Imperial College London

Department of Computing

3rd Year Group Project

Reconstruction of non-rigid objects from RGBD data

Authors:
Riku Murai
Alberto Spina
Matthew Brookes
Daniel Boulby
Thomas Bower
Alessandro Bonardi

Supervisor:
Dr. Bernhard Kainz

8th January, 2018

Acknowledgements

Many thanks to Daniel Grzech who gave much time and code to this project.

Contents

1 Executive Summary

DynamicFusion is a dense Simultaneous Localization and Mapping (SLAM) system capable of reconstructing deforming scenes. While implementations already exist[25], none are open-source, or publicly available. We are creating an implementation of the DynamicFusion algorithm which uses open source dependencies and can be run on custom data produced from an RGBD camera.

Over the last decade remarkable attempts have been made towards solving the SLAM problem, due to a new focus on efficient SLAM solvers and the availability of more efficient hardware for computation and sensing allowing the introduction of dense, large scale, and object-based SLAM algorithms[8]. These algorithms, however, are limited to the reconstruction of static scenes, failing to correctly recognise changes in the captured world and producing an incorrect map — in other words, existing implementations assume that the world is fixed and cannot change. DynamicFusion is the first dense SLAM system which can reconstruct and track dynamic and non-rigid scenes in real-time.

Our new proposed implementation of DynamicFusion allows the user to access and visualise the reconstructed scene directly from a live or offline recorded stream. It is expected to gain interest from various different sectors, as the currently only existing, functional implementation of the problem is proprietary. In particular, applications in:

- **Medicine** will help to diagnose early neurological and muscular disease in pre-term infants and children by reconstructing and modelling their movements and actions.
- **Robotics** will improve gesture recognition and interaction with unknown objects.
- **Machine Learning** has the potential to improve emotion classification.
- **VR Games and Communication** will allow a more immersive, realistic experience and better interaction between users.

2 Introduction

2.1 Motivation

DynamicFusion has become one of the most influential SLAM algorithms, winning Best Paper Award at CVPR15. This is because of the impact it can have in various areas, including robotics, medicine, virtual reality and machine learning, where the SLAM approaches currently used require a still scene or are subject to be captured without errors and reconstructed accurately.

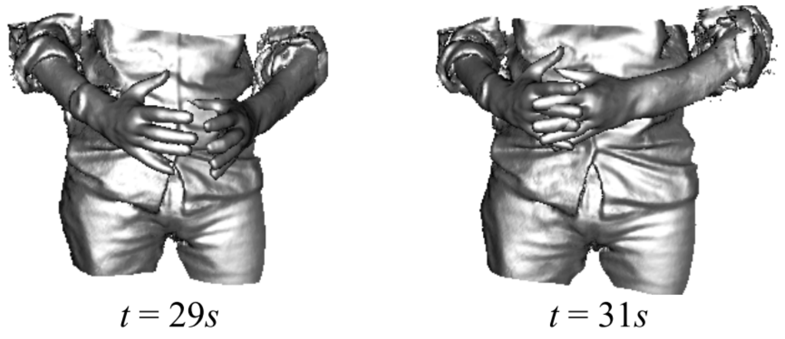


Figure 1: Two frames from the crossing finger reconstruction demo in the paper[25]

The lack of an open source implementation has led to many attempts from amateur programmers to produce one: as of now, to our knowledge, no code on *GitHub* successfully implements the algorithm; in addition companies interested in the features of DynamicFusion began looking into producing their own proprietary version or looking for engineers and researchers to achieve it for them. The interest in new implementations of DynamicFusion can easily be seen on the Internet in comments related to the original paper and *GitHub* repositories, in proposals for *Google SoC* projects, and even in job advertisements placed on forums[31].

2.2 Objectives

- Implement the DynamicFusion algorithm to run on a live or pre-recorded input stream.
- Use Open Source dependencies so that the released product is also Open Source.
- Produce a *Docker* container which can run the algorithm on a custom data source.
- Make the algorithm runnable on a range of hardware and operating systems.

2.3 Achievements

- Implemented DynamicFusion to run on around 20 frames in a reasonable time.
- Correctly localised and tracked an object's movement, computing the parameters of its motion.

