POINTIFY

Tagline 1
Tagline 2

Created by

Jacek Burys
Kabeer Vohra
Adam Hosier
Rui Liu
Ayman Moussa

Imperial College London

2016

Contents

1 Executive Summary **1**

2 Introduction **2**
2.1 Motivation . . . 2
2.2 Objectives . . . 2
2.3 Contributions . . . 2

3 Project Management **3**
3.1 Project Plan . . . 3
3.1.1 Iteration One . . . 3
3.1.2 Iteration Two . . . 3
3.1.3 Iteration Three . . . 3
3.1.4 Iteration Four . . . 3
3.1.5 Iteration Five . . . 3
3.2 Management techniques . . . 3
3.2.1 Version Control . . . 3
3.2.2 Task Board . . . 3
3.2.3 Continuous Integration and Deployment . . . 4
3.3 Team Meetings . . . 4
3.4 Task Allocation . . . 4

4 Design **5**

5 Implementation **6**
5.1 Calibration . . . 6
5.1.1 Marker position estimation . . . 6
5.1.2 Point cloud transformation . . . 6

1 Executive Summary

The ability to take an image and generate a 3D point cloud in real time, with multiple calibrated cameras, giving both depth and colour information has multiple use cases today. Through Pointify we have developed a system that utilises Microsofts proprietary Xbox Kinect hardware to generate these point clouds based on the XYZ and RGB data from the sensors on the devices. It is a truly cross-platform service which allows the camera clients and the server to run on Windows, OS X or Linux operating systems. We achieve this by using services and programming languages that run on all operating systems. Having multiple calibrated Kinect cameras allows for a full 360 degree view of the object or scene that is being photographed.

The system works by:

Client

- Uses industry standard open source computer vision library OpenCV to perform matrix operations as efficiently as possible.
- Uses open source libfreenect2 drivers from OpenKinect to extract the colour and depth camera data from the camera.
- Uses AruCo calibration as a module for OpenCV which we use to detect the marker within the scene and use it to calibrate the co-ordinate spaces of all cameras.
- Uses open-source C++11 implementation of socket.io client to communicate with the server and send the point cloud data when requested.

Server:

- Based on an AngularJS frontend and a NodeJS backend.
- Run using the Gulp build system to automatically host the server using localhost.
- Uses socket.io to calibrate the clients and synchronise the frames when taking a picture or streaming.
- Uses ThreeJS renderer to display the received point clouds in real-time.
- Can export current picture or video stream as a PLY file or STL file.

For static imaging there are two main use cases. In manufacturing, it is often the case that people wish to create duplicates of objects they already own. For example drill bits or trimmer attachments. Our system can achieve this by taking a picture and then exporting the STL file which can be immediately used to 3D print the object. For medical reasons, the system can create 360 degree images of the human body and then send it off to remote doctors for an online diagnosis.

For live imaging there are also two main use cases. For virtual reality gaming it can be useful to be able to map out the scene and work out the distance of the user from certain objects within their room. This could then be used to create live warnings in the virtual reality headset to alert the user when they are about to come into contact with an object in their gaming space. Also much like google street view our system can be used to create a live 3D view of certain points of interest or general roads which can be live streamed over the internet.

2 Introduction

2.1 Motivation

2.2 Objectives

2.3 Contributions

3 Project Management

3.1 Project Plan

Before we started looking into the specific requirements of this project, we met to discuss how we would go about designing and integrating our code, planning how we would use project management tools, as well as setting specific goals for each two week checkpoint.

3.1.1 Iteration One

We planned to spend the first iteration reviewing two codebases to familiarise ourselves with the problem, an existing partial solution, and the libraries we would use to solve this problem. The existing partial solution is a program called LiveScan3D, a research project from Warsaw University of Technology that performed the calibration and streaming elements of the project we were looking for. Our aim was to improve this system, by using a better calibration mechanism, called ArUco, and stream the scene in a way that can be viewed by anyone on any platform. We planned to investigate ArUco, and it's containing library OpenCV, as well as looking in to ways of efficiently streaming data over a network.

3.1.2 Iteration Two

We planned to start building our solution in the second iteration, specifically building a client application that could read from the Kinect cameras, and a server application to display a point cloud to the user. We planned to set up our project management techniques at the start of this iteration, so they would be ready to use when we started implementing our solution.

3.1.3 Iteration Three

We aimed to be able to stream a point cloud from a single camera to the client in the third iteration. This involved converting the data captured from the sensor into a format we could send across the network, and decoding this on the server to display to the user.

3.1.4 Iteration Four

In the fourth iteration we aimed to implement calibration, allowing us to connect multiple sensors and stream their captured data to the server simultaneously, with the point clouds they capture aligning.

3.1.5 Iteration Five

We planned to dedicate the final iteration to bug fixing and optimisation. There was a high chance that our final solution would have inefficiencies, as this application is performance critical. We wanted to ensure that we could stream the scene at a pleasant framerate with little to no errors or crashes.

