Developer Support Engineer Interview test

1. Please solve the following problems. You can use the Unity Documentation, Scripting Reference, Stack Overflow, Google, etc:

1.1. Extend the following vertex and fragment shader to use Light Probe illumination from the scene, affecting the object being rendered.

Shader "MyShader/Diffuse With LightProbes" {

Properties { [NoScaleOffset] \_MainTex ("Texture", 2D) = "white" {} }

SubShader {

Pass {

Tags {

"LightMode"="ForwardBase"

} CGPROGRAM

#pragma vertex v

#pragma fragment f

#include "UnityCG.cginc"

sampler2D \_MainTex;

struct v2f {

float2 uv : TEXCOORD0;

float4 vertex : SV\_POSITION;

};

v2f v (appdata\_base vertex\_data) {

v2f o;

o.vertex = UnityObjectToClipPos(vertex\_data.vertex);

o.uv = vertex\_data.texcoord;

return o;

}

fixed4 f (v2f input\_fragment) : SV\_Target {

}

} ENDCG

} }

fixed4 col = tex2D(\_MainTex, input\_fragment.uv);

return col;

1.2. Create a native plugin with a function written in C/C++, which is called from Unity in a C# script and receives the following struct from C#:

struct TwoStrings { string string1; string string2;

string concatenated; }

After calling the native function from C#, passing as argument an object of type TwoStrings, the variable “concatenated” of the object will store the two strings in

“string1” and “string2” concatenated.

1.3.  Create a Unity project using Unity’s C# Job System to calculate the sum of the R channel, for each texture element of a texture. To do this, split the texture into four regions of equal size, the operation should be processed by jobs **running in parallel**.

1.4.  Use VFX Graph to create a particle system that moves along a Bezier curve.

1.5.  Create two prefabs using cubes with a shared material, packing each prefab into a separate asset bundle. Use a script to load the prefabs and instantiate them in the scene. Do not use Addressables.

2. Please, try to answer the following questions in your own words:

2.1.  Describe what each of these technologies are and what they can be used for:

2.1.1.  Scriptable Build Pipeline

The Scriptable Build Pipeline lets you program and customize the way the project is compiled/built with your own C# code or with pre-made code that comes with Unity. The SBP is generally used to improve build time or have more control over the general building flow.

Can be used for: Improving building time

2.1.2.  Scriptable Render Pipeline

Similar to SBP, SRP is an API that allow us to programmatically control certain default processes that Unity usually handles automatically. In contrast to SBP, SRP is focused on how graphic rendering is computed; it let us control and schedule the rendering commands with C#.

Since the High Definition and Universal Render Pipelines are built over SRP you can customize them as well, optimizing your rendering process to be able to include more polygons and particles on screen, include higher detailed materials and light effects with the same processing power

Can be used for: Optimize rendering processes

2.1.3.  Addressables

Addressables is an asset management system that allows the programmer to reference an asset and its dependencies from anywhere with any dependencies asynchronically using its address. It is generally used with Asset Bundles to include (download) some content from an online server, lighting the app/game’s initial weight and allowing developers to modify or include assets without having to re-build the project and publishing it all over again.

Can be used for: Deliver on-line assets or modify files more dynamically since you only to have modify the addressabled asset instead of the whole project.

2.1.4.  IL2CPP

IL2CPP stands for Intermediate Language to C++, that means, it converts C# code to C++. Then, the converted C++ is used to create a native compilation for a specified platform.

If it is true that IL2CPP builds slower than Mono, the built product may run faster since it is compiled in native, so no translation is needed at runtime.

Another main feature is that IL2CPP enables 64 bits builds on Android and iOS, a now obligatory requirement for publishing apps on the Appstore and Google Play.

Can be used for: Create 64-bits apps/games, compile on native and allow Unity to include more devices on their cross-platform rooster.

2.1.5.  Nested Prefabs

Nested Prefabs are simply Prefabs inside another Prefab. Nested prefabs instances reference their own prefab asset while, at the same time, are part of the parent prefab. One useful implementation of nested prefabs is for standardizing interfaces and have more control over their changes (see example bellow)

As you can see here, (1) We can have a Canvas prefab with a header prefab nested.

(2) If we edit the children header prefab asset and insatiate it multiple times inside the parent prefab, I will maintain the properties of the children prefab asset.

(3) It doesn’t matter if both parent and children are under the control of two separated developing teams. Any change made either on the parent core, as well as the children will be reflected on the final prefab with the nested one.

Can be used for: If there is a big asset with many pieces that may change over the developing course, the use of nested prefabs can facilitate all the workflow by allowing different teams work on individual parts simultaneously.

2.2.  Mention at least two problems of Unity’s non-incremental Garbage Collector.

2.3.  Explain which of these is better and why? Unity LTS, TECH release, Beta or Alpha?

2.4.  What is your preferred version control system and why do you prefer it over others?

2.5.  What is your favorite IDE and why?

2.6.  What issues or limitations have you recently experienced using Unity?

2.7.  What strategies or best practices can be used to optimize the CPU and GPU usage in an application made with Unity?

2.8.  How do you catch and investigate crashes happening in a released game?

2.9.  Compare the following function and macro definitions. In what cases will they produce different results and/or side effects?

**int square(int val) { return val\*val; }**

**#define square(val) (val\*val)**

There are two cases where the macro definition will throw a different result:

1. In case ‘val’ is not a number, the function will throw an error while executing it. Macros do not have type checking nor compilation error checking so is possible any artifact may occur.
2. In case the ‘val’ value is a composed operand for example val= 1+2 the function will return 9 which is correct, but the macro definition will respond with a 5 because the definition will treat the calculation as 1+2\*1+2. To fix this I would declare the macro definition as follows:

**#define square(val) ((val)\*(val))**

2.10.  What is the package manager in Unity and what is the alternative way of adding a package than via the package manager UI?

2.11.  Examine the following function. What does it accomplish?

int someFunction(int i) {

int n = 0;

while (i) {  
i &= i-1;

n++; }

return n; }

It counts and returns how many bits (1) are in a specific integer.

How does it work:

Let’s choose int i=7 for this example:

* The while loop won’t stop until i=0; any other value will be considered as true. n will only count how many loops the function has gone through.
* The ‘&=’ operand can be translated to i = i & i-1;
* The & (instead of the && logical operand) is used for bit comparison, therefore i=7 will be treated as i=111 which is their binary equivalent. Each bit is compared with their pair in the exact same index.
* Therefore, the loop will go like this:

|  |  |  |
| --- | --- | --- |
| 1st run | 111 &  110=  110 | n=1  i=110 |
| 2nd run | 110 &  101=  100 | n=2  i=100 |
| 3rd run | 100 &  011=  000 | n=3  i=000 |

The 4th run never happens since i==0 and 0 is read as false in the while loop.