Interview questions

Posted on Nov 5, 2016

#rant (/tags/rant) #work (/tags/work)

Recently saw quite some twitter discussions about good & bad interview questions. Here's a few I found useful.

In general, the most useful questions seem to be fairly open-ended ones, that can either lead to a much larger branching discussions, or don't even have the "right" answer. Afterall, you mostly don't want to know the answer *(it's gonna be 42 anyway)*, but to see the problem solving process and/or evaluate general knowledge and grasp of the applicant.

The examples below are what I have used some times, and are targeted at graphics programmers with some amount of experience.

When a GPU samples a texture, how does it pick which mipmap level to read from?

Ok this one *does* have the correct answer, but it still can lead to a lot of discussion. The correct answer today is along the lines of:

> *GPU rasterizes everything in 2x2 fragment blocks, computes horizontal and vertical texture coordinate differences between fragments in the block, and uses magnitudes of the UV differences to pick the mipmap level that most closely matches 1:1 ratio of fragments to texels.*

If the applicant does not know the answer, you could try to derive it. "Well ok, if you were building a GPU or writing a software rasterizer, what would *you* do?".

Most people initially go for "based on distance to the camera", or "based on how big the triangle is on screen" and so on. This is a great first step (and was roughly how rasterizers in the really old days worked). You can up the challenge then by throwing in a "but what if UVs are not just computed per-vertex?". Which of course makes these suggested approaches not suitable anymore, and something else has to be thought up.

Once you get to the 2x2 fragment block solution, more things can be discussed (or just jump here if the applicant already knew the answer). What implications does it have for the programming models, efficiency etc.? Possible things to discuss:

- The 2x2 quad has to execute shader instructions in lockstep, so that UV differences can be computed. Which leads to branching implications, which leads to regular texture samples being disallowed (HLSL) or undefined (GLSL) when used inside dynamic branching code paths.
- Lockstep execution can lead into discussion how the GPUs work in general; what are the wavefronts / warps / SIMDs *(or whatever your flavor of API/GPU calls them)*. How branching works, how temporary variable storage works, how to optimize occuppancy, how latency hiding works etc. Could spend *a lot* of time discussing this.
- 2x2 quad rasterization means inefficiencies at small triangle sizes (a triangle that covers 1 fragment will still get four fragment shader executions). What implications this has for high geometry density, tessellation, geometric LOD schemes. What implications this has for forward vs deferred shading. What research is done to solve this problem, is the applicant aware of it? What would *they* do to solve or help with this?

You are designing a lighting system for a game/engine. What would it be?

This one does not even have the "correct" answer. A lighting system could be anything, there's at least a few dozen commonly used ways to do it, and probably millions of more specialized ways! Lighting encompasses a lot of things – punctual, area, environment light sources, emissive surfaces; realtime and baked illumination components; direct and global illumination; shadows, reflections, volumetrics; tradeoffs between runtime peformance and authoring performance, platform implications, etc. etc. It can be a *really* long discussion.

Here, you're interested in several things:

- General thought process and how do they approach open-ended problems. Do they clarify requirements and try to narrow things down? Do they tell what they do know, what they do not know, and what needs further investigation? Do they just present a single favorite technique of theirs and can't tell any downsides of it?
- Awareness of already existing solutions. Do they know what is out there, and aware of pros & cons of common techniques? Have they tried any of it themselves? How up-to-date is their knowledge?
- Exploring the problem space and making decisions. Is the lighting system for a single very specific game, or does it have to be general and "suitable for anything"? How does that impact the possible choices, and what are consequences of these choices? Likewise, how does choice of hardware platforms, minimum specs, expected complexity of content and all other factors affect the choices?

- Dealing with tradeoffs. Almost every decisions engineers do involve tradeoffs of some kind - by picking one way of doing something versus some other way, you are making a tradeoff. It could be performance, usability, flexibility, platform reach, implementation complexity, workflow impact, amount of learning/teaching that has to be done, and so on. Do they understand the tradeoffs? Are they aware of pros & cons of various techniques, or can they figure them out?

You are implementing a graphics API abstraction for an engine. How would it look like?

Similar to the lighting question above, there's no single correct answer.

This one tests awareness of current problem space (console graphics APIs, "modern" APIs like DX12/Vulkan/Metal, older APIs like DX11/OpenGL/DX9). What do they like and dislike in the existing graphics APIs (red flag if they "like everything" – ideal APIs do not exist). What would they change, if they could?

And again, tradeoffs. Would they go for power/performance or ease of use? Can you have both (if "yes" - why and how? if "no" - why?). Do they narrow down the requirements of who the abstraction users would be? Would their abstraction work efficiently on underlying graphics APIs that do not closely map to it?

You need to store and render a large city. How would you do it?

This one I haven't used, but saw someone mention on twitter. Sounds like an *excellent* question to me, again because it's very open ended and touches a lot of things.

Authoring, procedural authoring, baking, runtime modification, storage, streaming, spatial data structures, levels of detail, occlusion, rendering, lighting, and so on. Lots and lots of discussion to be had.

Well this is all.

That said, it's been ages since *I* took a job interview myself... So I don't know if these questions are as useful for the applicants as they seem for me :)

← Older (/blog/2016/09/13/Shader-Compression-Some-Data/)

Newer → (/blog/2016/12/09/Amazing-Optimizers-or-Compile-Time-Tests/)

Possibly Related Posts

- Mentoring: You Won't Believe What Happened Next! (/blog/2019/01/07/Mentoring-You-Wont-Believe-What-Happened-Next/), from 2019 January
- A case of slow Visual Studio project open times (/blog/2017/03/22/A-case-of-slow-Visual-Studio-project-open-times/), from 2017 March
- A Non-Uniform Work Distribution (/blog/2011/02/16/a-non-uniform-work-distribution/), from 2011 February

---

Have **feedback on this post**? Let me know on email (mailto:aras@nesnausk.org), mastodon (https://mastodon.gamedev.place/@aras) or twitter (https://twitter.com/aras_p).

Recent Posts

Float Compression 5: Science! (/blog/2023/02/03/Float-Compression-5-Science/)

Float Compression 4: Mesh Optimizer (/blog/2023/02/02/Float-Compression-4-Mesh-Optimizer/)

Float Compression 3: Filters (/blog/2023/02/01/Float-Compression-3-Filters/)

Float Compression 2: Oodleflate (/blog/2023/01/31/Float-Compression-2-Oodleflate/)

Float Compression 1: Generic (/blog/2023/01/29/Float-Compression-1-Generic/)

Float Compression 0: Intro (/blog/2023/01/29/Float-Compression-0-Intro/)

Swallowing the elephant into Blender (/blog/2022/07/20/Swallowing-the-elephant-into-Blender/)

All Posts (/all-posts)

Categories

blender (4) (/tags/blender)

code (115) (/tags/code)

compilers (16) (/tags/compilers)