Game Engine Architecture

Chapter 8

The Game Loop and Real-Time Simulation

Overview

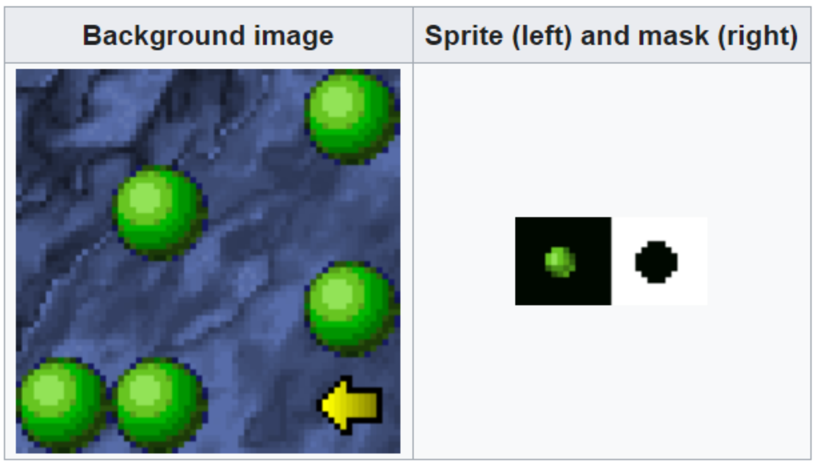
- Rendering Loop
- The Game loop
- Game Loop Architectural Styles
- Abstract Timelines
- Multiprocessor Game Loops

Rendering Loop

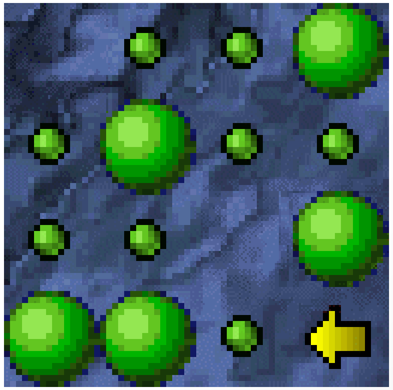
- In the early days of games, video cards were really slow
  - If they actually had a card
- Programmers optimized rendering using
  - Specialized hardware – allowed fixed number of sprites to be overlaid
  - XOR operations
  - Copy a portion of the background, drawing the sprite and later restoring 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))
    {
    incremenentScore(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

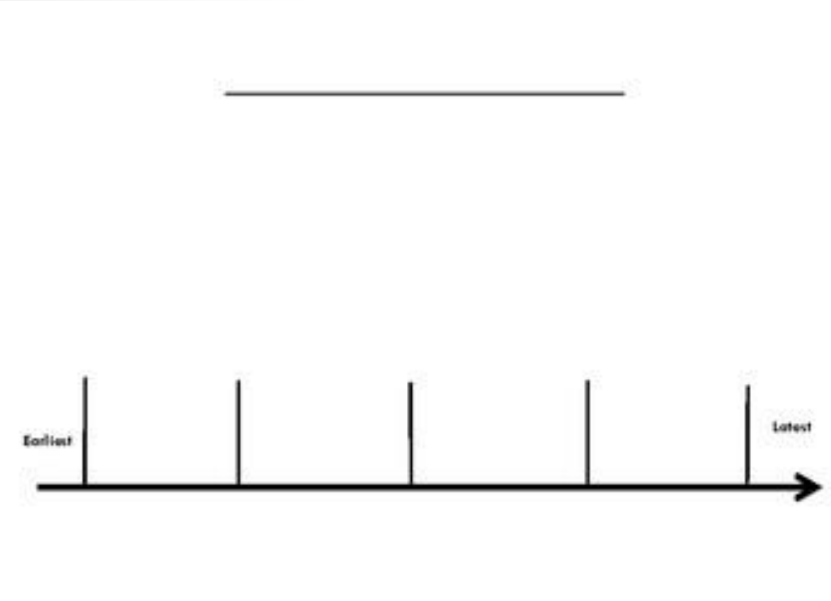
Callback-driven

- 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.
- At the lowest level, OGRE provides functions that can be called directly by a game engine programmer.
- OGRE also provides a framework that encapsulates knowledge of how to use the low-level OGRE library effectively.

Event-based updating

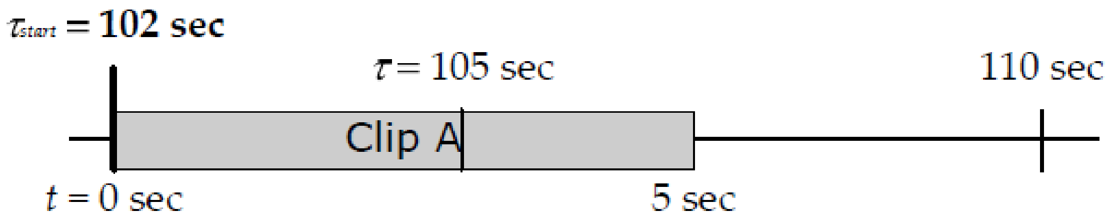
- Another way to design the loop is to use an event system
- Works like an input listener, but uses an event bus for the components to speak with one another
- Most game engines have an *event system*, which permits various engine subsystems to register interest in particular kinds of events and to respond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

Abstract timelines

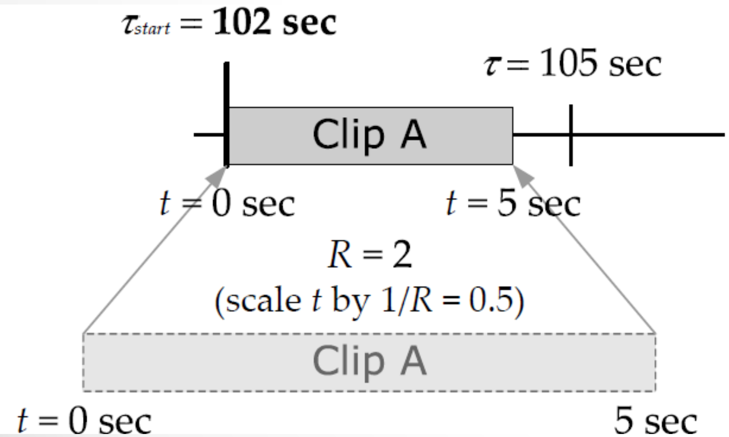


- Real time – measure by the CPU high resolution timer
- Game time – mostly controlled by real-time, but can be slowed, sped up, or paused
- Local and global time – animations have their own timeline. We can map it to global time in any way we want (translation, scaling)

