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

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

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

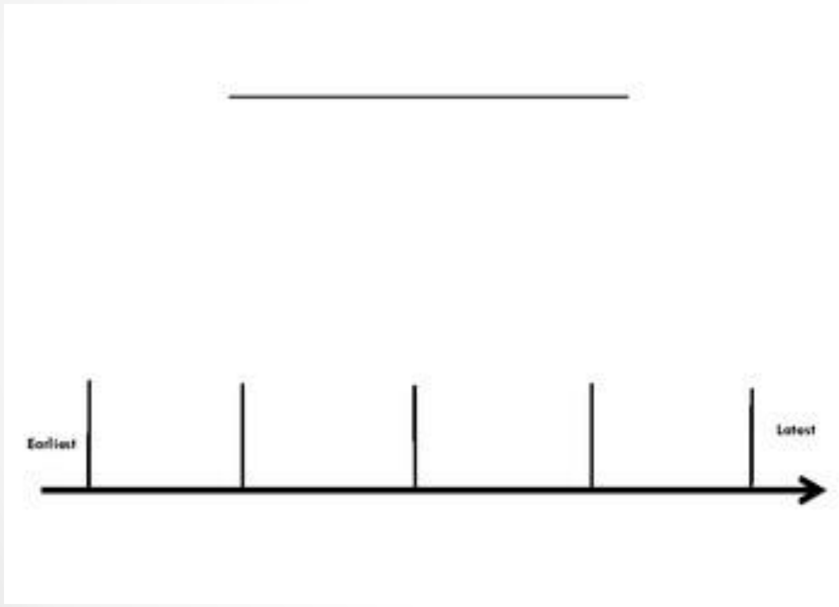
Callback-driven

- 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

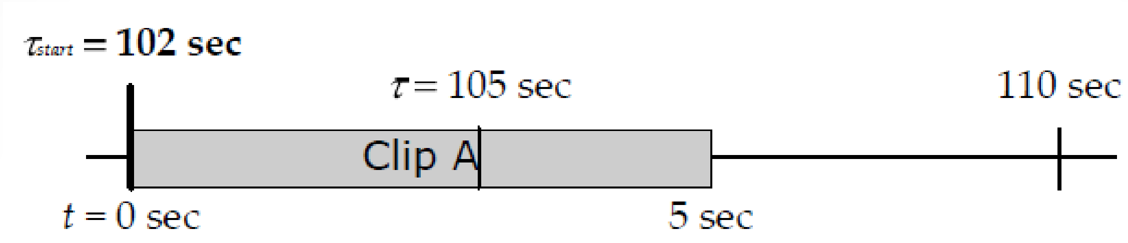
- Another way to design the loop is to use an event system
- Works like an input listener, but uses an event bus for the components to speak with one another
- Most game engines have an *event system*, which permits various engine subsystems to register interest in particular kinds of events and to respond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

Abstract timelines

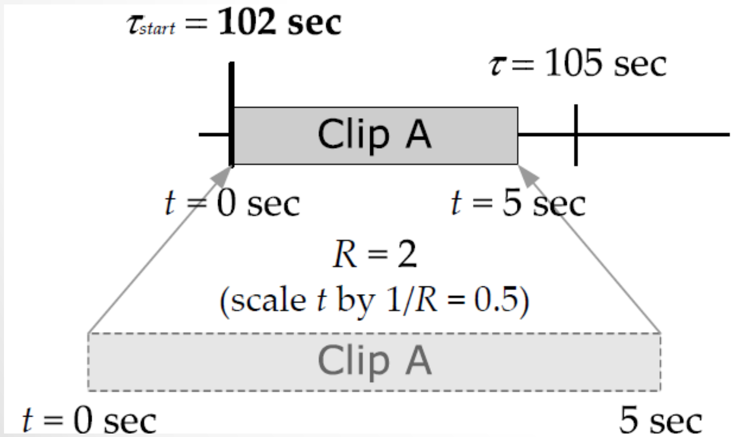


- Real time – measure by the CPU high resolution timer
- Game time – mostly controlled by real-time, but can be slowed, sped up, or paused
- Local and global time – animations have their own timeline. We can map it to global time in any way we want (translation, scaling)

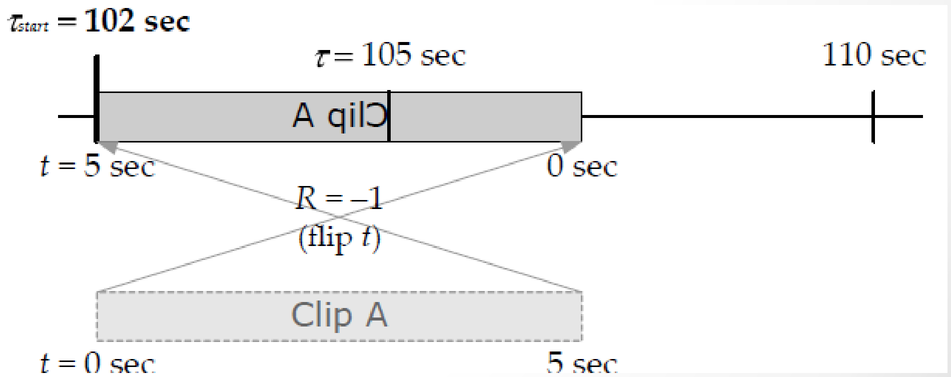
Mapping time



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: $x2 = x1 + \Delta x = x1 + v\Delta t$.

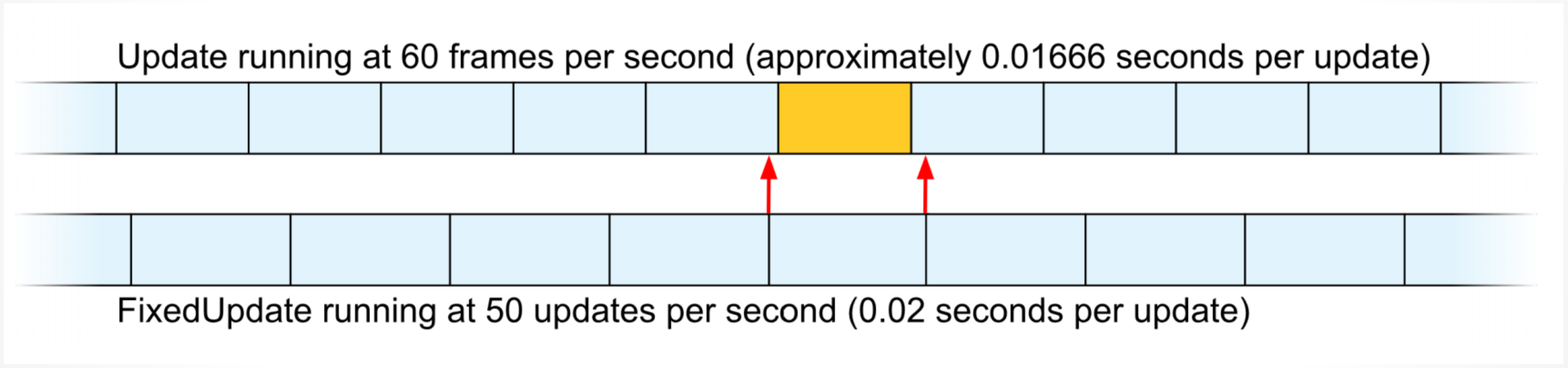
Unity Time Systems

- Unity has two systems which track time, one with a variable amount of time between each step, and one with a fixed amount of time between each step.

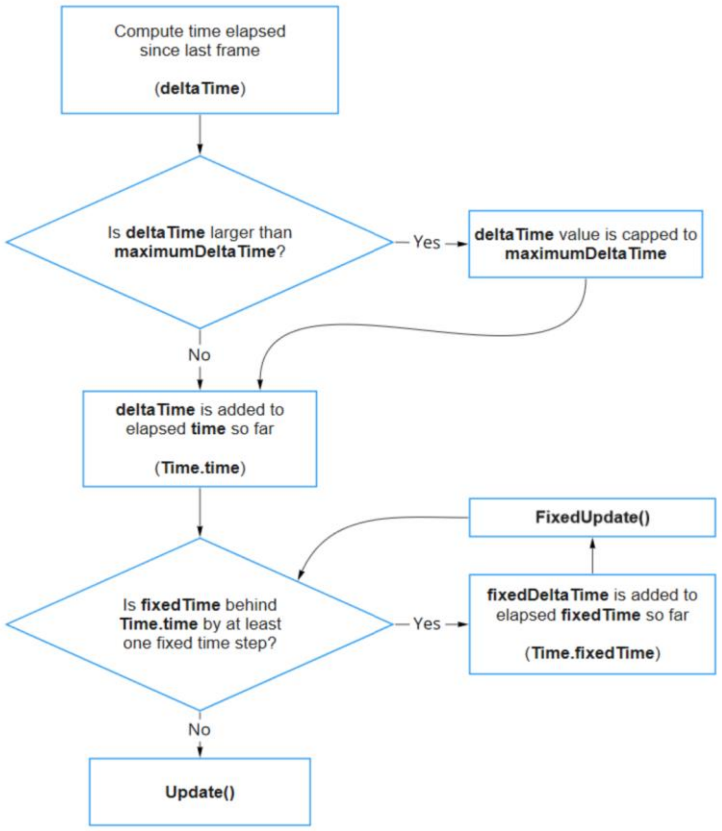
- The variable time step system operates on the repeated process of drawing a frame to the screen and running your app or game code once per frame.

- The fixed time step system steps forward at a pre-defined amount each step and is not linked to the visual frame updates. It is more commonly associated with the physics system, which runs at the rate specified by the fixed time step size, but you can also execute your own code each fixed time step if necessary.

Variable vs. Fixed Time Steps


Update running at 60 frames per second (approximately 0.01666 seconds per update)
FixedUpdate running at 50 updates per second (0.02 seconds per update)

Unity's Time Logic


Compute time elapsed since last frame (deltaTime)
Is deltaTime larger than maximumDeltaTime?
Yes
deltaTime value is capped to maximumDeltaTime
No
deltaTime is added to elapsed time so far (Time.time)
FixedUpdate()
Is fixedTime behind Time.time by at least one fixed time step?
Yes
fixedDeltaTime is added to elapsed fixedTime so far (Time.fixedTime)
No
Update()

Controlling and handling variations in time

- Elapsed time variations can be slight.
- For example, in a game running at 60 frames per second, the actual number of frames per second may vary slightly, so that each frame lasts between 0.016 and 0.018 seconds.
- Larger variations can occur when your app performs heavy computations or garbage collection, or when the resources it needs to maintain its frame rate are being used by a different app.

