed time)

The Game Loop – Running Average

start game

while(user doesn't exit)

{

how much time has elapsed?

update running average

get user input

get network messages

simulate game world(**average elapsed time)**

resolve collisions

move objects

draw graphics

play sounds

}

exit

- Elapsed time
  - An *estimate* of the **previous** frame used to calculate the **next** frame
  - Susceptible to frame rate spikes
    - Camera looks at complicated scene
    - Frame rate drops
- Calculate running/rolling average elapsed time
  - Over the last few frames
  - Soften impact of framerate spikes
  - Adapt to changing framerate

The Game Loop – Governed

start game

while(user doesn't exit)

{

how much time has elapsed?

get user input

get network messages

simulate game world(**elapsed time)**

resolve collisions

move objects

draw graphics

play sounds

sleep(desired-elapsed time)

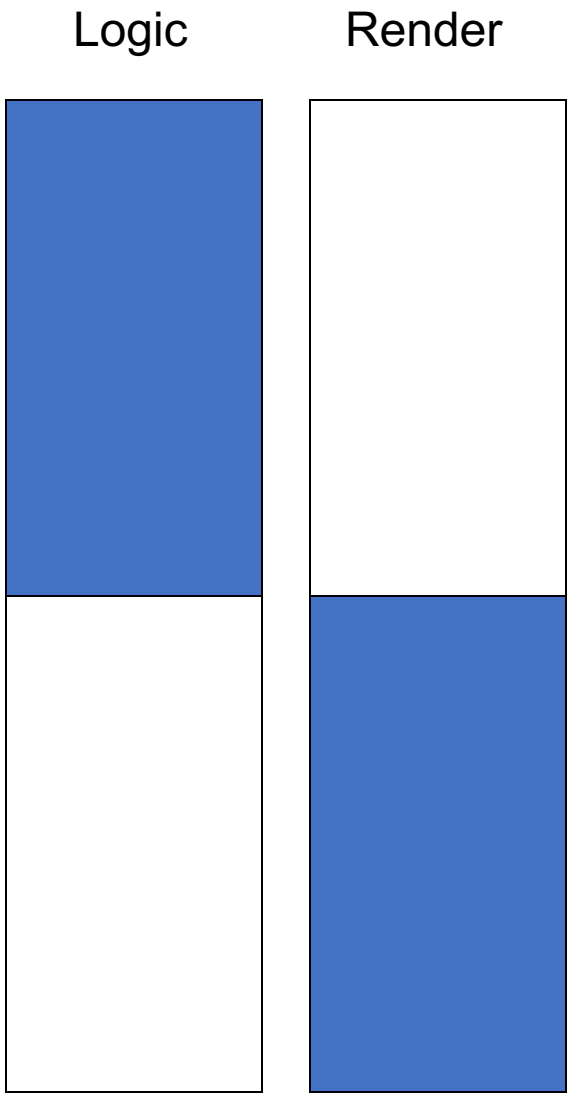
}

exit

- Fixed Framerate
  - Guarantee frame rate
    - 33.3ms (30 fps)
    - 16.6ms (60 fps)
- Measure duration of frame
  - Less than target
    - Sleep until next frame
  - Greater than target
    - Skip a frame
- Why?
  - Numerical integration might require fixed step (physics)
  - Avoid tearing / vSync
    - Image is updated at the same time as a monitor refresh