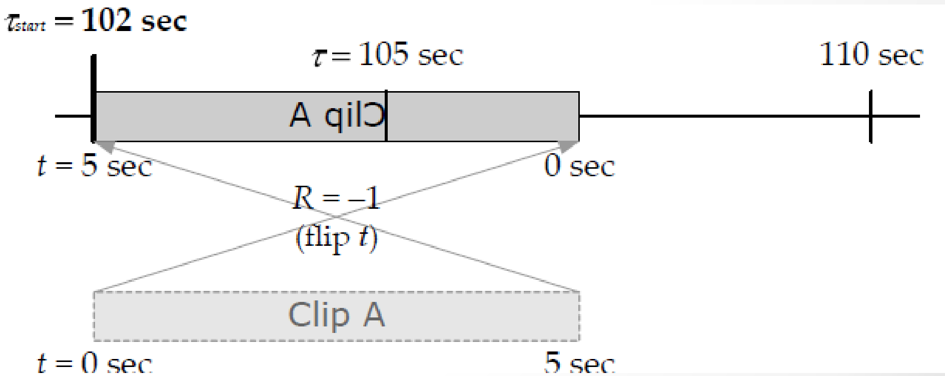
Mapping time



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*)&countsPer
Sec);
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

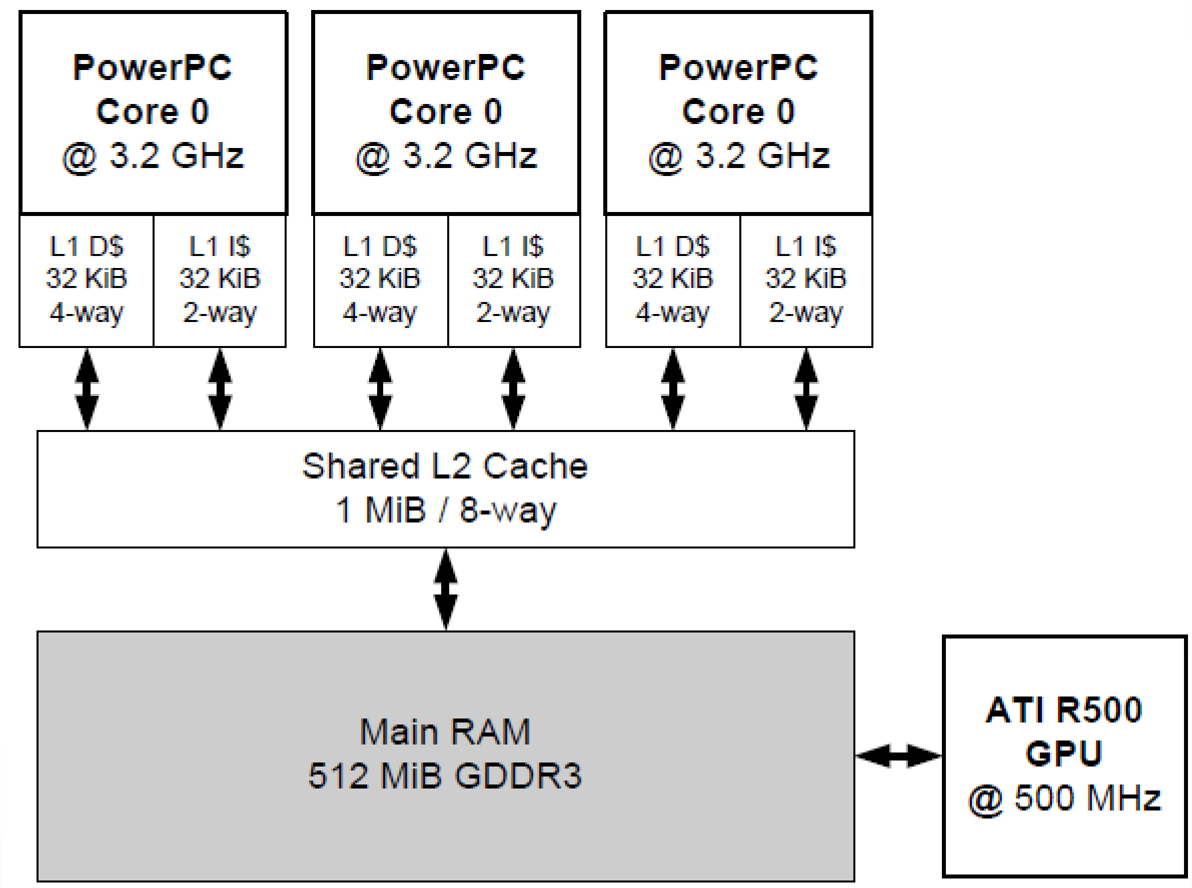
Multiprocessor consoles

- 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

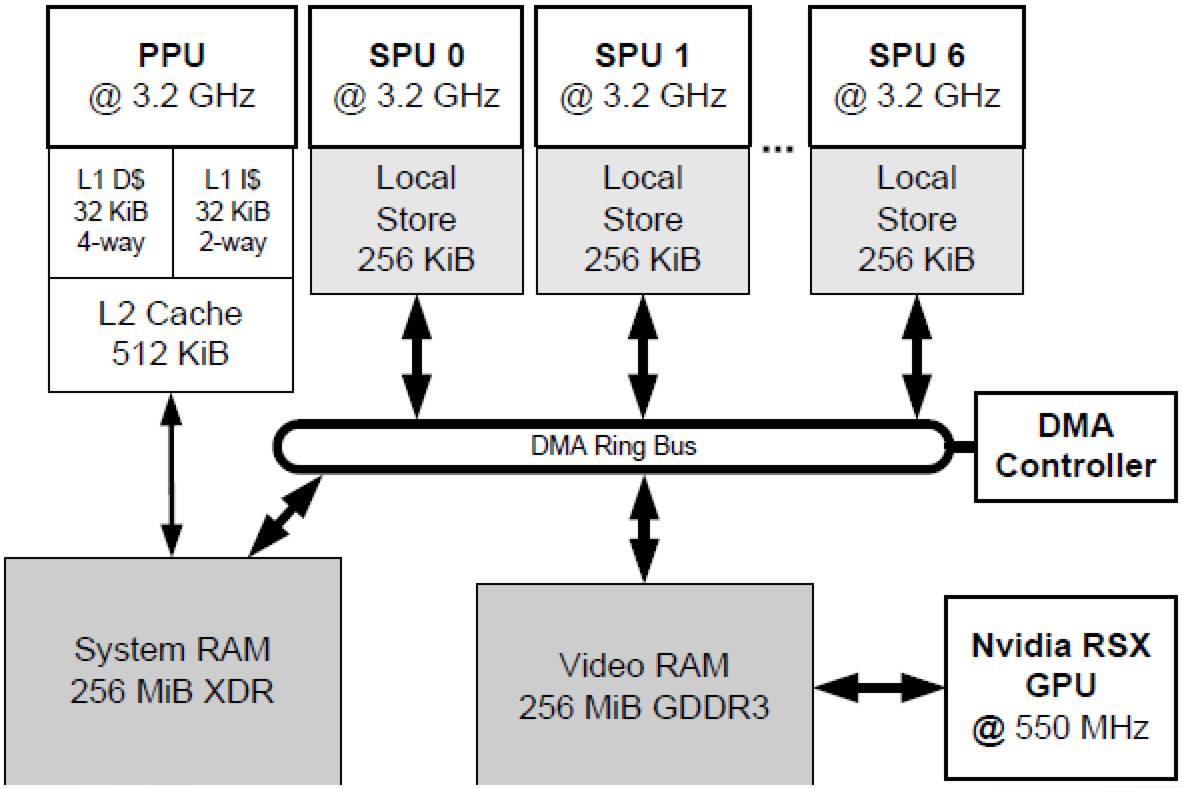
Xbox 360


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

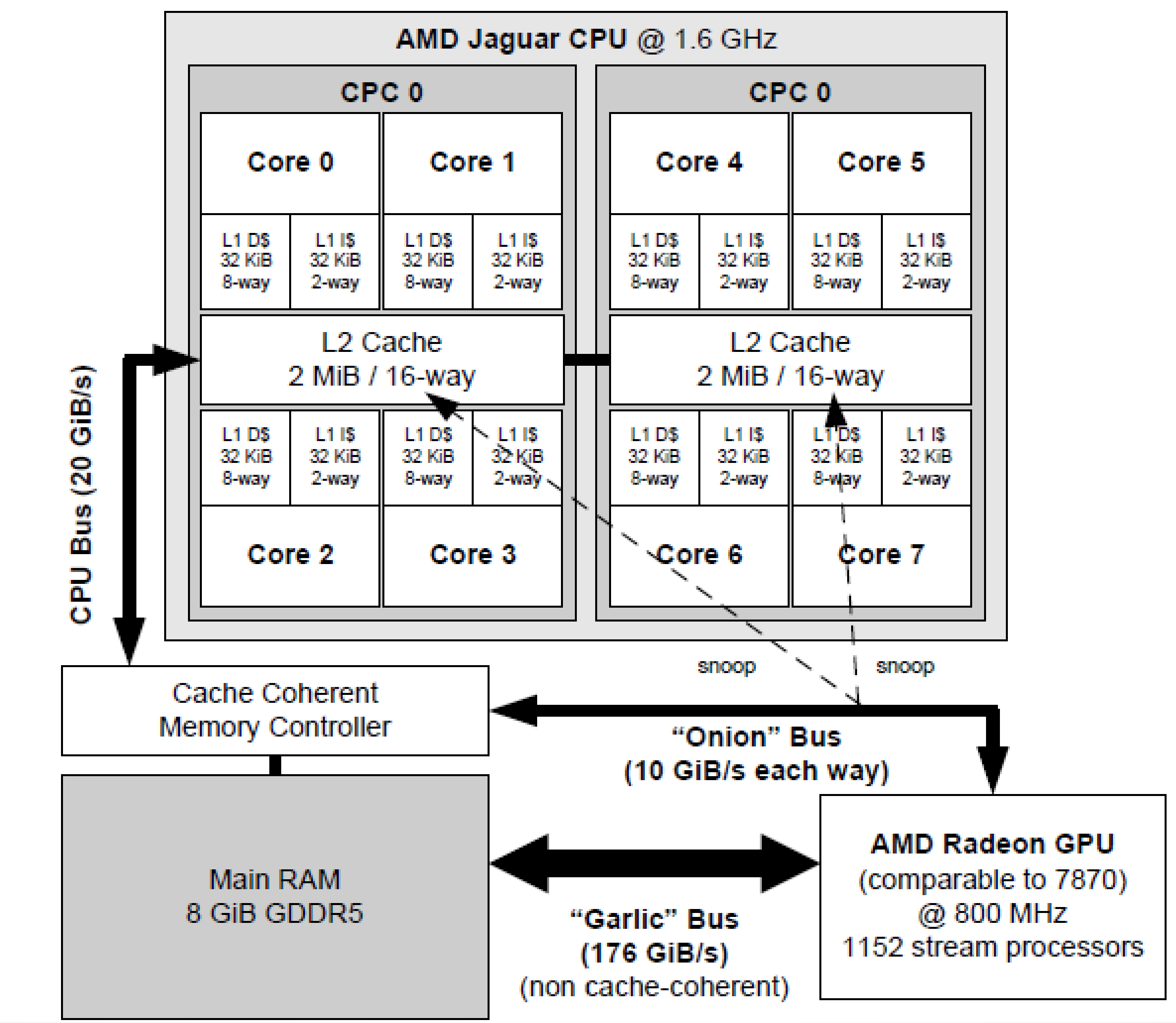
