Enhancing VR Immersion with the CPU in Star Trek™: Bridge Crew

Introduction
Traditionally in games, the CPU is not usually considered a large contributor to immersive and visually striking scenes. In the past, some few games exposed settings to users, allowing them to adjust CPU usage, but many developers believe it’s more trouble than it’s worth to implement multiple tiers of CPU based systems for different hardware levels. With this article, and the awesome work showcased in Star Trek™: Bridge Crew* in a partnership with Ubisoft’s Red Storm Entertainment*, we’re aiming to fix that misconception. Virtual Reality (VR) is a segment where the combination of environmental interaction, enhanced physics simulation, and destruction can become the frosting on the cake that keeps a player in your game and wanting more. Given the low-end hardware specifications required for Oculus*, it becomes more important than ever to eliminate the idea of keeping CPU work tailored to the minimum specification globally. Leveraging available system resources to enhance dynamism and immersion will help you create your ideal game while allowing as many players access as possible, and we’ve made it easier than ever.

This article will take you through each of the CPU-intensive features implemented in Star Trek™: Bridge Crew* with instructions on utilizing the systems they’ve been built upon. This is followed by with a brief section dedicated to determining how much CPU work is too much for each performance tier. The final section shows how to easily set up CPU performance categories to auto-detect where your end user’s hardware level sits.

Star Trek™: Bridge Crew* was built using Unity*, which will be the focus of this article, but all the concepts apply to other engines as well.

Check out the following video to see side-by-side comparisons of the game running these effects.

CPU Intensive Features in Star Trek™: Bridge Crew*
<Embedded video combining all side by side comparisons - https://www.youtube.com/watch?v=I-9B9jifj-k>

Bridge Damage – Combination of Physics Particles, Rigidbody Physics, and Realtime Global Illumination (GI)

Overview
The bridge of the USS Aegis is one of the main focal points of the game. Virtually all gameplay requires the player to be on the bridge, thus making it obvious that the bulk of the CPU work should be applied within it to give the player the most bang for their buck. The most time was spent focused on improving the bridge’s various destruction sequences adding elements of intensity to the scene. For example, when the bridge is damaged, big set pieces fly off; sparks ricochet off walls and floors; and fires spring up and throw a glow on surrounding set pieces not in direct view from the light source.
What Makes it CPU Intensive?
Applying damage to the bridge makes use of Unity’s* realtime **Rigidbody** physics, **Particle Systems** with large numbers of small particles with collision support enabled, as well as **realtime global illumination (GI)** updates created by the various fire and spark effects, which all scale across available CPU cores. Various debris objects are spawned that use **Rigidbody** physics during damage events, and the particle counts were pushed significantly higher when the high-end effects are active. World collision support for the particles was added along with collision primitives with the detail level needed to get the bouncing and scattering behavior desired for the sparks. Some of Unity’s* other particle features that use the CPU were added to enhance the scene, such as sub-emitters to add trails to fireballs and some sparks. The bridge damage particles were kept small in screen-coverage size to keep the GPU impact as low as possible while still achieving the desired look. When damage events occur, some of the lights and emissive surfaces flicker to simulate power interruption. The **GI** is updated while the lights are flickering and when there are fires active on the bridge. Next, we’ll go into each system and show how they can be leveraged separately.

Sparks

<Embed side by side comparison video – file – ParticlePhysicsComparison.mp4>

Overview
Unity’s* built-in **Particle System** component allows for a lot of variation in both aesthetics and behavior. It also just so happens that the built-in **Particle System** scales across available CPU cores very well under the hood. With the click of a button you can have your particle system collide and react to your environment, or if you want a more customized behavior, you can script the movement of each particle (more on this later). When using the built-in collision behavior shown below, the underlying engine will split the work up amongst available cores, allowing the system to go as wide as possible. Because of this, you can scale your particle counts based on the number of cores available, while also considering processor frequency and cache size. To activate collisions on your particles, simply go to the **Particle System** component of interest, check the **Collision** checkbox, and then select the desired settings associated with it.
### Particle System