Time Class in Unity

- Time.time indicates the amount of time elapsed since the player started, and so usually continuously and steadily rises.
- Time.deltaTime indicates the amount of time elapsed since the last frame, and so ideally remains fairly constant.
- Both these values are subjective measures of time elapsed within your app or game. This means they take into account any scaling of time that you apply.
  - For example, you could set the Time.timeScale to 0.1 for a slow-motion effect (which indicates 10% of normal playback speed). In this situation the value reported by Time.time increases at 10% the rate of "real" time. After 10 seconds, the value of Time.time would have increased by 1.
- You can set Time.timeScale to zero to pause your game, in which case the Update method is still called, but Time.time does not increase at all, and Time.deltaTime is zero.
- These values are also clamped by the value of the Time.maximumDeltaTime property. This means the length of any pauses or variations in frame rate reported by these properties will never exceed maximumDeltaTime.
  - For example, if a delay of one second occurs, but the maximumDeltaTime is set to the default value of 0.333, Time.time would only increase by 0.333, and Time.deltaTime would equal 0.333, despite more time having actually elapsed in the real world.

Lab1: Countdown timer

- Objective: Use time.deltaTime to build a countdown timer.
- Create a legacy text in the hierarchy (change the font color to white)
- Go to https://fonts.google.com/specimen/Press+Start+2P
- Download Press Start and import it under your "Assets" in the project tab and change the font of the text to Press Start.
- Create a C# script "Countdown" and attach it to the camera.

Countdown script

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

public class Countdown : MonoBehaviour
{
    public float timeStart = 60;
    public Text textBox;
    // Start is called before the first frame update
    void Start()
    {
        textBox.text = timeStart.ToString();
    }

    // Update is called once per frame
    void Update()
    {
        timeStart -= Time.deltaTime;
        textBox.text = math.round(timeStart).ToString();
    }
}
```

Lab2: Stopwatch

- Forget CountDown, CountUp
- Create a new scene.
- Add a legacy text and a Text Mesh Pro button.
- Create a C# script StopWatch and add it to the Camera (next page)
- Assign Camera to "onClick" event of the "Button" in the hierarchy and choose TimerButton.
- Open the main camera in the inspector
  - Assign text (Legacy) from the hierachy to “TextBox”
  - Assign TextButton from the hierarchy to “TextButton”

StopWatch Script

```
//Attach this script to the Camera Object and assign Camera to "onClick" event of the "Button" in the hierarchy and choose TimerButton

using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.UI;

public class StopWatch : MonoBehaviour
{
    public float timeStart;
    public Text textBox;

    [SerializeField]
    private TextMeshProUGUI textMeshPro;

    bool timerActive = false;

    // Start is called before the first frame update
    void Start()
    {
        //two digit text
        textBox.text = timeStart.ToString("F2");

        textMeshPro.text = "Start";
    }

    // Update is called once per frame
    void Update()
    {
        if (timerActive)
        {
            timeStart += Time.deltaTime;
            textBox.text = timeStart.ToString("F2");
        }

    }

    public void TimerButton()
    {
        timerActive = !timerActive;
        if (timerActive)
        {
            textMeshPro.text = "Pause";
            textMeshPro.color = Color.red;
        } else
        {
            textMeshPro.text = "Start";
            textMeshPro.color = Color.green;
        };
    }
}
```

Event System

- Note: You have a new GameObject called “Event System” in your Unity Hierarchy!
- The Event System in Unity is a manager that facilitates communication between Event System modules and sends events to objects in an application based on input.
- The Event System component is automatically created when a UI is added to a scene, and it contains Input Modules that determine how GameObjects in the UI respond to input.
- If your scene doesn't have an EventSystem, Unity UI won't recognize input such as navigating between different UI Buttons and clicking them. You can add one by selecting menu item GameObject > UI > EventSystem.

Time Settings

- The Time settings (menu: Edit > Project Settings, then the Time category) lets you set several properties that control timing within your game.

Property:	Function:
Fixed Timestep	A framerate-independent interval that dictates when physics calculations and **FixedUpdate()** events are performed.
Maximum Allowed Timestep	A framerate-independent interval that caps the worst case scenario when frame-rate is low. Physics calculations and **FixedUpdate()** events will not be performed for longer time than specified.
Time Scale	The speed at which time progresses. Change this value to simulate bullet-time effects. A value of 1 means real-time. A value of .5 means half speed; a value of 2 is double speed.
Maximum Particle Timestep	A framerate-independent interval that controls the accuracy of the particle simulation. When the frame time exceeds this value, multiple iterations of the particle update are performed in one frame, so that the duration of each step does not exceed this value. For example, a game running at 30fps (0.03 seconds per frame) could run the particle update at 60fps (in steps of 0.0167 seconds) to achieve a more accurate simulation, at the expense of performance.

Time Scale

- For special time effects such as slow-motion, time-based calculations in your code might happen at a slower pace.
- You may sometimes want to freeze game time completely, as when the game is paused.
- Unity has a Time Scale property that controls how fast game time proceeds relative to real time.
  - If you set the scale to 1.0, your in-game time matches real time. A value of 2.0 makes time pass twice as quickly in Unity (ie, the action will be speeded-up) while a value of 0.5 will slow gameplay down to half speed.
  - A value of zero will make your in-game time stop completely.
  - Note that the time scale doesn't actually slow execution but instead changes the time step reported to the Update and FixedUpdate functions with Time.deltaTime and Time.fixedDeltaTime. Your Update function may be called just as often when you reduce your time scale, but the value of deltaTime each frame will be less. Other script functions aren't affected by the time scale, so you can for example display a GUI with normal interaction when the game is paused.

Lab3: TimeScale

- Create a new scene
- Attach the TimeScript to the camera
- In this scipt:
    - Wait two seconds and display waited time.
    - This is typically just beyond 2 seconds.
    - Allow the speed of the time to be increased or decreased.
    - It can range between 0.5 and 2.0. These changes only happen when the timer restarts.

TimeScript

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


public class TimeScript : MonoBehaviour
{
    private float waitTime = 2.0f;
    private float timer = 0.0f;
    private float visualTime = 0.0f;
    private int width, height;
    private float value = 10.0f;
    private float scrollBar = 1.0f;

    void Awake()
    {
        width = Screen.width;
        height = Screen.height;
        Time.timeScale = scrollBar;
    }

    void Update()
    {
        timer += Time.deltaTime;

        // Check if we have reached beyond 2 seconds.
        // Subtracting two is more accurate over time than resetting to zero.
        if (timer > waitTime)
        {
            visualTime = timer;

            // Remove the recorded 2 seconds.
            timer = timer - waitTime;
            Time.timeScale = scrollBar;
        }
    }
```

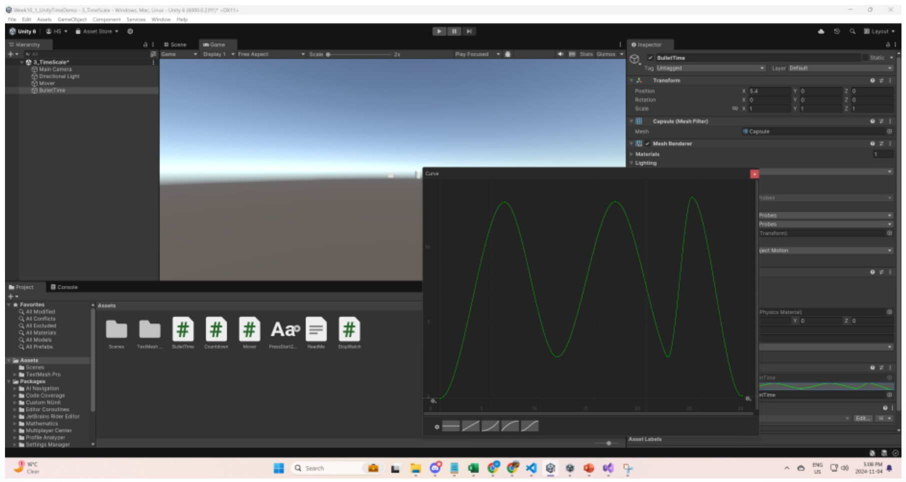
```
 void OnGUI()
    {
        GUIStyle sliderDetails = new GUIStyle(GUI.skin.GetStyle("horizontalSlider"));
        GUIStyle sliderThumbDetails = new
GUIStyle(GUI.skin.GetStyle("horizontalSliderThumb"));
        GUIStyle labelDetails = new GUIStyle(GUI.skin.GetStyle("label"));

        // Set the size of the fonts and the width/height of the slider.
        labelDetails.fontSize = 6 * (width / 200);
        sliderDetails.fixedHeight = height / 32;
        sliderDetails.fontSize = 12 * (width / 200);
        sliderThumbDetails.fixedHeight = height / 32;
        sliderThumbDetails.fixedWidth = width / 32;

        // Show the slider. Make the scale to be ten times bigger than the
needed size.
        value = GUI.HorizontalSlider(new Rect(width / 8, height / 4, width - (4 *
width / 8), height - (2 * height / 4)),
            value, 5.0f, 20.0f, sliderDetails, sliderThumbDetails);

        // Show the value from the slider. Make sure that 0.5, 0.6... 1.9, 2.0 are
shown.
        float v = ((float)Mathf.RoundToInt(value)) / 10.0f;
        GUI.Label(new Rect(width / 8, height / 3.25f, width - (2 * width / 8), height
- (2 * height / 4)),
            "timeScale: " + v.ToString("f1"), labelDetails);
        scrollBar = v;

        // Display the recorded time in a certain size.
        labelDetails.fontSize = 14 * (width / 200);
        GUI.Label(new Rect(width / 8, height / 2, width - (2 * width / 8), height - (2
* height / 4)),
            "Timer value is: " + visualTime.ToString("f4") + " seconds.", labelDetails);
    }
}
```

Lab4: Animation Curve

- Change the time scale based on an animation curve
- Create a new Scene
- Add a 3D capsule object “Bullet Time”
- Attach the BulletTime Script (Next Page)
- Note: Time.unscaledTime is The timeScale-independent time for a frame. This is the time in seconds since the start of the game.

