

COMP130: Game Architecture

## **4: Game engine architecture**

**The main loop**

# Basic game architecture

- ▶ CPUs execute **sequences of instructions**
  - ▶ At the CPU level, there is no such thing as “when  $X$  happens, do  $Y$ ”
  - ▶ Instead: “keep checking whether  $X$  has happened; if so, do  $Y$ ”
- ▶ Thus games need to have a **main loop**
- ▶ The main loop is usually part of the game’s **engine**
  - ▶ E.g. Unreal implements the main loop for you
  - ▶ E.g. PyGame isn’t a game engine, so you have to implement your own main loop

# The basic main loop

The most basic main game loop does **two** things:

1. **Update** the state of the game
  - ▶ Handle player input
  - ▶ Update physics, collision detection, AI etc.
  - ▶ Do any game-specific updates
2. **Render** the game to the screen
  - ▶ Game world
  - ▶ User interface

It does these **once per frame** (typically 30 or 60 times per second)

# The basic main loop

```
bool running = true;

while (running)
{
    update();
    render();
}
```

# Rendering

- ▶ This is where you draw the **current state of the game** to the screen
- ▶ Also draw any **heads-up display (HUD)** elements, e.g. score, lives, mini-map, etc.
- ▶ Most modern game engines **clear the screen and redraw everything** on every frame

# Screen refresh rate

- ▶ Old CRT monitors worked by scanning an electron beam down the screen
  - ▶ [https://www.youtube.com/watch?v=lRidfW\\_l4vs](https://www.youtube.com/watch?v=lRidfW_l4vs)
- ▶ Hence the term **(vertical) refresh rate**
- ▶ Refresh rate is measured in **cycles per second** i.e. **Hz**
- ▶ Other monitor technologies work differently, but still refresh the screen at regular intervals

# Frame rate

- ▶ We generally want to sync it up so that  
**one display refresh = one main loop iteration**
- ▶ If the main loop runs too slowly, we get “lag”
- ▶ If the main loop runs too quickly, we waste resources on drawing things faster than the display can show them



# Limiting the frame rate

- ▶ The game's main loop is generally synchronised to the screen refresh rate
- ▶ However, refresh rates can vary
  - ▶ Older TVs:  $\sim 30\text{Hz}$
  - ▶ HDTVs and standard monitors:  $60\text{Hz}$
  - ▶ VR headsets:  $90\text{Hz}$
  - ▶ High-end gaming monitors:  $120\text{Hz}$  or higher

# Limiting the update rate

- ▶ Having the update frequency depend on the refresh rate would be bad!
  - ▶ The game could appear to run in slow or fast motion, completely changing the gameplay
- ▶ This was the situation on older consoles:  
American/Japanese versions of games actually ran a little faster than European versions, due to the NTSC TV standard having a higher refresh rate than PAL!

# Variable time step

- ▶ Have the update step depend on the **elapsed time since the last update**
- ▶ Also known as the **delta time**
- ▶ E.g. if the game is running at a steady 60FPS, delta time =  $\frac{1}{60}$
- ▶ So instead of this:

```
player.positionX += player.velocityX;
```

- ▶ do this:

```
player.positionX += player.velocityX * deltaTime;
```

# Variable time step

```
bool running = true;
float lastFrameTime = getCurrentTime();

while (running)
{
    float currentFrameTime = getCurrentTime();
    float deltaTime = currentFrameTime - lastFrameTime;

    update(deltaTime);
    render();
    waitForVerticalRefresh();

    lastFrameTime = currentFrameTime;
}
```

# Variable time step — the downside

- ▶ `deltaTime` will inevitably **fluctuate** slightly
- ▶ Recall from COMP110 session 11: floating point numbers are **not perfectly accurate**
  - ▶ E.g.  $0.1 + 0.2 == 0.30000000000000004$
- ▶ This can cause some systems (particularly physics simulations) to become **nondeterministic**
  - ▶ I.e. to give different results from the same inputs
  - ▶ Very bad for online multiplayer, physics puzzles, replay features, etc.
- ▶ If `deltaTime` becomes large (e.g. if the system cannot keep up with the frame rate), physics simulations can be prone to **numerical instability**
  - ▶ E.g.  
<https://www.youtube.com/watch?v=vuTReFhBMPg>