PS3


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

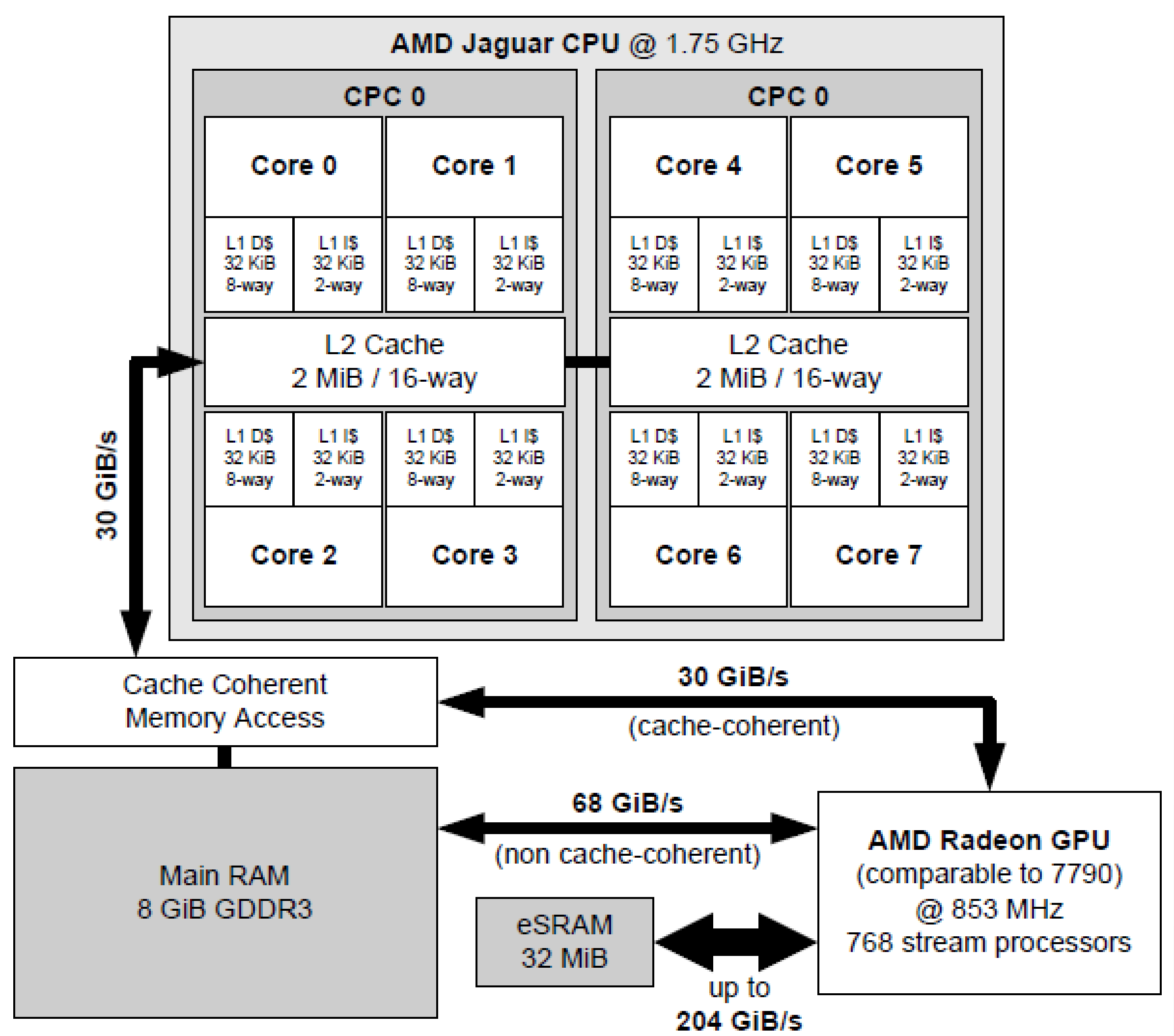
PS 4


AMD Jaguar CPU @ 1.6 GHz
CPC 0
Core 0
Core 1
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L2 Cache 2 MiB / 16-way
Core 2
Core 3
CPC 0
Core 4
Core 5
L2 Cache 2 MiB / 16-way
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
Core 0
Core 1
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L2 Cache 2 MiB / 16-way
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
Core 2
Core 3
CPC 0
Core 4
Core 5
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L2 Cache 2 MiB / 16-way
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
Core 6
Core 7
30 GiB/s
Cache Coherent Memory Access
30 GiB/s
(cache-coherent)
Main RAM 8 GiB GDDR3
68 GiB/s
(non cache-coherent)
AMD Radeon GPU (comparable to 7790) @ 853 MHz 768 stream processors
eSRAM 32 MiB
up to 204 GiB/s