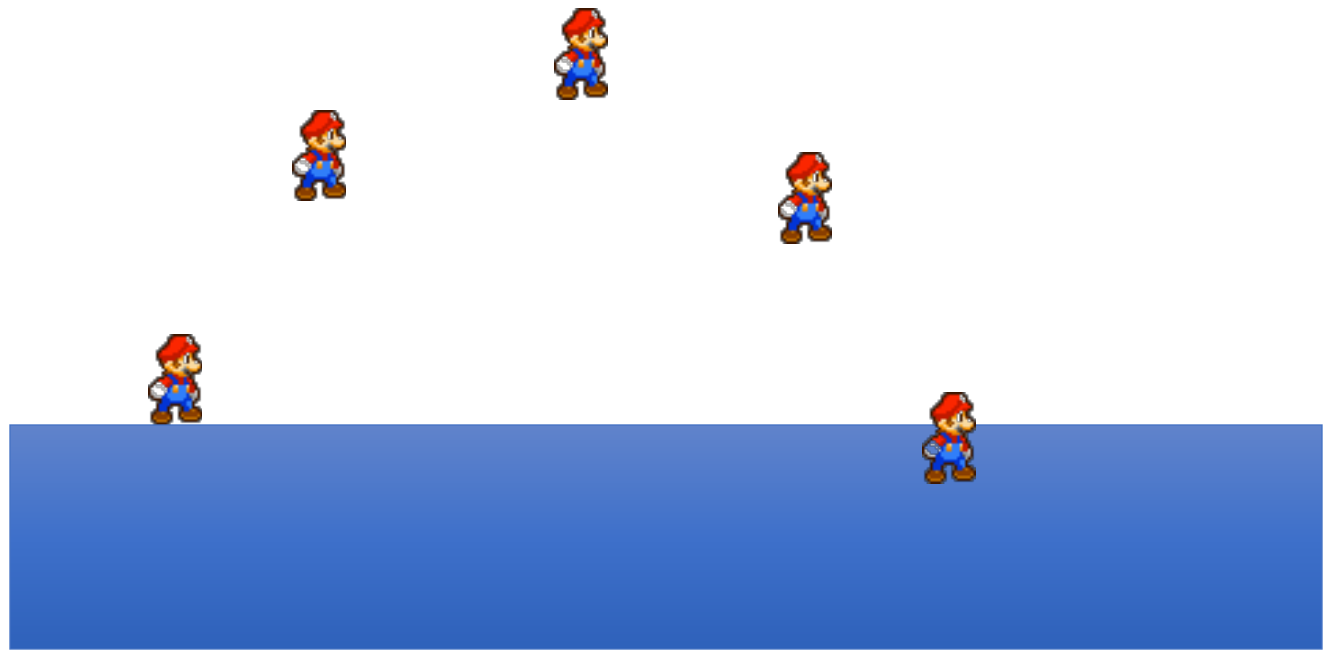
The Game Loop (attempt 1)

