Game Engine Architecture

Chapter 8

The Game Loop and Real-Time Simulation

Overview

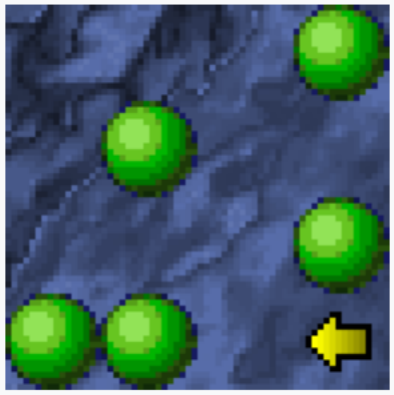
- Rendering Loop
- The Game loop
- Game Loop Architectural Styles
- Abstract Timelines
- Multiprocessor Game Loops

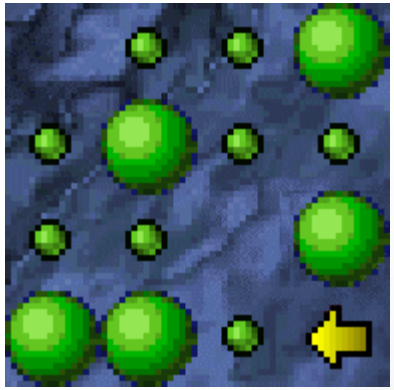
Rendering Loop

- In the early days of games, video cards were really slow
  - If they actually had a card
- Programmers optimized rendering using
  - Specialized hardware – allowed fixed number of sprites to be overlaid
  - XOR operations
  - Copy a portion of the background, drawing the sprite and later restoring the background
- Today, when the camera moves the entire scene changes
  - Everything is invalidated
  - It is faster to redraw everything than trying to figure out what to redraw

Bit blit

- Bit blit (stands for bit block transfer) is a data operation commonly used in computer graphics in which several bitmaps are combined into one using a boolean function
- A classic use for blitting is to render transparent sprites onto a background.
- In this example a background image, a sprite, and a 1-bit mask are used.
- A loop that examines each bit in the mask and copies the pixel from the sprite only if the mask is set
- The sprite is drawn in various positions over the image to produce this:

Background image	Sprite (left) and mask (right)
	



Simple loop

```
while(!quit){
    updateCamera();
    updateSceneElements();
    renderScene();
    swapBuffers();
}
```

Game Loop

- Games have many subsystems that interact
- These systems need to update at different rates
  - Physics – 30,60, or up to 120Hz
  - Rendering – 30-60Hz
  - AI – 0.5 – 1 Hz
- Lots of ways to do variable updating
- Let's consider a simple loop

Pong

- Started as Tennis for Two in 1958
  - William A Higinbotham at Brookhaven National Labs
- Later turned into *Table Tennis* on the Magnavox Odyssey
- Then became Atari Pong

Pong game loop

```
void main(){
    initGame();
    while(true){
        readHumanInterfaceDevices(
            );
        If (quitButtonPressed())
            break;
        movePaddles();
        moveBall();
        collideAndBounceBall();
        handleOutOfBounds();
        renderPlayfield();
    }
}
```

handleOutOfBounds()

```
{
    if
    (ballImpactedSide(LEFT_PLAYER))
    {
    incrementScore(RIGHT_PLAYER);
    resetBall();
    }
    else if
    (ballImpactedSide(RIGHT_PLAYER)
    )
    {
    incrementScore(LEFT_PLAYER);
    resetBall();
    }
}
```

Game loop styles

- Windows message pump

```
while(true){
    MSG msg;
    while(PeekMessage(&msg, NULL, 0, 0) > 0){
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    RunOneIterationOfGameLoop();
}
```

- Messages take precedence – everything else pauses

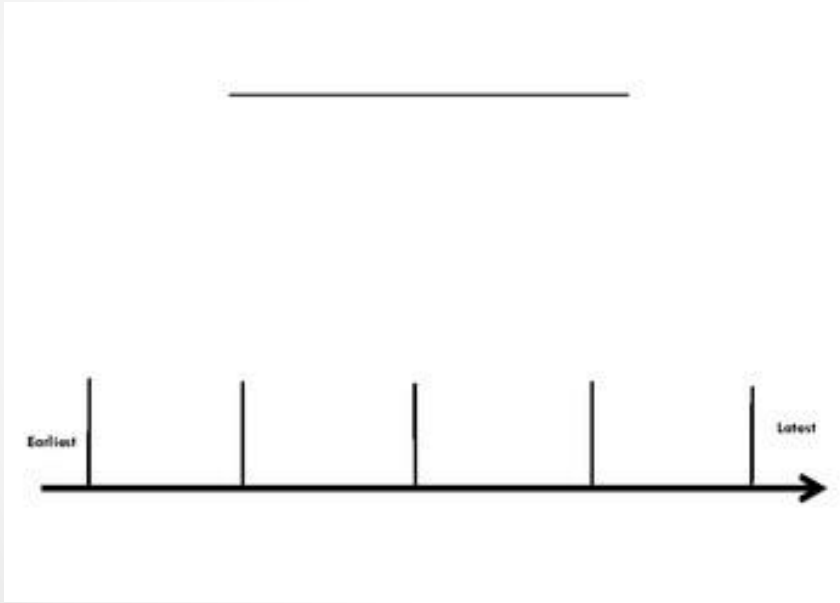
Callback-driven

- Most game engine subsystems and third-party game middleware packages are structured as *libraries*.
- A library is a suite of functions and/or classes that can be called in any way the application programmer sees fit.
- Libraries provide maximum flexibility to the programmer.
- Libraries are sometimes difficult to use, because the programmer must understand how to properly use the functions and classes they provide.
- In contrast, some game engines and game middleware packages are structured as *frameworks*.
- A framework is a partially constructed application—the programmer completes the application by providing custom implementations of missing functionality within the framework (or overriding its default behavior).
- Programmers has little or no control over the overall flow of control within the application, because it is controlled by the framework.
- In a framework-based rendering engine or game engine, the main game loop has been written for us, but it is largely empty.
- The game programmer can write callback functions in order to "fill in" the missing details.
- The OGRE rendering engine is an example of a library that has been wrapped in a framework.

Event-based updating

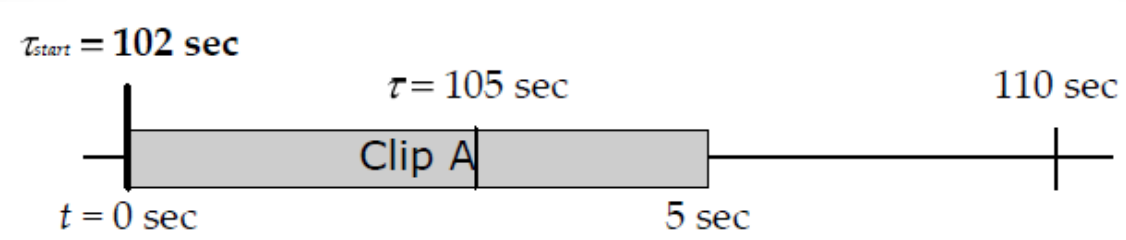
- Another way to design the loop is to use an event system
- Works like an input listener, but uses an event bus for the components to speak with one another
- Most game engines have an *event system*, which permits various engine subsystems to register interest in particular kinds of events and to respond to those events when they occur

Abstract timelines

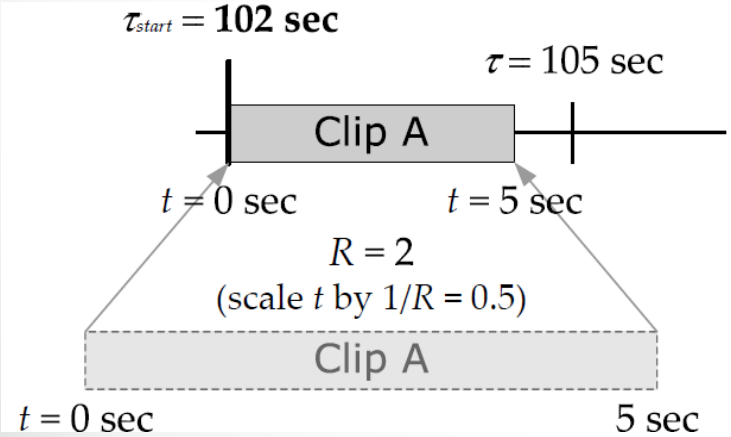


- Real time – measure by the CPU high resolution timer
- Game time – mostly controlled by real-time, but can be slowed, sped up, or paused
- Local and global time – animations have their own timeline. We can map it to global time in any way we want (translation, scaling)

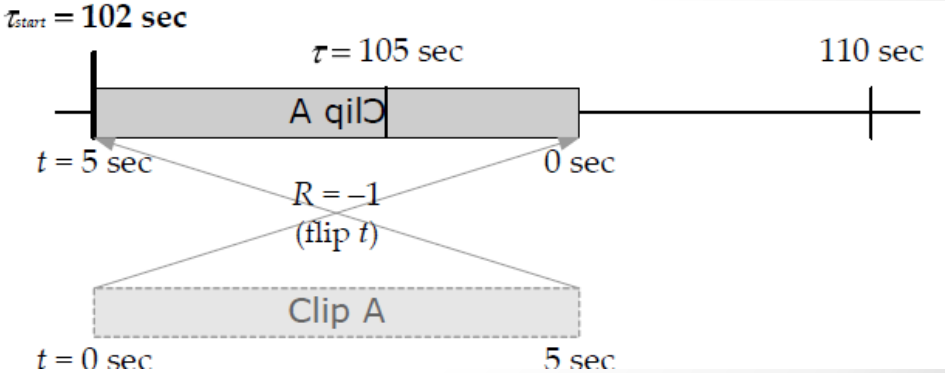
Mapping time



Simple Mapping



Scaled Mapping



Reverse Mapping

Measuring time

- By now we understand the concept of FPS. This also leads to the idea of deltaTime (the time between frames)
- We can use deltaTime to update the motion of objects in the game to keep the perception of time constant despite the frame rate (only if we are measuring it)

From Frame Rate to Speed

- Let's imagine that we want to make a spaceship fly through our game world at a constant speed of 40 meters per second
- or in a 2D game, we might specify this as 40 *pixels* per second.
- One simple way to accomplish this is to multiply the ship's speed v (measured in meters per second) by the duration of one frame Δt (measured in seconds), yielding a change in position $\Delta x = v\Delta t$ (which is measured in *meters per frame*).
- This position delta can then be added to the ship's current position x1, in order to find its position next frame: $x2 = x1 + \Delta x = x1 + v\Delta t$.