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>5.00</td>
</tr>
<tr>
<td>Leaping</td>
<td>✓</td>
</tr>
<tr>
<td>Protrude</td>
<td></td>
</tr>
<tr>
<td>Start Delay</td>
<td>0</td>
</tr>
<tr>
<td>Start Lifetime</td>
<td>5</td>
</tr>
<tr>
<td>Start Speed</td>
<td>5</td>
</tr>
<tr>
<td>3D Start Size</td>
<td></td>
</tr>
<tr>
<td>Start Size</td>
<td>1</td>
</tr>
<tr>
<td>3D Start Rotation</td>
<td></td>
</tr>
<tr>
<td>Start Rotation</td>
<td>0</td>
</tr>
<tr>
<td>Randomize Rotation</td>
<td></td>
</tr>
<tr>
<td>Start Color</td>
<td></td>
</tr>
<tr>
<td>Gravity Modifier</td>
<td>0</td>
</tr>
<tr>
<td>Simulation Space</td>
<td>Local</td>
</tr>
<tr>
<td>Simulation Speed</td>
<td>1</td>
</tr>
<tr>
<td>Scaling Mode</td>
<td>Local</td>
</tr>
<tr>
<td>Play On Awake</td>
<td>✓</td>
</tr>
<tr>
<td>Max Particles</td>
<td>1000</td>
</tr>
<tr>
<td>Auto Random Seed</td>
<td>✓</td>
</tr>
<tr>
<td>Emission</td>
<td>✓</td>
</tr>
<tr>
<td>Shape</td>
<td>✓</td>
</tr>
<tr>
<td>Velocity over Lifetime</td>
<td></td>
</tr>
<tr>
<td>Limit Velocity over Lifetime</td>
<td></td>
</tr>
<tr>
<td>Inherit Velocity</td>
<td></td>
</tr>
<tr>
<td>Force over Lifetime</td>
<td></td>
</tr>
<tr>
<td>Color over Lifetime</td>
<td></td>
</tr>
<tr>
<td>Color by Speed</td>
<td></td>
</tr>
<tr>
<td>Size over Lifetime</td>
<td></td>
</tr>
<tr>
<td>Size by Speed</td>
<td></td>
</tr>
<tr>
<td>Rotation over Lifetime</td>
<td></td>
</tr>
<tr>
<td>Rotation by Speed</td>
<td></td>
</tr>
<tr>
<td>External Forces</td>
<td></td>
</tr>
<tr>
<td>Noise</td>
<td></td>
</tr>
<tr>
<td>Collision</td>
<td>✓</td>
</tr>
<tr>
<td>Collision Mode</td>
<td>3D</td>
</tr>
<tr>
<td>Collision Node</td>
<td></td>
</tr>
<tr>
<td>Dampen</td>
<td>0</td>
</tr>
<tr>
<td>Bounce</td>
<td></td>
</tr>
<tr>
<td>Lifetime Less</td>
<td>0</td>
</tr>
<tr>
<td>Min Kill Speed</td>
<td></td>
</tr>
<tr>
<td>Max Kill Speed</td>
<td>10000</td>
</tr>
<tr>
<td>Reduce Scale</td>
<td>1</td>
</tr>
<tr>
<td>Collides With</td>
<td>Everything</td>
</tr>
<tr>
<td>Interior Collisions</td>
<td>✓</td>
</tr>
<tr>
<td>Max Collision Shapes</td>
<td>256</td>
</tr>
<tr>
<td>Collision Quality</td>
<td>High</td>
</tr>
<tr>
<td>Enable Dynamic Collide</td>
<td></td>
</tr>
<tr>
<td>Send Collision Message</td>
<td></td>
</tr>
<tr>
<td>Triggers</td>
<td></td>
</tr>
<tr>
<td>Sub Emitters</td>
<td></td>
</tr>
<tr>
<td>Texture Sheet Animation</td>
<td></td>
</tr>
<tr>
<td>Lights</td>
<td></td>
</tr>
<tr>
<td>Trails</td>
<td></td>
</tr>
<tr>
<td>Renderer</td>
<td></td>
</tr>
<tr>
<td>Resimulate</td>
<td>✓</td>
</tr>
<tr>
<td>Selection</td>
<td></td>
</tr>
<tr>
<td>Bounds</td>
<td></td>
</tr>
</tbody>
</table>
There are quite a few options involved in the **Collision** settings group. The main setting to consider should be between colliding against the **World** or if you’d like to define a set of planes that the particles will collide with. The former setting will produce the most realistic simulation as virtually every existing collider in your scene will be considered in each particle update calculation, but this of course comes with additional CPU cost. Usually, games will define a set of key planes that will act as an approximation of the surrounding geometry to keep compute as low as possible to make room for other CPU intensive effects. The setting you choose depends entirely on the layout of your game and what you’d like to achieve visually. For example, the following defines three planes as colliders: a floor and two walls.