Animation Curve

- An Animation Curve has multiple keys which are control points that the curve passes through. These are visualized in the Curve Editor as small diamond shapes on the curves. A frame in which one or more of the shown curves have a key is called a keyframe



BulletTime Script

```
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;
         }

    }
}
```

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

Capture frame rate

- A special case of time management is where you want to record gameplay as a video. Since the task of saving screen images takes considerable time, the game's normal frame rate is reduced, and the video doesn't reflect the game's true performance.

- To improve the video's appearance, use the Capture Framerate property. The property's default value is 0, for unrecorded gameplay.

- For recording. When you set the property's value to anything other than zero, game time is slowed, and the frame updates are issued at precise regular intervals. The interval between frames is equal to 1 / Time.captureFramerate, so if you set the value to 5.0 then updates occur every fifth of a second.
- With the demands on frame rate effectively reduced, you have time in the Update function to save screenshots or take other actions:

Lab5: CaptureFrame

- Create a new scene
- Add a 3D sphere
- Attach a Mover.cs script

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

- Attach CaptureFrameScript to the camera (next page)

CaptureFrame

```
using UnityEngine;
using System.Collections;

public class CaptureFrameScript : MonoBehaviour
{
    // Capture frames as a screenshot sequence. Images are
    // stored as PNG files in a folder - these can be combined into
    // a movie using image utility software (eg, QuickTime Pro).
    // The folder to contain our screenshots.
    // If the folder exists we will append numbers to create an empty folder.
    string folder = "ScreenshotFolder";
    int frameRate = 25;

    void Start()
    {
        // Set the playback frame rate (real time will not relate to game time after this).
        Time.captureFramerate = frameRate;

        // Create the folder
        System.IO.Directory.CreateDirectory(folder);
    }

    void Update()
    {
        // Append filename to folder name (format is '0005 shot.png"')
        string name = string.Format("{0}/{1:D04} shot.png", folder, Time.frameCount);

        // Capture the screenshot to the specified file.
        ScreenCapture.CaptureScreenshot(name);
    }
}
```

Lab6: Coroutines

```
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting.Antlr3.Runtime;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.SocialPlatforms;

public class Coroutines : MonoBehaviour
{

    public bool Run;

    // Toggles the time scale between 1 and 0.5
    // 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()
    {
        //This can be used for slow motion effects or to speed up your application.
        //When timeScale is 1.0, time passes as fast as real time.
        //When timeScale is 0.5 time passes 2x slower than realtime.
        //When timeScale is set to zero your application acts as if paused if all your functions are frame rate
independent. Negative values are ignored.
        //Time.timeScale = 0.5f;
        //To call the coroutine
        StartCoroutine(MyCoroutine());
    }

IEnumerator MyCoroutine()
  {
      Debug.Log(Time.frameCount);
      //Print the time of when the function is first called.
      Debug.Log("Started Coroutine at timestamp : " + 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 5 seconds.
      //yield on a new Yield Instruction that waits for 5 seconds.
      //It suspends the coroutine eeecution for the given amount of seconds using scaled time.
      //yield return new WaitForSeconds(5);

      //Suspends the coroutine execution for the given amount of seconds using unscaled time.
      //yield return new WaitForSecondsRealtime(5f);


      //Debug.Log("Run is false");

      //This one take a predicate
      yield return new WaitUntil(() => Run == true);

      //Debug.Log("Run is true");

      Debug.Log(Time.frameCount);
      Debug.Log("Finished Coroutine at timestamp : " + Time.time);
  }

  void Update()
  {
      if (Input.GetButtonDown("Fire1"))
      {
          if (Time.timeScale == 1.0f)
              Time.timeScale = 0.5f;
          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);
      }
  }
}
```

Coroutines

- A coroutine allows you to spread tasks across several frames. 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.
- By default, Unity resumes a coroutine on the frame after a yield statement. If you want to introduce a time delay, use WaitForSeconds.
- To stop a coroutine, use StopCoroutine and StopAllCoroutines.
- A coroutine also stops if you've set SetActive to false to disable the GameObject the coroutine is attached to.
- Calling Destroy(example) (where example is a MonoBehaviour instance) immediately triggers OnDisable and Unity processes the coroutine, effectively stopping it.

Job System

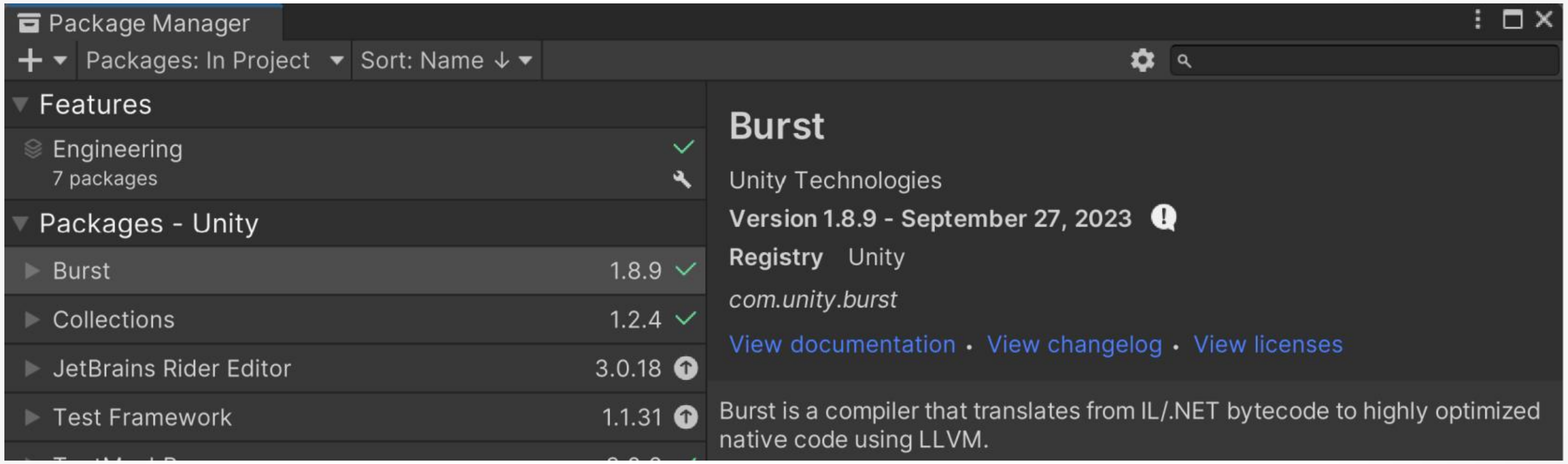
- The job system lets you write simple and safe multithreaded code so that your application can use all available CPU cores to execute your code. This can help improve the performance of your application.
- You can use the job system by itself, but for improved performance, you should also use the Burst 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.

Lab7: Job Systems

- Install "com.unity.jobs" by name in the package manager.
- Install "Collections", "Burst", and "Mathematics" packages to the package manager.
- Burst is a compiler that translates from IL/.NET bytecode to highly optimized native code using LLVM.
- Select "Burst" in the package manager and click on "View ChangeLog"
- Click on drop down "Burst 1.6.6" and select the latest version…1.8.9 or later
- Go to explorer, Go to packages folder and open the .json file, and under "dependencies" add/update "com.unity.burst": "1.8.9",

Burst

- Restart the editor, and you should see:



Package Manager

Packages: In Project | Sort: Name ↓

Features

Engineering ✓
7 packages

Packages - Unity

Burst	1.8.9	✓
Collections	1.2.4	✓
JetBrains Rider Editor	3.0.18	↑
Test Framework	1.1.31	↑

Burst

Unity Technologies

Version 1.8.9 - September 27, 2023

Registry Unity

com.unity.burst

View documentation · View changelog · View licenses

Burst is a compiler that translates from IL/.NET bytecode to highly optimized native code using LLVM.

Lab7–1: No Job System

- In this example, we create an expensive calculation that you must do in each frame.
- Create a new C# script IJob1 (next page) and attach it to the camera or a game object.
- Check Window > Analysis > Profiler to see the performance.

No Job System

```
uşîŋĝ Şỳşţêŋ Cộľľêçţîộŋş
uşîŋĝ Şỳşţêŋ Cộľľêçţîộŋş Ğêŋêsîç
uşîŋĝ ÛŋîţỳÉŋĝîŋê
uşîŋĝ Ûŋîţỳ Ňắţĥêŋắţîçş


řučľîç çľắşş Íĸộčˌ   ŇộŋộBêĥắŵîộus

    ğľộắţ ŋş   ˳ğ

    řučľîç ŵộîđ ÉỵřêŋşîŵêCắľçuľắţîộŋˌ


        ğľộắţ şţắşţŢîŋê   Ţîŋê sêắľţîŋêŞîŋçêŞţắşţuř
        ğľộắţ ŵắľuê    ˳ ˳ğ
        ğộs îŋţ î ˳  î _˳˳˳˳  î

            ŵắľuê   ŋắţĥ êỵřˌ˳ ŋắţĥ şrşţ ŵắľuê


        ŋş    Ţîŋê sêắľţîŋêŞîŋçêŞţắşţuř   şţắşţŢîŋê   ˌ˳˳˳ğ
        Dêčuĝ Ľộĝ   Ţîŋê sêắľţîŋêŞîŋçêŞţắşţuř   şţắşţŢîŋê    ˌ˳˳˳ğ     ŋş

    ŵộîđ Ûřđắţê

        ÉỵřêŋşîŵêCắľçuľắţîộŋˌ


    ŵộîđ ÔŋĞÛÍ

        ĞÛÍĽắỳộuţ Ľắčêľ  Ŋộ Ķộč Şỳşţêŋ      ŋş ŢộŞţsîŋĝ  ğˌ      ŋş
```

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, like 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.

IJob

- An interface that allows you to schedule a single job that runs in parallel to other jobs and the main thread.
- After a job is scheduled, the job's Execute method is invoked on a worker thread.
- You can use the returned JobHandle to make sure that the job has completed.
- You can also pass the JobHandle to other jobs as a dependency, which ensures that jobs are executed one after another on the worker threads.

