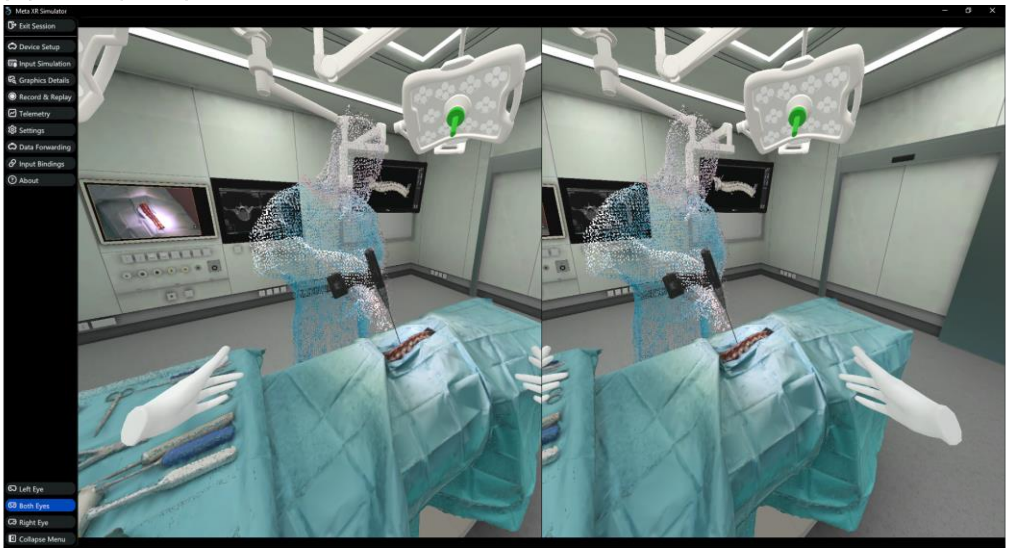
Overview document for VR renderer of animated point-clouds

Project Overview

The goal of this project was to have a real-time VR app showcasing the digital twin of the ORX operating room. The 3d model was created in Blender which needed to be exported to Unity 2022.3. Out of the available VR and AR headsets the Meta Quest Pro was chosen and was used over the Quest Link [1] due to performance requirements. To interface with the Pro in Unity the Meta SDK was used [2] [3]. As to not always needing to start up the headset for testing, the Meta XR Simulator [4] was used.
In addition to the 3D models of the room, the project also contains an animation system that loads the captured point cloud data (with timing information) and superposes it onto the already rendered geometry. To be able to write this as a custom render feature the Universal Render pipeline (URP) was chosen.



Existing technologies

Unity does not support Point Cloud rendering out of the box. While there are existing packages (some not free) that enable point cloud rendering, they have the following issues:

- Do not support VR cameras, leading to bugs such as rendering to only one eye
- Use progressive rendering (adding more points from frame to frame) which assumes a stationary camera for at least a few frames at a time to converge to a good result. This is not the case in VR with headtracking
- No support for animation i.e. switching point clouds regularly
- Some only support rendering points as tiny triangles which is not well suited for the hardware rendering pipeline

These issues lead to the decision to write a custom renderer in Unity which outputs two textures that will be superimposed into the two rendered VR views.

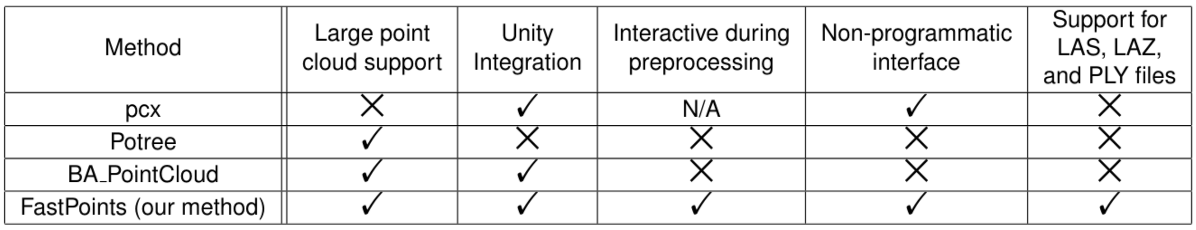
[1]: https://developer.oculus.com/documentation/unity/unity-link/

