Integration of OpenGL 4.4 with Virtual reality components on a previously made Robotic Arm

A Virtual Reality Project

Denis Beqiraj, Diego Moranda, Andrea Riccardi, Lorenzo Ronzani
Virtual Reality Course - SUPSI DTI
Client: Peternier Achille

Spring Semester - Academic year 2021/2022
13. May 2022

Introduction

The main goal of this group project was to develop a **working** Virtual Reality application, based on the **previously** made Robotic Arm.

The **requirements** of the project were about a first phase of **upgrading** the current robotic arm to implement the **OpenGL 4.4 features**, adding the GPU support to **improve** the **overall speed** of the project, the second phase started with the **implementation** of the various requirements, such as a **Skybox**, adding **shaders**, **per-pixel lighting**, **stereoscopic rendering** and the **OpenVR integration** capable of using an **headset** such as Riftcat, with the possibility to **interact** with the virtual environment using the **Leap motion** infrared sensor.

At the end of the project we got a **working** robotic arm, with an **interactive instrument panel** used to control such an arm with Leap motion, giving the **feel** of being there near the arm.

Keywords: OpenGL, C++, Skybox, Shader, OpenVR, Riftcat

Requirements

The **requirements** of the project were to upgrade the current working previously made robotic arm with the new **features** learnt in this course. All this by using OpenGL 4.4 and the C++ language. The project followed the outline of the features seen during the lectures. The **features** needed are the following:

- Working Virtual Reality robotic arm
- OpenGL 4.4 functions
- Shaders integration
- Per-pixel lighting
- 2 lights (Omnidirectional, Spot)
- Skybox
- File configuration parameters
- OpenVR integration
- Leap motion integration

The final product must be able to run in **Windows** as a **standalone** project, so a *.dll* must be provided. It also needs to be able to run in **OpenVR** and have the **interactivity** of the Leap motion infrared sensor.

Architecture

The architecture is divided into **client** and **engine**. Using the previous engine we built the new **features** on it. Here we are going to list the **main** ones.

Meshes

To upgrade the old engine to OpenGL 4.4 we **removed** the old functions that would cause a **pipeline stall**, implementing the feature of **VBOs** and **VAO** on the meshes. So we have a **single VAO** for Mesh that has all the **VBOs** inside, such as the **Vertex** vbo, **Normal** vbo, **Texture** vbo and **Faces** vbo.
The render sets the **uniform** of our mesh like **modelview** and the **inverse transpose** of that one, for a **normal** matrix.
After that we render our **materials**, **bind** our **VAO** and draw elements.

Materials

The new version of our engine instead of using old methods to set **diffuse**, **specular**, **ambient** and **emission**, we put them in their proper locations through **uniforms**.
If the material has **texture** we bind it, otherwise we set the **void texture** created for that purpose.

AABB inclusion

To keep track of the various **collisions** we implemented a class called **AABB** “Axis-Aligned Bounding box”. Which in our case are used in the phase of **keeping track** of the **user touch** on the controls, and also to keep track of the **scene rendering**, because it’s a **bounding-box culling**, in which we have a sphere, in which if the **bounding box** of the object is located inside it will get rendered, otherwise it won't.

Leap motion integration

To use the Leap Motion the “Minimal Leap Motion API wrapper” given by the professor.
We created in 3d studio max **23 spheres** representing the **elbow**, the **wrist** and the **hand** with all the **fingers,** that translates depending on leap vectors. To implement the Leap Motion we also had to use a **thread** because of the lagging hand so we got a higher **refresh rate**. We made that change to **clear** the leap motion buffer faster thus making it go faster.

Line 403 engine.ccp file

```
// Render hands using spheres:
    mutex.lock();
```

As we implemented the thread, we also made the program **thread safe** watching the possible **check-then-act**/**race condition** problems.

Shaders

To display our scene we use **shaders**, in particular we created six shader files, **three vertex** and **three fragments**.
We used a **vertex_cubemap** and a **fragment_cubemap** shaders to implement the **skybox** in the background which returns a **texture** that will be **rendered** behind everything.

