G54GAM Games

Building Games

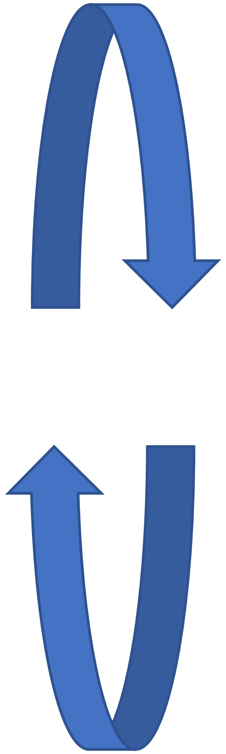
Game Loops

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



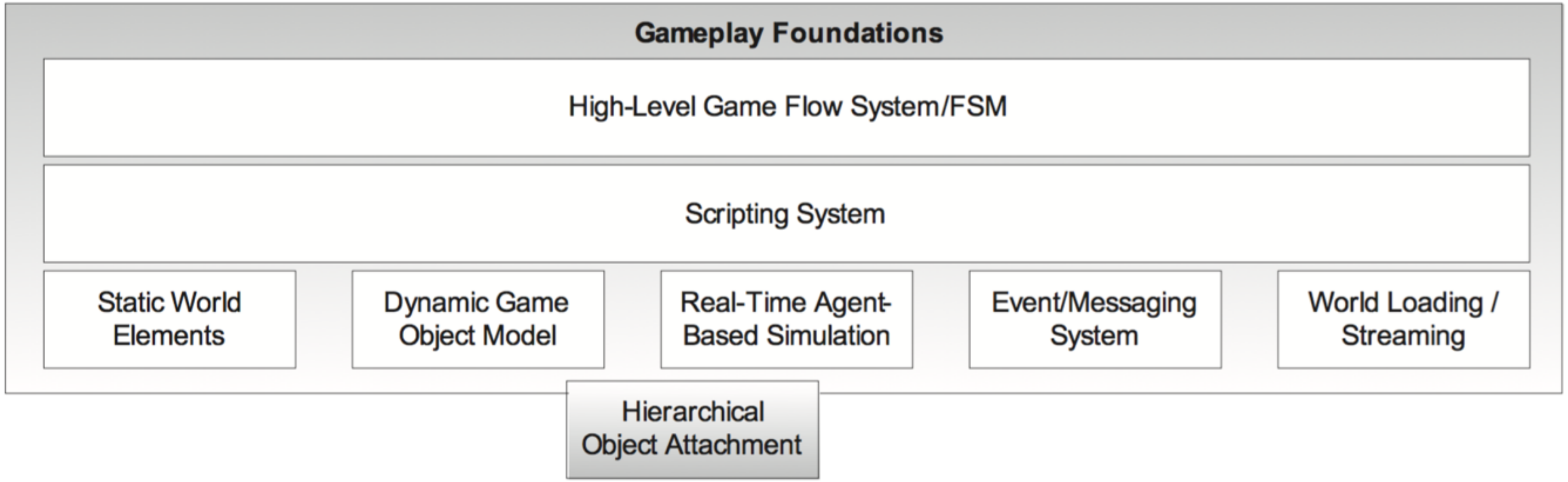
Player Action

...