< CODE – Embed particle system controller sample here >

```csharp
ParticleSystem[] ParticleSystems;
public Transform[] CollisionPlanes;

public void Awake()
{
    ParticleSystems = gameObject.GetComponentsInChildren<ParticleSystem>();
    Debug.Log("Initializing ParticleSystemController");
}

public void SetCPULevel(CPUCapabilityManager.SYSTEM_LEVELS sysLevel)
{
    for (int i = 0; i < ParticleSystems.Length; i++)
    {
        var particleSysMain = ParticleSystems[i].main;
        var particleSysCollision = ParticleSystems[i].collision;
        var particleSysEmission = ParticleSystems[i].emission;
        if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.HIGH)
        {
            particleSysEmission.rateOverTime = 400.0f;
            particleSysMain.maxParticles = 20000;
            particleSysCollision.enabled = true;
        }
    }
```
particleSysCollision.type = ParticleSystemCollisionType.World;
else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.MEDIUM)
{
  particleSysEmission.rateOverTime = 300.0f;
  particleSysMain.maxParticles = 10000;
  particleSysCollision.enabled = true;
  particleSysCollision.type = ParticleSystemCollisionType.World;
}
else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.LOW)
{
  particleSysEmission.rateOverTime = 200.0f;
  particleSysMain.maxParticles = 5000;
  particleSysCollision.enabled = true;
  particleSysCollision.type = ParticleSystemCollisionType.Planes;
  for (int j = 0; j < CollisionPlanes.Length; j++)
  {
    particleSysCollision.SetPlane(j, CollisionPlanes[j]);
  }
}
else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.OFF)
{
  particleSysEmission.rateOverTime = 100.0f;
  particleSysMain.maxParticles = 3000;
  particleSysCollision.enabled = false;
}

</ CODE – Embed particle system controller sample here >

See more optimized version in CPUCapabilityTester sample

**Realtime Global Illumination (GI)**

<Embed side by side comparison video – GIComparison.mp4>

**Overview**

**Realtime GI** is the simulation of light rays bouncing within a scene and indirectly illuminating objects. This feature was something the team really wanted to leverage because the big window at the front of the Aegis would allow for astral bodies and damage effects to update the interior of the bridge. Moving the Aegis in front of a massive sun or nebula changes the appearance of the bridge to reflect the incoming light, increasing immersion by giving the scene a cohesive look and making the vistas feel much more real.

**What Makes it CPU Intensive?**

Unity’s* **realtime GI** is computed heavily on the CPU and leverages a percentage of the available cores depending on the fidelity desired.
Is it Built into Unity*?

Yes. When the **realtime Gi** effects are enabled, the application uses the highest CPU usage setting Unity* allows with an immediate update rate to get the best results.

How it’s Done

To enable this effect, check the **Realtime Lighting** checkbox in the Lighting window (Window > Lighting). (Note: Editor performance settings for Realtime GI were hidden in recent versions of Unity* and handled under the hood. Scripted update settings are still available – see sample for details) On older versions of Unity*, check the **Precomputed Realtime GI** checkbox (still within Window > Lighting).

There are two settings which both heavily affect CPU usage. **Realtime Resolution** and **CPU Usage**.

- **Realtime Resolution** determines how many texels per unit should be computed. Unity* published a tutorial that goes into detail on how to properly set this value. A useful rule of thumb is that visually rich indoor scenes require more texels per unit to achieve as much realism as possible. In large outdoor scenes, indirect lightning transitions are not as noticeable, allowing the compute power to be spent elsewhere.