We used a **vertex_fbo** and a **fragment_fbo** which display the virtual environment if you **don't have** the headset on, if you have the **vr on**, it will show on the desktop the left eye view.
We also implemented the **main** shaders: the **vertex** and **fragment**. In the fragment we use the **single pass rendering**, using the **per-pixel lighting**, especially for the fragment shader we calculate the sum of all **lights**/**materials** in the scene, for the **omni** light we calculate as the example in vr series and for **spotlights** we calculate the **angle** by the **acos** of the dot of the **direction** of the **light** and the minus direction of **spotlight's** one, so we check if this value is less than the **cutoff**. The source code of shaders is written on **file** for greater **dynamism**. This allows us to make faster **changes**.

Setup shaders

We set up the shaders at the end of the **init engine** process, we do this because a shader needs to know where the **values** that we **pass** are.
For the ***passthroughShader*** (used for the fbo render) we did this steps: we **binded** the **vbos**, that contain **box** and **texture coordinates**, and we binded the **texture properties** (we use one FBO for each eye).
For the other two shaders we simply binded the **needed parameters** like the **modelView** and the **projection** matrix, the **light** and **material settings** and the **texture parameter**.

VR or LeapMotion

We use a simple txt file (with **0/1** numbers) to make the engine **understand** if we are using the **VR** part or not and if we are using the **leapMotion** or not. This file is read in the engine **init** method.

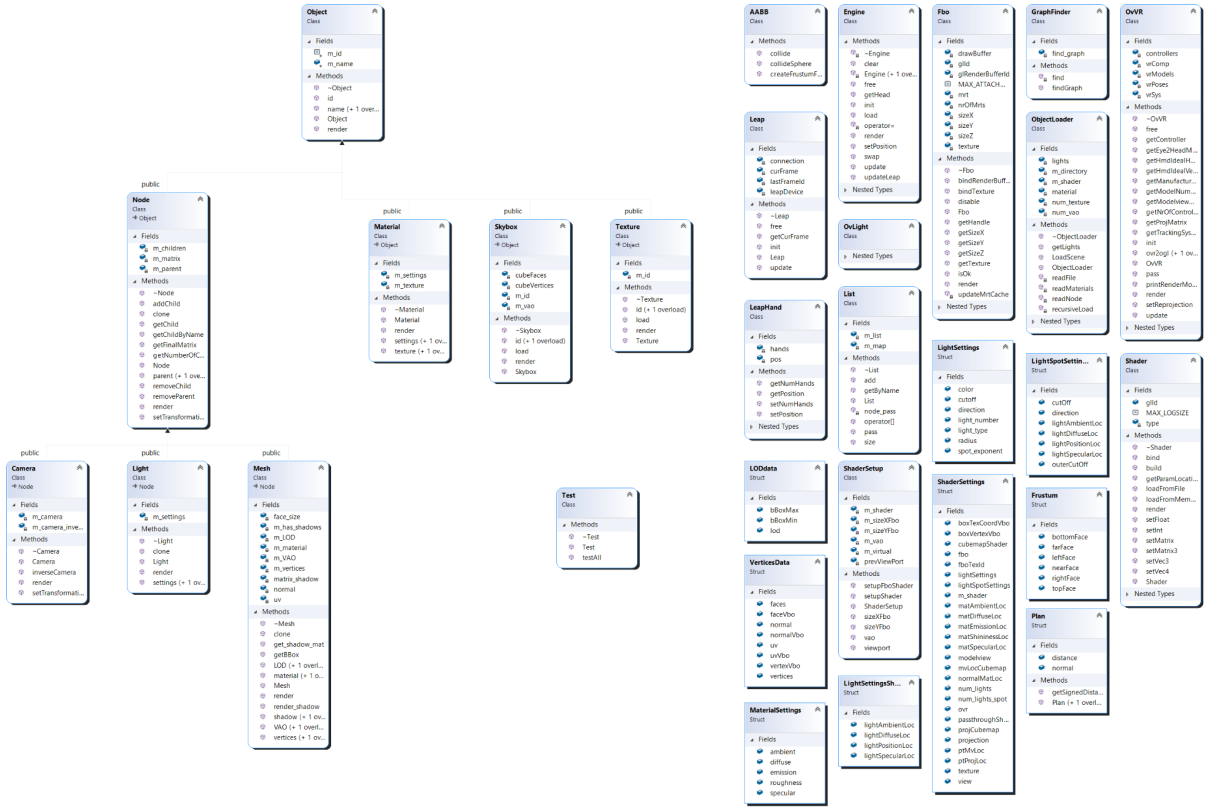
Stereoscopic rendering

For this kind of rendering we make use of two **fbo** and of the **eyes** and **projection** matrices given by **openVR**, that are different for each eye. The first step is to **render** the scene for the point of **view** of the left eye and pass the correlated **fbo** to **openVR**, then we do the same thing for the right eye.