System Outcome

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

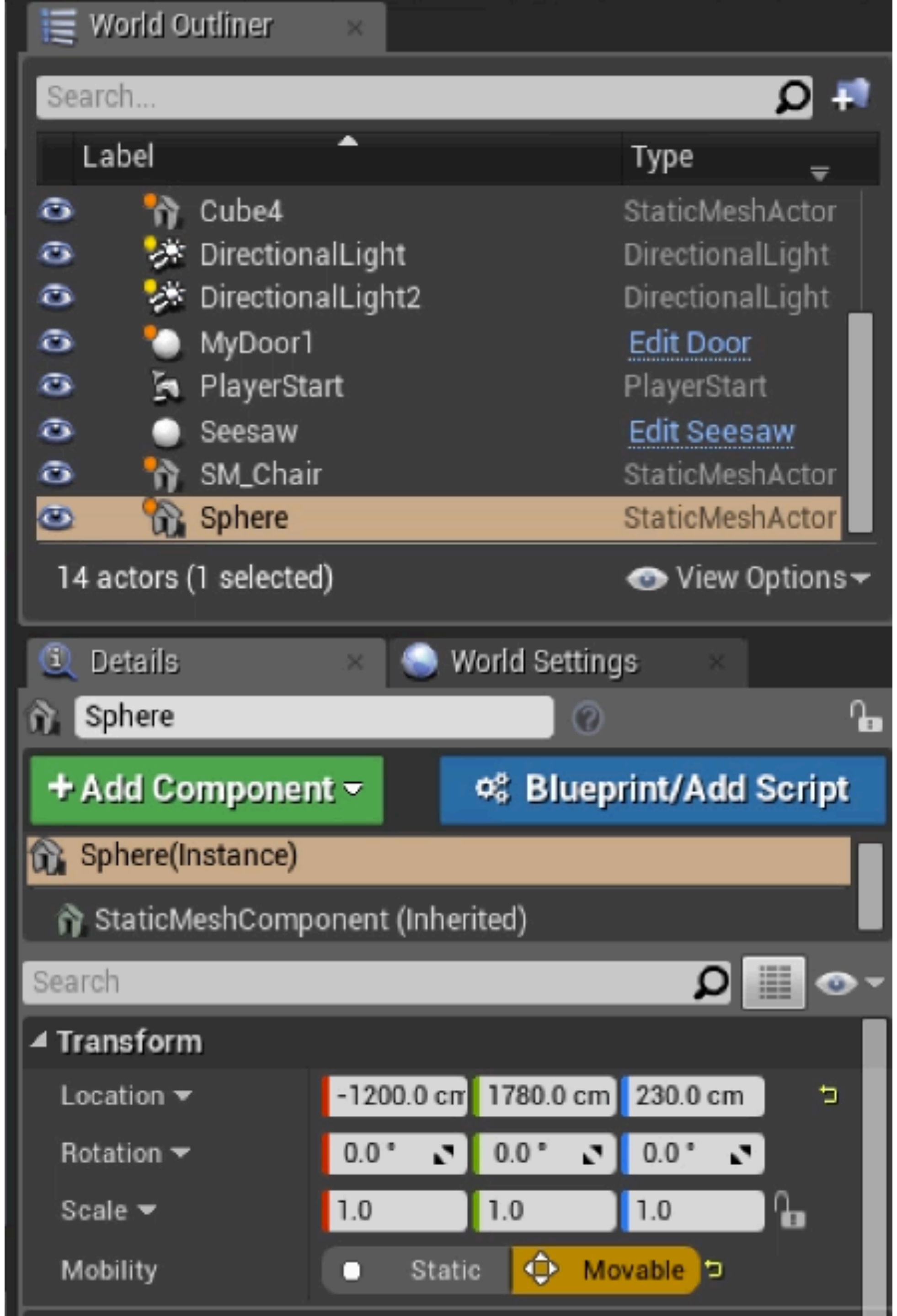
Gameplay / Game State



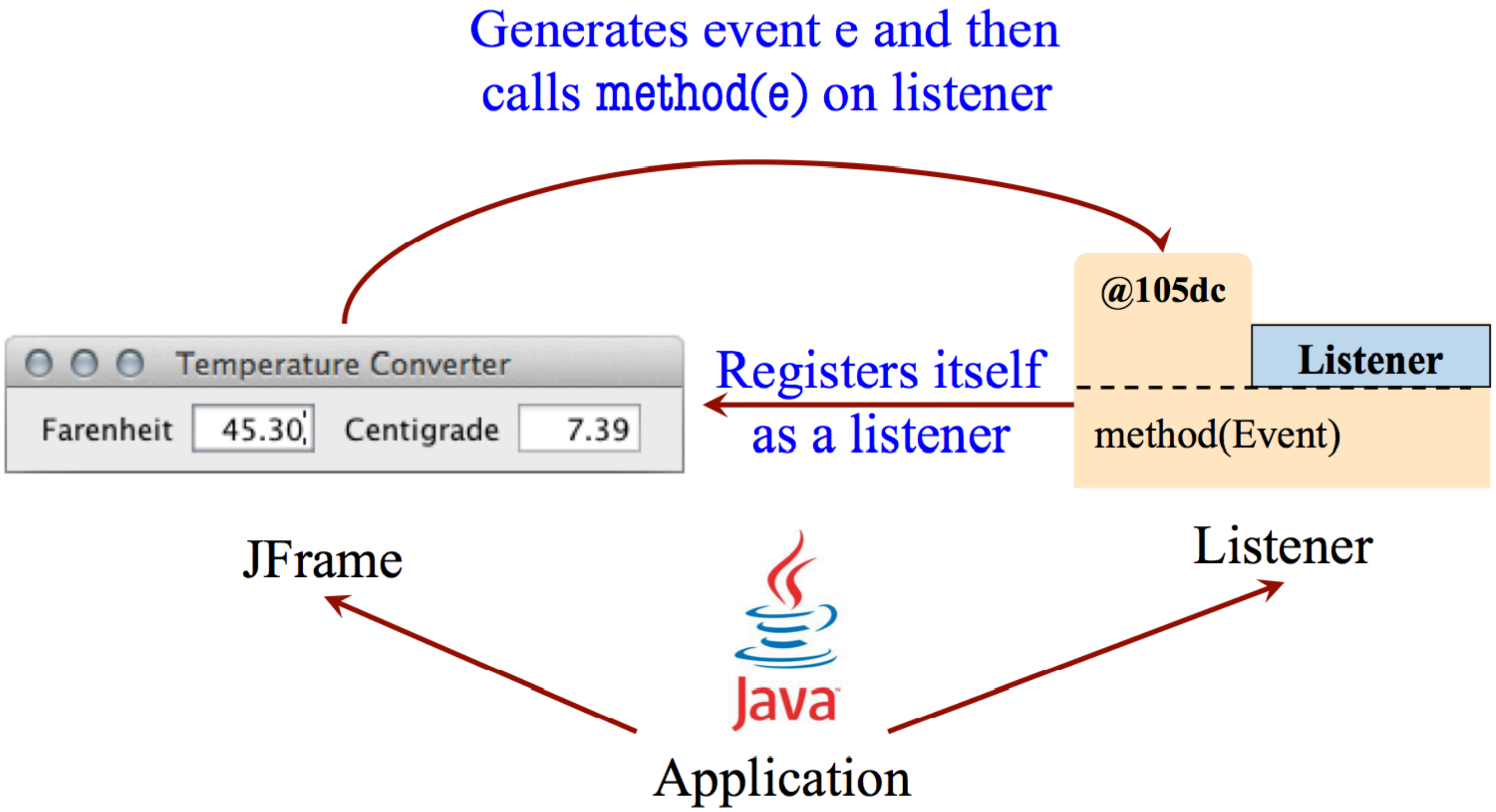
- A game-object model
  - The “state” of the game
    - How is the simulation driven (onTick)
  - What are the core game objects?
    - Actors, created / destroyed
- Pathfinding (navmesh), scripting (blueprint), events
- Levels and game modes
- Promotes “data driven design”