Lab7–2: IJob

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

//Step1
using Unity.Jobs;
using Unity.Collections;

//Step2: create a structure that defines the job
public struct ExpensiveCalculation2 : IJob
{
   public NativeArray<float> Value;

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

   }
}
```

```
public class IJob2 : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        //for (int i = 0; i < 100000; i++)
        //{
        //    float f = Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);
        //}

        //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);

        ExpensiveCalculation2 job = new ExpensiveCalculation2()
        {
            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();


    }
}

}
```

Allocator

- Persistent: Slower (de)allocation, but can stay in memory. Needs to be deallocated manually.
- Temp Job: Faster (de)allocation and used for single jobs. Can be in memory for few frames.
- Temp: Fastest (de)allocation. Can be in memory for max 1 frame.

Check FPS using Job System

- using JOB system doesn't really change anything! Why? Becuase we are not really doing anything in parallel!

Lab7–3: IJob Performance

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

//Step1
using Unity.Jobs;
using Unity.Collections;
using System.Runtime.CompilerServices;

//Step2: create a structure that defines the job
public struct ExpensiveCalculation2_1 : 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));
        }

    }
}


public class IJob2_1 : MonoBehaviour
{
    float ms = 0f;
    // Start is called before the first frame update
    void Start()
    {

    }

  // Update is called once per frame
    void Update()
    {
        float startTime = Time.realtimeSinceStartup;

        ExpensiveCalculation2_1 job = new
ExpensiveCalculation2_1();

        JobHandle jHandle = job.Schedule();

        //the complete function ensures that the job is
completed.
        jHandle.Complete();
        ms = (Time.realtimeSinceStartup - startTime) * 1000f;
        Debug.Log(((Time.realtimeSinceStartup - startTime) *
1000f) + "ms");

    }


    void OnGUI()
    {
        GUILayout.BeginArea(new Rect(0, 20, 300, 100));
        GUILayout.Label("Using Job Systems: " + ms.ToString("f2") +
"ms");
        GUILayout.EndArea();

    }
}
```

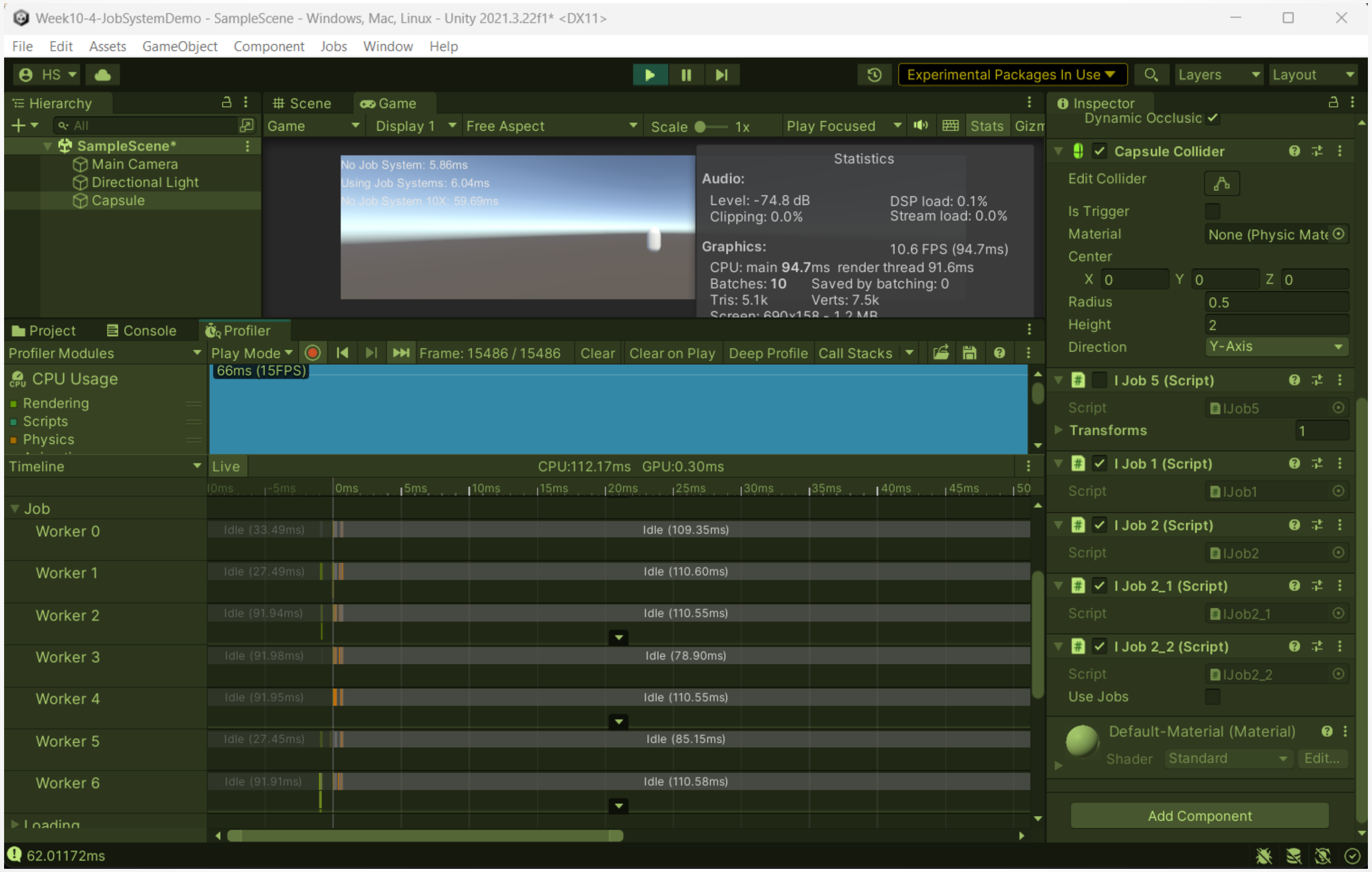
Lab 7-4: Run 10 Jobs in parallel

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

public struct ExpensiveCalculation2_2 : 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));
      }

   }
}


public class IJob2_2 : MonoBehaviour
{
   [SerializeField] private bool useJobs;
   float ms = 0f;
   // Start is called before the first frame update
   void Start()
   {
   }
 void Update()
   {
      float startTime = Time.realtimeSinceStartup;

      if (useJobs)
      {
         NativeList<JobHandle> jobHandleList = new NativeList<JobHandle>(Allocator.Temp);
         for (int i = 0; i < 10; i++)
         {
            JobHandle jobHandle = ReallyToughTaskJob();
            jobHandleList.Add(jobHandle);

            //the complete function ensures that the job is completed.
            //jHandle.Complete();
         }
         JobHandle.CompleteAll(jobHandleList);
         jobHandleList.Dispose();
      }
      else
      {
         for (int i = 0; i < 10; i++)
         {
           ReallyToughTask();
         }
      }

      ms = (Time.realtimeSinceStartup - startTime) * 1000f;
      Debug.Log(((Time.realtimeSinceStartup - startTime) * 1000f) + "ms");

   }
void OnGUI()
   {
      GUILayout.BeginArea(new Rect(0, 40, 300, 100));
      if (useJobs)
      {
         GUILayout.Label("Using Job System 10X: " + ms.ToString("f2") + "ms");
      }
      else
      {
         GUILayout.Label("No Job System 10X: " + ms.ToString("f2") + "ms");
      }
      GUILayout.EndArea();

   }


   private void ReallyToughTask()
   {

      //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));
      }

   }

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

   }
}
```

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 94.7ms  render thread 91.6ms
  Batches: 10      Saved by batching: 0
  Tris: 5.1k       Verts: 7.5k
  Screen: 690x158 - 1.2 MB
```

Profiler — Frame: 15486 / 15486 — 66ms (15FPS)

Timeline — Live — CPU:112.17ms GPU:0.30ms

```
Job
  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)
```

Inspector — Capsule Collider: Is Trigger unchecked; Material: None (Physic Material); 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

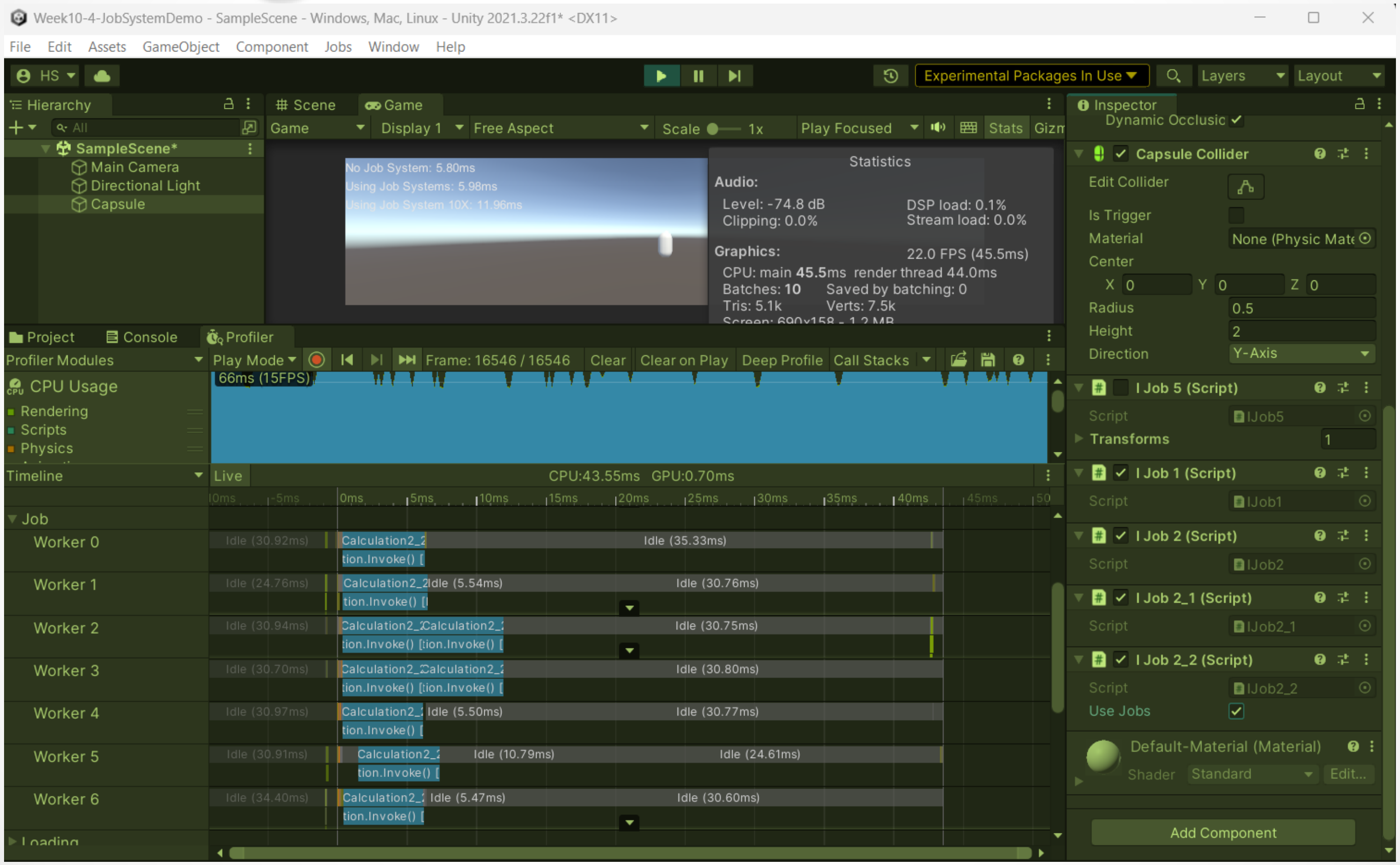
I Job 2 (Script) — Script: IJob2

I Job 2_1 (Script) — Script: IJob2_1

I Job 2_2 (Script) — Script: IJob2_2 — Use Jobs unchecked

62.01172ms

Using Jobs in the Profiler



```
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

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)
```

```
Dynamic Occlusion ✓
Capsule Collider ✓
  Is Trigger
  Material     None (Physic Material)
  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
```

Burst

- Burst is a compiler that you can use with Unity's job system to create code that enhances and improves your application's performance. It translates your code from IL/.NET bytecode to optimized native CPU code that uses the LLVM compiler.