Old school

- Early games did not measure the amount of real time that elapsed
- Game objects were moved a fixed amount per iteration
  - Movement rate of the objects were dependent on the CPU speed
- Sucked when you upgraded the computer because the game ran too fast to play
- The turbo button was used to solve this problem in some cases
  - Turn off turbo to slow the computer down and make an older game playable

Updating Based on Elapsed Time

- One method that is often used is to read time directly and compute deltaTime
- Has some problems
  - We use this frame's time as an estimate of next frames time
  - Can lead to cascade delays and instability
- We could use a running average
  - Smoothes things out a bit
  - Long averages smooth out time, but are less reactive

Govern the rate

- It is best to govern the rate by sleeping between frames in order to standardize the time
- Need to have fairly consistent frame rates
- Has the advantage that everything is consistent
- Also easier to make a record and playback function

Blanking

- Many games govern their frame rate to the v-blank interval
- On CRT displays, the v-blank interval is the time during which the electron gun is "blanked" (turned off) while it is being reset to the top-left corner of the screen.
- In a raster graphics display, the vertical blanking interval (VBI), also known as the vertical interval or VBLANK, is the time between the end of the final visible line of a frame or field and the beginning of the first visible line of the next frame. It is present in analog television, VGA, DVI and other signals.
- Rendering Engines wait for VBI of the monitor before swapping buffers.
- Modern LCD, plasma and LED displays no longer use an electron beam, and they don't need any time between finishing the draw of the last scan line of one frame and the first scan line of the next. But the v-blank interval still exists (video standards)
- Waiting for the v-blank interval is called *v-sync* (just another form of frame-rate governing).
- This prevents tearing and limits the repaints to the maximum possible updating of the screen
  - Why render frames that never get displayed

Measuring time

- Most operating systems provide a function for querying the system time, such as the C standard library function time().
- time() returns an integer representing the number of *seconds* since midnight, January 1, 1970, so its resolution is one second—far too coarse, considering that a frame takes only tens of milliseconds to execute.
- All modern processors have a special register that holds a clock tick count since power on
- These can be super high resolution because the clock can tick 3 billion times per second
- Most of these registers are 64-bit so hold $1.8X10^{19}$ ticks before wrapping – 195 years on a 3.0 GHz processor

Accessing time

- Different CPUs have different ways to get time
- Pentiums use *rdtsc* (read time-stamp counter)
  - Wrapped in Windows with QueryPerformanceCounter()
- On PowerPC (Xbox 360 or Playstation 3) use the instruction "*mftb*" (move from time base register)
- On other PowerPCs use *mfspr* (move from special-purpose register)

Query Performance Counter in Windows

- Counters are used to provide information as to how well the operating system or an application, service, or driver is performing.
- QPC is used for measure time intervals or high resolution time stamps.
- QPC measures the total computation of response time of code execution.
- QPC in independent and isn't synchronized to any external time reference.
- If we want to use synchronize time stamps then, we must use GetSystemTimePreciseAsFileTime.
- QPC has two types of API's:
- Application: QueryPerformanceCounter(QPC)
- Device Driver: KeQueryPerformanceCounter

TIMING AND ANIMATION

- For accurate time measurements, we use the performance timer (or performance counter).
- To use the Win32 functions for querying the performance timer, we must #include <windows.h>
- The performance timer measures time in units called counts. We obtain the current time value, measured in counts, of the performance timer with the QueryPerformanceCounter function.
- It retrieves the current value of the performance counter, which is a high resolution (<1us) time stamp that can be used for time-interval measurements.

```
void GameTimer::Start()
{
__int64 startTime;
QueryPerformanceCounter((LARGE_INTEGER*)&startTime);


// Accumulate the time elapsed between stop and start pairs.
//
//                     |<-------d------->|
// ----*---------------*-----------------*------------> time
//  mBaseTime       mStopTime        startTime

if( mStopped )
{
mPausedTime += (startTime - mStopTime);

mPrevTime = startTime;
mStopTime = 0;
mStopped  = false;
}
}
```

LARGE_INTEGER

LARGE_INTEGER is a portable 64-bit integer. (i.e., you want it to function the same way on multiple platforms)

If the system doesn't support 64-bit integer, then it's defined as two 32-bit integers, High Part and a Low Part.

If the system does support 64-bit integer then it's a union between the two 32-bit integer and a 64-bit integer called the QuadPart.

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
        LONG HighPart;
    } DUMMYSTRUCTNAME;
    struct {
        DWORD LowPart;
        LONG HighPart;
    } u;
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

QueryPerformanceFrequency

- To get the frequency (counts per second) of the performance timer, we use the QueryPerformanceFrequency function:

```
LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
LARGE_INTEGER Frequency; //Default frequency supported by your hardware

QueryPerformanceFrequency(&Frequency);
QueryPerformanceCounter(&StartingTime);

// Activity to be timed

QueryPerformanceCounter(&EndingTime);
ElapsedMicroseconds.QuadPart = EndingTime.QuadPart - StartingTime.QuadPart;


//
// We now have the elapsed number of ticks, along with the
// number of ticks-per-second. We use these values
// to convert to the number of elapsed microseconds.
// To guard against loss-of-precision, we convert
// to microseconds *before* dividing by ticks-per-second.
//

ElapsedMicroseconds.QuadPart *= 1000000;
ElapsedMicroseconds.QuadPart /= Frequency.QuadPart;  //in microsecond
```

Game Timer Class

```
class GameTimer
{
public:
GameTimer();

float TotalTime()const; // in seconds
float DeltaTime()const; // in seconds

void Reset(); // Call before message loop.
void Start(); // Call when unpaused.
void Stop();  // Call when paused.
void Tick();  // Call every frame.

private:
double mSecondsPerCount;
double mDeltaTime;

__int64 mBaseTime;
__int64 mPausedTime;
__int64 mStopTime;
__int64 mPrevTime;
__int64 mCurrTime;

bool mStopped;
};
```

GameTimer()

- The constructor, in particular, queries the frequency of the performance counter.

```
GameTimer::GameTimer()
: mSecondsPerCount(0.0), mDeltaTime(-1.0), mBaseTime(0),
  mPausedTime(0), mPrevTime(0), mCurrTime(0), mStopped(false)
{
__int64 countsPerSec;
QueryPerformanceFrequency((LARGE_INTEGER*)&countsPerSec);
mSecondsPerCount = 1.0 / (double)countsPerSec;
}
```

Time Elapsed Between Frames

- When we render our frames of animation, we will need to know how much time has elapsed between frames so that we can update our game objects based on how much time has passed.

```
void GameTimer::Tick()
{
if( mStopped )
{
mDeltaTime = 0.0;
return;
}

__int64 currTime;
QueryPerformanceCounter((LARGE_INTEGER*)&currTime);
mCurrTime = currTime;

// Time difference between this frame and the previous.
mDeltaTime = (mCurrTime - mPrevTime)*mSecondsPerCount;

// Prepare for next frame.
mPrevTime = mCurrTime;

// Force nonnegative.  The DXSDK's CDXUTTimer mentions that if the
// processor goes into a power save mode or we get shuffled to another
// processor, then mDeltaTime can be negative.
if(mDeltaTime < 0.0)
{
mDeltaTime = 0.0;
}
}
```

The Application Message Loop

- The function Tick is called in the application message loop as follows. In this way, Δt is computed every frame and fed into the UpdateScene method so that the scene can be updated based on how much time has passed since the previous frame of animation.

```
int D3DApp::Run()
{
MSG msg = {0};

mTimer.Reset();

while(msg.message != WM_QUIT)
{
// If there are Window messages then process them.
if(PeekMessage( &msg, 0, 0, 0, PM_REMOVE ))
{
            TranslateMessage( &msg );
            DispatchMessage( &msg );
}
// Otherwise, do animation/game stuff.
else
        {
mTimer.Tick();

if( !mAppPaused )
{
CalculateFrameStats();
Update(mTimer);
Draw(mTimer);
}
else
{
Sleep(100);
}
        }
    }

return (int)msg.wParam;
}
```

Multi-cores

- These clocks can drift so take caution
- On multi-core processors all of the clocks are independent of one another
- Try not to compare the times because you will get strange results

Time units

- You should standardize time units in your engines and decide on the correct data type for the storage
  - 64-bit integer clock
  - 32-bit integer clock
  - 32-bit floating point clock

64-bit integer clock

- Worth it if you can afford the storage
- Direct copy of the register in most machines so no conversion
- Most flexible time representation