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


World Outliner
Search...
Label
Type
Cube4
StaticMeshActor
DirectionalLight
DirectionalLight
DirectionalLight2
DirectionalLight
MyDoor1
Edit Door
PlayerStart
PlayerStart
Seesaw
Edit Seesaw
SM_Chair
StaticMeshActor
Sphere
StaticMeshActor
14 actors (1 selected)
View Options
Details
World Settings
Sphere
+ Add Component
Blueprint/Add Script
Sphere(Instance)
StaticMeshComponent (Inherited)
Search
Transform
Location
-1200.0 cm
1780.0 cm
230.0 cm
Rotation
0.0 °
0.0 °
0.0 °
Scale
1.0
1.0
1.0
Mobility
Static
Movable

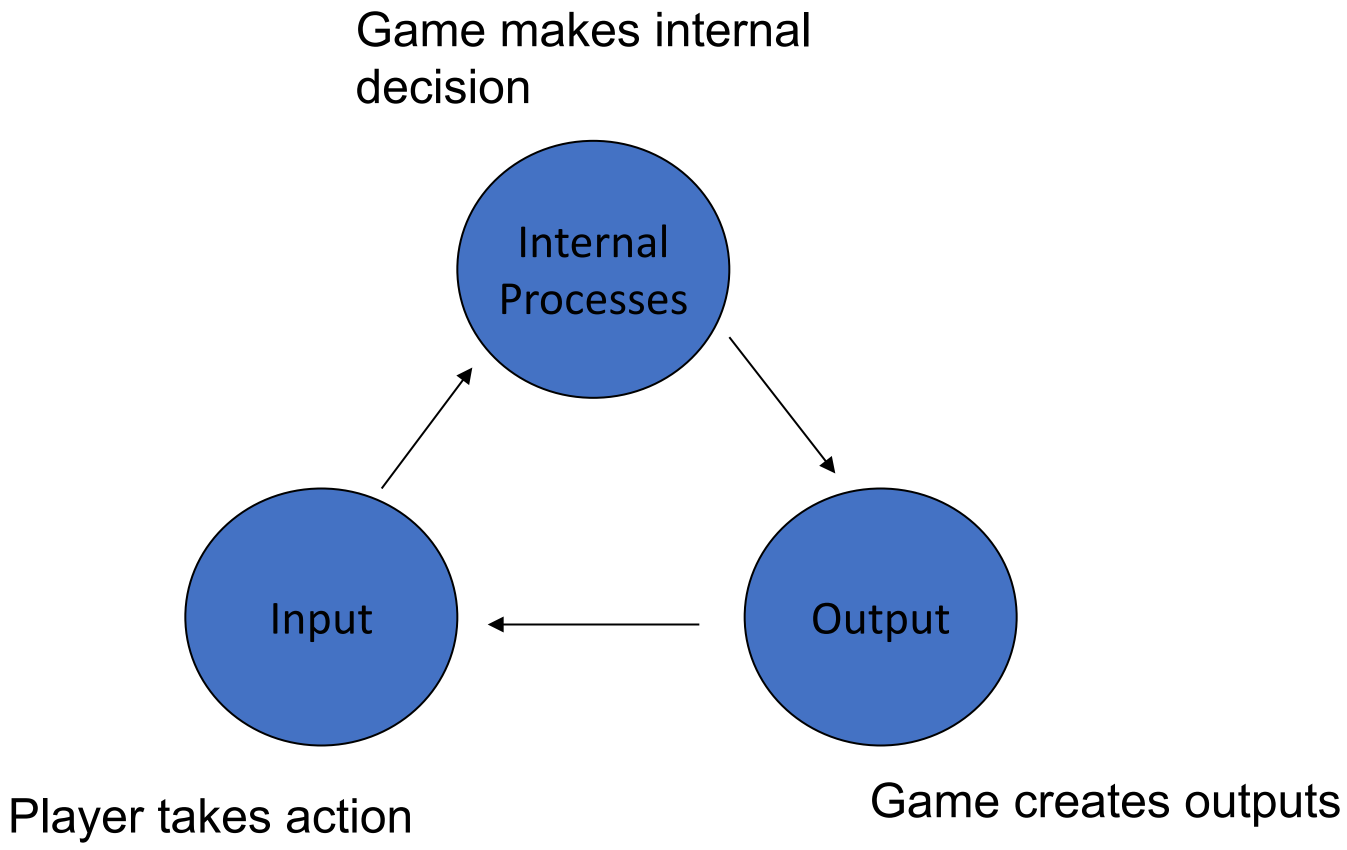
Event Driven Design?