Lab 7-5: Using Burst Compiler

- When you install “Collections” from the package manager, it installs “Burst” package as well.
- Go to Editor, Jobs → Burst → EnableCompilation
- using Unity.Burst;

- Add Burst Compile attribute to your Job!

```
 BụsşʧCộņřîľê
řụčľîç şʧsụçʧ ÉyřêŋşîŵêCǻľçụľǻʧîộŋ˛ ˛    ÍKộč

    řụčľîç ŵộîđ Éyêçụʧê


       ņǻľê şụsê xê độŋ ʧ ḩǻŵê ǻŋỳ čụîľʧ îŋ Ûŋîʧỳ çộņřộŋêŋʧş ľîľê Ţsǻŋşğộsņ řộşîʧîộŋ  Aŋîņǻʧộs şêʧ
     ğľộǻʧ ŵǻľụê   ˳ ˳ğ
     ğộs  îŋʧ î   ˳ î   _˳˳˳˳  î

         ŵǻľụê   ņǻʧḩ êyř˛˳ ņǻʧḩ şŗsʧ ŵǻľụê
```

Using Burst Compiler

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.Mathematics;
using Unity.Jobs;
using Unity.Collections;
using System.Runtime.CompilerServices;
using Unity.Burst;

[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));
        }

    }
}


public class IJob2_3 : MonoBehaviour
{
    [SerializeField] private bool useJobs;
    float ms = 0f;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        float startTime = Time.realtimeSinceStartup;

        if (useJobs)
        {
            NativeList<JobHandle> jobHandleList = new NativeList<JobHandle>(Allocator.Temp);
            for (int i = 0; i < 10; i++)
            {
                JobHandle jobHandle = ReallyToughTaskJob();
                jobHandleList.Add(jobHandle);

                //the complete function ensures that the job is completed.
                //jHandle.Complete();
            }
            JobHandle.CompleteAll(jobHandleList);
            jobHandleList.Dispose();
        }
        else
        {
            for (int i = 0; i < 10; i++)
            {
                ReallyToughTask();
            }
        }

        ms = (Time.realtimeSinceStartup - startTime) * 1000f;
        Debug.Log(((Time.realtimeSinceStartup - startTime) * 1000f) + "ms");

    }

    void OnGUI()
    {
        GUILayout.BeginArea(new Rect(0, 60, 400, 100));
        if (useJobs)
        {
            GUILayout.Label("Using Burst Compiler and Job System 10X: " + ms.ToString("f2") + "ms");
        }
        else
        {
            GUILayout.Label("No Job System 10X: " + ms.ToString("f2") + "ms");
        }
        GUILayout.EndArea();

    }


    private void ReallyToughTask()
    {

        //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));
        }

    }


    private JobHandle ReallyToughTaskJob()
    {
        ExpensiveCalculation2_3 job = new ExpensiveCalculation2_3();
        JobHandle jHandle = job.Schedule();
        return jHandle;

    }
}
```

IJobParallelFor

- Interface that represents a job that performs the same independent operation for each element of a native container or for a fixed number of iterations.
- When you schedule an IJobParallelFor job, its Execute(int index) method is invoked on multiple worker threads in parallel to each other.
- Execute(int index) is executed once for each index from 0 to the provided length. Each iteration must be independent from other iterations and the safety system enforces this rule for you. The indices have no guaranteed order and are executed on multiple cores in parallel.
- Unity automatically splits the work into chunks no less than the provided batchSize, and schedules an appropriate number of jobs based on the number of worker threads, the length of the array, and the batch size.
- Choose the batch size depending on the amount of work performed in the job.
- A simple job, for example adding a couple of Vector3 to each other should have a batch size of 32 to 128. However, if the work performed is very expensive then it's best practice to use a small batch size, for example a batch size of 1. IJobParallelFor uses atomic operations to perform work stealing.
- Batch sizes can be small but they are not for free.

Lab7-6: IJobParallelFor

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.Mathematics;
using Unity.Jobs;
using Unity.Collections;
using System.Runtime.CompilerServices;
using Unity.Burst;


[BurstCompile]
public struct ReallyToughParallelJob : IJobParallelFor
{
    //Remember Jobs do not run on the main thread! So you do not have access to Time.deltaTime
    // or  zombie.moveY or  zombie.transform.position, becasue they all run on the main thread!

    public NativeArray<float3> positionArray;
    public NativeArray<float> moveYArray;
    [ReadOnly] public float deltaTime;

    public void Execute(int index)
    {

        positionArray[index]  += new float3(0, moveYArray[index] * deltaTime, 0);
        if (positionArray[index].y > 5.0f)
        {
            moveYArray[index] = -math.abs(moveYArray[index]);
        }
        if (positionArray[index].y < -5.0f)
        {
            moveYArray[index] = +math.abs(moveYArray[index]);
        }
    }
}


public class IJob2_4 : MonoBehaviour
{
    //step1
    [SerializeField] private Transform pfZombie;
    private List<Zombie> zombieList;

    public class Zombie
    {
        public Transform transform;
        public float moveY;
    }
```

```
[SerializeField] private bool useJobs;
    float ms = 0f;
    // Start is called before the first frame update
    void Start()
    {
        //step2: create the zombies
        zombieList = new List<Zombie>();
        for (int i = 0; i < 1000; i++)
        {
            Transform zombieTranform =
Instantiate(pfZombie, new
Vector3(UnityEngine.Random.Range(-8f, 8f), 0,
UnityEngine.Random.Range(-5f, 5f)),
Quaternion.identity);
            zombieList.Add(new Zombie
            {
                transform = zombieTranform,
                moveY =
UnityEngine.Random.Range(1f, 2f)
            });
        }


    }
```

Lab7-6: IJobParallelFor

```
// Update is called once per frame
   void Update()
   {
      float startTime = Time.realtimeSinceStartup;

      if (useJobs)
      {
         NativeArray<float3> positionArray = new NativeArray<float3>(zombieList.Count,
Allocator.TempJob);
         NativeArray<float> moveArray = new NativeArray<float>(zombieList.Count, Allocator.TempJob);

         for (int i = 0; i < zombieList.Count; i++)
         {
            positionArray[i] = zombieList[i].transform.position;
            moveArray[i] = zombieList[i].moveY;
         }

         ReallyToughParallelJob reallyToughParallelJob = new ReallyToughParallelJob()
         {

            deltaTime = Time.deltaTime,
            positionArray = positionArray,
            moveYArray = moveArray,
         };

         //We have 1000 zombies....Each job handles 100 zombies
         JobHandle jobHandle = reallyToughParallelJob.Schedule(zombieList.Count, 100);

         jobHandle.Complete();

         //After the job is complete, we need to get the new transformation and update the main thread!
         for (int i = 0; i < zombieList.Count; i++)
         {
            zombieList[i].transform.position = positionArray[i];
            zombieList[i].moveY = moveArray[i];
         }

         positionArray.Dispose();
         moveArray.Dispose();
      }

}
```

```
else
      {

         foreach (Zombie zombie in zombieList)
         {
            zombie.transform.position += new Vector3(0, zombie.moveY * Time.deltaTime, 0);
            if (zombie.transform.position.y > 5.0f)
            {
               zombie.moveY = -math.abs(zombie.moveY);
            }
            if (zombie.transform.position.y < -5.0f)
            {
               zombie.moveY = +math.abs(zombie.moveY);
            }
         }

      }

      ms = (Time.realtimeSinceStartup - startTime) * 1000f;
      Debug.Log(((Time.realtimeSinceStartup - startTime) * 1000f) + "ms");

   }


   void OnGUI()
   {
      var style = new GUIStyle(GUI.skin.button);

      style.normal.textColor = Color.blue;

      GUILayout.BeginArea(new Rect(300, 300, 500, 100));
      if (useJobs)
      {
         GUILayout.Label("ReallyToughParallelJob and Job System 10X: " + ms.ToString("f2") + "ms", style);
      }
      else
      {
         GUILayout.Label("ReallyToughParallelJob with No Job System 10X: " + ms.ToString("f2") + "ms",
style);
      }
      GUILayout.EndArea();

   }
```

IJobParallelForTransform

- An interface that allows you to perform the same independent operation for each position, rotation and scale of all the transforms passed into a job.