Interaction

To make the virtual environment **interactive** we implemented using the leap motion and the hand a fully **functional hand**, which thanks to the **AABB** explained before are able to **touch** the buttons to operate the robotic arm.
To move around we still use **WASD**.

Engine class diagram



Engine

The new version of engine has some **changes**, in fact now we **initialize** all the settings of Leap Motion and SteamVR on **init method**, even with **uniform** locations.
The **render** method renderize in the **FBO** the scene, and **optimizes** it with our **sphere culling**, after that step we get the FBO **texture** and **render** it to the screen.

Client

The client can use some **new methods** of our engine, the difference between the previous version is:

- **updateLeap** that updates the hand spheres
- **setPosition** that sets the headPosition of SteamVR head
- **getHead** that returns modelview matrix of SteamVR head

Results

We developed a **generic** graphic engine that is able to load a scene from an .ovo file and render it. Our graphic engine works with **OpenGL 4.4**, which is mandatory because we should improve the **performances** and **qualities**.
OpenGL 4.4 forces us to implement **Vbo**, **Vao** and **Shaders** (at least 1 **Vertex** and 1 **Fragment**).
Thanks to the use of shaders, we have implemented the **per-pixel lighting.**
Our engine supports 2 types of **lights** (**Spot**, **Omni**) using **one-pass rendering** technique.
We implemented **fbo** to prepare data for OpenVr.
We simulate **stereoscopic rendering** generating 2 images (1 for left and 1 for right eye), then we insert them in a **fbo** and lastly, they are taken by **OpenVr** and used.
We added two options (in a file) for choosing if we **want to use** OpenVr **or not** and the same for leap motion. That is very **useful** for the testing phase.
We added a **skybox** that allows us to make our scene “infinite”.
Lastly we added the **motion capture** with **Leap Motion** and we implemented the interaction with the **bounding box** techniques.
Our engine works only on **Windows** because it is the main platform for Virtual Reality applications.
We used our generic graphic engine to create a **three segment** robotic arm. We **control** it using a small button panel placed on the side.
We use **OpenVr** with Riftcat to test our application.
Our engine is able to render **shadows** of objects, with the fake shadow technique.

Demo Screenshot

In this screenshot we can see the **Skybox** as the universe, we can see the **Hand** composed of 23 **Spheres**, there is also the **control** panel in which we can press the **buttons** to move the robotic arms around. We have a **dynamic light**(cyan) mounted on the arm, currently **pointing** at the rotating planet Earth.

Conclusion

The pipeline from the older engine to the new one was pretty **clear**, we had some difficulties when we tried to implement the **spotlights**, because of the coordinate space, the solution was to change the world coordinates in eye coordinates, otherwise everything worked **without** too many **problems**.
Finally, we got a **working VR graphic engine** with some **interactions** with the virtual environment thanks to LeapMotion. We **learned** a lot of **important** theory notions about how the **VR engines** are built, how we could manage interactions with the user and how important it is to create some kind of **immersion** for a **better experience**.
Unfortunately we didn't have **enough time** for studying and implementing the **physics** in our engine. That could be **the next step**, because our engine is quite **flexible** and we could **integrate** this feature via an **external library**.
We've implemented the **frustum shape** from the camera, but for time reasons it's still a work in progress.

References

https://learnopengl.com/Getting-started/Camera
http://www.peternier.com/
https://theswissbay.ch/pdf/Gentoomen%20Library/Game%20Development/Programming/OpenGL%20SuperBible%204th%20Edition.pdf
https://www.youtube.com/watch?v=45MIykWJ-C4
https://learnopengl.com/Advanced-OpenGL/Cubemaps
https://learnopengl.com/