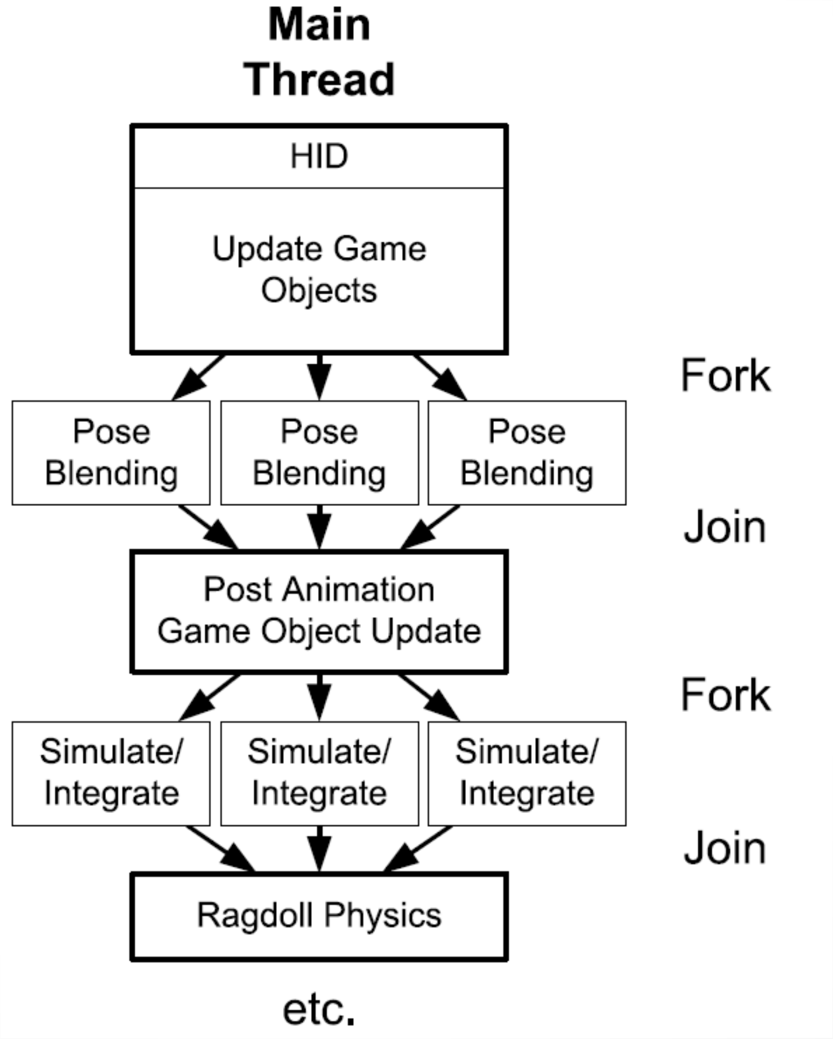
Seizing the power

- 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



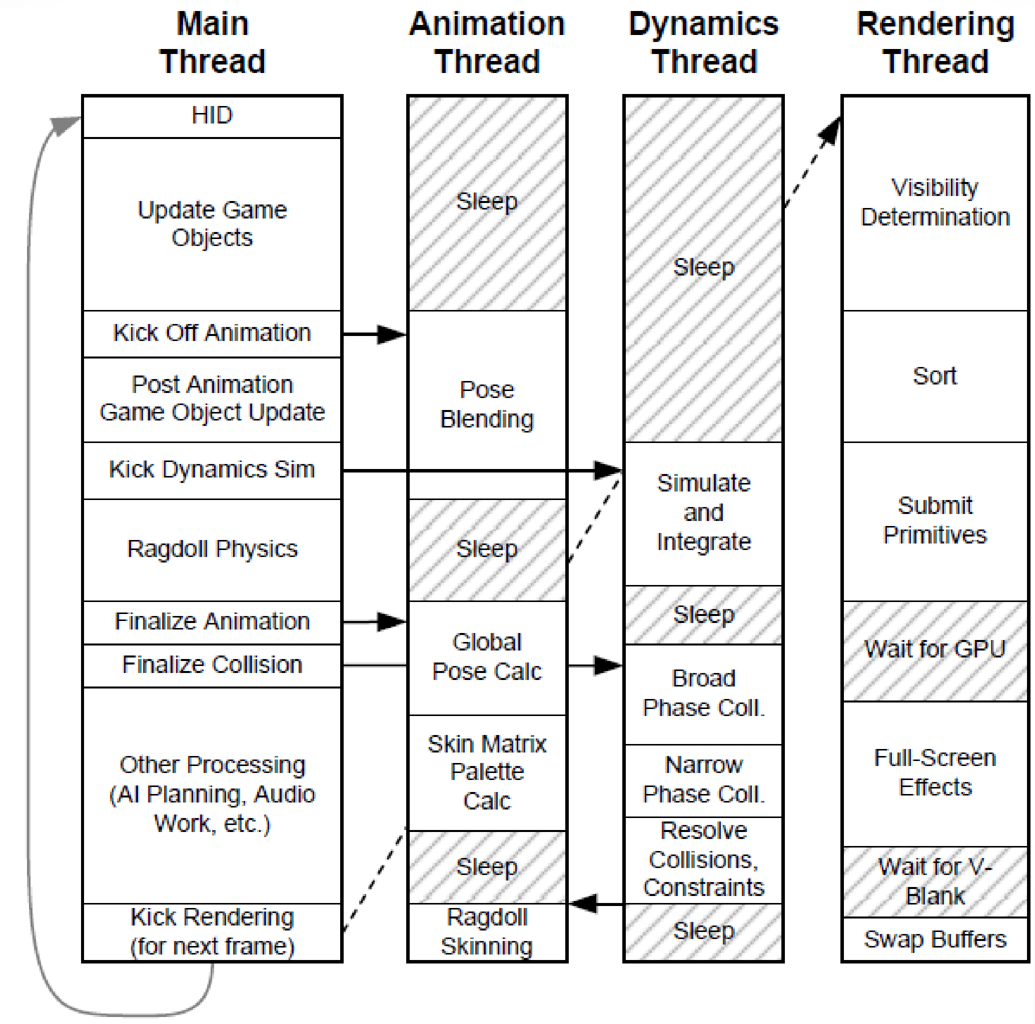
Example

- 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



Main 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)

Animation Thread

Sleep

Pose Blending

Sleep

Global Pose Calc

Skin Matrix Palette Calc

Sleep

Ragdoll Skinning

Dynamics Thread

Sleep

Simulate and Integrate

Sleep

Broad Phase Coll.

Narrow Phase Coll.

