–Feburary Paper,2015–

Virtual Reality 3D Scanning

Abstract

We propose a device and accompanying software that provides real-time feedback as a scene is 3D scanned. This method allows viewing the current scan from multiple angles and viewpoints, while also providing a feeling of presence through a virtual reality interface.

1. Introduction

...

2. Approach

...

3. Implementation

The program was written in C++, using C++11 features such as lambdas. It uses OpenGL for rendering, SimpleDirectMedia Layer (SDL) to create a window and maintain an OpenGL context, the Oculus Rift SDK to render to the Oculus Rift and read position/rotation tracking data, the OpenGL Mathematics (GLM) library for mathematical structures and operations and the Intel Double Springs 4 SDK to read depth images from the Intel Double Springs 4 camera.

The implementation is divided into 4 main systems, each represented by a class: `Game`, `VR`, `DS`, `Scene`. `Game` runs the main event loop, `VR` handles virtual reality rendering, `DS` reads depth camera data and `Scene` manages scene elements and draws them.

3.1. `Game`

`Game` maintains an SDL window with an OpenGL context, listens for events and notifies event handlers, and provides a frames-per-second display. The constructor initializes SDL, creates a window and an OpenGL context, and the destructor quits SDL.

In its public interface, the most important part of `Game` is the `Game::update(update, draw)` method, which runs the main event loop, taking two functions `update` and `draw`. The `update` function is called every loop, and the `draw` function is called whenever the scene must be re-rendered. These are separate callbacks to allow updating at a higher frequency than drawing, or to allow update even when not drawing, such as when the window is minimized.

Other than this, `Game` provides methods to retrieve the SDL window, access command-line arguments and quit the program.

3.2. VR

VR is the main interface to the virtual reality display. It uses the Oculus Rift SDK to communicate with the Oculus Rift device, and the implementation has been designed so there are no references to the Oculus Rift SDK elsewhere in the code and where required the device is communicated with only through the VR class. The constructor creates an OpenGL framebuffer to render to that is shared with the Oculus SDK, resizes and repositions the window to render to the Oculus Rift, and enables position and rotation tracking. The destructor deinitializes all of this.

The public VR::draw(drawer) method draws to the virtual reality display, taking a function drawer that draws the actual scene as if drawing to a conventional screen. It achieves this by calling drawer twice, once for each eye, each time setting up the projection and view matrices according to the eye rendered from. The view matrix is dependent upon the position and rotation of the head-mounted display and takes into account the displacement of each eye from the center of the head. To calculate the view matrix it computes the inverse of the matrices returned by VR::eye_transforms().

The public VR::eye_transforms() returns two matrices in an std::array, each representing the camera matrix of an eye. It also supports an optional boolean parameter mid. If mid is true, it returns the camera matrix of the 'mid' eye, as if viewing the scene from the center of the head with the same eye orientation.

3.3. DS

DS is the main interface to the depth camera. It uses the Intel Double Springs 4 SDK to communicate with the Intel Double Springs 4 depth camera. The constructor probes for the depth camera configuration, enables Z (depth) capture and sets the camera resolution to 480x360.

The public DS::points() method captures a depth image from the camera and returns an std::vector representing a point cloud. These points are placed in the camera-space of the depth camera, with the z-axis pointing forward, the x axis pointing right and the y axis pointing up, in a frame that looks 'down' through the camera.

3.4. Scene

Scene handles the maintenance and drawing of scene elements. In the current implementation, there are no scene elements other than the point cloud itself. The scene can be rotated and zoomed in and out of with the mouse. The Scene::update() function must be called every update to handle mouse events. Scene::draw(points, vr) draws the point cloud, taking the array of points points and the current VR instance vr. The VR instance is required so that the points can be placed in their correct world-space position using the 'mid' eye camera matrix returned by VR::eye_transforms(). Since the depth camera is attached to the front of the head-mounted display, this achieves the effect of registering multiple captures from the depth camera.

...

References