[2]: https://developer.oculus.com/documentation/unity/unity-env-device-setup/

[3]: https://developer.oculus.com/documentation/unity/unity-package-manager/

[4]: https://developer.oculus.com/documentation/unity/xrsim-intro/

[5]: https://developer.oculus.com/documentation/unity/unity-sf-locomotion/



Method	Large point cloud support	Unity Integration	Interactive during preprocessing	Non-programmatic interface	Support for LAS, LAZ, and PLY files
pcx	✗	✓	N/A	✓	✗
Potree	✓	✗	✗	✗	✗
BA_PointCloud	✓	✓	✗	✗	✗
FastPoints (our method)	✓	✓	✓	✓	✓

Comparison of different Pointcloud libraries [6]

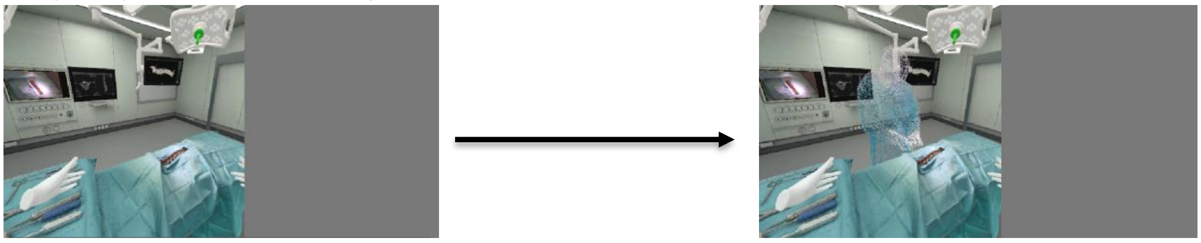
Even though the room size is quite limited, movement via controllers needed to be implemented as the Quest Pro is connected to a laptop via a Link Cable (**TEST OUT LINK AIR)**. For this purpose, the movement implementation (Dual-stick teleport) from the starter samples was used [5].

Technical overview

Post-processing

As the scene contains both traditional mesh geometry and a point cloud, we need to add the rendered points to the existing camera texture generated by Unity. This is done as a post-processing step after all other geometry has been handled. It is implemented by blitting/copying to the camera texture using Unity's URP Custom Render feature.

For correct occlusion culling between scene geometry and points, we also need to compare the depth of the rendered points with the scene depth. The point color is only copied if it's closer than what has already been rendered to the camera. To avoid points corresponding to the animated mesh hiding the actual mesh, the depth of the points is biased into the negative. The occlusion culling could instead be done while rendering the points to the temporary texture for better performance.



Compute shader

The point cloud is rendered into a temporary texture (RWStructuredBuffer<uint>) which is later blitted into the camera's render texture. This is done by running a compute shader with each thread processing a single point. The temporary texture contains the depth values in the most significant bits and color information in the remaining least significant bits. This is done to write to the texture atomically using InterlockedMin to write both depth and color at the same time. InterlockedMin also correctly writes points in the same pixel closer to the camera over points further away.

[6]: https://arxiv.org/pdf/2302.05002

Technical difficulties

Blit VR camera texture

An issue with most of Unities tutorials on blitting and some of its post processing is that they don't explain how to apply them correctly to XR where there are two cameras. In our use case, we want to copy from a different texture depending on which camera a pixel corresponds to. This is indicated by `unity_StereoEyeIndex` in the fragment shader [7].

Projection matrices

The project uses one camera for both eyes (Multiview optimization) and thus getting the projection matrix requires the use of `GetStereoProjectionMatrix` instead of `projectionMatrix`. The view matrix however is still accessed from the two, separate cameras (even though those don't render anything).

DX11

When running Meta via the Link, the only supported graphics API is DX11 [8]. This limits us in which extensions are available to us [9] and does not support 64-bit integer atomic operations (InterlockedMin). Thus, the per-pixel size of our temporary texture is limited to 32 bits. We use 24 bits for the color thus we only have 8 bits left for depth (0-255).

Depth

Depth computed by perspective division (z/w) is non-linear and needs to be converted to linear depth [10] so we can store it with more precision when we cast it to an integer with a range from 0 to 255. Be careful to also linearize the depth of the camera depth texture.

JSON Package

Unity's built-in Json solution is quite limited in what it can parse with it not being able to handle dictionaries and multidimensional arrays. Therefore, for more complex parsing use the also supported package Newtonsoft-Json which supports more features [11].