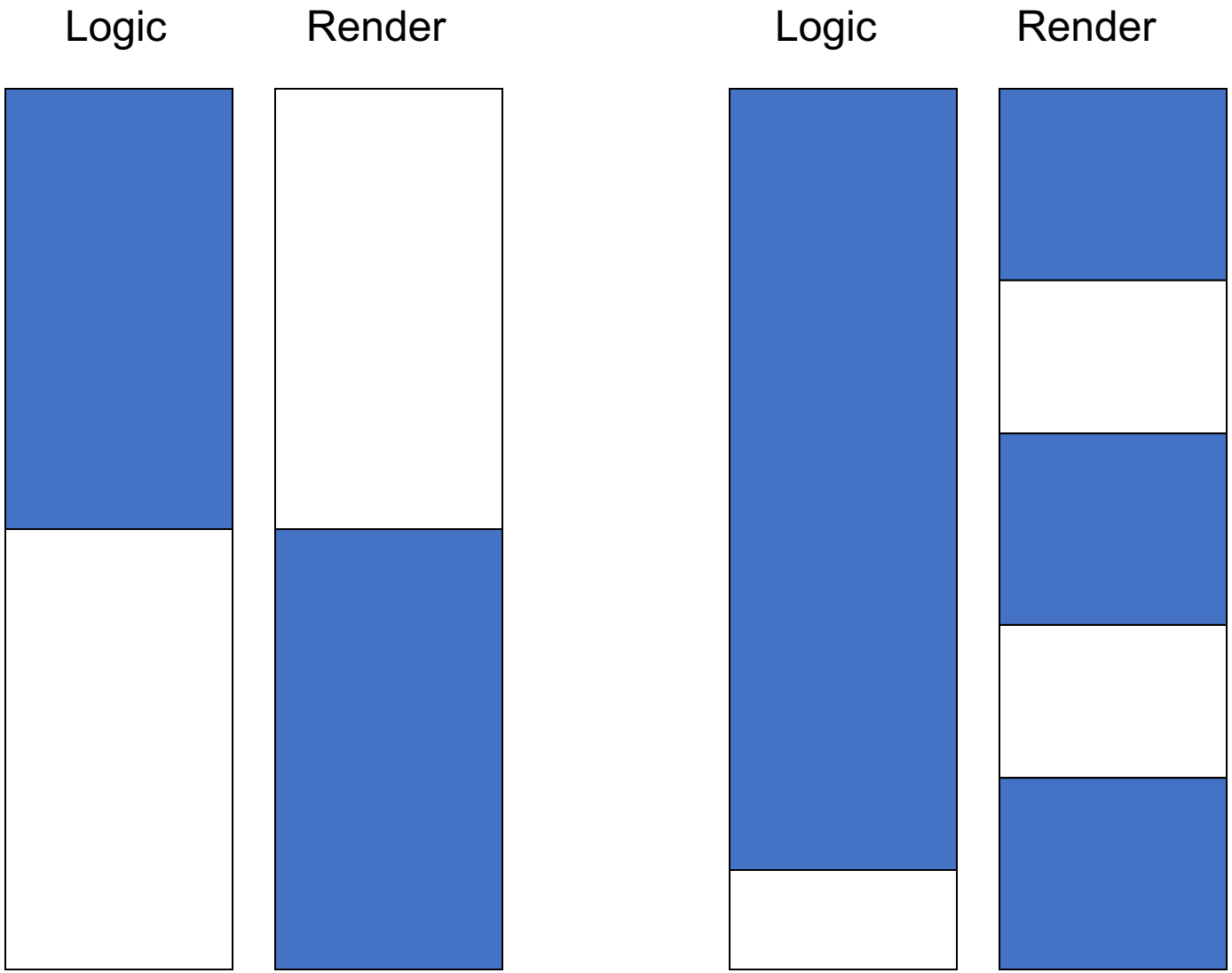
- Simplest case – two routines **coupled**
  - Input, Update / Logic
  - Rendering
- Advantages of the *coupled* approach
  - Both routines are given equal priority
  - Logic and presentation are fully coupled
    - Easier to code, no concurrency
- Disadvantages
  - Variation in complexity in one of the routines affects the other one
  - No control over how often a routine is updated