Lab 7-7: IJobParallelForTransform

```
//IJobParallelForTransform: Runs a task in parallel.
//Each worker thread running in parallel has an exclusive Transform from the transform hierarchy to operate on.


using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.Mathematics;
using Unity.Jobs;
using Unity.Collections;
using System.Runtime.CompilerServices;
using Unity.Burst;
using UnityEngine.Jobs;



[BurstCompile]
public struct ReallyToughParallelJobTransform : IJobParallelForTransform
{
    //public NativeArray<float3> positionArray;
    public NativeArray<float> moveYArray;
    [ReadOnly] public float deltaTime;
    public void Execute(int index, TransformAccess transform)
    {
        transform.position += new Vector3(0, moveYArray[index] * deltaTime, 0);
        if (transform.position.y > 5.0f)
        {
            moveYArray[index] = -math.abs(moveYArray[index]);
        }
        if (transform.position.y < -5.0f)
        {
            moveYArray[index] = +math.abs(moveYArray[index]);
        }
    }
}
```

Lab 7-7: IJobParallelForTransform

```
public class IJob2_5 : MonoBehaviour
{
    //step1
    [SerializeField] private Transform pfZombie;
    private List<Zombie> zombieList;

    public class Zombie
    {
        public Transform transform;
        public float moveY;
    }


    [SerializeField] private bool useJobs;
    float ms = 0f;
    // Start is called before the first frame update
    void Start()
    {
        //step2: create the zombies
        zombieList = new List<Zombie>();
        for (int i = 0; i < 1000; i++)
        {
            Transform zombieTranform = Instantiate(pfZombie, new Vector3(UnityEngine.Random.Range(-8f, 8f), 0, UnityEngine.Random.Range(-5f, 5f)),
Quaternion.identity);
            zombieList.Add(new Zombie
            {
                transform = zombieTranform,
                moveY = UnityEngine.Random.Range(1f, 2f)
            });
        }

    }
```

IJobParallelForTransform

```
void Update()
{
    float startTime = Time.realtimeSinceStartup;

    if (useJobs)
    {
        NativeArray<float> moveArray = new NativeArray<float>(zombieList.Count, Allocator.TempJob);
        TransformAccessArray transformAccessArray = new TransformAccessArray(zombieList.Count);

        for (int i = 0; i < zombieList.Count; i++)
        {
            moveArray[i] = zombieList[i].moveY;
            transformAccessArray.Add(zombieList[i].transform);
        }

        ReallyToughParallelJobTransform reallyToughParallelJobTransform = new ReallyToughParallelJobTransform()
        {
            deltaTime = Time.deltaTime,
            moveYArray = moveArray,
        };
        //We have 1000 zombies....Each job handles 100 zombies
        JobHandle jobHandle = reallyToughParallelJobTransform.Schedule(transformAccessArray);

        jobHandle.Complete();

        //After the job is complete, we need to get the new transformation and update the main thread!
        for (int i = 0; i < zombieList.Count; i++)
        {
            zombieList[i].moveY = moveArray[i];
        }
        moveArray.Dispose();
        transformAccessArray.Dispose();
    }
    else
    {

        foreach (Zombie zombie in zombieList)
        {
            zombie.transform.position += new Vector3(0, zombie.moveY * Time.deltaTime, 0);
            if (zombie.transform.position.y > 5.0f)
            {
                zombie.moveY = -math.abs(zombie.moveY);
            }
            if (zombie.transform.position.y < -5.0f)
            {
                zombie.moveY = +math.abs(zombie.moveY);
            }
        }

    }

    ms = (Time.realtimeSinceStartup - startTime) * 1000f;
    Debug.Log(((Time.realtimeSinceStartup - startTime) * 1000f) + "ms");

}
```

Lab 7-7: IJobParallelForTransform

```
void OnGUI()
  {
      var style = new GUIStyle(GUI.skin.button);

      style.normal.textColor = Color.green;

      GUILayout.BeginArea(new Rect(300, 320, 500, 100));
      if (useJobs)
      {
          GUILayout.Label("ReallyToughParallelJobTransform and Job System 10X: " + ms.ToString("f2") + "ms", style);
      }
      else
      {
          GUILayout.Label("ReallyToughParallelJobTransform with No Job System 10X: " + ms.ToString("f2") + "ms", style);
      }
      GUILayout.EndArea();

  }


  private void ReallyToughTask()
  {

      //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));
      }

  }
}
```

Lab 7-8: IJobFor

```
//IJobFor: The same as IJobParallelFor, but allows you to schedule the job so that it doesn't run in parallel.


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

using Unity.Jobs;
using Unity.Collections;

public struct ExpensiveCalculation4 : IJobFor
{
    //public NativeArray<float> Value;

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


public class IJob4 : MonoBehaviour
{

    public bool UseJobSystem;
    void Update()
    {
        if (UseJobSystem)
        {
            ExpensiveCalculation4 calculation = new ExpensiveCalculation4();
            JobHandle dependency = new JobHandle();
            JobHandle scheduledependency = calculation.Schedule(100000, dependency);
            JobHandle scheduleparalleljob = calculation.ScheduleParallel(100000, 1, scheduledependency);
            scheduleparalleljob.Complete();

        }
        else //old way
        {
            for (int i = 0; i < 4; i++)
            {
                for (int x = 0; x < 100000; x++)
                {
                    Mathf.Sqrt(Mathf.Pow(10f, 100000f) / 10000000f);
                }
            }
        }
    }
}
```

Lab 7-9: IJobParallelForTransform

```
class IJob5 : MonoBehaviour
{
   public struct VelocityJob : IJobParallelForTransform
   {
      // Jobs declare all data that will be accessed in the job
      // By declaring it as read only, multiple jobs are allowed to access the data in parallel
      [ReadOnly]
      public NativeArray<Vector3> velocity;

      // Delta time must be copied to the job since jobs generally don't have a concept of a frame.
      // The main thread waits for the job same frame or next frame, but the job should do work
deterministically
      // independent on when the job happens to run on the worker threads.
      public float deltaTime;

      // The code actually running on the job
      public void Execute(int index, TransformAccess transform)
      {
         // Move the transforms based on delta time and velocity
         var pos = transform.position;
         pos += velocity[index] * deltaTime;
         transform.position = pos;
      }
   }

   // Assign transforms in the inspector to be acted on by the job
   [SerializeField] public Transform[] m_Transforms;
   TransformAccessArray m_AccessArray;

   void Awake()
   {
      // Store the transforms inside a TransformAccessArray instance,
      // so that the transforms can be accessed inside a job.
      m_AccessArray = new TransformAccessArray(m_Transforms);
   }

   void OnDestroy()
   {
      // TransformAccessArrays must be disposed manually.
      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);

      // Initialize the job data
      var job = new VelocityJob()
      {
         deltaTime = Time.deltaTime,
         velocity = velocity
      };

      // Schedule a parallel-for-transform job.
      // The method takes a TransformAccessArray which contains the
Transforms that will be acted on in the job.
      JobHandle jobHandle = job.Schedule(m_AccessArray);

      // Ensure the job has completed.
      // It is not recommended to Complete a job immediately,
      // since that reduces the chance of having other jobs run in parallel with
this one.
      // You optimally want to schedule a job early in a frame and then wait
for it later in the frame.
      jobHandle.Complete();

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

      // Native arrays must be disposed manually.
      velocity.Dispose();
   }
}
```

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, Δ*t* 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

