Initial Thoughts on Quake 3 and Quake 4 SDK source codes

During this week, I had a look at Quake 3 Arena source code and Quake 4 SDK source code. I spent most of the time looking at Quake 3 Arena source code, and only a brief time looking at Q4 SDK.
Here are some random thoughts

Quake 3

1. Everything here is just a guess. It could be wrong ☺
2. Quake 3 is divided into 8 smaller modules. Overall, they build to two dlls and one exe. The two dlls correspond to client game, and server game. The exe is the actual game. The 8 modules are
    a. **Botlib**: The area awareness system (AAS) and other functionality needed to create the bots. The bot AI uses these functions to make their decisions (like visibility, routing….etc)
    b. **Cgame**: The client game. All actions made here are not final until the server has approved the actions.
    c. **Game**: The server game. This is always running
    d. **Q3_ui**: The actual UI used by Quake 3 (i.e. the specific menus and actions…)
    e. **Quake3**: This is the main exe for the game. It contains simple things like the cinematics
    f. **Renderer**: Responsible for drawing everything. It contains the BSP code, animations, and texture handling
    g. **Splines**: Used to generate curves and interpolate between them. I am not sure if this for actually "rendering" curves, or for generating curved paths for projectiles.
    h. **Ui**: This is the general GUI system. The Q3_UI makes use of this one to generate the specific GUI for Quake 3 Arena
3. Since Quake 3 Arena is entirely written in C, it is very hard to guess where are the main components in the engine. There are rumors that the code is so much better than Quake 2, but, for me, it was still very hard to follow.
4. It was pretty easy to identify the very low level things (like cinematics, keyboard handling, rendering the in-game console, and taking commands). I could modify those behaviors easily (disabling cinematics, making keys stick…)
5. I had problems making changes to the DLL parts of the game! My changes didn't reflect to the game, and I couldn't set break points anywhere in the DLL part of the game! No idea why!
6. Some parts in the game were easy to identify and some parts were very hard to identify
    a. For example, it is easy to identify and modify the part of the code related to taking a powerup or armor, including the respawn system… etc.

b. On the other hand, it was very difficult to comprehend what happens in the virtual machine part
c. Some others, I had mixed success with, like for example, the shader parser was easy to read, but the whole system was complicated to follow.

7. Some .c files were identified with the most important functions, like for example in the BSP loader, we get this note in the beginning

```
/*
Loads and prepares a map file for scene rendering.

A single entry point:

void RE_LoadWorldMap( const char *name );
*/
```

This really helps make things clearer

8. On the other hand, some files are just so hard to comprehend, and it is not easy to guess which functions are internal, and which functions are essential for interfacing (i.e. C doesn't have a public/private keyword! It would have been great to have it!)
9. Some parts in the code are really shocking… For example,

```
if ( tr.numLightmaps == 1 ) {
        //FIXME: HACK: maps with only one lightmap turn up
fullbright for some reason.
        //this avoids this, but isn't the correct solution.
        tr.numLightmaps++;
}
```

Apparently, they didn't have much time to correct this! ☺

10. This code was not written with the idea that other people will be reading it! I think it was suitable for the internal coders, where everyone had to focus on one specific area of the code. Here is an example function…

```
//===========================================================================
//
// Parameter:                           -
// Returns:                             -
// Changes Globals:              -
//===========================================================================
int AAS_OriginOfMoverWithModelNum(int modelnum, vec3_t origin)
{
      int i;
      aas_entity_t *ent;

      for (i = 0; i < aasworld.maxentities; i++)
      {
            ent = &aasworld.entities[i];
            if (ent->i.type == ET_MOVER)
            {
                  if (ent->i.modelindex == modelnum)
                  {
                        VectorCopy(ent->i.origin, origin);
                        return qtrue;
                  } //end if
            } //end if
      } //end for
```

```
	return qfalse;
} //end of the function AAS_OriginOfMoverWithModelNum
```

The comments above are left blank by coders, and there is no way of knowing the importance of this function, and whether it is used by people outside this file or not. It would have been nice if we had dOxygen style comments in the code.

11. Overall, I wasn't very impressed with this code, especially after looking at the Quake 4 SDK source code (see below!)
12. I think, given more time with the code, I will be able to get more familiar with it. My thoughts here, are just initial reactions. Maybe next week my opinion will be totally different.

Quake 4

1. Quake 4 on the other side, was much more pleasant to read than Quake 3 source code. The reasons for this could be
    a. The nice directory structure, where files are grouped by concern. This makes it much easier to locate a certain functionality
    b. Quake 4 is implemented in C++, so, looking at the inheritance hierarchy makes it easier to understand what happens in the code. Also, ignoring private methods in the beginning makes it much easier to understand the big picture. In addition, after a while, we learn that "Save()", and "Restore()" are the basics for the persistence system, which makes life easier…
    c. The code base here is much smaller than Quake 3 Arena, because this is just the SDK and not the full game
2. I didn't spend as much time on Quake 4 as I did on Quake 3, so maybe I missed some things, yet, I feel I understand the structure of Quake 4 SDK better than Quake 3's.
3. I had a closer look at the script compiler and interpreter. The interpreter was really easy to follow. The compiler was more difficult but still much easier to follow than the virtual machine in Quake 3.
4. The description comments on top of each .h file made it really easier to understand what each class was doing. Jumping ahead to the public section made it very easy to understand what each class is exactly responsible for. Looking at the virtual functions made it easy to spot what I need to implement to extend those classes!
5. Some parts were still hard to comprehend. I guess this is because of the very brief time I spent with this code.
6. Again, these are all initial thoughts. They may completely change next week, after I had had the chance to look more into the code ☺ Also, the SDK is much smaller than the whole game source code, so, it is like having the important parts pointed out! This could be another reason why it was easier to follow it.