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?

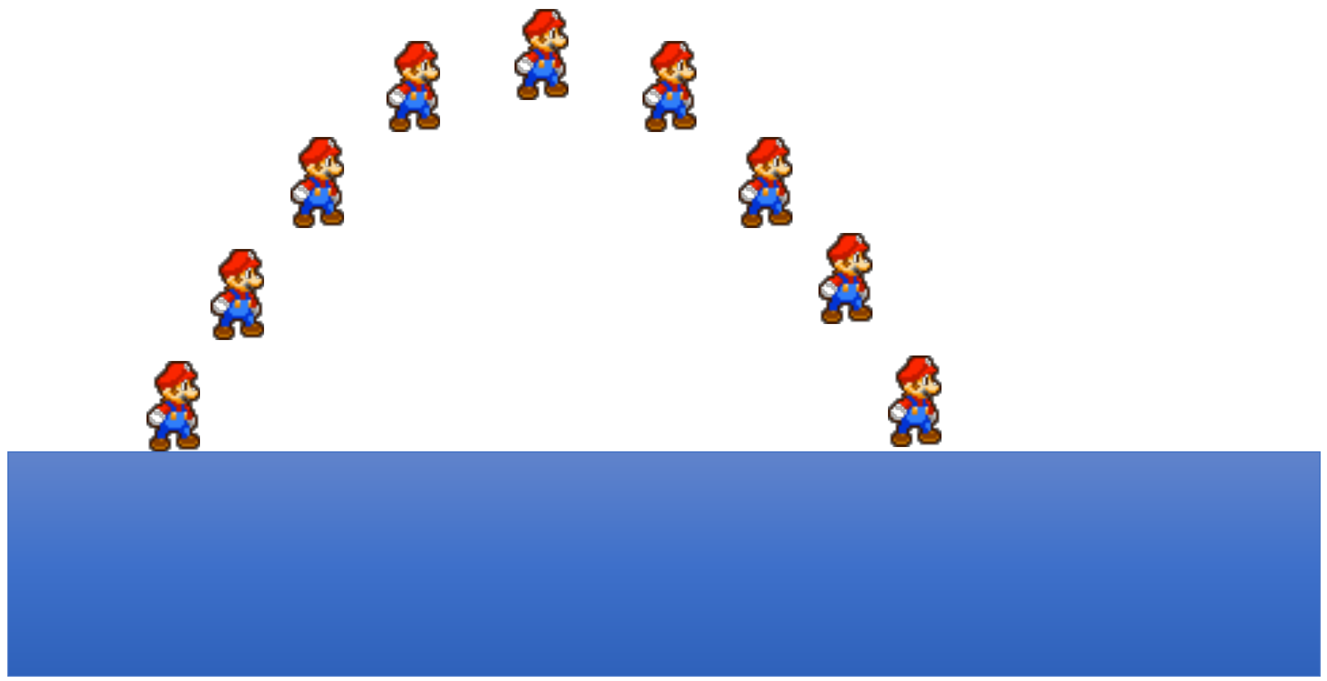
draw graphics

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)