Resolve Collisions, Constraints

Sleep

Rendering Thread

Visibility Determination

Sort

Submit Primitives

Wait for GPU

Full-Screen Effects

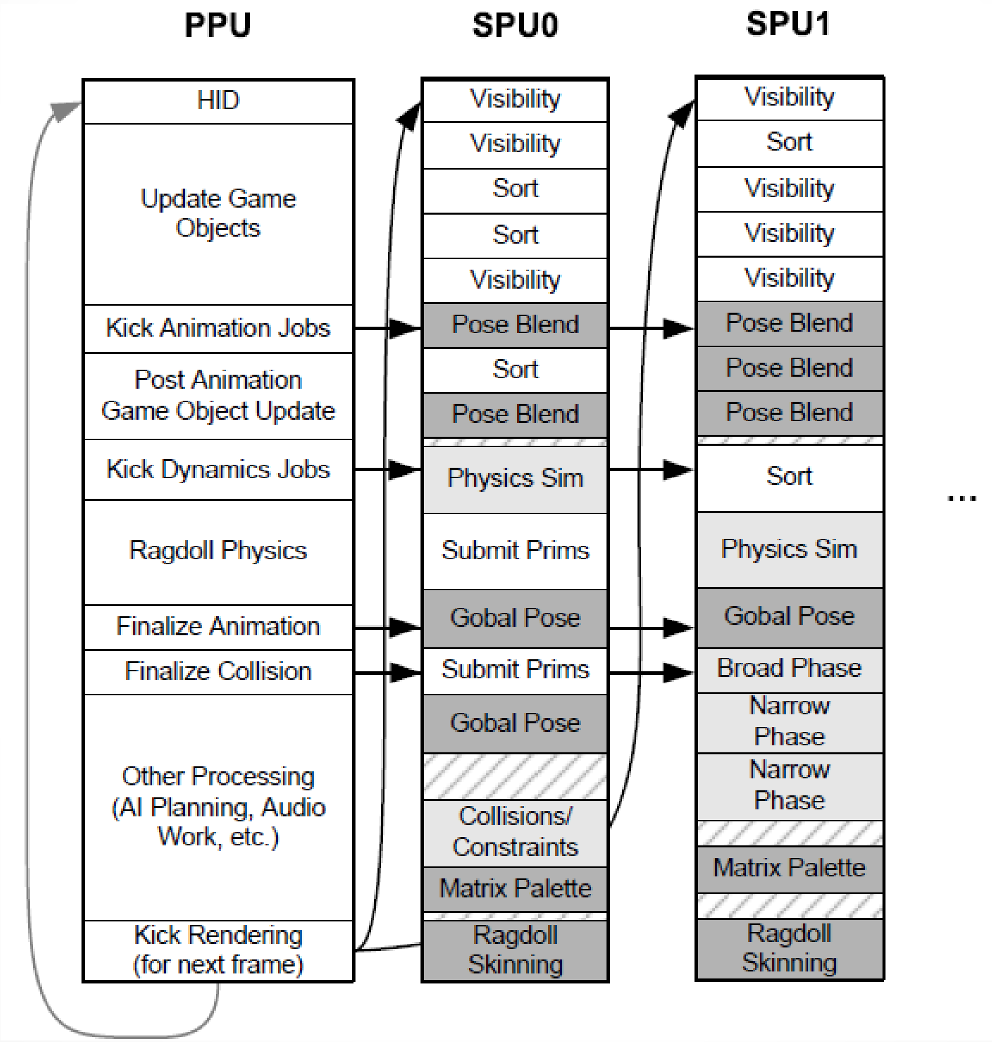
Wait for V-Blank

Swap Buffers

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

- 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)

SPU0

- 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

SPU1

- 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

Lab1: Animation Curve

- Animation Curve
  - Creates an animation curve from an arbitrary number of keyframes.
- Store a collection of Keyframes that can be evaluated over time.
- You can set up the keyframes in the inspector or in C# code.

  -
    ```
    public AnimationCurve bulletTimeScale = new
    AnimationCurve(new Keyframe(0,0), new Keyframe(3, 5),
    new Keyframe(5,2));
    ```
- Evaluate: Evaluate the curve at time.

  -
    ```
    m_UnscaledElapsedTime += Time.unscaledDeltaTime;
    ```

  -
    ```
    Time.timeScale =
    bulletTimeScale.Evaluate(m_UnscaledElapsedTime);
    ```

Lab4: Job System

- The job system lets you write simple and safe multithreaded code so that your application can use all available CPU cores to execute your code. This can help improve the performance of your application.
- You can use the job system by itself, but for improved performance, you should also use the Burst compiler, which is specifically designed to compile jobs for Unity's job system.
- The Burst compiler has improved code generation, which results in increased performance and a reduction of battery consumption on mobile devices.
- You can also use the job system with Unity's Entity Component System to create high performance data-oriented code.

Job Systems

- Add “com.unity.jobs” by name to the package manager.
- Add “Collections” package to the package manager.

No Job System

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


public class IJob1 : MonoBehaviour
{
    float ms = 0f;

    public void ExpensiveCalculation1()
    {

        float startTime = Time.realtimeSinceStartup;
        float value = 0.0f;
        for(int i=0; i<50000; i++)
        {
            value = math.exp10(math.sqrt(value));
        }

        ms = (Time.realtimeSinceStartup - startTime) *1000f;
        Debug.Log(((Time.realtimeSinceStartup - startTime) * 1000f) + "ms");
    }
    void Update()
    {
        ExpensiveCalculation1();
    }

    void OnGUI()
    {
        GUILayout.Label("No Job System: " + ms.ToString("f2") + "ms");
    }

}
```

Unity’s Job System

- A job is a small unit of work that does one specific task.
- A job receives parameters and operates on data, similar to how a method call behaves.
- Jobs can be self-contained, or they can depend on other jobs to complete before they can run.
- In Unity, a job refers to any struct that implements the IJob interface.
- Only the main thread can schedule and complete jobs.
- It can’t access the content of any running jobs, and two jobs can’t access the contents of a job at the same time.
- To ensure efficient running of jobs, you can make them dependent on each other.
- Unity’s job system allows you to create complex dependency chains to ensure that your jobs complete in the correct order.

Job types

- IJob: Runs a single task on a job thread.
- IJobParallelFor: Runs a task in parallel. Each worker thread that runs in parallel has an exclusive index to access shared data between worker threads safely.
- IJobParallelForTransform: Runs a task in parallel. Each worker thread running in parallel has an exclusive Transform from the transform hierarchy to operate on.
- IJobFor: The same as IJobParallelFor, but allows you to schedule the job so that it doesn't run in parallel.

Using Job System

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.Mathematics;
using Unity.Jobs;
using Unity.Collections;

public struct ExpensiveCalculation2_1 : IJob
{
    public void Execute()
    {
        float value = 0.0f;
        for (int i = 0; i < 50000; i++)
        {
            value = math.exp10(math.sqrt(value));       }

    }
}


public class IJob2_1 : MonoBehaviour
{
    float ms = 0f;
    void Update()
    {
        float startTime = Time.realtimeSinceStartup;

        ExpensiveCalculation2_1 job = new ExpensiveCalculation2_1();
        JobHandle jHandle = job.Schedule();
        jHandle.Complete();
        ms = (Time.realtimeSinceStartup - startTime) * 1000f;
        Debug.Log(((Time.realtimeSinceStartup - startTime) * 1000f) + "ms");