Logic
Render

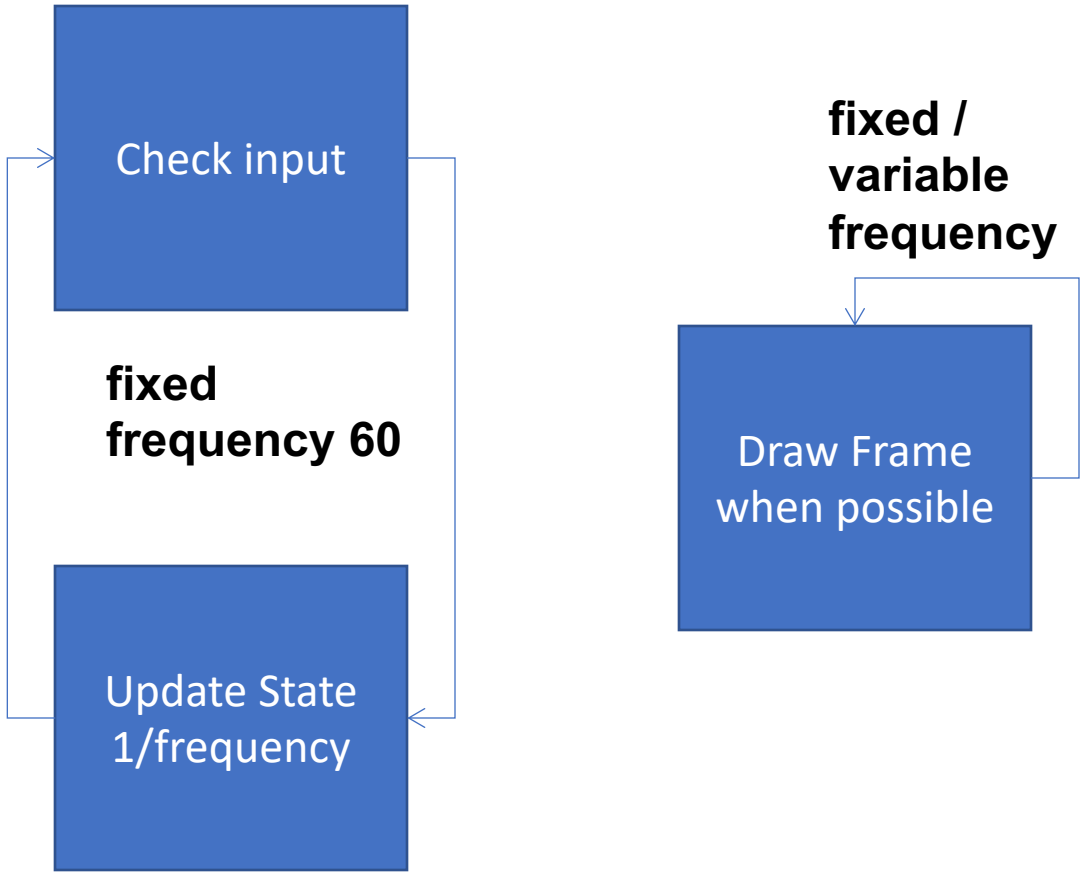
Time Constraints

- Graphics rendering must happen at ~30 FPS to achieve the illusion of motion
- Frequency of other subsystems may be different
  - AI (~10), input (~40), audio (~50), physics (~60), haptic feedback (~3000)
- The game engine services these subsystems
  - In charge of calling the components at the correct time




Logic
Render
Logic
Render

Decoupling


Check input
fixed
frequency 60
Update State
1/frequency
fixed /
variable
frequency
Draw Frame
when possible

The Game Loop (attempt 2)

- Multiple threads, decoupled
- Advantages of the multi-threaded approach
  - Both update and render loops run at their own frame rate
- Disadvantages
  - Not all machines are that good at handling threads
    - Single CPU, precise timing problems
  - Synchronization issues
    - Two threads accessing the same data

The Game Loop (attempt 2)

start game
start render thread
while(user doesn't exit)
{
    how much time has elapsed?
    get user input
    get network messages
    simulate game world(**elapsed time)**
    resolve collisions
    move objects
    play sounds
    sleep(desired-elapsed time)
}
exit

while(user doesn't exit)
{
    draw graphics
    sleep(desired-elapsed time)
}
exit