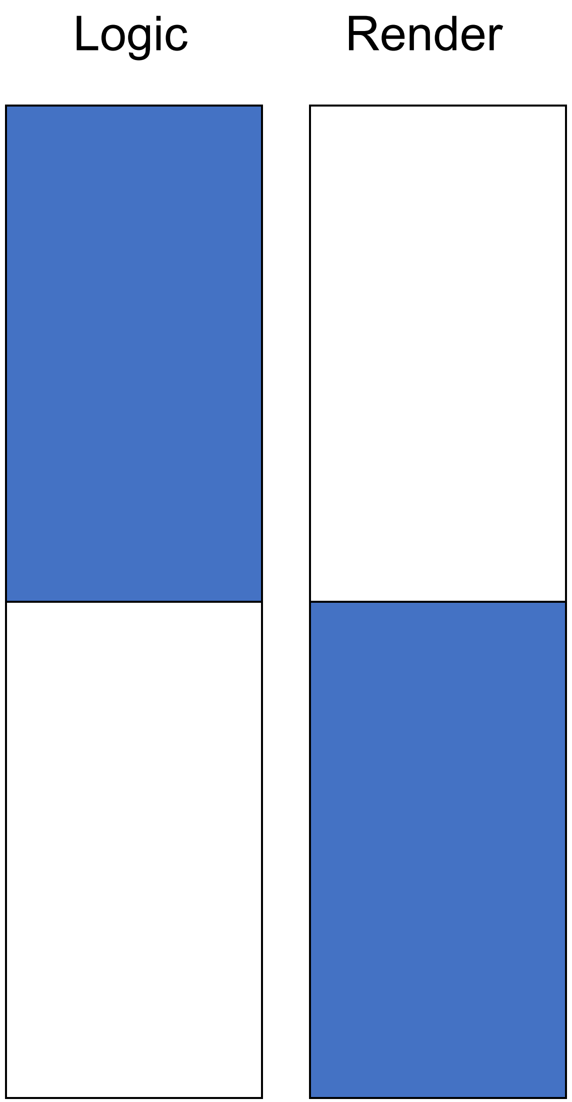
}

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

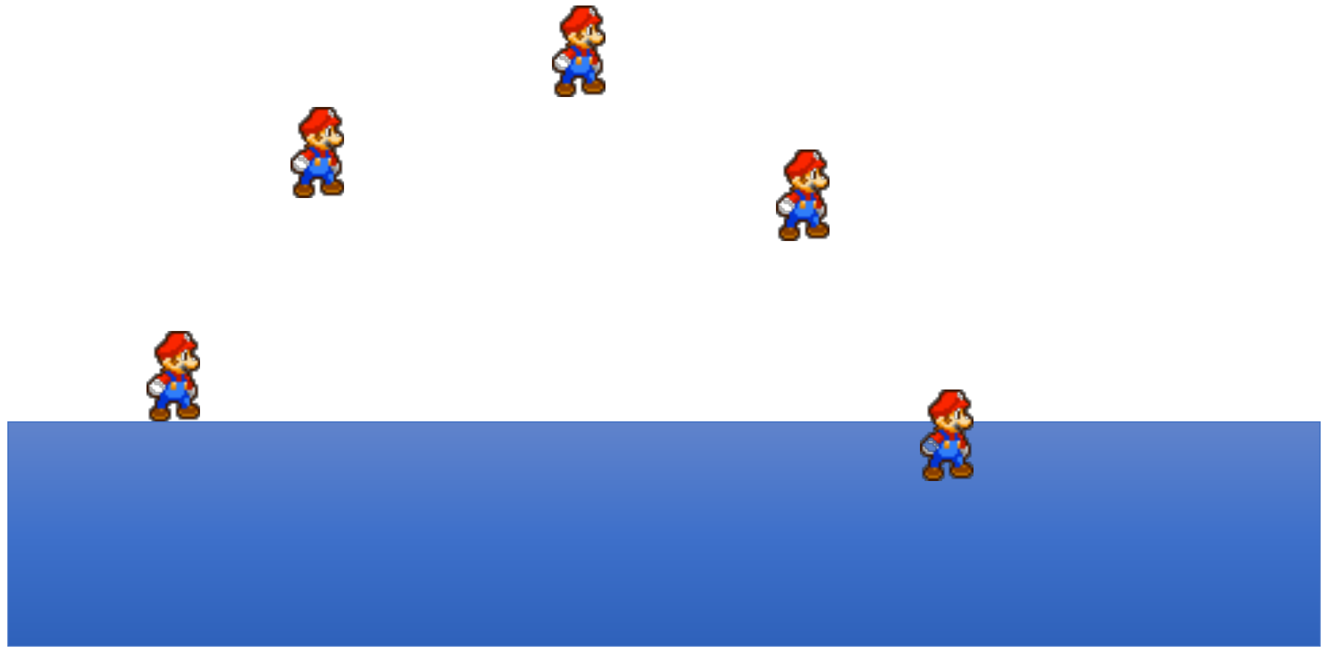
The Game Loop (attempt 1)