32-bit integer clock

- Often we can use a 32-bit integer clock to measure short duration events
- For eample, to profile the performance of a block of code, we might do something like this:

```
U64 begin_ticks = readHiResTimer();
doSomething();
U64 end_ticks = readHiResTimer();
U32 dt_ticks = static_cast<U32>(end_ticks – begin_ticks);
```

- Careful because it wraps after just 1.4 seconds

32-bit floating point

- Another common approach is to store small values in a 32-bit float in units of seconds

```
U64 begin_ticks = readHiResTimer();
doSomething();
U64 end_ticks = readHiResTimer();
F32 dt_seconds = (F32) (end_ticks – begin_ticks) / (F32)
        getHiResTimerFreq();
```

- Subtract the two U64s before the cast to prevent overflow

About floats

- Keep in mind that precision and magnitude are inversely related for floats
  - As the exponent increases the fraction space decreases
- Reset the float every once in a while to avoid decreased precision

Other time units

- Some game engines define their own time units
  - Allows integers to be used, but is fine-grained
  - Precise enough to be used for most game engine calculations
  - Large enough so it doesn't cause a 32-bit integer to wrap to often
- Common choice is 1/300th of a second
  - Still fine grained
  - Wraps every 165.7 days
  - Multiple of NTSC and PAL refresh rates

Handling breakpoints

- When you hit a breakpoint the clock keeps running
- Can cause bad things to happen when coming out
  - Hours could have elapsed – poor physics engine
- You can avoid this problem by using an upper bound and then clamp the time

```
if(dt > 1.0f)
{
        dt = 1.0f/30.0f;
}
```

Multiprocessor game loops

- In 2004 CPU manufacturers ran into a problem with heat as they attempted to increase CPU speed
- At first they felt they had reached the limits of Moore's law
  - # of transistors on a chip will double every 18 to 24 months
- But it was speed, not transistors that was limited
- Many moved to parallel architectures

Parallel games

- Many game companies were slow to transition to using multiple cores
  - Harder to program and debug
- The shift was slow, only a few subsystems were migrated at first
- Now many companies have engines that take advantage of the extra compute power

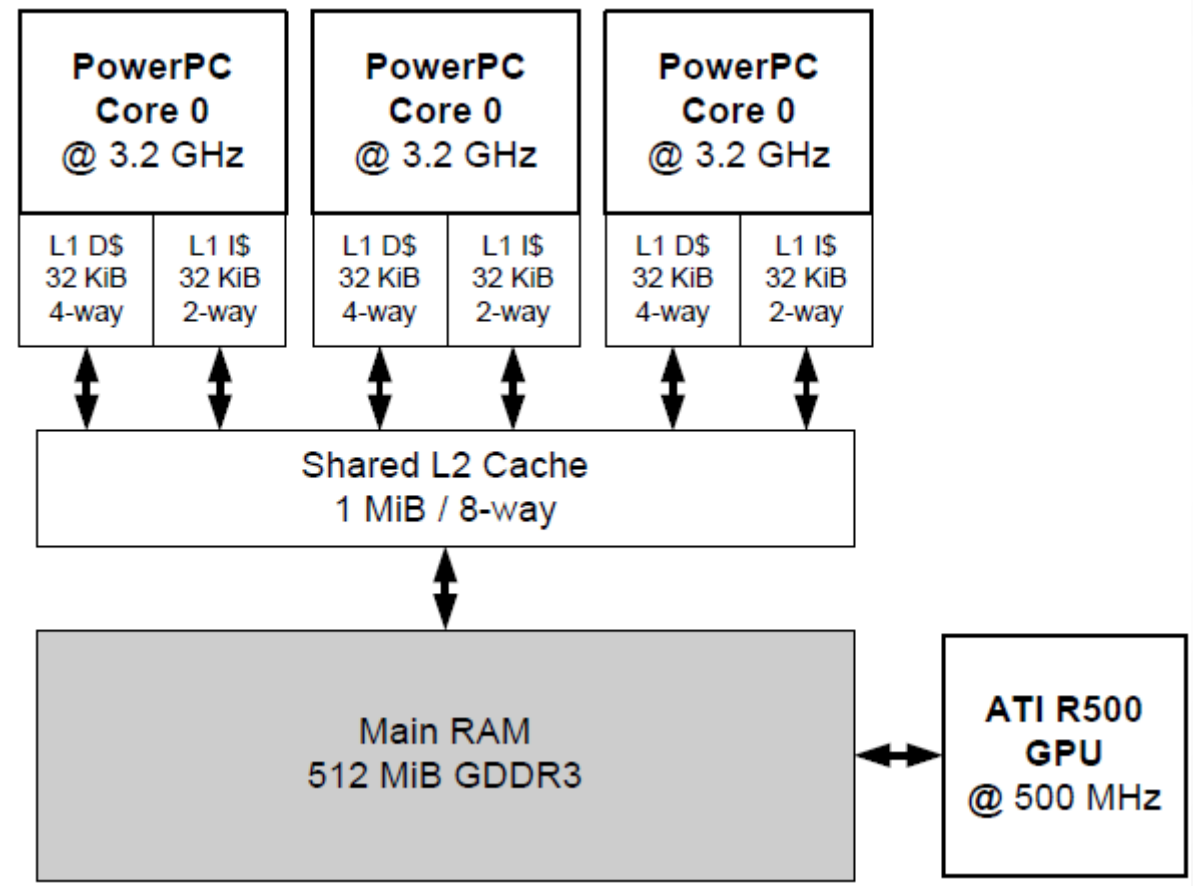
Multiprocessor consoles

- Xbox 360
- Xbox One
- PlayStation 3
- PlayStation 4

XBox 360

- 3 identical PowerPC cores
  - Each core has a dedicated L1 cache
  - They share a common L2 cache
- Has a dedicated 512MB RAM – used for everything in the system

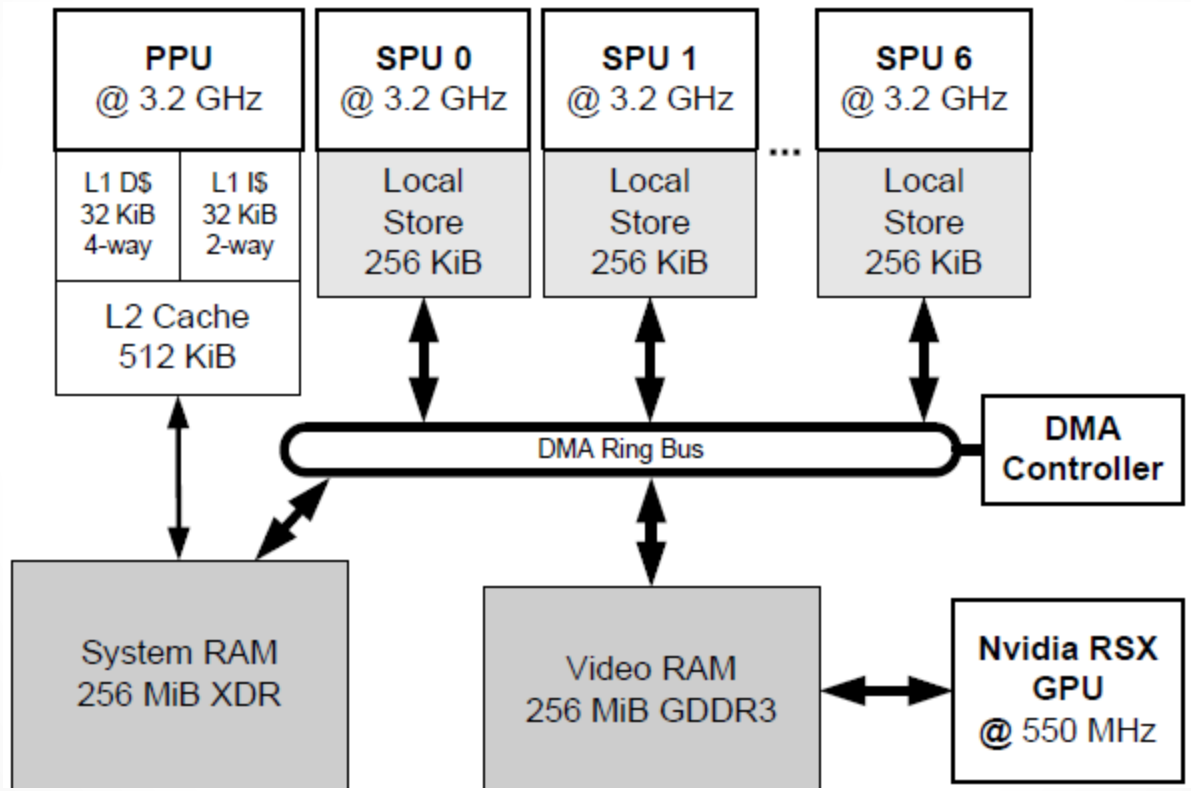
Xbox 360


PowerPC
Core 0
@ 3.2 GHz
PowerPC
Core 0
@ 3.2 GHz
PowerPC
Core 0
@ 3.2 GHz
L1 D$
32 KiB
4-way
L1 I$
32 KiB
2-way
L1 D$
32 KiB
4-way
L1 I$
32 KiB
2-way
L1 D$
32 KiB
4-way
L1 I$
32 KiB
2-way
Shared L2 Cache
1 MiB / 8-way
Main RAM
512 MiB GDDR3
ATI R500
GPU
@ 500 MHz

PlayStation 3

- Uses the Cell Broadband Engine (CBE) architecture
- Uses multiple processors each of which is specially designed
- The Power Processing Unit (PPU) is a PowerPC CPU
- The Special Processing Units (SPU) are based on the PowerPC with reduced and streamlined instruction sets also has 256K of L1 speed memory
- Communication done through a DMA bus which does memory copies in parallel to the PPU and SPUs