- **CPU Usage** determines how many of the engine’s available worker threads will be leveraged for the realtime GI computation. It is best practice to determine the amount of CPU power available on various system levels and set this accordingly. For lower-end systems it’s best to keep this low/medium; for higher-end systems it’s better to use high or unlimited. Descriptions of these settings can be found in the Unity* documentation shipped with versions that expose them.

```
void Start () {
  SetCPULevel(CPUCapabilityManager.Singleton.CPUCapabilityLevel);
}

public void SetCPULevel(CPUCapabilityManager.SYSTEM_LEVELS sysLevel)
{
  if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.HIGH)
  {
    DynamicGI.updateThreshold = 0;
  }
```
```csharp
else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.MEDIUM)
{
    DynamicGI.updateThreshold = 25;
}
else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.LOW)
{
    DynamicGI.updateThreshold = 50;
}
else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.OFF)
{
    DynamicGI.updateThreshold = 100;
}
Debug.Log("\n(" + gameObject.name + ") System capability set to: " + CPUCapabilityManager.Singleton.CPUCapabilityLevel + ", so setting GI update threshold to: " + DynamicGI.updateThreshold);
}
</ CODE – Embed GI sample here G>

Dynamic Asteroids

<Embed side by side comparison video – AsteroidVid.mp4>

AsteroidVid.mp4

Overview
When the Aegis navigates asteroid fields, additional asteroids are generated outside the view frustum of the player and launched into view. These asteroids collide with existing in-place asteroids and kick off dust.

Many of the games maps also contain asteroid field generators; these field generators scatter large static asteroids within a cylindrical or spherical zone. When high-end CPU effects are enabled, these zones also place dynamic asteroids with Rigidbody physics at a certain distance away from the ship while it’s moving. This helps give the impression that the asteroid field is full of smaller fragments colliding with each other and the larger asteroids. There is, additionally, a small chance a dynamic asteroid will spawn with a velocity already applied to keep things moving and the scene active. Finally, some asteroids will break apart into smaller fragments when colliding with the player’s ship or other asteroids, while others will bounce off but remain intact.

These changes have the effect of pulling the player’s attention away from the skybox, creating a sense that the player truly is in space; all without disrupting gameplay.

What Makes it CPU Intensive?

Having large numbers of dynamic asteroid fragments flying around in asteroid fields using Rigidbody physics, instantiating un-pooled fragments while moving and generating additional fragments when asteroids break apart all use a lot of CPU time.
Is it Built into Unity*?
The dynamic asteroids use Unity’s* **Rigidbody Physics** and **Particles Systems**, but the system to generate the asteroids was written and customized by the Star Trek™: Bridge Crew* team. Check out the sample below to see how you can implement a similar system yourself.

How it’s Done
If the player’s machine is capable, previously static models in the scene that don’t need to remain static can have **Rigidbody** physics enabled. This can be done dynamically in script by adding new **Rigidbody** components to existing objects, or by generating prefabs of preconfigured objects on-the-fly. Dynamic objects and objects that can be interacted with do a lot to increase immersion in games, especially in VR.

```
public GameObject[] PotentiallyDynamicObjects;
    int NumDynamicObjects = 0;

void Start (){
    SetCPULevel(CPUCapabilityManager.Singleton.CPUCapabilityLevel);
}

public void SetCPULevel(CPUCapabilityManager.SYSTEM_LEVELS sysLevel)
{
    if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.HIGH)
    {
        NumDynamicObjects = PotentiallyDynamicObjects.Length;
    }
    else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.MEDIUM)
    {
        NumDynamicObjects = PotentiallyDynamicObjects.Length / 2;
    }
    else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.LOW)
```

< CODE – Embed staticdynamiccontroller sample here>
```csharp
{ 
    NumDynamicObjects = PotentiallyDynamicObjects.Length / 3;
}
else if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.OFF)
{
    NumDynamicObjects = 0;
}

Debug.Log("(Obj: " + gameObject.name + ") System capability set to: " +
CPUCapabilityManager.Singleton.CPUCapabilityLevel + ", so setting number of dynamic
objects to: " + NumDynamicObjects);

for (int i = 0; i < NumDynamicObjects; i++)
{
    Rigidbody objRigidBody =
PotentiallyDynamicObjects[i].AddComponent<Rigidbody>();
    objRigidBody.useGravity = true;
    objRigidBody.mass = 10;
    PotentiallyDynamicObjects[i].AddComponent<CustomAsteroidMovement>();
}

</CODE – Embed staticdynamiccontroller sample here>

Cloud Wakes and Solar Flares
<Embed side by side comparison video – WakeEffectComparison.mp4 & SolarFlareComparison.mp4>

Overview
Cloud wakes increase immersion by creating the illusion that enemy ships and the Aegis are displacing
dust as they move through space. Solar flares accomplish the same thing by distracting the eye from the
skybox, making the player feel like they are in the far reaches of space.