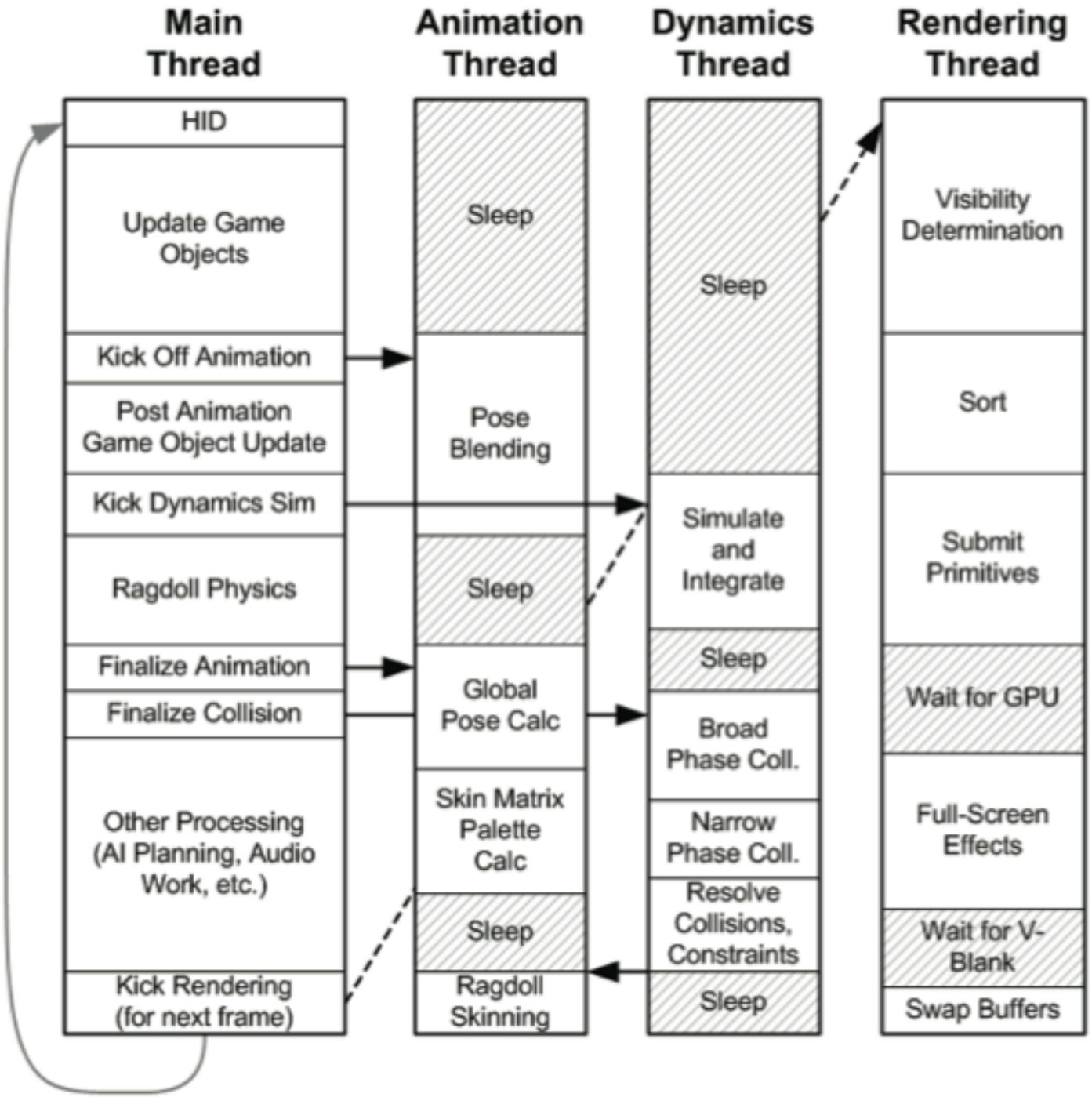
The Game Loop (attempt 3)

- Game Loop as Scheduler
- Advantages of the frequency dependent single - threaded decoupled approach
  - Allow an individual frequency for each entity in the game
    - Single thread per sub-system
  - Same mechanism can be applied to rendering
  - Generic automatic registration mechanism
- Disadvantages
  - Need to specify the frequency ‘manually’ for each entity
    - Relatively coarse-grained chunks of work
    - Can place restrictions on how various processors are utilised
  - The game engine needs an entry point for each entity to update (might be large)