# Fixed time step

- ▶ Perform the update at a **fixed rate**, e.g. 50 times per second
- ▶ If refresh rate  $< 50\text{Hz}$ , update several times per frame
- ▶ If refresh rate  $> 50\text{Hz}$ , update once every few frames

# Fixed time step

```
bool running = true;
float lastUpdateTime = getCurrentTime();
float timePerUpdate = 1.0f / 50.0f;

while (running)
{
    float currentFrameTime = getCurrentTime();

    while (currentFrameTime - lastUpdateTime >= timePerUpdate)
    {
        update();
        lastUpdateTime += timePerUpdate;
    }

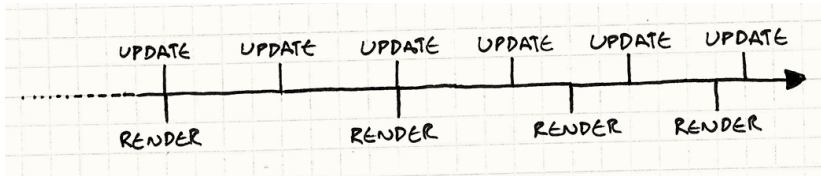
    render();
    waitForVerticalRefresh();
}
```

# Stalling

- ▶ What if `update` takes longer than `timePerUpdate` to execute?
- ▶ The `while` loop will perform more and more iterations in an effort to catch up, eventually grinding the game to a halt
- ▶ **Solution:** `break` out of the loop after a maximum number of iterations (e.g. 10)



# Interpolation



- ▶ Rendering at “irregular” intervals (with respect to update) can result in jerky movement
- ▶ Solution: **interpolate** between the two previous updates
  - ▶ E.g. if the render falls exactly halfway between two updates, render each object exactly halfway between its positions **before** and **after** the most recent update

# Further information on fixed time steps

- ▶ <http://gafferongames.com/game-physics/fix-your-timestep/>
- ▶ <http://gameprogrammingpatterns.com/game-loop.html>

# Substepping

- ▶ Aims to be a “best of both worlds” between fixed and variable time steps
- ▶ Use fixed time step at low frame rates, variable at high frame rates
- ▶ E.g. again with a target update rate of 50Hz
- ▶ If refresh rate  $< 50\text{Hz}$ , update several times per frame, with a `deltaTime` of  $\frac{1}{50}$
- ▶ If refresh rate  $> 50\text{Hz}$ , update once every frame, with `deltaTime` measured as for variable time step

# Substepping

```
bool running = true;
float lastFrameTime = getCurrentTime();
float timePerUpdate = 1.0f / 50.0f;

while (running)
{
    float currentFrameTime = getCurrentTime();
    float deltaTime = currentFrameTime - lastFrameTime;

    if (deltaTime <= timePerUpdate)
    { // Variable time step
        update(deltaTime);
    }
    else
    { // Fixed time step
        while (deltaTime > 0)
        {
            update(timePerUpdate);
            deltaTime -= timePerUpdate;
        }
    }

    render();
    waitForVerticalRefresh();
    lastFrameTime = currentFrameTime;
}
```

# Time steps in Unreal

- ▶ Unreal uses **variable time step** by default
- ▶ Can be switched to use **substepping**
- ▶ `Tick` function on C++ classes is called once per frame
- ▶ `FCalculateCustomPhysics` delegate can be used to do something once per substep
- ▶ Blueprints are updated once per frame; substep updates require C++
- ▶ More info: <http://www.aclockworkberry.com/unreal-engine-substepping/>

# Time steps in Unity

- ▶ Unity uses both **fixed** and **variable time step**
- ▶ `Update` function on `MonoBehaviour`s is tied to frame rate
- ▶ `FixedUpdate` function is tied to fixed physics updates