What Makes it CPU Intensive?
The cloud wakes and solar flares use scripted particle behaviors which require updating the particles
individually using a script on the main thread. Looping through several hundred to a few thousand
particles and updating their properties through script uses a lot of CPU time, but allows custom behavior
for particles that wouldn’t be possible using the normal particle system properties offered out of the box
in Unity*. Keep in mind that this must currently be done on the main thread, so this system can't go as
wide on the cores as the previously mentioned particle collision system. Stay tuned for Unity’s* new C#
job system mentioned at Unite Europe 2017 which will extend the Unity* API to allow better multi-
threading in script code.
Is it Built into Unity?

Cloud wakes and solar flares use Unity’s *Particle System*, but how the particles move and change over time was scripted by Red Storm Entertainment. The wake effect emits particle trails from several emitter points on the ship using a single particle system. The size and lifetime of the particles in a single trail are based on its emitter. The trail particles are emitted in world space, but the emitter points stay attached to the ship so that they continue to emit from the correct locations as the ship turns and banks. The custom particle behavior script adds virtual “attractor” objects behind the ship that oscillate randomly to pull nearby particles towards them, introducing turbulence to the trails behind the ships while passing through clouds. The solar flares also use the attractor behavior to either splash the particles outward or pull them back towards the sun’s surface after initially having been emitted outward. The following simple example shows how to make all particles head towards the world origin.

```csharp
public class ParticleBehavior : MonoBehaviour {
    publicParticleSystem myParticleSystem;
    ParticleSystem.Particle[] myParticles = new ParticleSystem.Particle[4000];
    public float mySpeed = 10.0f;

    void Update () {
        int numParticles = myParticleSystem.GetParticles(myParticles);
        for (int i = 0; i < numParticles; i++)
        {
            myParticles[i].position = myParticles[i].position + ParticleSpeed
                * Time.deltaTime;
        }
        myParticleSystem.SetParticles(myParticles, numParticles);
    }
}
```

Ship Destruction

<Embed side by side comparison video - ShipDestructionComparisonnewVids>
Overview
The ship destruction feature enhances the game by giving players a more satisfying feeling when defeating an enemy. Traditionally in games, a trick is used to occlude exploding objects with an explosion effect to mask the popping effect when removing a discarded GameObject from the scene. With the available CPU power in higher-end setups, we can split the model into pieces and launch them all in different directions, and even add sub-destruction. Each piece can collide with scene dressings and then ultimately disappear or linger if the system can handle it.

What Makes it CPU Intensive?
The ships are broken into many different parts by the artists that all contain Rigidbody components, and animated via physics forces when they're initialized. Collision with other objects (i.e., asteroids, ships) was enabled to ensure realistic behavior when animated in the environment. Furthermore, each exploded ship part had particle trails attached to them.

Is it Built into Unity*?
The Rigidbody and physics aspect of this feature are entirely built in, and Unity*-specific methods are used to add explosion forces to the ship parts. Afterwards, they are animated and collide with objects using Unity's* Rigidbody Physics system. A Unity* Particle System is used to emit particles that have sub-emitters to create trails behind the pieces, but the top-level particle positions are managed in script to ensure they remained attached to the exploded ship parts without worrying about parent coordinate spaces.

How it’s Done
Build out your models in pieces separated by various break points. Outfit each game object containing a mesh renderer in Unity* with a Rigidbody component. When the object should be destroyed, enable the Rigidbody components on each mesh and apply an explosive force to all of them. See Unity’s* Rigidbody documentation for more details.

< CODE – Embed explosionController sample here>

```csharp
// Explosion arguments
public float ExplosiveForce;
public float ExplosiveRadius;
public Transform ExplosiveTransform; // Centerpoint of explosion

public Rigidbody BaseRigidBody;
public GameObject[] PotentiallyDetachablesCubes;
List<Rigidbody> ObjRigidbodies = new List<Rigidbody>();
bool IsCPUcapable = false;
bool HasExploded = false;

void Start ()
{
```
SetCPULevel(CPUCapabilityManager.Singleton.CPUCapabilityLevel);
}

public void SetCPULevel(CPUCapabilityManager.SYSTEM_LEVELS sysLevel) {
    // Only use if CPU deemed medium or high capability
    if (sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.HIGH
        || sysLevel == CPUCapabilityManager.SYSTEM_LEVELS.MEDIUM)
    {
        IsCPUCapable = true;