3.2 Management techniques

3.2.1 Version Control

When working in a team this size, some form of version control is essential. We chose to use git, hosted on GitHub because of our teams' familiarity with the platform. We tried to follow the Git Flow branching model, as our project involved adding many features, one at a time, so following the "branch per feature" practice seemed logical. We mainly worked on our own separate branches, merging to master before a checkpoint, or when an important feature was complete and working.

3.2.2 Task Board

We found a Kanban-style board very useful to help us manage tasks throughout the project. We chose to use a web platform called Trello for this, as it allowed for all team members to view and edit the board anywhere, as well as providing all the tools we were looking for. We classified each task in to three lists, one labeled ”Queued” for tasks that have been planned but not started, then ”In progress” for tasks that are currently being worked on, and a list for tasks that had been finished, called ”Complete”. We would assign these tasks to team members during our meetings, then update the status of the task as we worked on it. We aimed to keep the amount of ”In progress” tasks as low as possible, focusing on finishing tasks that were semi-completed before moving on to something new. Trello would automatically notify team members when their tasks had been updated, which allowed us to effectively stay updated with the state of the project between meetings.

3.2.3 Continuous Integration and Deployment

As our project involved building a client/server style system, we wanted somewhere that was able to constantly run the server to help when developing and demonstrating the tool. We used the Department of Computing’s CloudStack instance for this, because of it’s easy availability and powerful resources.
We wanted to spend as little time as possible manually running automated tests and following deployment processes, so decided to use a continuous integration and deployment tool to automate this for us. We chose TeamCity as our tool, as it integrated well with our GitHub repository, and would run in the background of our deployment server. TeamCity would listen for pushes to the master branch of the GitHub repository, then try to build the code and run all automated tests over it. If all these steps were successful, and all the tests passed, we would push the changes to the deployment instance, which would be available to anyone in the college. This proved to be very useful when a single member was testing the functionality of the client software, as it required a connection to the server to run, so having a persistent instance of the server available at all times was essential.

3.3 Team Meetings

As our project involved a lot of interaction between different parts of the code base, we had to meet often in order to discuss how this would work. We aimed to hold formal group meetings weekly, to update the team on our progress and allocate tasks for the next week, as well as meet with our supervisor to demonstrate our progress and listen to his advice. The meetings helped us keep track of what each team member was doing, as well as giving us insight into when specific parts of the project would be completed. We also held more frequent meetings during the week to discuss specific tasks that needed collaboration, and sometimes work on important features together, in a pair programming environment to ensure their quality.

3.4 Task Allocation

After we had investigated the task in more detail, and understood exactly what was needed of us, we were in a position to allocate tasks to group members. The main tasks involved building the client system, the server system, working with the ArUco calibration library and ensuring the software could be built cross platform. We informally assigned team members to these tasks, with each member putting most of their focus on to their task. From these general aims, we would break them down into more specific tasks, such as ”Improve frontend playback framerate”. We tried to write our tasks such that each of them would give a real benefit to the end user, and improve their experience with the product.

4 Design

5 Implementation

5.1 Calibration

5.1.1 Marker position estimation

When streaming with more than one camera, we required the point clouds they send to be aligned such that they can be displayed seamlessly on top of each other, building a 3 dimensional representation of the scene. The idea behind this involves applying a transformation to each point in the scene, to bring it from the camera's perspective to some common perspective shared by all the cameras. We discussed two ways of doing this, one of which chooses one of the cameras as the "master" camera, then synchronises the other cameras around it. The alternative method is by centering all of the cameras on a common point, in our case the calibration marker we use. An advantage of using the first method is that one less camera would need to be calibrated, as all of the other cameras would be centered around the master, but this would lead to more complicated code, and more difficult communication between the cameras. The second approach would give more elegant and maintainable code, as each of the cameras performs the same calibration, but sacrificing a small amount of efficiency on one of the cameras. For these reasons, we chose to use the second approach, calibrating all cameras around the marker placed in the scene.
This involved working with a third party library called ArUco, which is a well established computer vision library for c++ that helps with the calibration process. In the method we chose, the cameras are all centered around a stationary marker placed in the scene, so we required ArUco to detect these markers and calculate the camera's position relative to the marker. The aim is to find the transformation that brings points centered around the camera, into a space with the marker as the origin, then once this is calculated we can apply this transformation to each point cloud to align them. We start this process by detecting any markers visible in the scene, then finding rotation and translation vectors from the camera to this marker, adjusting for intrinsic camera parameters such as focal length and principal point offset. From this we can build up a transformation matrix to combine both the rotation and translation, before inverting it to get the translation from camera to marker as required.

5.1.2 Point cloud transformation

Once we have found the transformation matrix as in the previous part, we must apply it to each point cloud before they are is sent to the server. We experimented with doing this by multiplying each xyz point by the transformation matrix to give the transformed point, but this was causing some performance issues due to the amount of points needed to be transformed at once. A better way of doing this would be to apply the transformation to all points in one operation with a more complex matrix operation. We used an optimised implementation of this in opencv called perspectiveTransform to do this efficiently, which when given the set of input points, and the transformation function we calculated earlier, would return the transformed points we needed. When each client calculated the transformation to bring their origin to the marker, and applied this transformation to the point clouds captured from the connect, they could be successfully combined such that they would align.