Gimbal lock

When loading the transform matrices for the drill from JSON, they need to be converted into the transform of the object [12]. The rotation is handled differently by using the `Quaternion.LookRotation` function using a transformed point as the coordinate changes from the captured data to Unity needs to be applied. It's important to use the Quaternions instead of `Transform.LookAt` as it uses Euler angles and can lead to gimbal lock.

[7]: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@13.1/manual/renderer-features/how-to-fullscreen-blit-in-xr-spi.html
[8]: https://forum.unity.com/threads/unity-vr-compatible-graphic-apis.1441978/
[9]:https://docs.google.com/document/d/1yHARKE5NwOGmWKZY2z3EPwSz5V_ZxTDT8RnRl521iyE/edit#heading=h.39v0g15e9zo1
[10]: https://stackoverflow.com/questions/6652253/getting-the-true-z-value-from-the-depth-buffer/6657284#6657284
[11]: https://docs.unity3d.com/Packages/com.unity.nuget.newtonsoft-json@2.0/manual
[12]: https://forum.unity.com/threads/how-to-assign-matrix4x4-to-transform.121966/#post-1830992

Efficient loading

When a new point cloud is needed for the animation (read from timing data of the recording) we read the whole binary ply file into memory. Then we check the header and read all the points into our custom point array (containing only position and color in only 32 bits). For performance reasons, this is done as an asynchronous job.

Exporting Blender to Unity

Back-face culling

Blender does not have back-face culling enabled by default in contrast to Unity. To enable back-face culling in Blender use Viewport shading; click Shading (v) -> Back-face culling. To display the normals in Blender in edit mode go to Gizomos (v) -> Display Normals.
To fix the normals in edit mode open the Normals window (Alt+N).

- For most objects use Recalculate Outside (looking at an object from outside of it).
- For walls: Recalculate Inside (as we are inside of the room, looking out)
- If there are still issues persisting with back-face culling: Duplicate the mesh, Flip the normal (again normal window), and then join the meshes in the object model

Complex Materials

Blender's export for Materials only supports Principled BRDF and not more complex materials. To export, do the following [13]:

1. Under the UV Editing tab select all vertices, UV-> Smart UV Project
2. Under Shading, create an Image Texture node and create a new image
3. Use the Cycle renderer to Bake the Diffuse Color into the texture
4. Use that texture as input into a principled BSDF and use this as your material output
5. Export

Emissive Materials

The blender export does not support emissive materials. This needs to be set in Unity by hand (to enable editing of the materials: Extract Materials from Prefab).

Baked GI

To get Global Illumination we use baked GI. For that, each object calculates its Lightmap UVs. By default, these might overlap. Either recalculate them with better values or switch the object with problematic geometry to use Light probes (and add some to the scene). While baking switch all material's side to both and go back to the front face setting after you finished baking.

How to run

Saschas Laptop: Press Boost button next to Power button for optimal performance

Setup

1. Start up Meta Quest Link App and the Unity project **(Internet required)**
2. Connect the Quest via Cable
3. Start Quest
4. Create a new Room scale boundary
5. Start up Link from Quicksettings (second from Left) -> Quest Link
    a. Allow USB Debugging

[13]: https://docs.unity3d.com/Packages/com.unity.nuget.newtonsoft-json@2.0/

    b. Launch
6. In the menu, select + and the Unity Project (Not the Unity Hub)
7. Start the project in the Editor ▶

Controls

Right Control stick: Left/Right rotate view
Left Control stick: Pull to get Teleport laser, Release to Teleport (if line green), Can rotate view direction after Teleport with Right Control stick while holding the Teleport
Press ≡ to reset controllers if misaligned
Press A button to hide surgeon and drill

Further improvements

Antialiasing

Further away the point cloud suffers from aliasing. This can be avoided by having one pass where one only writes the depth (also enables more precise depth) and then one where one adds color to the current color in the temporary texture and counts the pixels. In the blit, the sum is then divided by the amount thus anti-aliasing.

Temporal stability

The down-sampled point clouds were uniformly down-sampled independently from frame to frame, which could be a reason for the slight flickering. This could be improved by using optical flow to ensure that points from one frame are always also included in the next. Once that is the case, one could also interpolate between as the application runs at 90 FPS and the recording was made at 30 FPS.