        // add rigidbodies to all little cubes
        for (int i = 0; i < PotentiallyDetachableCubes.Length; i++)
        {
            Rigidbody CurrRigidbody = PotentiallyDetachableCubes[i].AddComponent<Rigidbody>();
            CurrRigidbody.isKinematic = true;
            CurrRigidbody.useGravity = false;
            ObjRigidbodies.Add(CurrRigidbody);
        }

        Debug.Log("(ExplosionController) System capability set to: " +
            CPUCapabilityManager.Singleton.CPUCapabilityLevel + ", so object (" + gameObject.name + ") is destructible");
    }
    else
    {
        Debug.Log("(ExplosionController) System capability set to: " +
            CPUCapabilityManager.Singleton.CPUCapabilityLevel + ", so object (" + gameObject.name + ") not destructible");
    }
}

public void ExplodeObject()
{
    HasExploded = true;
    if (IsCPUCapable)
    {
        BaseRigidbody.useGravity = false;
        BaseRigidbody.isKinematic = true;
        BoxCollider[] BaseColliders = GetComponents<BoxCollider>();
        for (int i = 0; i < BaseColliders.Length; i++)
        {
            BaseColliders[i].enabled = false;
        }

        for (int i = 0; i < ObjRigidbodies.Count; i++)
        {
            Rigidbody CurrRigidbody = ObjRigidbodies[i];
            CurrRigidbody.isKinematic = false;
            CurrRigidbody.useGravity = true;
            CurrRigidbody.AddExplosionForce(ExplosiveForce, ExplosiveTransform.position, ExplosiveRadius);
            ObjRigidbodies[i].gameObject.AddComponent<BoxCollider>();
        }
    }
    else
    {
        // Boring destruction implementation
BaseRigidBody.AddExplosionForce(ExplosiveForce, ExplosiveTransform.position, ExplosiveRadius);
}

void OnCollisionEnter(Collision collision)
{
    if(!HasExploded)
    {
        ExplosiveTransform.position = collision.contacts[0].point;
        ExplodeObject();
    }
}

</CODE – Embed explosionController sample here>

**CPU Capability Detection Plugin**

Ok, so we’ve been through each of the features added to Star Trek™: Bridge Crew®, but how do we determine what our target system can handle? To make this as painless as possible, we’ve created an easy-to-use Unity® plugin with source. The code comes with example code written for both Unity® and native implementations and acts as a toolbox to easily get you system metrics to help you define your target system categories. Many of the above examples are integrated into the sample to make it easy to hit the ground running. Here are the steps:

1. Define your CPU performance tiers.

   <CODE – Embed SYSTEM_LEVELS code sample here – remove picture>

   ```csharp
   public enum SYSTEM_LEVELS
   {
       OFF,
       LOW,
       MEDIUM,
       HIGH,
       NUM_SYSTEMS
   };
   </CODE – Embed SYSTEM_LEVELS code sample here – remove picture>
2. Set your CPU value thresholds. Various metrics are supplied from the plugin, such as logical/physical core count, max frequency, system memory, etc. However, you can always add your own if you’d like to consider other factors. For most basic uses, the supplied metrics should suffice.


3. Initialize the plugin and determine if the user is running on an Intel® processor.
void QueryCPU()
{
    InitializeResources();
    if (IsIntelCPU())
    {
        // Your performance categorization code
    }
    else
    {
        Debug.Log("You are not running on an Intel CPU");
    }
}

StringBuilder cpuNameBuffer = new StringBuilder(BufferSize);
GetProcessorName(cpuNameBuffer, ref BufferSize);
SysLogicalCores = GetNumLogicalCores();
SysUsablePhysMemoryGB = GetUsablePhysMemoryGB();
SysMaxBaseFrequency = GetMaxBaseFrequency();
SysCacheSizeMB = GetCacheSizeMB();

StringBuilder cpuNameBuffer = new StringBuilder(BufferSize);
GetProcessorName(cpuNameBuffer, ref BufferSize);
SysLogicalCores = GetNumLogicalCores();
SysUsablePhysMemoryGB = GetUsablePhysMemoryGB();
SysMaxBaseFrequency = GetMaxBaseFrequency();
SysCacheSizeMB = GetCacheSizeMB();
5. Compare your threshold values to determine which previously defined performance tier the system tested belongs in.