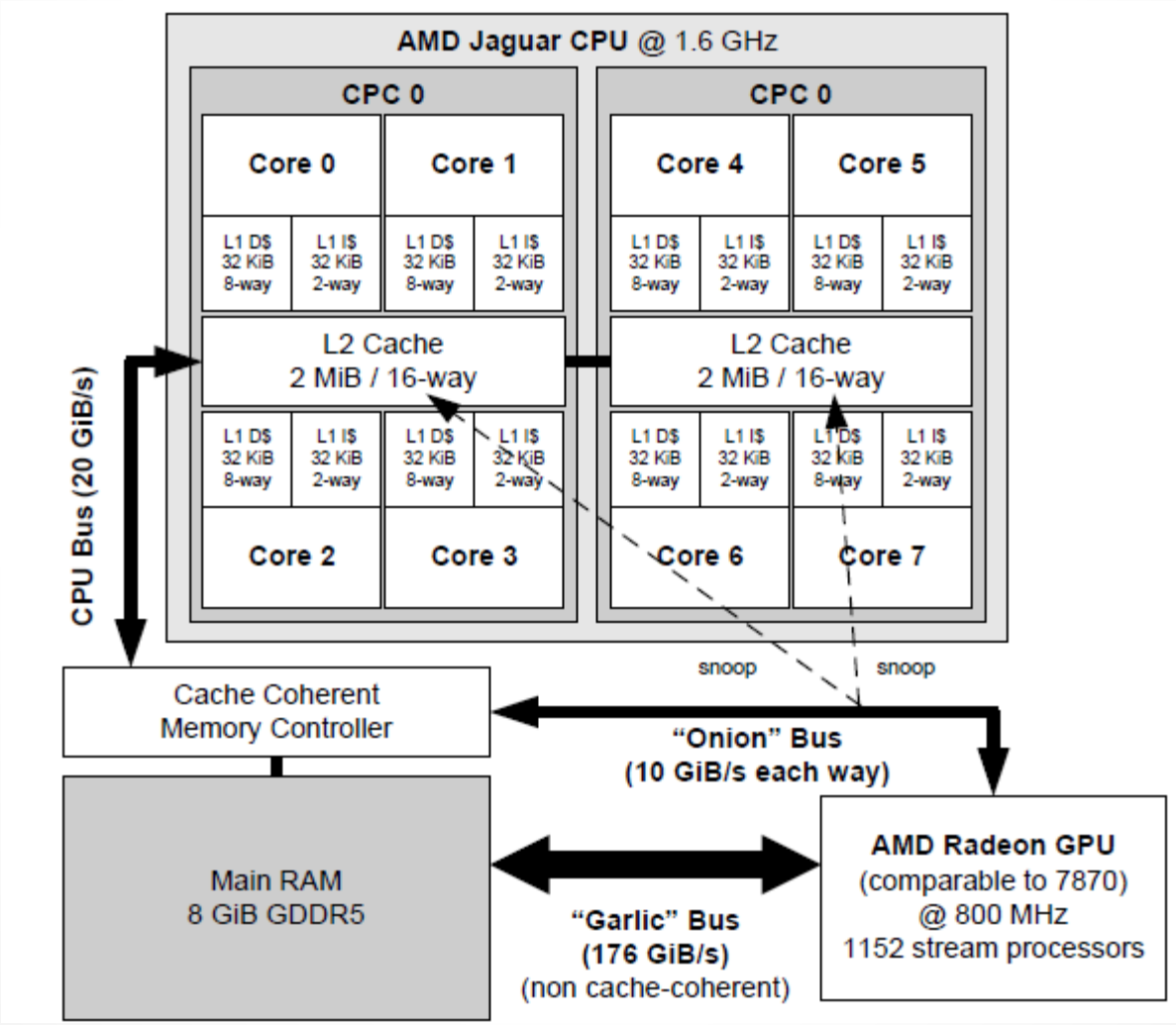
PS3


PPU
@ 3.2 GHz
L1 D$
32 KiB
4-way
L1 I$
32 KiB
2-way
L2 Cache
512 KiB
SPU 0
@ 3.2 GHz
Local
Store
256 KiB
SPU 1
@ 3.2 GHz
Local
Store
256 KiB
...
SPU 6
@ 3.2 GHz
Local
Store
256 KiB
DMA Ring Bus
DMA
Controller
System RAM
256 MiB XDR
Video RAM
256 MiB GDDR3
Nvidia RSX
GPU
@ 550 MHz

PlayStation 4

- Very different from Cell architecture
- Utilizes an eight core AMD Jaguar CPU
  - Has built in code optimization
- Modern GPGPU
  - Close to an AMD Radeon 7870
- Uses Intel instruction set instead of PowerPC
- Shared 8GiB block of GDDR5 Ram
- Employs three buses
  - 20 GiB/second CPU->RAM bus
  - 10GiB/second “onion” bus between the GPU and CPU caches
  - 176 GiB/second “garlic” bus between the GPU and RAM

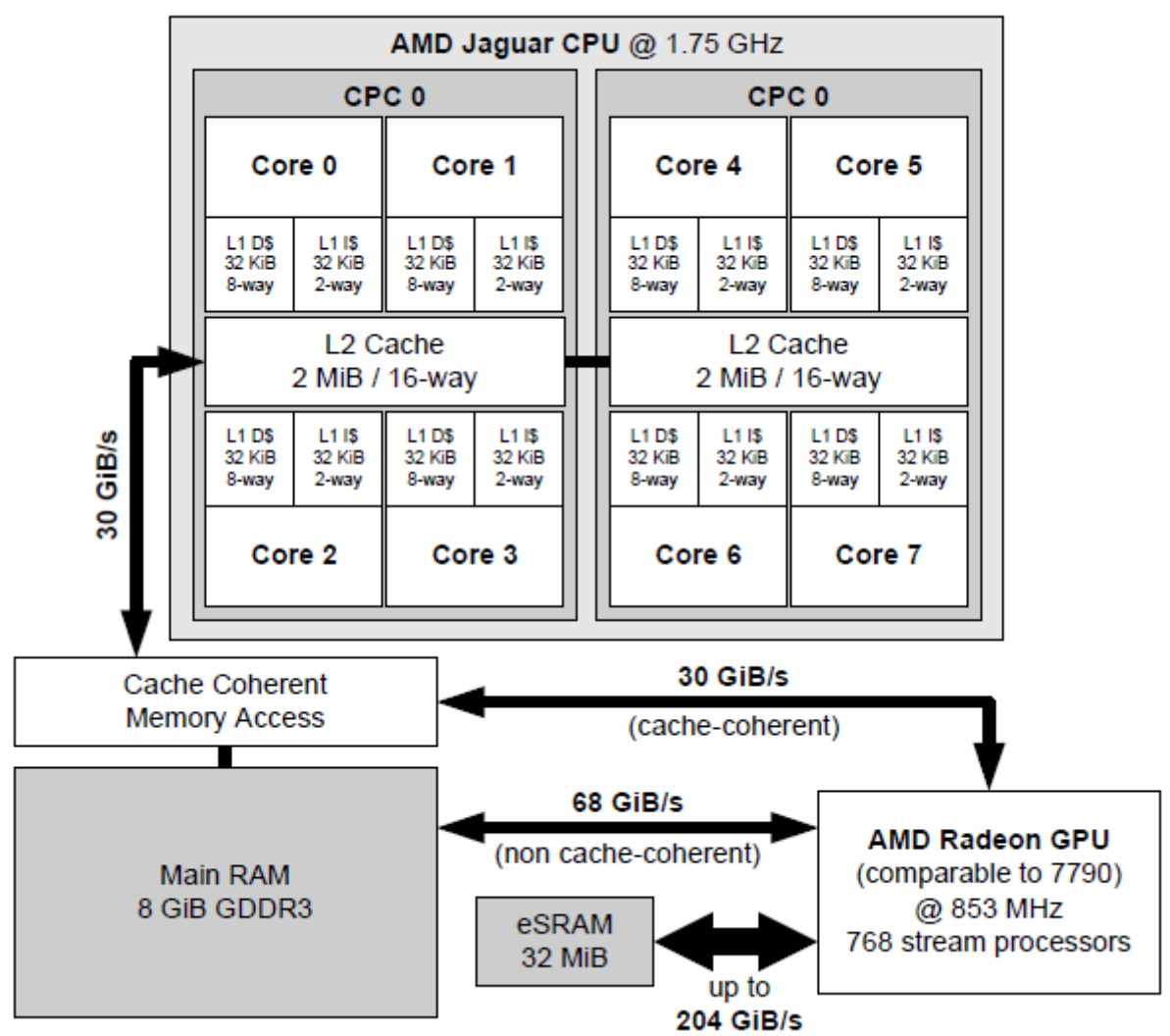
PS 4


AMD Jaguar CPU @ 1.6 GHz
CPC 0
CPC 0
Core 0
Core 1
Core 4
Core 5
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L2 Cache 2 MiB / 16-way
Core 2
Core 3
Core 6
Core 7
CPU Bus (20 GiB/s)
snoop
snoop
Cache Coherent Memory Controller
"Onion" Bus (10 GiB/s each way)
Main RAM 8 GiB GDDR5
"Garlic" Bus (176 GiB/s) (non cache-coherent)
AMD Radeon GPU (comparable to 7870) @ 800 MHz 1152 stream processors