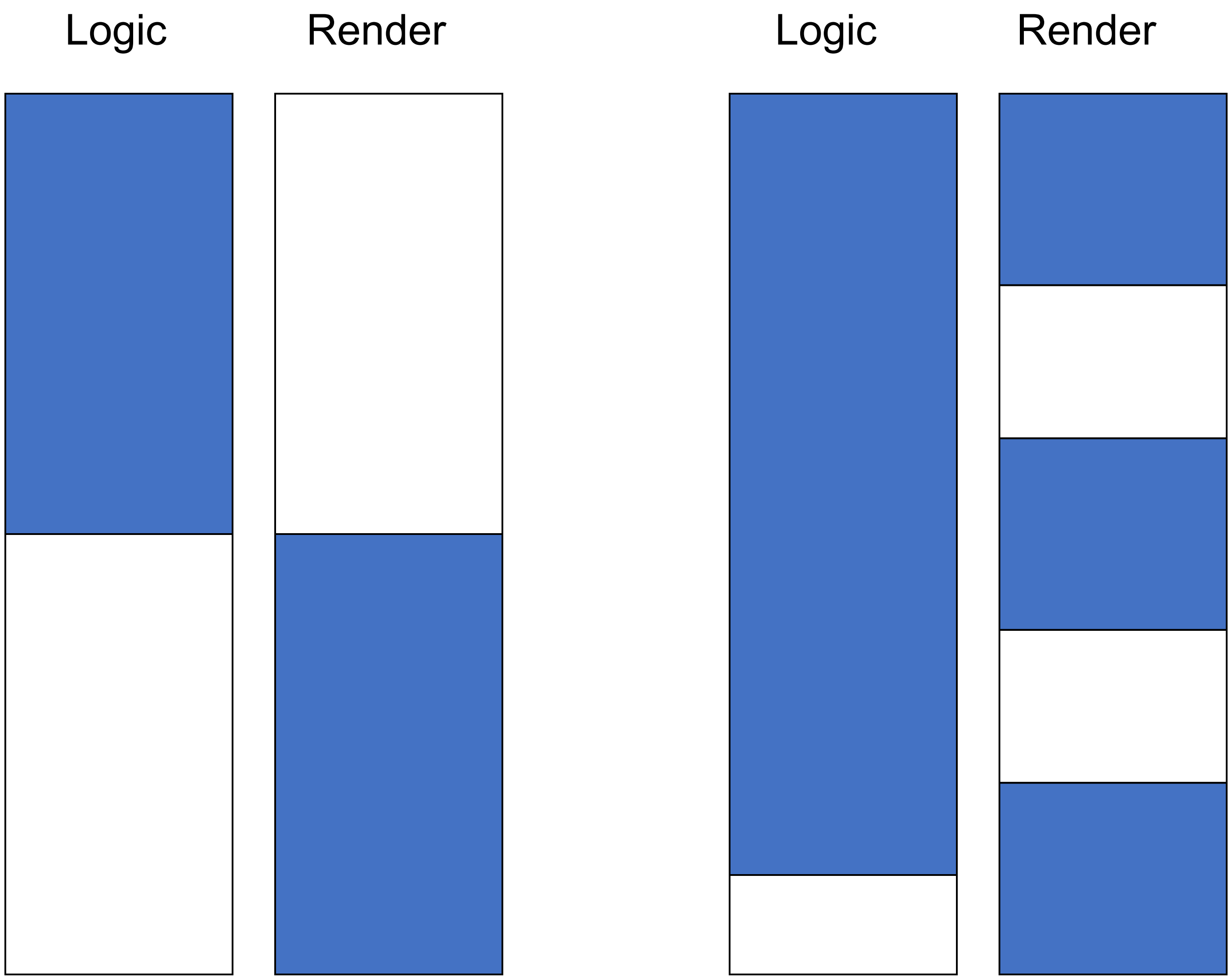
- Simplest case – two routines **coupled**
  - Input, Update / Logic
  - Rendering
- Advantages of the *coupled* approach
  - Both routines are given equal priority
  - Logic and presentation are fully coupled
    - Easier to code, no concurrency
- Disadvantages
  - Variation in complexity in one of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


