COMP.SGN.320.2021-2022 3D and Virtual Reality

Extended reality

1 General Instructions

The goal of this assignment is to cover some of the basic concepts related to Augment Reality and Mixed Reality which are both sitting under the umbrella term Extended Reality.

Tasks

This assignment consists of several tasks in which you must create (dynamic) computer-generated content and blend it with a real-world environment. All tasks may be performed at home. Completing tasks 2.1–2.4 is required to pass the exercise with grade 1 and completing each additional task will increase the grade by 2, up to grade 5.

Deliverables

Each group should return:

1 – **Matlab script** that does the required work. Your script should execute without any errors. Please, **list clearly which steps you have completed** in the comments, at the beginning of the main Matlab script.

The submission must contain all **Matlab scripts**, and **any additional data** required by your scripts to run successfully, everything within a single **ZIP** file. See Moodle for the exact deadline. Possible extensions to the deadline should be negotiated **before** the deadline and may be awarded at the discretion of the assistant if valid arguments are presented.

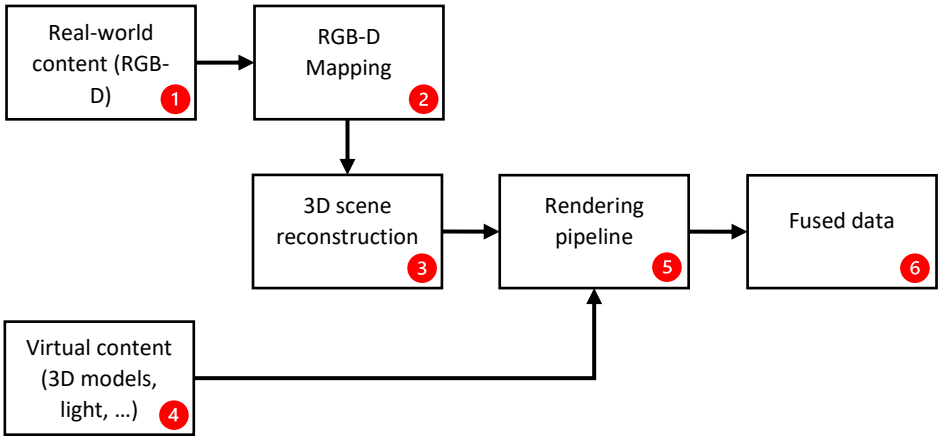

Figure 1 – Main stages of the framework to fuse virtual content and real-world content captured with RGB-D sensor. 1) Color image and depth image from RGB-D camera system; 2) Data mapping depth camera ↔ color camera; 3) Scene reconstruction using registered depth and color data; 4) Creation of all virtual elements, including 3D models, light sources, …; 5) (Real-time) rendering pipeline using Z-buffer to blend virtual content and real-world content; 6) The fused data (mixed reality).

2 Homework tasks

2.1. Virtual content (mandatory)

Create a **custom** 3D model (a cube) consisting of N=8 vertices and M=12 polygons (or triangles) by manually defining the points, their connectivity, and other attributes. Use the auxiliar library for this task.

- Each vertex is a 3D point (X_N, Y_N, Z_N) in 3D space and the set of vertices is stored in a 3xN matrix of coordinates.
- Each polygon (or face) is a connection between three vertices, and it is stored as triplets of indices into the array of vertices, i.e. 3xN matrix where each column represents a connection between the indexed vertices.
- The coordinates are given in meters and relative to the *world origin* (0, 0, 0). For simplicity, the *world origin* coincides with the depth camera location. Therefore, everything will be relative to the depth camera location. *Note: depth camera (axes) is different than the world axes.*
- Assign different colors in a 3xN matrix to each polygon so that each face of the cube has a different color. You may also assign the color for each vertex.

In addition to the 3D model, you are encouraged to play with the other scene components, including materials and lights.

To preview your model, you can use *patch* function.

2.2. Real-world content (mandatory)

For simplicity, we are going to use synthetic RGB-D in this assignment. However, real RGB-D data should work as well despite all the extra challenges to obtain clean and reliable data.

Start by loading the synthetic RGB-D data and respective calibration. *Note:* do not forget that the given synthetic "depth" image (distance map) must be converted to the actual depth (just like in LW2 – H1 b.). Implement this conversion inside *DistToDepth* function.

Then, apply the mapping of color data to depth image plane (just like in LW2 – H6). Implement this mapping inside *GetMappedColor* function.

Finally, we are going to keep foreground objects only to reduce the rendering complexity.

2.3. 3D scene reconstruction (mandatory)

Since our goal is to fuse both real-world content and virtual content, we must first reconstruct the real-world environment so that virtual elements (e.g. light sources, collisions, …) can interact with it. To this end, we first make use of *delaunayTriangulation* function. This algorithm is fast, and it is enough to transform our real scene (i.e., point-cloud) into a mesh. Although, keep in mind that there are more sophisticated algorithms for this job such as Poisson Surface Reconstruction. You can try and decide yourself on how to apply *delaunayTriangulation* efficiently, either in the 3D data (X_N, Y_N, Z_N) or 2D data (u, v).

You can use *trisurf* function to preview your triangulated data (mesh).

2.4. Rendering pipeline (mandatory)

So far, all the input data (real-world content and virtual content) is static. To make things more interesting and to test whether the real-world data and virtual content are blended correctly, we can add some animations. Thus, we need a rendering loop which updates the state of the different components created up until now (e.g., change color, move, rotate, scale, add or remove 3D model(s) over time), update input controls (not used in this assignment), window callbacks, etc.

In this task, your goal is to add more virtual content (3D models) to the scene and animate them. You can also change any 3D model attribute that you may like (e.g., material).

2.5. Fused data (+2)

After the previous task, the only thing missing in the rendering pipeline is to fuse both virtual and real-world content. *Note: please use the XRView window to display the fused results!*

In addition to blending the data, you must make sure that the light source (which can be static or dynamic) is illuminating the scene as expected.

Use *patch* and/or *trisurf* to accomplish this task.

2.6. Collisions (+2)

Find a way to implement a small collision detector and make the virtual content interact with itself and with real-world content. For example, you can simulate gravity and apply it to simple objects (e.g., cubes) and define a planar object (floor) where those objects cannot pass through.

2.7. Downsampling 3D reconstruction data (+2)

As you may have noticed, the reconstructed 3D data from real-world content is computationally heavy due to the large amount of data points (vertices). Therefore, in this task, you are asked to find a way to reduce the number of vertices. As a result, the performance should improve (i.e., the rendering loop should run faster) and the quality of the real-world content should decrease.