```
bool IsSystemHigherThanThreshold(SystemThreshold threshold)
{
    if (threshold.NumLogicalCores < SysLogicalCores && threshold.MaxBaseFrequency < SysMaxBaseFrequency
    {
        return true;
    }
    return false;
}
```

```
SYSTEM_LEVELS SystemLevel;
if (IsSystemHigherThanThreshold(HighSettings) || IsWhitelistedCPU(SYSTEM_LEVELS_HIGH))
{
    SystemLevel = SYSTEM_LEVELS_HIGH;
}
else if (IsSystemHigherThanThreshold(MedSettings) || IsWhitelistedCPU(SYSTEM_LEVELS_MEDIUM))
{
    SystemLevel = SYSTEM_LEVELS_MEDIUM;
}
else if (IsSystemHigherThanThreshold(LowSettings) || IsWhitelistedCPU(SYSTEM_LEVELS_OFF))
{
    SystemLevel = SYSTEM_LEVELS_LOW;
}
else
{
    SystemLevel = SYSTEM_LEVELS_OFF;
}
Debug.Log("Your system level has been categorized as: " + SystemLevel);
```
```csharp
SYSTEM_LEVELS  MySystemLevel = SYSTEM_LEVELS.OFF;

if (IsSystemHigherThanThreshold(HighSettings) || IsWhitelistedCPU(SYSTEM_LEVELS.HIGH))
{
    MySystemLevel = SYSTEM_LEVELS.HIGH;
}
else if (IsSystemHigherThanThreshold(MedSettings) ||
         IsWhitelistedCPU(SYSTEM_LEVELS.MEDIUM))
{
    MySystemLevel = SYSTEM_LEVELS.MEDIUM;
}
else if (IsSystemHigherThanThreshold(LowSettings) ||
         IsWhitelistedCPU(SYSTEM_LEVELS.OFF))
{
    MySystemLevel = SYSTEM_LEVELS.LOW;
}
else
{
    MySystemLevel = SYSTEM_LEVELS.OFF;
}