    }

}
```

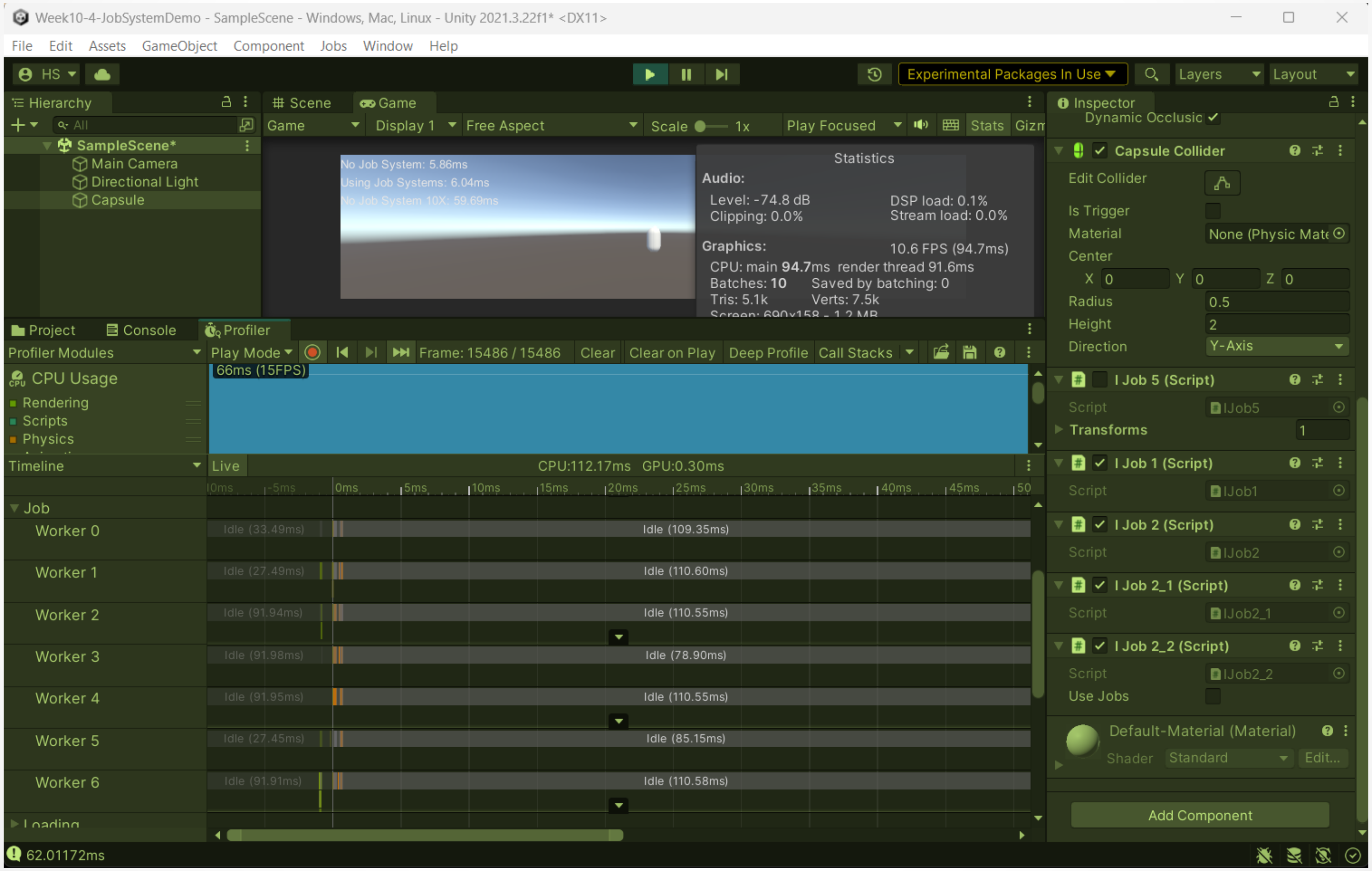
Running Jobs in Parallel

```
  private JobHandle ReallyToughTaskJob()
  {
      ExpensiveCalculation2_2 job = new ExpensiveCalculation2_2();
      JobHandle jHandle = job.Schedule();
      return jHandle;