Logic
Render

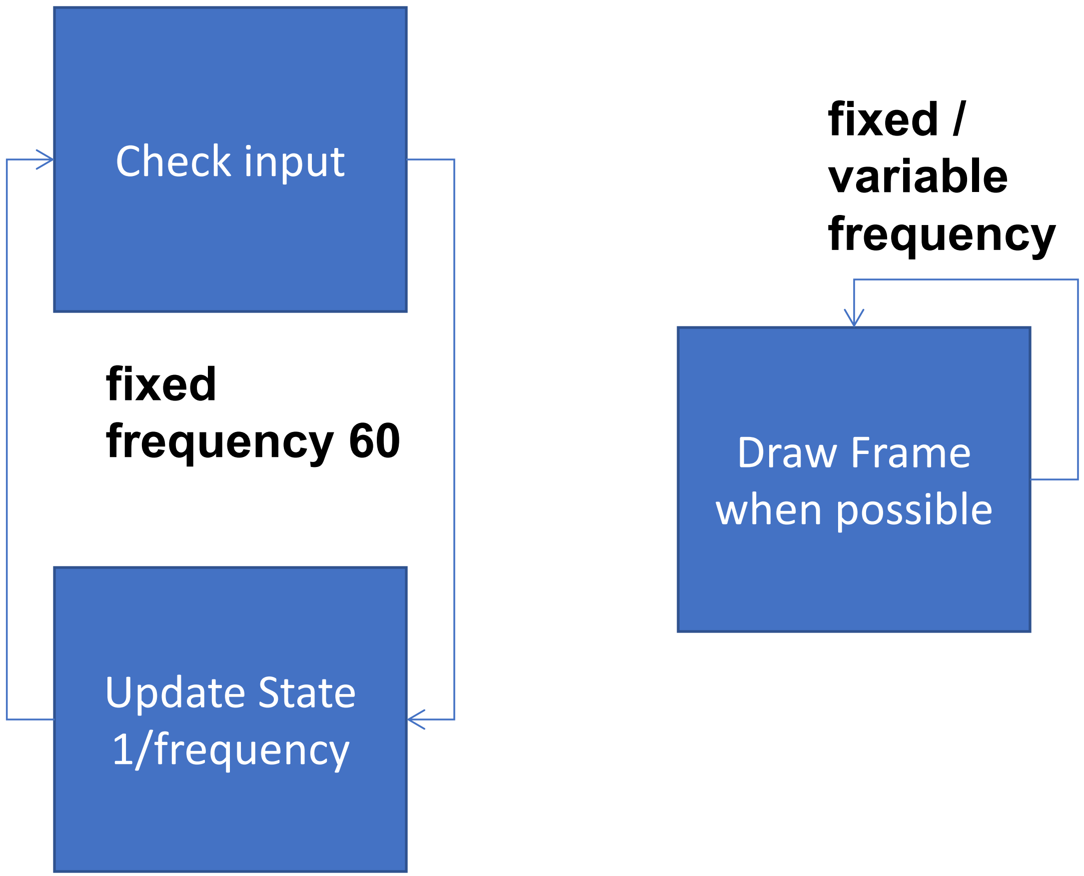
Time Constraints

- 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

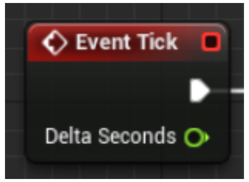
  move objects

  play sounds

  sleep(desired-elapsed time)

}

exit



while(user doesn't exit)

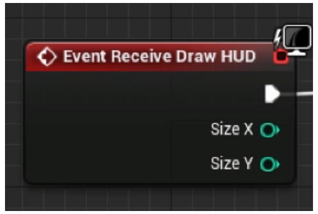
{

  draw graphics

  sleep(desired-elapsed time)

}

exit



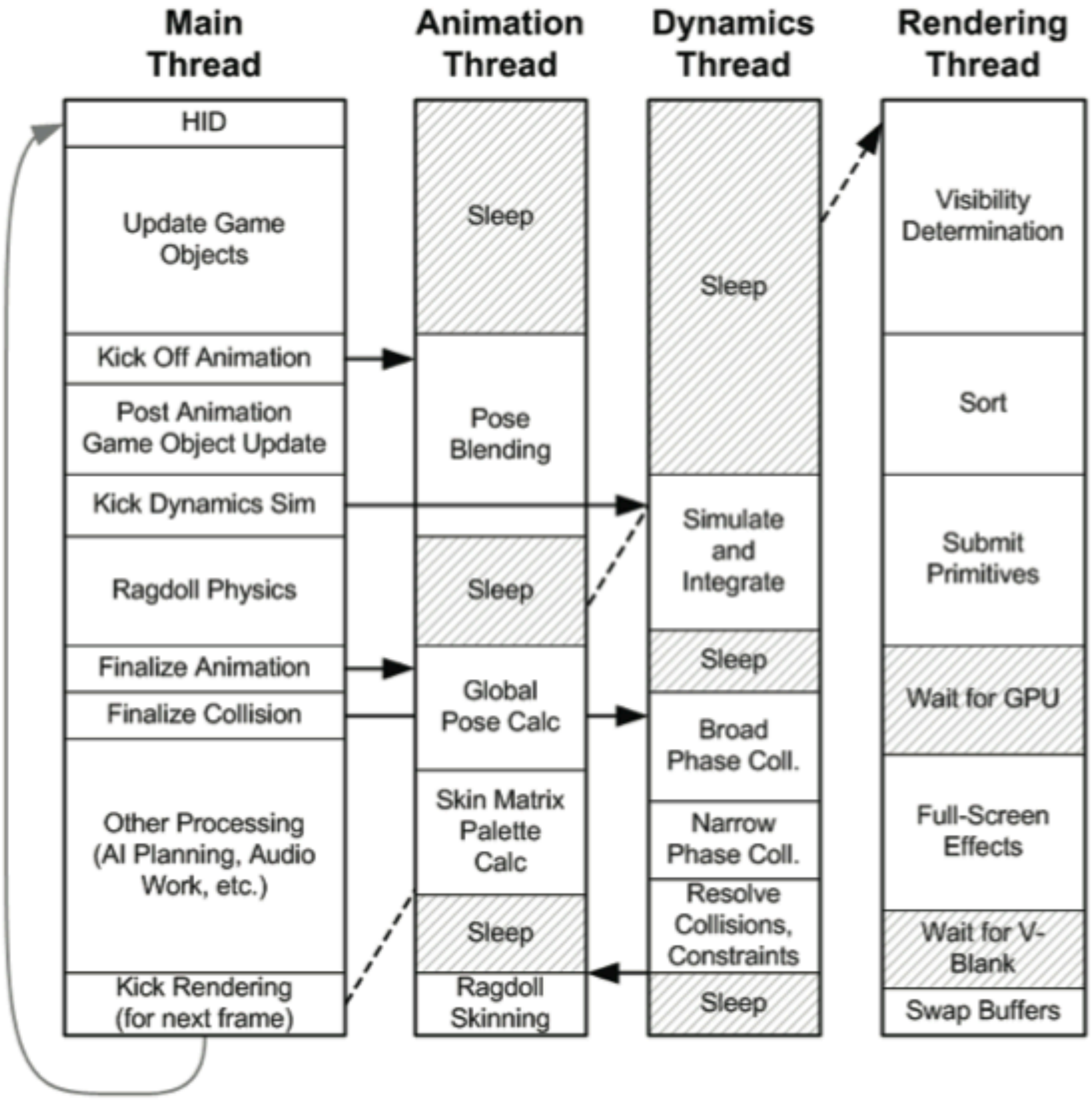
The Game Loop (attempt 3)