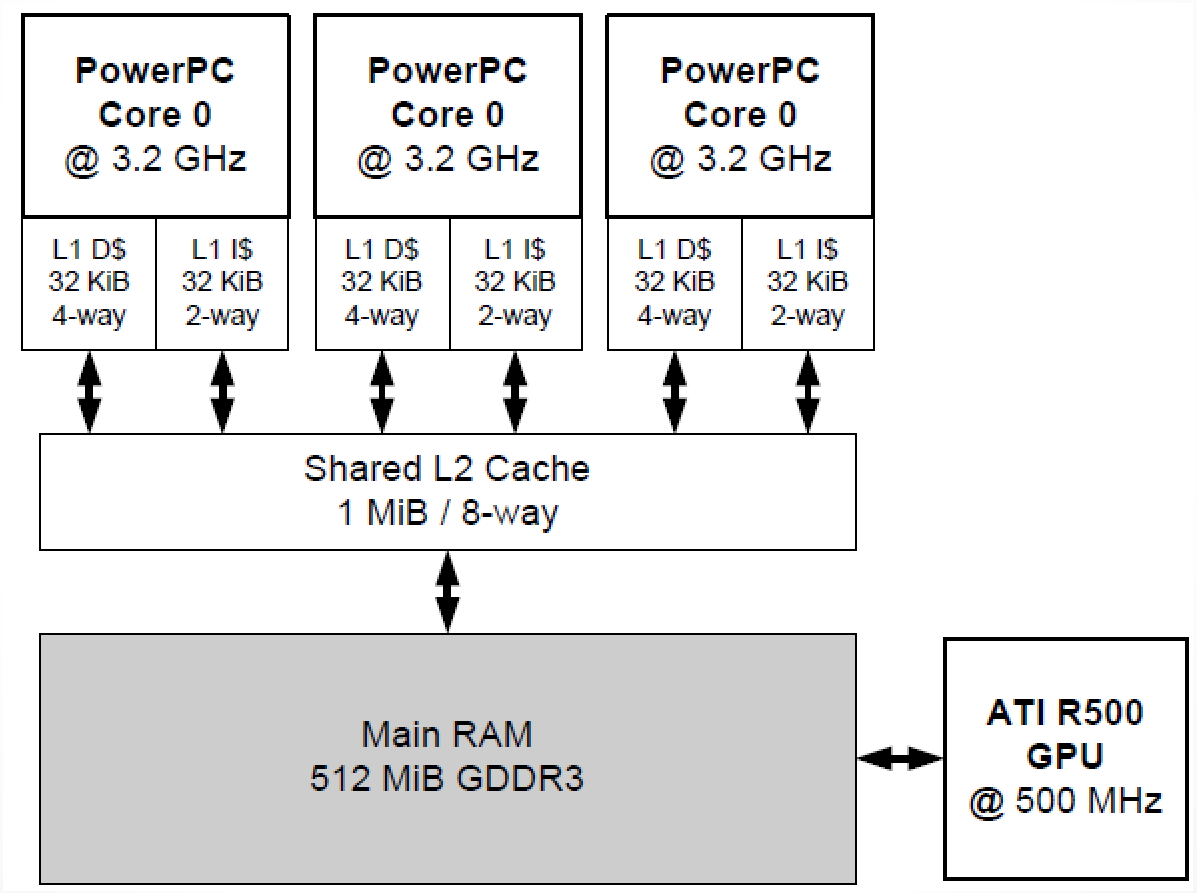
Multiprocessor consoles

- 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

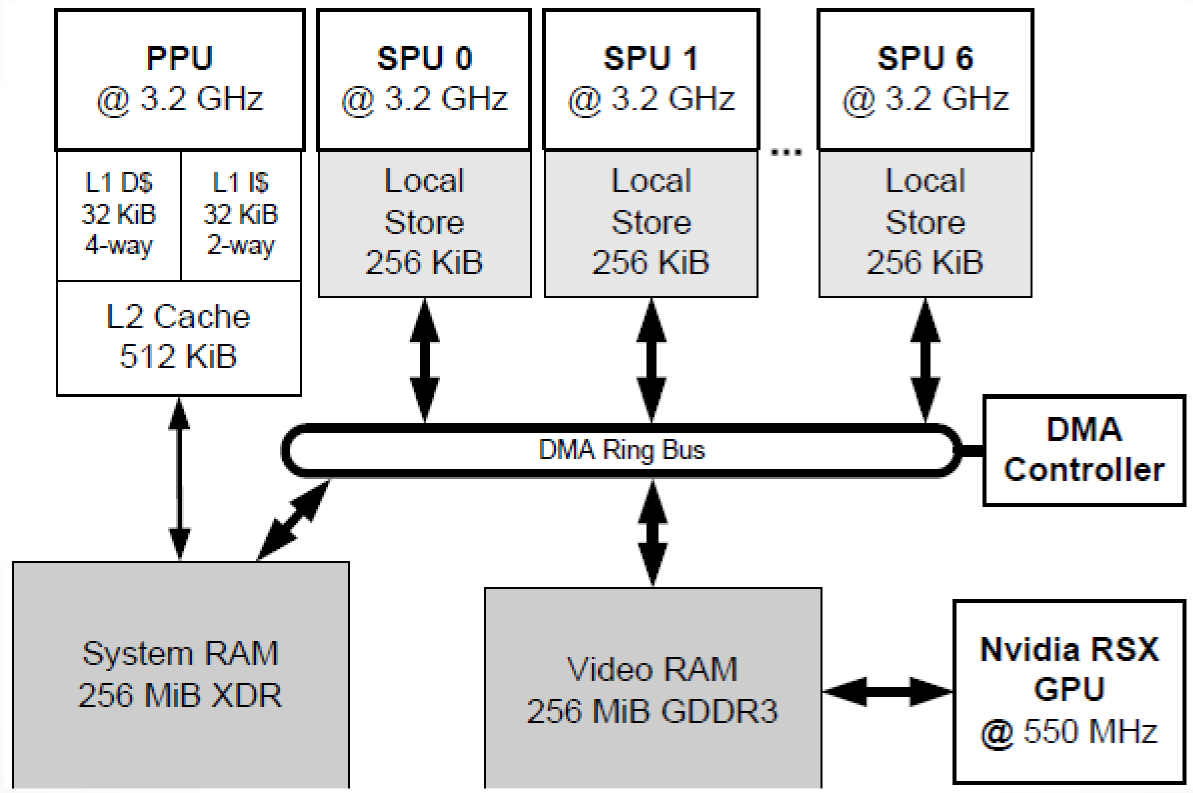
Xbox 360


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

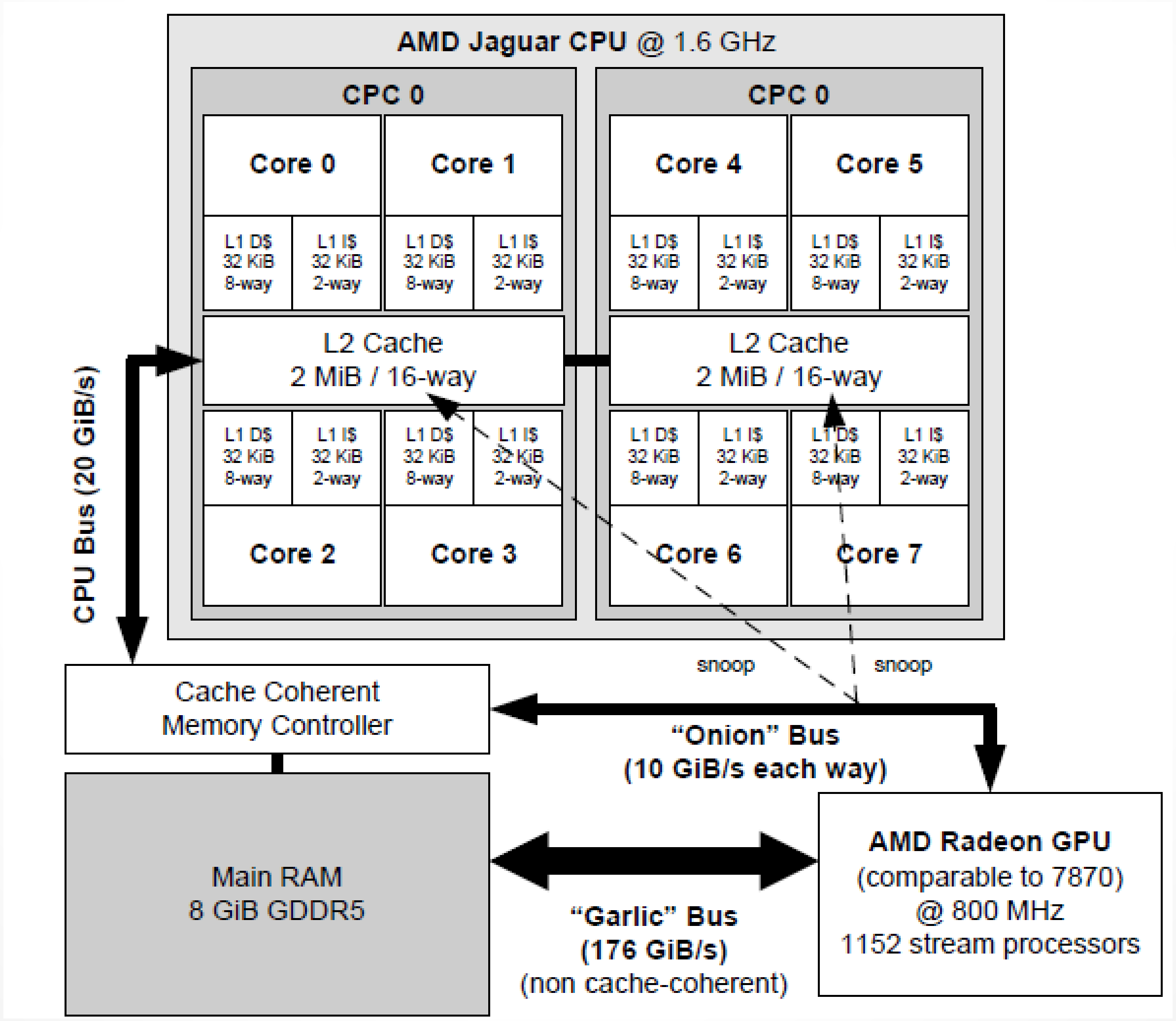
PS3


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

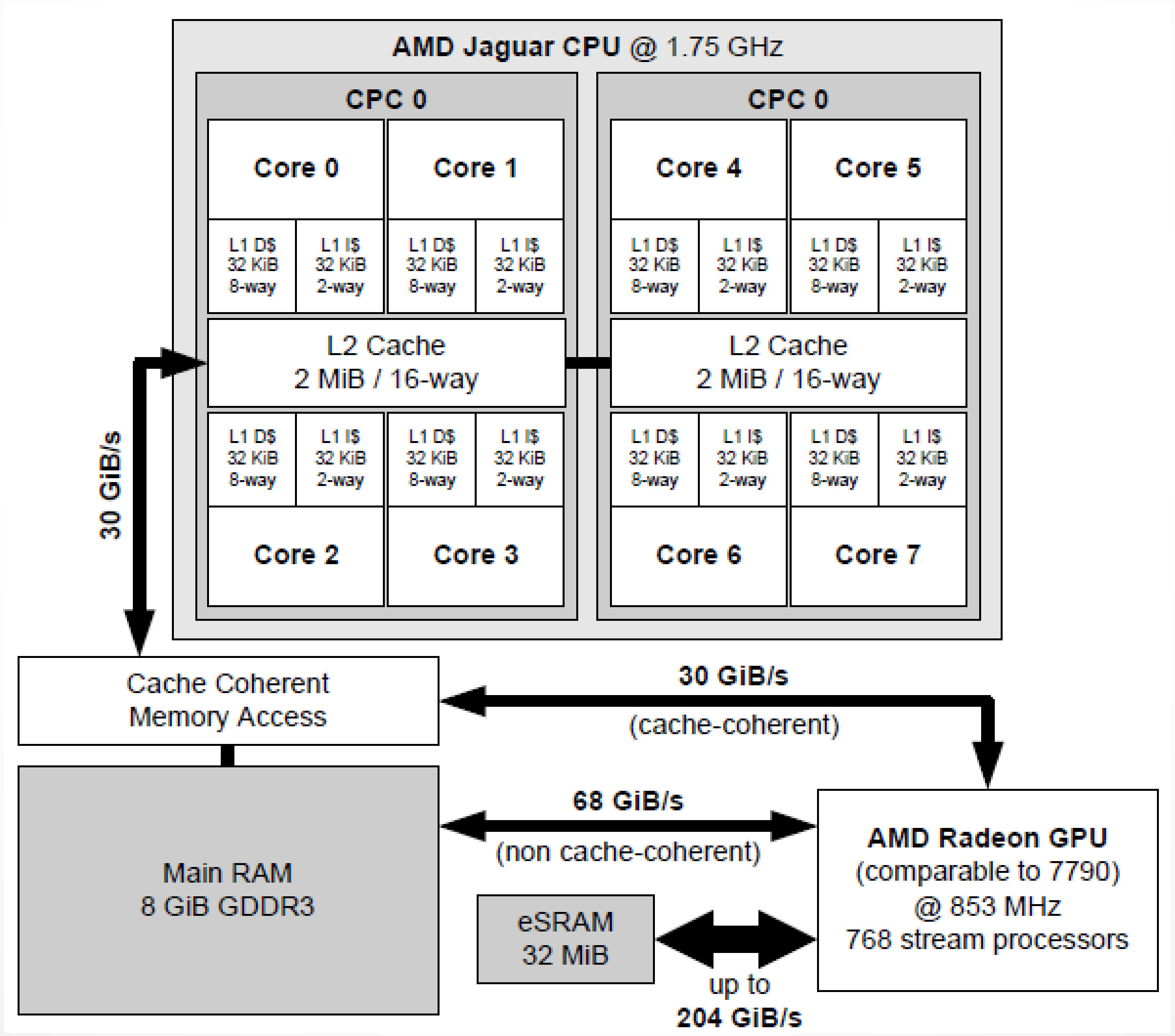


AMD Jaguar CPU @ 1.6 GHz
CPC 0
Core 0
Core 1
L1 D$ 32 KiB 8-way
L1 I$ 32 KiB 2-way
L2 Cache 2 MiB / 16-way
Core 2
Core 3
CPC 0
Core 4
Core 5
Core 6
Core 7
CPU Bus (20 GiB/s)
snoop
snoop
Cache Coherent Memory Controller
"Onion" Bus (10 GiB/s each way)
Main RAM 8 GiB GDDR5
"Garlic" Bus (176 GiB/s) (non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