Xbox One

- Very similar to the PS 4 – both based on the AMD Jaguar
- Important differences
  - *CPU Speed:* 1.75 Ghz vs 1.6 Ghz on the PS4
  - *Memory type:* GDDR3 RAM (slower), but has 32MiB eSRAM on the GPU (faster)
  - *Bus Speed:* faster main bus (30GiB/sec vs 20GiB/sec)
  - *GPU:* not quite as powerful (768 processors vs 1152 processors), but runs faster (853Mhz vs 800 Mhz)
  - OS and ecosystem: Xbox Live vs PlayStation Network (PSN). Really a matter of taste

Xbox One


AMD Jaguar CPU @ 1.75 GHz
CPC 0
CPC 0
Core 0
Core 1
Core 4
Core 5
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L2 Cache 2 MiB / 16-way
Core 2
Core 3
Core 6
Core 7
30 GiB/s
Cache Coherent Memory Access
30 GiB/s
(cache-coherent)
68 GiB/s
(non cache-coherent)
Main RAM 8 GiB GDDR3
eSRAM 32 MiB
up to 204 GiB/s
AMD Radeon GPU
(comparable to 7790)
@ 853 MHz
768 stream processors

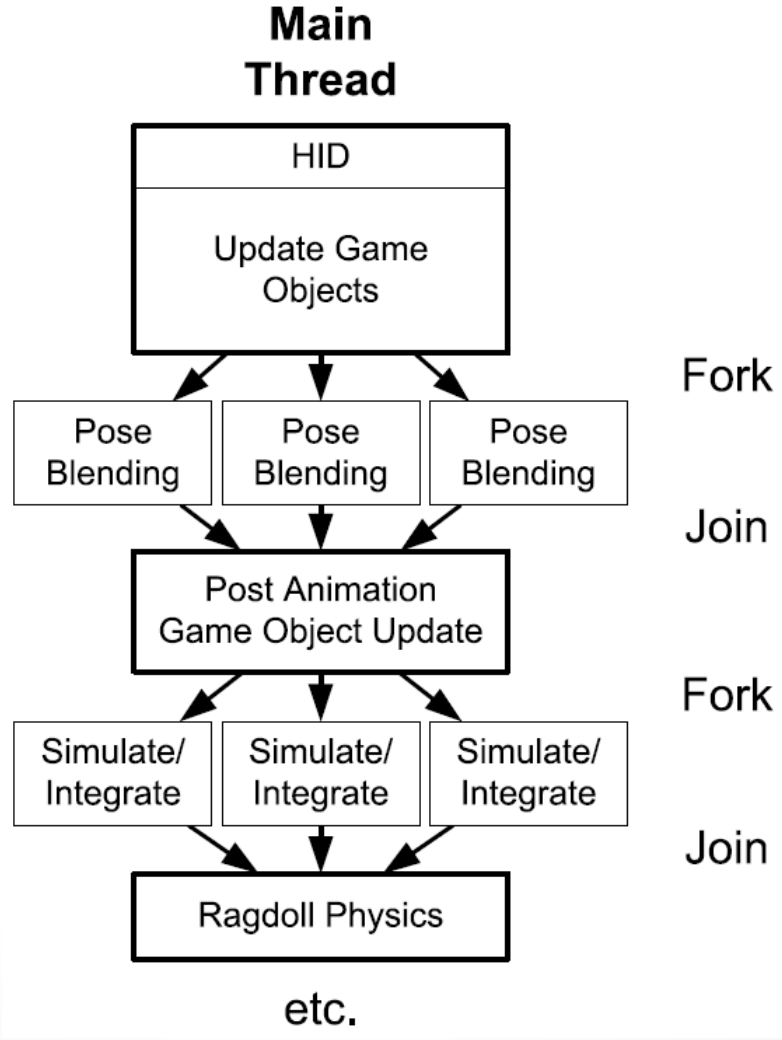
Seizing the power

- Fork and Join
  - Split a large task into a set of independent smaller tasks and then join the results together
- One thread per subsystem
  - Each major component runs in a different thread
- Jobs
  - Divide into multiple small independent jobs

Fork and Join

- Divide a unit of work into smaller subunits
- Distribute these onto multiple cores
- Merge the results

Fork and Join



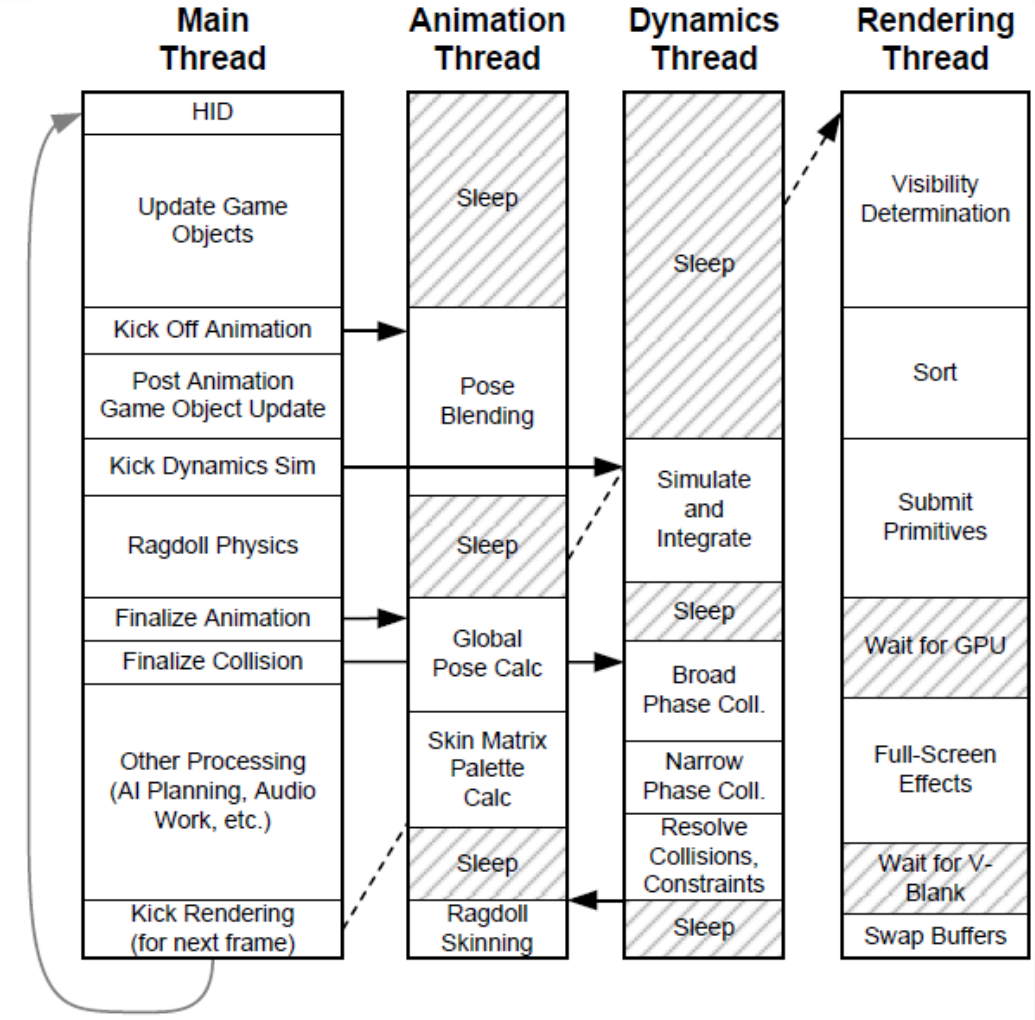
Example

- LERPing can be done on each joint independent of the others
- Imagine having 5 characters each with 100 joints that need to have blended poses computed
- We could divide the work into N batches, where N is the number of cores
- Each computes 500/N LERPs
- The main thread then waits (or not) on a semaphore
- Finally, the results are merged and the global pose calculated

Thread per Subsystem

- Have a master thread and multiple subsystem threads
  - Animation
  - Physics
  - Rendering
  - AI
  - Audio
- Works well if the subsystems can act mostly independently of one another