- Game Loop as Scheduler
- Advantages of the frequency dependent single - threaded decoupled approach
  - Allow an individual frequency for each entity in the game
  - Same mechanism can be applied to rendering
  - Generic automatic registration mechanism
- Disadvantages
  - Need to specify the frequency ‘manually’ for each entity
  - The game engine needs an entry point for each entity to update (might be large)

The Game Loop (attempt 3)

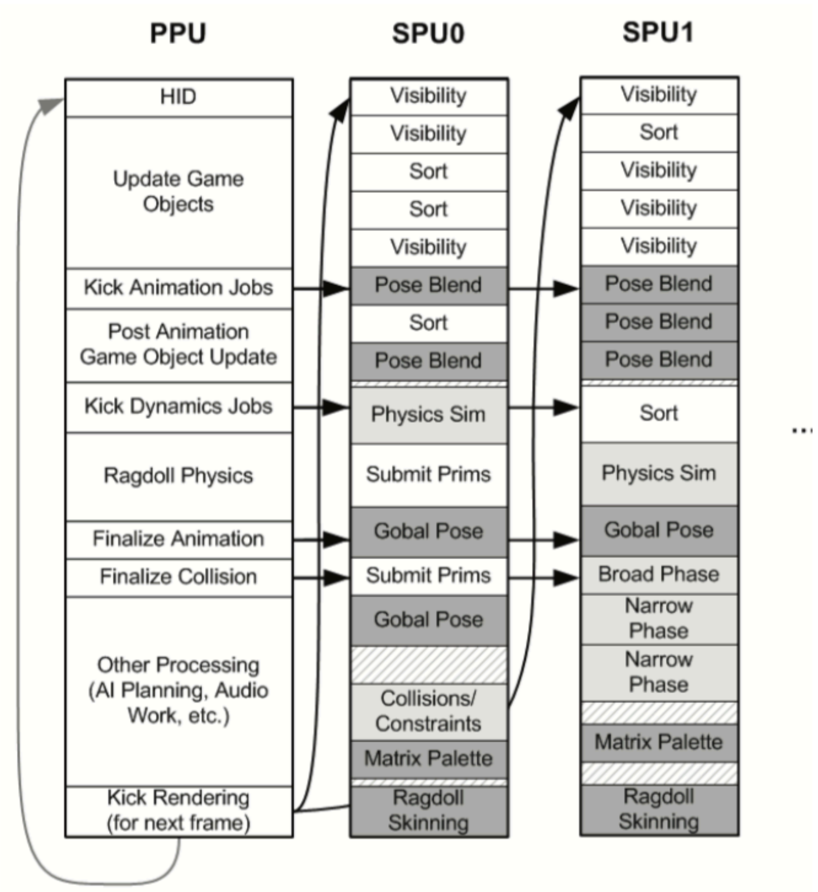
```
start game
Physics.setFrequency(60);
AI.setFrequency(10);
while( user doesn't exit )
{
    get user input
    get network messages
    Physics.update();
    AI.update();
    // rendering frequency is "as fast as possible"
    Renderer.render();
    lastCall = getTime();
}
exit
```

- 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

Reading

- Game Engine Architecture, Jason Gregory 2014, chapter 7