  }


NativeList<JobHandle> jobHandleList = new
NativeList<JobHandle>(Allocator.Temp);
 for (int i = 0; i < 10; i++)
 {
     JobHandle jobHandle = ReallyToughTaskJob();
     jobHandleList.Add(jobHandle);
}
 JobHandle.CompleteAll(jobHandleList);
 jobHandleList.Dispose();
```

Profiler

- Window→ Analysis → Profiler
- Click on “Live” button next to “TimeLine” at the bottom
- Open the Job tab to see the threads

Not using Jobs in the Profiler



Week10-4-JobSystemDemo - SampleScene - Windows, Mac, Linux - Unity 2021.3.22f1* <DX11>

No Job System: 5.86ms
Using Job Systems: 6.04ms
No Job System 10X: 59.69ms

Statistics

Audio:
Level: -74.8 dB   DSP load: 0.1%
Clipping: 0.0%   Stream load: 0.0%

Graphics: 10.6 FPS (94.7ms)
CPU: main **94.7**ms render thread 91.6ms
Batches: **10**   Saved by batching: 0
Tris: 5.1k   Verts: 7.5k
Screen: 690x158 - 1.2 MB

Frame: 15486 / 15486

66ms (15FPS)

CPU:112.17ms GPU:0.30ms

Worker 0	Idle (33.49ms)	Idle (109.35ms)
Worker 1	Idle (27.49ms)	Idle (110.60ms)
Worker 2	Idle (91.94ms)	Idle (110.55ms)
Worker 3	Idle (91.98ms)	Idle (78.90ms)
Worker 4	Idle (91.95ms)	Idle (110.55ms)
Worker 5	Idle (27.45ms)	Idle (85.15ms)
Worker 6	Idle (91.91ms)	Idle (110.58ms)

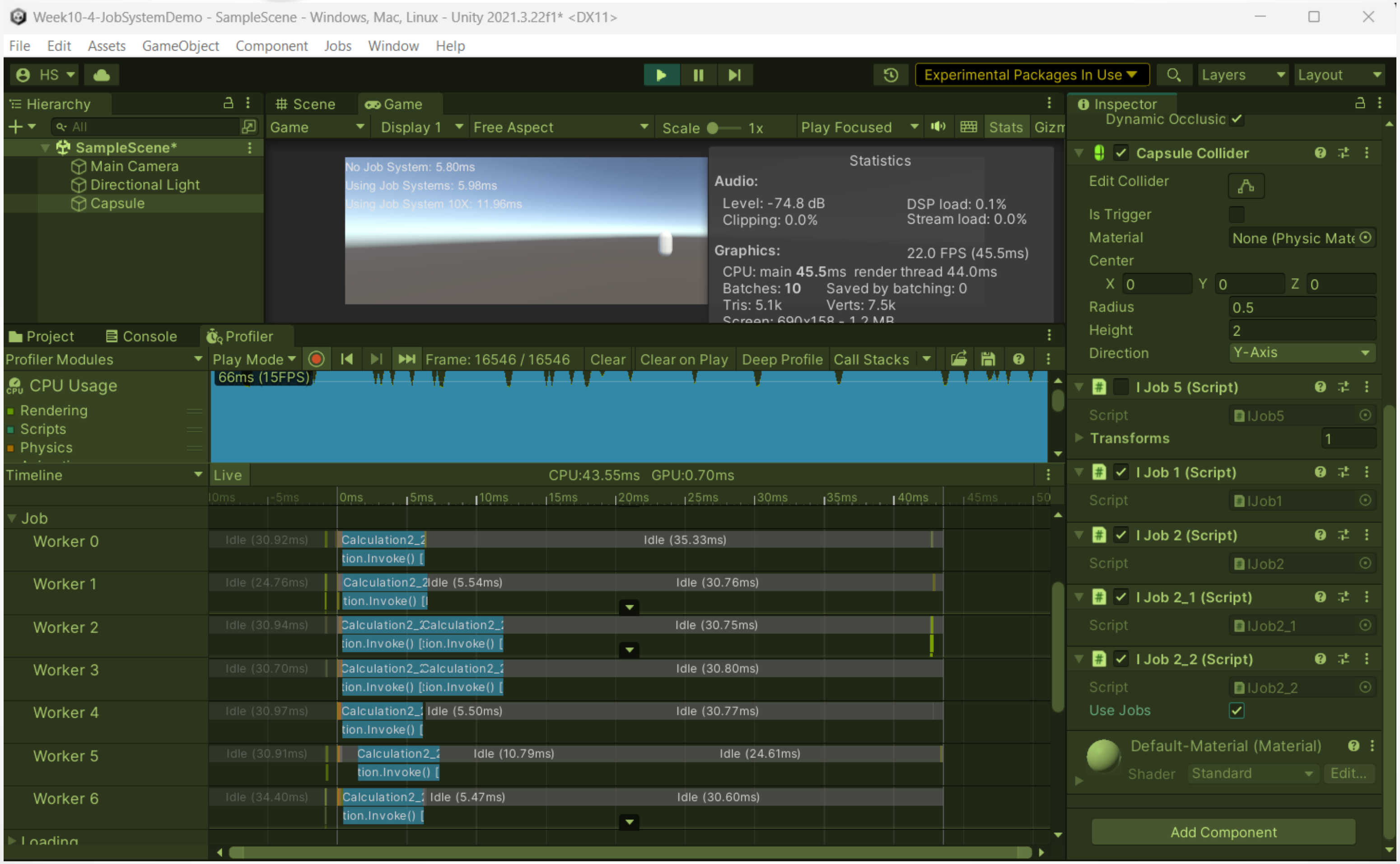
62.01172ms

Capsule Collider
Is Trigger ☐
Material: None (Physic Mate
Center: X 0 Y 0 Z 0
Radius: 0.5
Height: 2
Direction: Y-Axis

I Job 5 (Script) — Script: IJob5; Transforms: 1
I Job 1 (Script) — Script: IJob1
I Job 2 (Script) — Script: IJob2
I Job 2_1 (Script) — Script: IJob2_1
I Job 2_2 (Script) — Script: IJob2_2; Use Jobs ☐

Default-Material (Material) — Shader: Standard

Using Jobs in the Profiler



Week10-4-JobSystemDemo - SampleScene - Windows, Mac, Linux - Unity 2021.3.22f1* <DX11>

SampleScene*
Main Camera
Directional Light
Capsule

Game
Scale 1x

No Job System: 5.80ms
Using Job Systems: 5.98ms
Using Job System 10X: 11.96ms

```
Statistics
Audio:
  Level: -74.8 dB            DSP load: 0.1%
  Clipping: 0.0%             Stream load: 0.0%
Graphics:                    22.0 FPS (45.5ms)
  CPU: main 45.5ms  render thread 44.0ms
  Batches: 10     Saved by batching: 0
  Tris: 5.1k      Verts: 7.5k
  Screen: 690x158 - 1.2 MB
```

Frame: 16546 / 16546

66ms (15FPS)

CPU:43.55ms   GPU:0.70ms

-10ms -5ms 0ms 5ms 10ms 15ms 20ms 25ms 30ms 35ms 40ms 45ms 50

```
Job
  Worker 0   Idle (30.92ms)  Calculation2_2 / tion.Invoke() [                      Idle (35.33ms)
  Worker 1   Idle (24.76ms)  Calculation2_2 / tion.Invoke() [   Idle (5.54ms)       Idle (30.76ms)
  Worker 2   Idle (30.94ms)  Calculation2_2 Calculation2_2 / tion.Invoke() [tion.Invoke() [   Idle (30.75ms)
  Worker 3   Idle (30.70ms)  Calculation2_2 Calculation2_2 / tion.Invoke() [tion.Invoke() [   Idle (30.80ms)
  Worker 4   Idle (30.97ms)  Calculation2_2 / tion.Invoke() [   Idle (5.50ms)       Idle (30.77ms)
  Worker 5   Idle (30.91ms)  Calculation2_2 / tion.Invoke() [   Idle (10.79ms)      Idle (24.61ms)
  Worker 6   Idle (34.40ms)  Calculation2_2 / tion.Invoke() [   Idle (5.47ms)       Idle (30.60ms)
Loading
```

Dynamic Occlusic ✓

☑ Capsule Collider
Edit Collider
Is Trigger □
Material: None (Physic Mate
Center: X 0   Y 0   Z 0
Radius: 0.5
Height: 2
Direction: Y-Axis

□ I Job 5 (Script)
Script: IJob5
Transforms: 1

☑ I Job 1 (Script)
Script: IJob1

☑ I Job 2 (Script)
Script: IJob2

☑ I Job 2_1 (Script)
Script: IJob2_1

☑ I Job 2_2 (Script)
Script: IJob2_2
Use Jobs ☑

Default-Material (Material)
Shader: Standard   Edit...

Using Burst Compiler

- Go to Editor, Jobs → Burst → EnableCompilation
- using Unity.Burst;

- Add Burst Compile attribute to your Job!

```
[BurstCompile]
public struct ExpensiveCalculation2_3 : IJob
{
    public void Execute()
    {

        //make sure we don't have any built-in Unity components like Transform.position, Animator.set
        float value = 0.0f;
        for (int i = 0; i < 50000; i++)
        {
            value = math.exp10(math.sqrt(value));
        }

    }
}
```