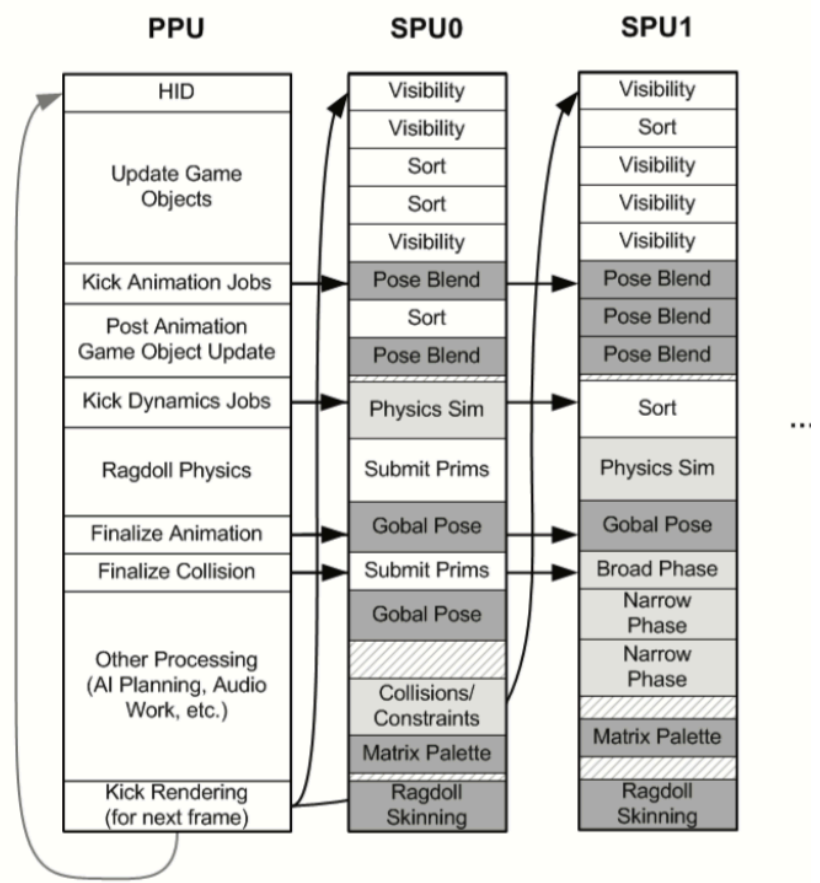
The Game Loop (attempt 3)

```
start game
Physics.setFrequency(60);
AI.setFrequency(10);
while( user doesn't exit )
{
    get user input
    get network messages
    Physics.update();
    AI.update();
    // rendering frequency is “as fast as possible”
    Renderer.render();
    lastCall = getTime();
}
exit
```

- One thread per subsystem
  - Relatively isolated repeating subsystems
- Main game loop thread
  - Controls and synchronises subsystem threads


Main Thread
Animation Thread
Dynamics Thread
Rendering Thread
HID
Update Game Objects
Kick Off Animation
Post Animation Game Object Update
Kick Dynamics Sim
Ragdoll Physics
Finalize Animation
Finalize Collision
Other Processing (AI Planning, Audio Work, etc.)
Kick Rendering (for next frame)
Sleep
Pose Blending
Sleep
Global Pose Calc
Skin Matrix Palette Calc
Sleep
Ragdoll Skinning
Sleep
Simulate and Integrate
Sleep
Broad Phase Coll.
Narrow Phase Coll.
Resolve Collisions, Constraints
Sleep
Visibility Determination
Sort
Submit Primitives
Wait for GPU
Full-Screen Effects
Wait for V-Blank
Swap Buffers

Job Architecture (PS3)



- Parallel hardware
  - PS3 cell processor
  - Scale out rather than scale up
- Maximise processor utilisation?
  - Main game loop runs on modest PPU
  - SPUs used as job processors
    - Jobs are fine grained and independent

The Game Loop

- What if the time between two loops is significantly larger than the required frequency of any component?
  - Do nothing, the game is slowed down
  - Update the game logic according to the actual time elapsed since the last call
    - Visual gaps
- Solutions
  - Speed-up update
    - Decrease update frequency, use LoD
  - Speed-up rendering
    - Use graphics LoD, perform fewer special effects etc
      - Delegate decision to player

References

- Game Engine Architecture, Jason Gregory 2014, chapters 1, 7