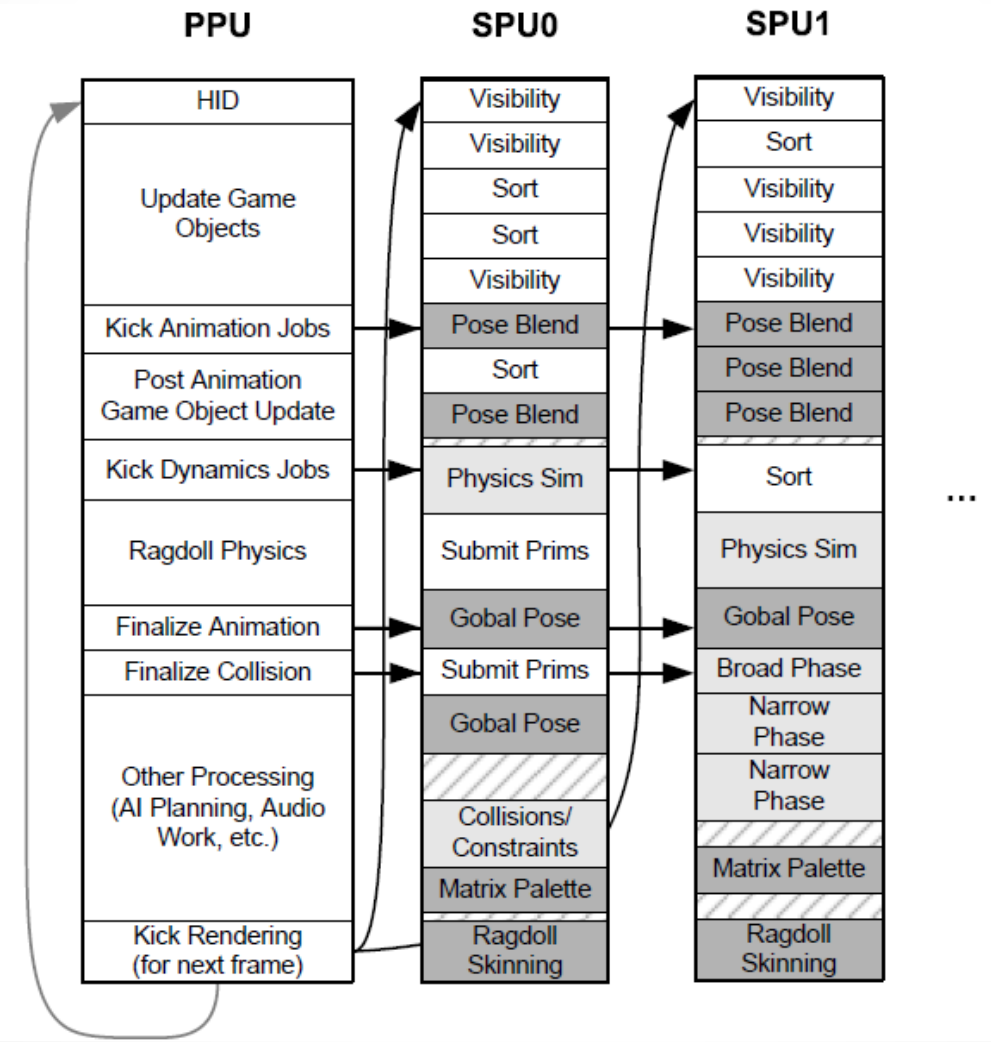
Thread per subsystem



Jobs

- Multithreading can sometimes be too course grained
  - Cores sit idle
  - More computational threads can block other subsystems
- We can divide up large tasks and assign them to free cores
- Works well on the PS3 – uses SPURS model for task assignment to the SPUs

Jobs


PPU
SPU0
SPU1
HID
Update Game Objects
Kick Animation Jobs
Post Animation Game Object Update
Kick Dynamics Jobs
Ragdoll Physics
Finalize Animation
Finalize Collision
Other Processing (AI Planning, Audio Work, etc.)
Kick Rendering (for next frame)
Visibility
Visibility
Sort
Sort
Visibility
Pose Blend
Sort
Pose Blend
Physics Sim
Submit Prims
Gobal Pose
Submit Prims
Gobal Pose
Collisions/ Constraints
Matrix Palette
Ragdoll Skinning
Visibility
Sort
Visibility
Visibility
Visibility
Pose Blend
Pose Blend
Pose Blend
Sort
Physics Sim
Gobal Pose
Broad Phase
Narrow Phase
Narrow Phase
Matrix Palette
Ragdoll Skinning
...

Networked Multiplayer Game Loops

- Client-Server
  - Can be run as separate processes or threads
  - Many games use a single thread
    - The client and server can be updating at different rates
- Peer-to-Peer
  - Each system acts as a client and server
  - Only one systems has authority over each dynamic object
  - Internal details of the code have to handle the case of having authority and not having authority for each object
  - Authority of an object can migrate

GameLoop In Unity

- To calculate distance when given speed, multiply the time it takes to travel that distance.
- In Unity the MonoBehaviour method Update is called once per frame.
- With methods that are called repeatedly such as Update, the variable Time.deltaTime is changed so that it is equal to the time between these calls.
- When it is used in Update, Time.deltaTime is equal to the time between frames.

DeltaTime

- Create a Cube GameObject and Mover Script in Unity and attach the script to the game object and run!
- In this example the script's transform has its position added to every Update call - that is, every frame - in the forward direction.
- The amount that is added is the speed variable multiplied by Time.deltaTime which equals a distance.
- This distance will change depending on the length of the frame.
- If the frame takes a long time then Time.deltaTime will be larger and so the calculated distance will be longer.
- Conversely, if the frame is very quick then Time.deltaTime will be smaller and so the calculated distance will be shorter.

```
using UnityEngine;
public class Mover :
MonoBehaviour
{
    public float speed = 5f;
    void Update()
    {
transform.position +=
Vector3.forward *
(speed * Time.deltaTime);
    }
}
```

Time Scale

- For the most part it is assumed that time continues at the same speed throughout a project but on occasions there can be a need for it to run at a different rate.
- The most common example of this is the concept of bullet-time where everything slows down in order to emphasize the speed things are happening.
- In order to facilitate this, the Time class has a variable called timeScale.
  - The default time scale is 1, this means that 1 second of game time will take 1 second of real time.
  - Reducing the time scale will slow down aspects of gameplay such as physics and animation.
  - A timescale of 0.5 will mean that 0.5 seconds of game time will take 1 second of real time - things happen half as fast.
  - Likewise a timescale of 2 will mean that 2 seconds of game time will take 1 second of real time - things happen twice as fast.

Scaled vs unscaled time

- Having a global control that slows down or speeds up aspects of gameplay can be extremely useful. However, there are often instances where some parts need to run at a normal speed and others need to be scaled.
- A common example of this is a game menu when gameplay is paused.
  - In this case, the menu would run using unscaled time. Unscaled time always runs in parallel with real time.
  - In this scenario the game play elements would be running with scaled time and when the time scale is set to 0 those elements would pause.
  - Meanwhile the menu would be running with unscaled time and so menu operations would continue as normal.
-

Time Scale Example

- This example is a very basic bullet time system.
- When the StartBulletTime method is called externally, it resets the timer and sets the bool that controls what happens in the Update method.
- If bullet time is being used then the if statement will be entered. It will set the time scale to the evaluation of the animation curve. The timer will then be incremented by the unscaledDeltaTime. It is important to note that this must be the unscaledDeltaTime because the deltaTime will be affected by the timeScale.
- If the animation curve sets the time scale to zero then the deltaTime will be zero too. The timer is then compared to the time of the last keyframe on the animation curve and if it is exceeded then bullet time is stopped.
-

BulletTime

```
using UnityEngine;
public class BulletTime : MonoBehaviour
{
    public AnimationCurve
bulletTimeScale = new
AnimationCurve(new Keyframe(0,0), new
Keyframe(3, 5), new Keyframe(5,2));
    public GameObject bulletObject;

    bool m_IsUsingBulletTime;
    float m_UnscaledElapsedTime;
    float chronometer = 0.0f;
    public void StartBulletTime()
    {
        m_UnscaledElapsedTime = 0f;
        m_IsUsingBulletTime = true;
    }

    private void Start()
    {
        StartBulletTime();
    }
```

```
 void Update()
    {
        if (m_IsUsingBulletTime)
        {
            Time.timeScale =
bulletTimeScale.Evaluate(m_UnscaledElapsedTime);
            m_UnscaledElapsedTime +=
Time.unscaledDeltaTime;
            if (m_UnscaledElapsedTime >
bulletTimeScale[bulletTimeScale.length - 1].time)
            {
                m_IsUsingBulletTime = false;
            }

            Vector3 bulletPosition = new
Vector3(m_UnscaledElapsedTime, Time.timeScale, 0);
            bulletObject.transform.position =
bulletPosition;
        }
        else
        {
            Time.timeScale = 1f;
            chronometer += Time.deltaTime;
            //Vector3 bulletPosition = new
Vector3(chronometer,
bulletTimeScale.Evaluate(chronometer), 0);
            //bulletObject.transform.position =
bulletPosition;
        }
    }
}
```

Coroutines

1. Coroutines in Unity are basically delayed functions
2. A coroutine allows you to spread tasks across several frames.
3. In Unity, a coroutine is a method that can pause execution and return control to Unity but then continue where it left off on the following frame.
4. However, it's important to remember that coroutines aren't threads. Synchronous operations that run within a coroutine still execute on the main thread.
5. Create a "Coroutines" C# script and attach it to a GameObject.
6. This coroutine makes program to wait for one frame

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Coroutines : MonoBehaviour
{
    void Start()
    {
        //To call the coroutine
        StartCoroutine(MyCoroutine());
    }
    IEnumerator MyCoroutine()
    {
        Debug.Log(Time.frameCount);
        //we use yield to delay the function.
        // The yield function make a pause for a single frame.
        yield return null;
        Debug.Log(Time.frameCount);
    }
}
```

WaitForSeconds

When you look at the Console, you see 1, and 2 which represents Frame 1 and Frame 2,
meaning Coroutine make a delay for one frame count.

```
1
UnityEngine.Debug:Log (object)

2
UnityEngine.Debug:Log (object)
```

If you want your program wait for 3 seconds, notice the frame number could be very different!

```
IEnumerator MyCoroutine()
    {
        Debug.Log(Time.frameCount);
        Debug.Log(Time.time);

        yield return new WaitForSeconds(3f);

        Debug.Log(Time.frameCount);
        Debug.Log(Time.time);
    }
```

```
1
UnityEngine.Debug:Log
(object)

0
UnityEngine.Debug:Log
(object)

826
UnityEngine.Debug:Log
(object)

3.003252
UnityEngine.Debug:Log
(object)
```

Time.timeScale

Change the Time Scale to 5 and run the application. You will notice that it gets to 3 seconds much quicker!

```
void Start()
    {
        Time.timeScale = 5.0f;
        //To call the coroutine
        StartCoroutine(MyCoroutine());
    }
```

WaitForSecondRealTime

However,if we change the WaitForSecond to WaitForSecondRealTime, this will ignore the time scale.
Meaning it actually waits for three seconds regardless of Time Scale.