Debug.Log("Your system level has been categorized as: " + MySystemLevel);
```

Performance Profiling and Considerations

Just like with GPU work, we need to verify that our feature set’s combined CPU utilization doesn’t exceed our low, medium, and high targets to cause constant Asynchronous Spacewarp (trying very hard to resist terrible Star Trek™ pun) and reprojection triggers. We wanted to make sure that the game maintained a consistent 90 frames per second while still maximizing the CPU, no matter what machine the game was running on. The Star Trek™: Bridge Crew* team decided on three levels of feature sets: Off, Partial, and Full. So, we tested the Full group of features on a machine which matched with our Off threshold.
The above GPUView screenshot shows ~22 ms of time passing from present to present (highlighted). Present indicates when the final frame has been generated and is ready for submission to the head mounted display (HMD). This can be thought of in terms of frame rate (converting to 45 fps). Going from 90 to 45 fps means that we are consistently triggering ASW with this configuration running on our ‘Off’ tier system. Looking at three test runs over Mission 5, we see an average of ~5.5k synthetic frames being generated because of ASW triggers. Squeezing these immersive features onto Oculus min-spec didn’t work out, as we expected. But rather than keeping feature sets off across all configurations,
we bound feature sets to hardware levels we could determine at run-time to activate the appropriate set, allowing players with all hardware levels to experience the game as best as it should be experienced. If we look at the same configuration running on our high-end target (Intel® Core™ i7-7700K processor), we see things change.

GPUView showing work distribution on desktop system with KBL i7-7700K CPU + GTX 1080 GPU

<table>
<thead>
<tr>
<th>CPU</th>
<th>Graphics Card</th>
<th>Scenario</th>
<th>Configuration</th>
<th>Run</th>
<th>Refresh Intervals</th>
<th>New Frames</th>
<th>Dropped Frames</th>
<th>Synthetic Frames Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBL i7-7700k</td>
<td>GTX1080</td>
<td>Mission 5 after initial warp</td>
<td>Full Settings</td>
<td>1</td>
<td>11703</td>
<td>11666</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>11654</td>
<td>11617</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>11700</td>
<td>11672</td>
<td>28</td>
<td>0</td>
</tr>
</tbody>
</table>

**Averages**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>KBL i7-7700k</td>
<td>GTX1080</td>
<td></td>
<td></td>
<td></td>
<td>11685.67</td>
<td>11651.67</td>
<td>34.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Number of synthetic frames generated being zero indicates the CPU work never exceeded 11.1 ms per frame threshold with full feature set on the higher end CPU.

With the additional logical cores, increased frequency, and bigger cache size of our high-end target, all the work can sprawl out and complete within the allotted 11.1 ms required to hit 90 fps. The average duration of CPU work per frame ranges from 9-10.3 ms from head to tail. This means we are pushing our high-end target nearly to its limit, but still maintaining a solid 90 fps and utilizing all of the resources available to us. We’ve hit the sweet spot!
Ok, so we’ve got our ‘Off’ and ‘Full’ feature sets tested. At this point, we needed to select a subset of the ‘Full’ features to enable on the Intel Core i7-7700HK processor-based VR-ready notebooks. This is our mid-target for the ‘Partial’ feature set. We wanted to keep features that really affected the inside of the bridge, so we prioritized those and slowly removed the others one by one until we hit the sweet spot. Eventually, we only had to cut the dynamic wake effects and dynamic asteroids to comfortably push out 90 fps on the laptop. Here is a screen capture of GPUView* showing the ‘Partial’ feature set running on our test VR-ready notebook.

### GPUView showing work distribution on VR Gaming Laptop with KBL i7-7820HK CPU + GTX 1080 GPU

<table>
<thead>
<tr>
<th>CPU</th>
<th>Graphics Card</th>
<th>Scenario</th>
<th>Configuration</th>
<th>Run</th>
<th>Refresh Intervals</th>
<th>New Frames</th>
<th>Dropped Frames</th>
<th>Synthetic Frames Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBL i7-7820HK</td>
<td>GTX1080</td>
<td>Mission 5 after initial warp</td>
<td>Full Settings</td>
<td>1</td>
<td>11887</td>
<td>11242</td>
<td>116</td>
<td>529</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>11881</td>
<td>11315</td>
<td>110</td>
<td>456</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>11792</td>
<td>10912</td>
<td>125</td>
<td>755</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>Averages</strong></td>
<td><strong>11853.33</strong></td>
<td><strong>11156.33</strong></td>
<td><strong>580.00</strong></td>
</tr>
</tbody>
</table>

Number of synthetic frames generated being non-zero indicates the CPU work exceeded 11.1 ms per frame threshold with the full feature set on the VR ready laptop.
Conclusion

Overall, CPU usage increases the most from the use of more realistic, higher resolution simulations as well as the existence of more dynamic entities; physics simulations previously thought to be too expensive are now something that can be enabled on many CPUs. Additionally, various other CPU intensive systems like animation/Inverse Kinematics (IK), cloth simulation, flocking, fluid simulation, procedural generation, and more can be used to create a more rich and realistic world. The industry has had settings tiers for graphics for a while now and it’s time we start thinking the same way about CPU settings. When developing a game, think about all your untapped compute potential on different hardware levels and consider how it can harnessed to make your game something special. Check the links below for more information. Happy hunting.

- Special Thanks to Kevin Coyle and the rest of the Red Storm Entertainment team who worked with us on this partnership and helped put together this article **

Additional Resources

“Set Graphics to Stun: Enhancing VR Immersion with the CPU in Star Trek™: Bridge Crew”
The author presented the information in this article at Unite 2016.

< Embed: https://youtu.be/gtWDeJl90q8 >

Session Description – Many games and experiences these days put a huge emphasis on GPU work and let the many cores built in to modern mainstream CPUs sit idle on the sideline. This talk explores how Ubisoft's Red Storm studio and Intel partnered to push immersion as far as possible in Star Trek™: Bridge Crew using Unity* to take advantage of these available resources. Learn how you can achieve stunning visuals with minimal performance impact on the GPU in your own games!

Catlike Coding
Catlike Coding offers a number of great CPU/math-heavy tutorials that anybody can pick up and run with. The tutorials are Unity*-focused but can apply to any other engine as the meat of the content doesn’t depend on any particular API. It’s highly recommended for those interested in procedural generation, leveraging curves/splines, mesh deformation, texture manipulation, noise, and more.

Fluid Simulation for Video Games (Series)
This is a well-written tutorial on implementing fluid simulation for video games that leverage many cores. The article is great for beginners, walking them through everything from concept to
implementation. By the end of the article the reader will have source code to add to their own engine and an understanding of how to manipulate the code to emulate various fluid types.


Notices

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

This sample source code is released under the Intel Sample Source Code License Agreement.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, and Core are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

© 2017 Intel Corporation