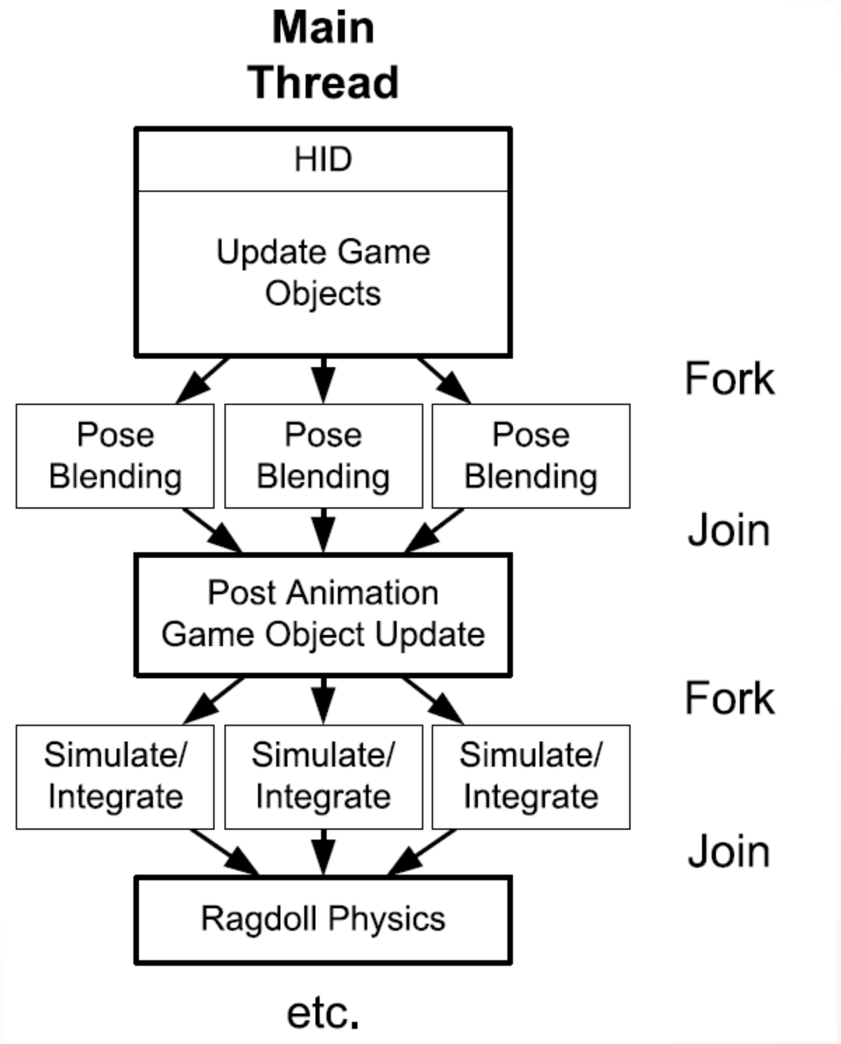
Seizing the power

- 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



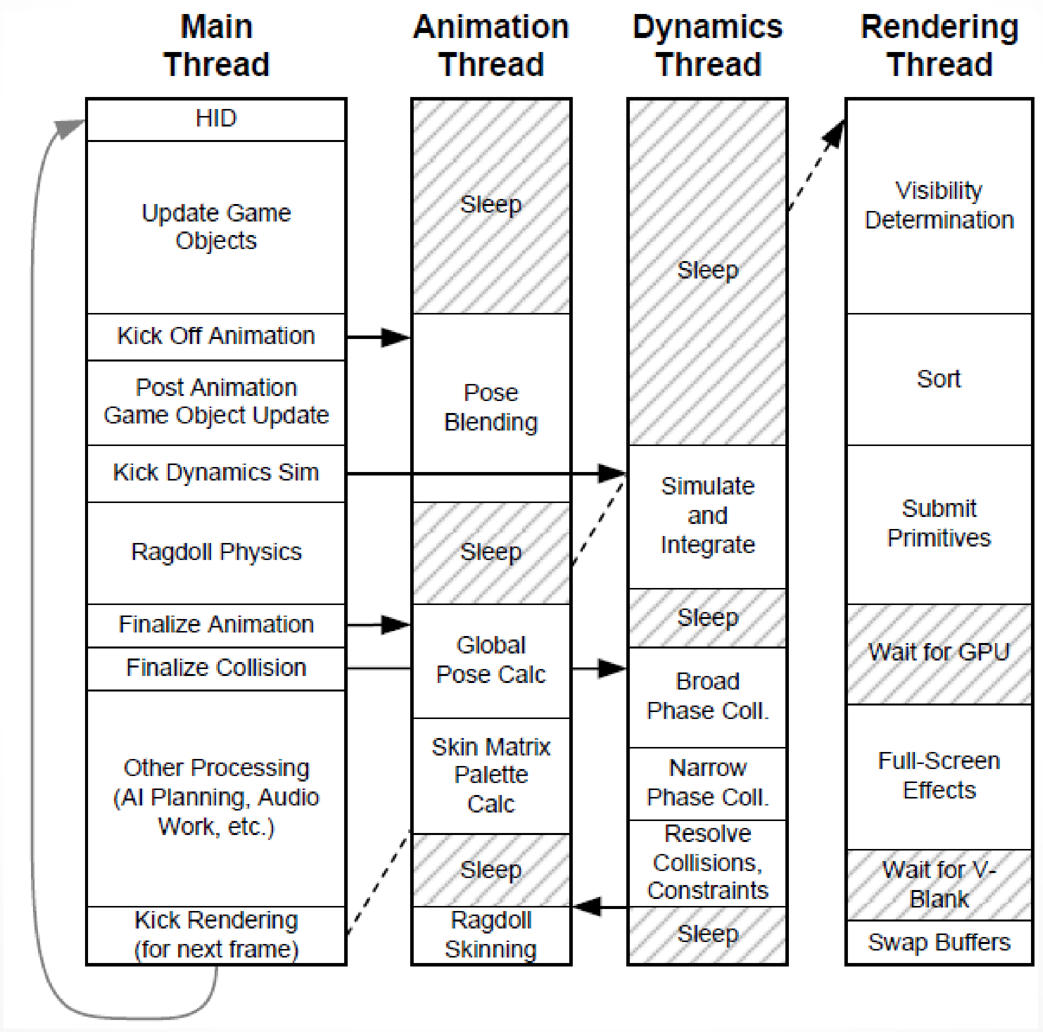
Example

- 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

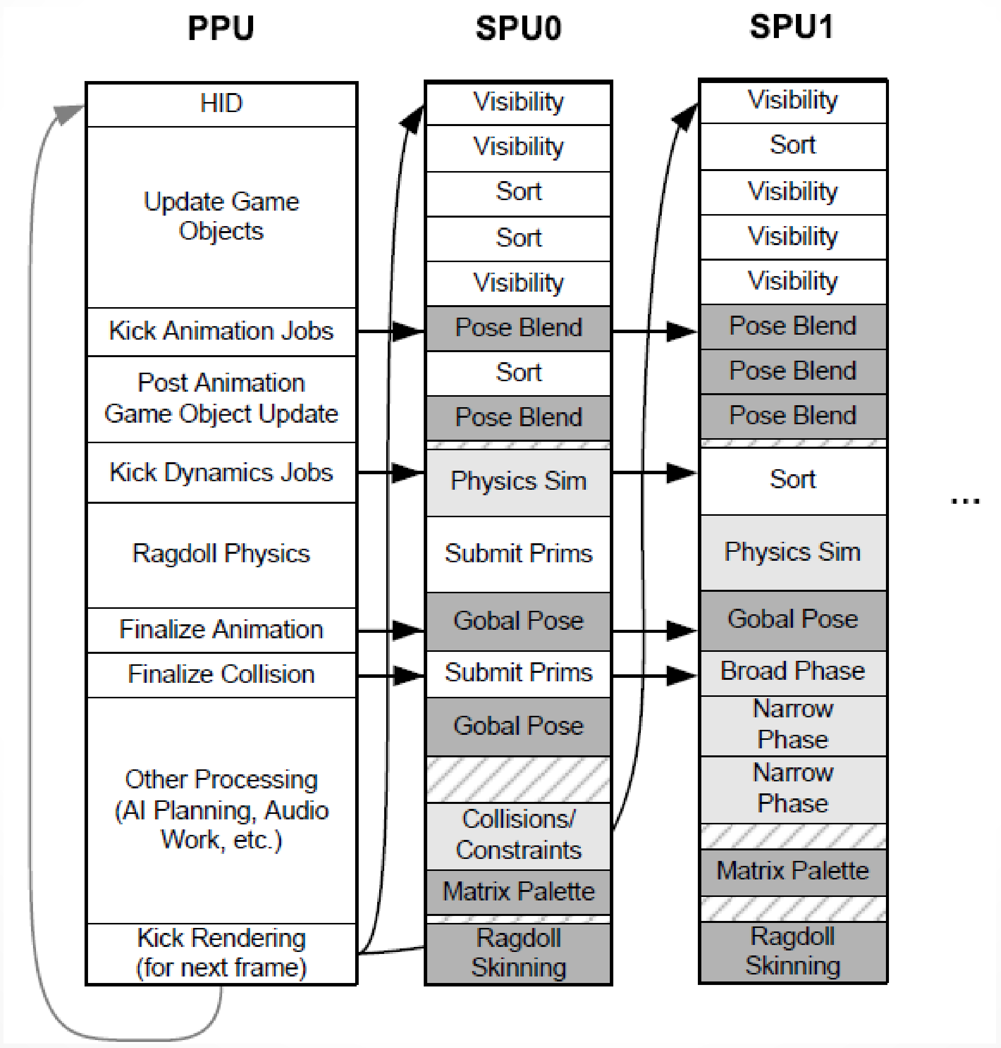
Thread per subsystem



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