```
void Start()

    {

        Time.timeScale = 2.0f;

        //To call the coroutine

        StartCoroutine(MyCoroutine());

    }

    //wait for seconds

    IEnumerator MyCoroutine()

    {

        Debug.Log(Time.frameCount);

        Debug.Log(Time.time);

        //we use yield to deplay the function. The yield function make a pause for a single frame.

        //yield return null;

        //we use yield to deplay the function. The yield function make a pause for three seconds.

        //yield return new WaitForSeconds(3f);

        yield return new WaitForSecondsRealtime(3f);

        Debug.Log(Time.frameCount);

        Debug.Log(Time.time);

    }
```

WaitUntil take a predicate

we create a public Run boolean so we can change it at Run Time:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Coroutines : MonoBehaviour
{
    public bool Run;
   void Start()
    {
        StartCoroutine(MyCoroutine());
    }
    IEnumerator MyCoroutine()
    {
        Debug.Log("Run is false");
        //This one take a predicate
        yield return new WaitUntil(() => Run == true);

        Debug.Log("Run is true");

    }
}
```

Run the application and you see “Run is false”, check the checkbox and you will “Run is True”

Time.FixedDeltaTime

Adjust fixed delta time according to timescale example

```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Coroutines : MonoBehaviour

{

    public bool Run;

    // Toggles the time scale between 1 and 0.7

    // whenever the user hits the Fire1 button.

    private float fixedDeltaTime;

    void Awake()

    {

        // Make a copy of the fixedDeltaTime, it defaults to 0.02f, but it can be changed in the editor

        this.fixedDeltaTime = Time.fixedDeltaTime;

        Debug.Log(Time.fixedDeltaTime);

    }

    // Start is called before the first frame update

    void Start()

    {

        //Time.timeScale = 2.0f;

        //To call the coroutine

        StartCoroutine(MyCoroutine());

    }

//wait for seconds

    IEnumerator MyCoroutine()

    {

        Debug.Log("Run is false");

        //This one take a predicate

        yield return new WaitUntil(() => Run == true);

        Debug.Log("Run is true");

}

    void Update()

    {

        if (Input.GetButtonDown("Fire1"))

        {

            if (Time.timeScale == 1.0f)

                Time.timeScale = 0.7f;

            else

                Time.timeScale = 1.0f;

            // Adjust fixed delta time according to timescale

            // The fixed delta time will now be 0.02 real-time seconds per frame

            Time.fixedDeltaTime = this.fixedDeltaTime * Time.timeScale;

            Debug.Log(Time.fixedDeltaTime);

        }

    }

}
```

The Basic IJob Interface in Unity

1. Open the package managers
2. Go to package manager, Click on Settings→ Project settings and "Enable Pre-release Packages"
2. Click on + to add a package by name
3. type: com.unity.jobs, and click “add” to install
4. Create a GameObject
5. Create a script called IJobTutorial

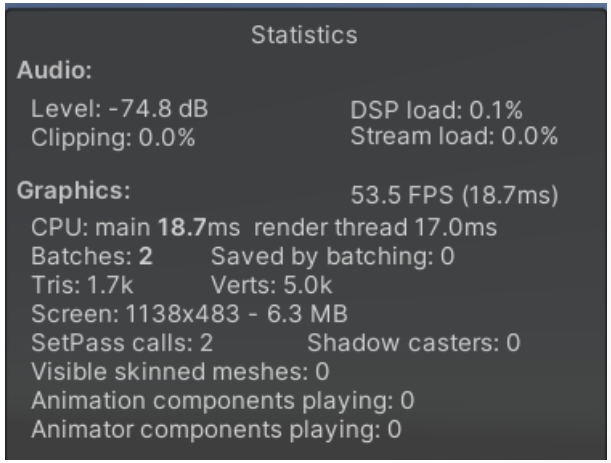
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class IJobTutorials : MonoBehaviour
{
void Update()
    {
        for(int i=0; i<100000;i++)
        {
            float f = Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);
        }
    }
}
```

6. attach the script to the game object

Statistics

- 7. Run the application and checkout the stats…yup, 55 FPS if you are lucky
- 8. Add `using Unity.Jobs;`
- 9. Right above the IJobTutorialClass add this struct:

```
public struct ExpensiveCalculation : IJob
{
   public void Execute()
    {

    }
}
```



ExpensiveCalculation

11.Now move the calculation inside the execute function:

```
public struct ExpensiveCalculation : IJob
{
   public void Execute()
    {
        //make sure we don't have any built-in Unity components like Transform.position, Animator.set
        for (int i = 0; i < 100000; i++)
        {
            float f = Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);
        }
    }
}
public class IJobTutorials : MonoBehaviour
{
void Update()
    {
}
}
```

NativeArray

12. Now we want to communicate execute result to the main class! In order to do that, we need a Native Conatiner.Make sure that you have using Unity.Collections;

```
using Unity.Jobs;

using Unity.Collections;


public struct ExpensiveCalculation : IJob

{

    public NativeArray<float> Value;

   public void Execute()

    {

        for (int i = 0; i < 100000; i++)

        {

            // float f = Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);

             Value[0] = Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);

        }

    }

}


public class IJobTutorials : MonoBehaviour

{

}
```

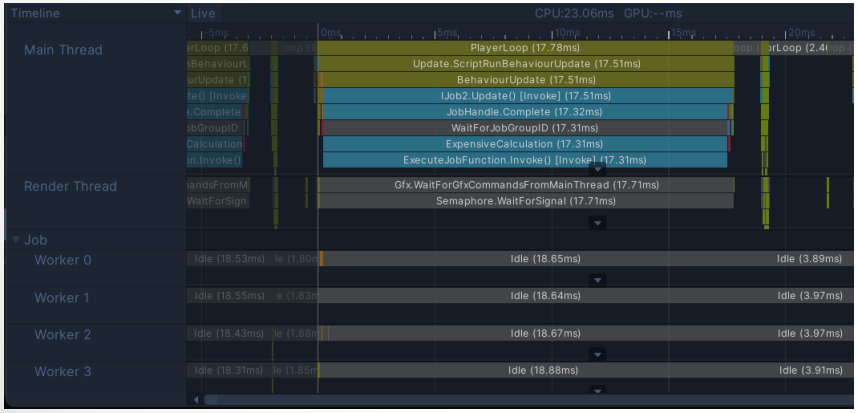
JobHandle

13. In order to create a job and run it, we need a Job Handle to make sure that we ran the job, and job is completed.

```
public class IJobTutorials : MonoBehaviour

{

    // Update is called once per frame

    void Update()

    {

        //Allocator.TempJob used when you do not want persistent data between frame otherwise use (Allocator.Persistent)!

        NativeArray<float> _Value = new NativeArray<float>(1, Allocator.TempJob);

        ExpensiveCalculation job = new ExpensiveCalculation()

        { Value = _Value };

        JobHandle jHandle = job.Schedule();

        //the complete function ensures that the job is completed.

        jHandle.Complete();

        //After the job is completed, we can get the value

        Debug.Log(job.Value[0]);

        //Once your job is done, you have to manually dispose the native container

        _Value.Dispose();

    }

}
```

Check out the Profiler

- Window → Analysis → Profiler → Click on a time on a time line
- Click on the timeline drop down!



14. Go back to Unity Editor and run the scene.
15. You will notice that nothing is changed in terms of framerate! Because we had only one thing. We didn't do two jobs in parallel to take advantage of the job system.
16. Only main thread is busy!!! None of the worker threads are doing any calculations.

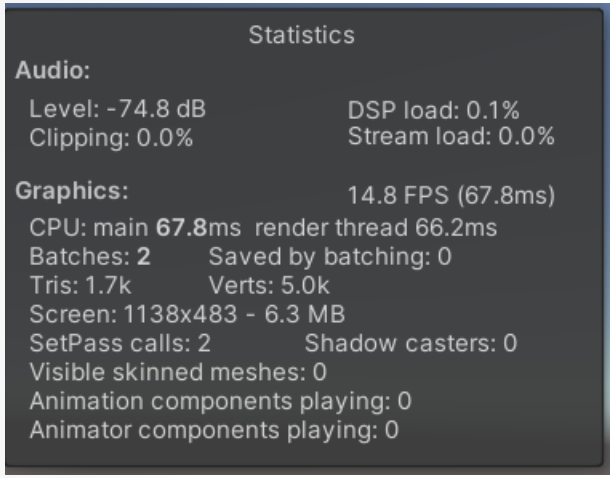
Adding 4 jobs

16. Now we run four jobs..first old school (bool UseJobSystem is equal to false!)

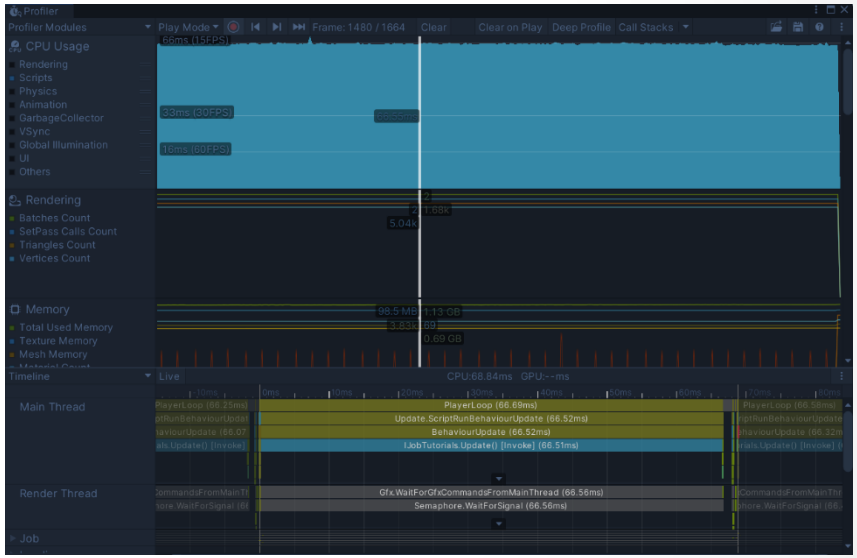
```
using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using Unity.Jobs;

using Unity.Collections;

using JetBrains.Annotations;

using UnityEngine.Pool;

public struct ExpensiveCalculation : IJob

{

public void Execute()

    {

        for (int i = 0; i < 100000; i++)

        {

            Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);

        }

    }

}
```

```
public class IJobTutorials : MonoBehaviour

{

    public bool UseJobSystem;

    void Update()

    {

        if (UseJobSystem)

        {

            NativeList<JobHandle> Jobs = new NativeList<JobHandle>(Allocator.Temp);

            for (int i = 0; i < 4; i++)

            {

                Jobs.Add(new ExpensiveCalculation().Schedule());

            }

            JobHandle.CompleteAll(Jobs);

        }

        else //user old way

        {

            for (int i = 0; i < 4; i++)

            {

                for (int x = 0; x < 100000; x++)

                {

                    Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);

                }

            }

        }

    }

}
```

Run the Profiler

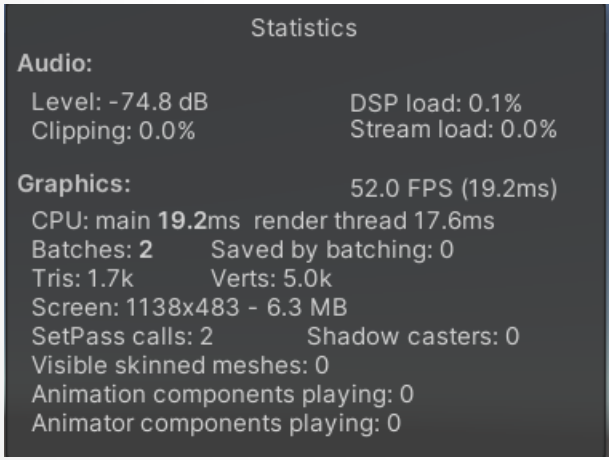
- 

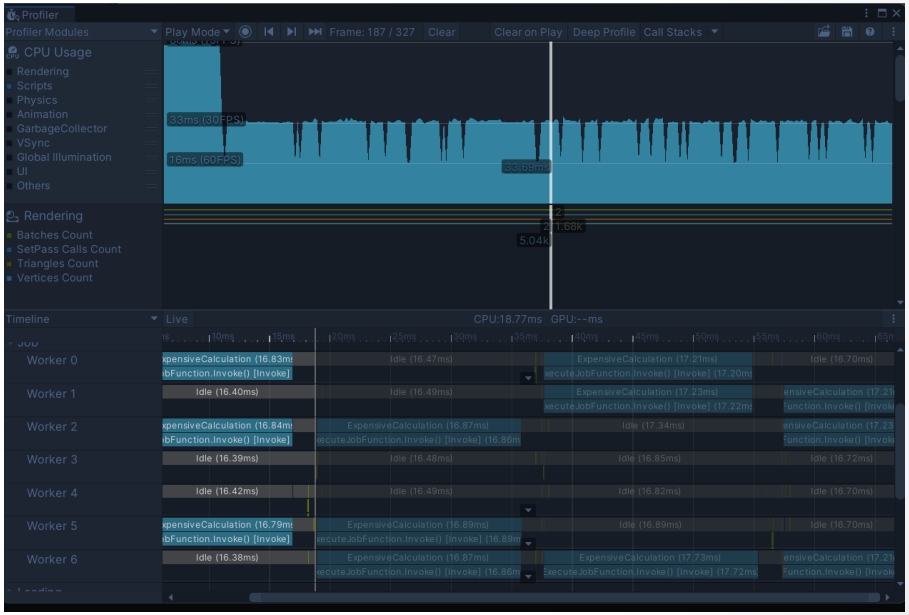
- 17. Run the profiler (Ctrl+7) or Window → Analysis → Profiler
- 18. Click on a timeline to select a point



Check out the profiler

17. Now if we set the Boolean to true, you see the performance boost 4 times and you see how executive calculation are done in different threads. Meaning instead of 400,000 on the same thread, we splitted them to 100000 on 4 different threads





The IJobFor Interface in Unity

The IJobFor interface is meant to iterarte over so many objects.

First we start with a clean script. Run and make sure everything is working

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.Jobs;
using Unity.Collections;
using JetBrains.Annotations;
using UnityEngine.Pool;

public struct ExpensiveCalculation : IJobFor
{


   public void Execute()
    {
            Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);
    }
}
```

```
public class IJobTutorials : MonoBehaviour
{
    public bool UseJobSystem;
    void Update()
    {
        if (UseJobSystem)
        {
        }
        else //old way
        {
            for (int i = 0; i < 4; i++)
            {
                for (int x = 0; x < 100000; x++)
                {
                    Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);
                }
            }
        }
    }
}
```

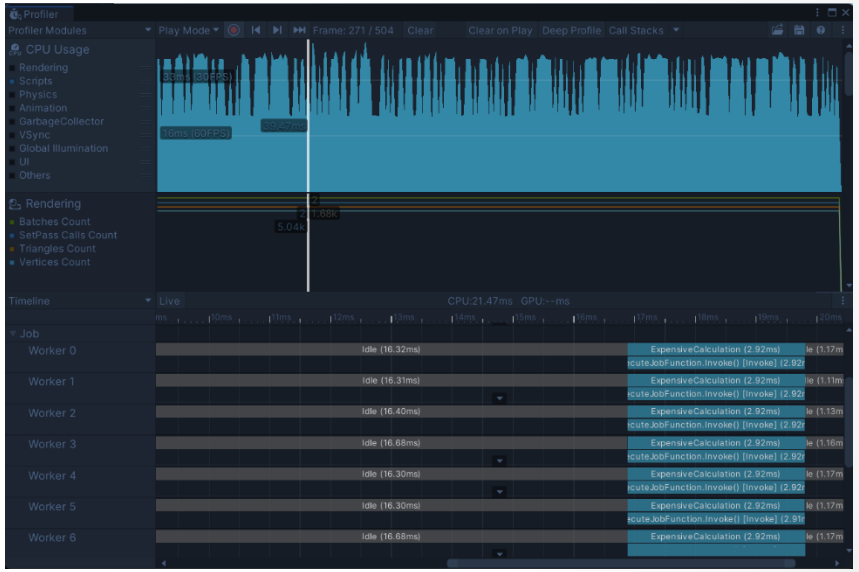
Add an index to Execute

```
public void Execute(int i)

Now we need to schedule it!

//IJobFor vs. IJobParallelFor  -===> IJobParallelFor i going to be
deprecated soon!

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using Unity.Jobs;

using Unity.Collections;

using JetBrains.Annotations;

using UnityEngine.Pool;


public struct ExpensiveCalculation : IJobFor

{


   public void Execute(int i)

    {

            Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);

    }

}
```

```
public class IJobTutorials : MonoBehaviour

{

    public bool UseJobSystem;

    void Update()

    {

        if (UseJobSystem)

        {

            ExpensiveCalculation calculation = new ExpensiveCalculation();

JobHandle dependency = new JobHandle();

            JobHandle scheduledependency = calculation.Schedule(100000,dependency);

JobHandle scheduleparalleljob = calculation.ScheduleParallel(100000, 1, scheduledependency);

            scheduleparalleljob.Complete();

        }

        else //user old way

        {

            for (int i = 0; i < 4; i++)

            {

                for (int x = 0; x < 100000; x++)

                {

                    Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);

                }

            }

        }

    }

}
```

Checkout the profiler

- Check the Use Job System in the inspector to see the difference. If you look at the profiler, the expensive calculation is using all the worker threads available to your computer and they are all running in parallel.



IJobParallelForTransform

- Parallel-for-transform jobs allow you to perform the same independent operation for each position, rotation and scale of all the transforms passed into the job.

```
using UnityEngine;
using Unity.Collections;
using Unity.Jobs;
using UnityEngine.Jobs;

class IJob5 : MonoBehaviour
{
        public struct VelocityJob : IJobParallelForTransform
        {
            [ReadOnly]
            public NativeArray<Vector3> velocity;

    public float deltaTime;
    public void Execute(int index, TransformAccess transform)
            {
    var pos = transform.position;
                pos += velocity[index] * deltaTime;
                transform.position = pos;
            }
        }
        [SerializeField] public Transform[] m_Transforms;
        TransformAccessArray m_AccessArray;

        void Awake()
        {
    m_AccessArray = new TransformAccessArray(m_Transforms);
        }

        void OnDestroy()
        {
    m_AccessArray.Dispose();
        }

        public void Update()
        {
            var velocity = new NativeArray<Vector3>(m_Transforms.Length,
    Allocator.Persistent);

            for (var i = 0; i < velocity.Length; ++i)
                velocity[i] = new Vector3(0f, 10f, 0f);
    var job = new VelocityJob()
            {
                deltaTime = Time.deltaTime,
                velocity = velocity
            };
    JobHandle jobHandle = job.Schedule(m_AccessArray);
    jobHandle.Complete();

            Debug.Log(m_Transforms[0].position);

    velocity.Dispose();
        }
    }
```