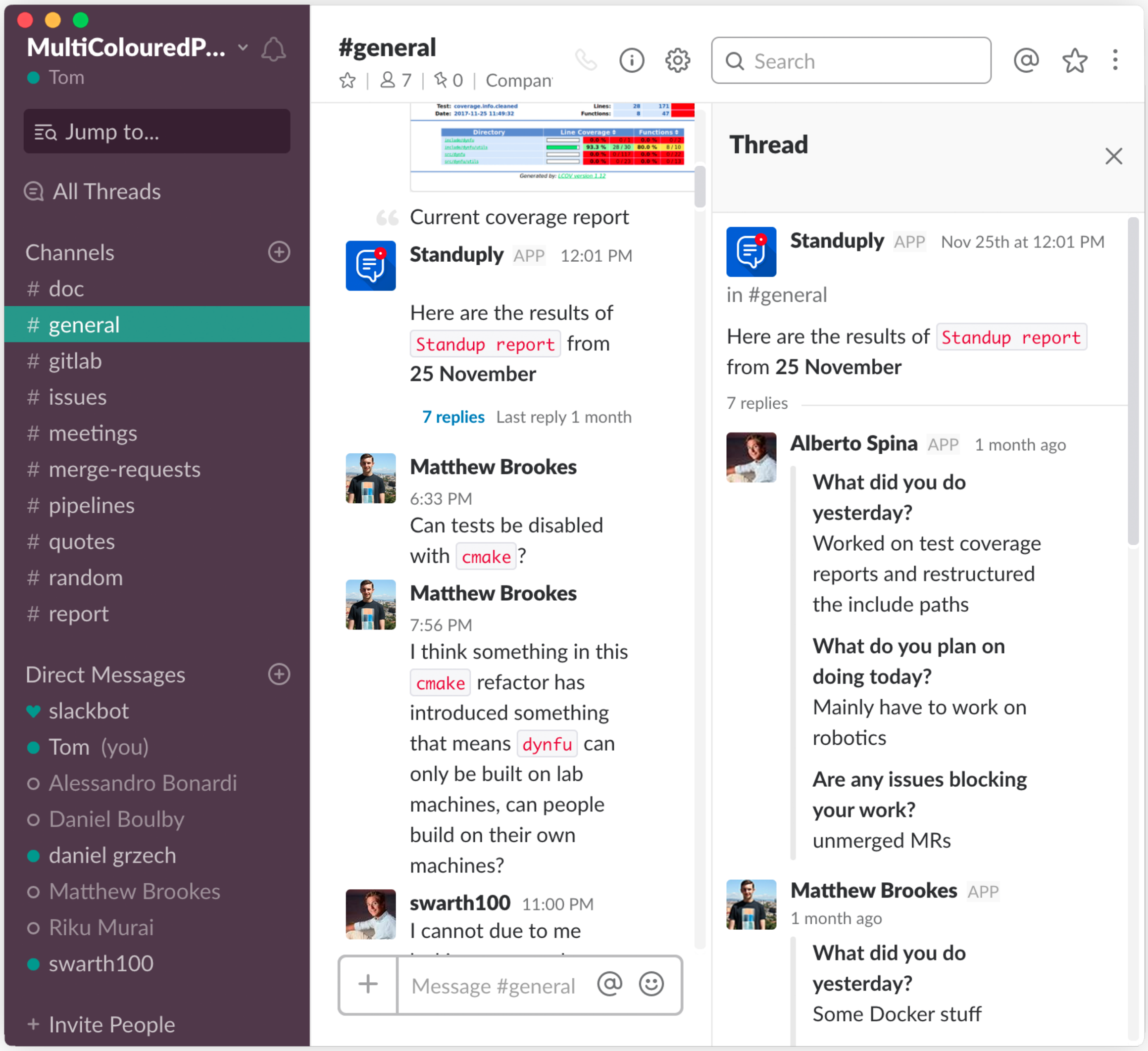
- Pipelined all the components of the project, in order to get the output reconstruction from the live camera input or PNG images.
- Dependencies are all released under Open Source licences with the exception of *CUDA*.
- Published a *Docker* container to *Docker Hub* which runs DynamicFusion on RGBD data.
- Created a visualiser to display the output.
- Documented build steps so that `dynfu` can be built in different environments.

3 Project Management

3.1 Project Organisation

3.1.1 Methodology

We decided to use the Agile methodology for this project because it is the methodology most of the group had familiarity with from both previous projects and industrial experience. In particular, we decided to use the *Scrum* paradigm which involves having *sprints* with clearly defined goals of what to achieve within a given timeframe. This tied in well with the Checkpoint format of the project, and allowed us to meet with the product owner – our supervisor – at the end of every sprint for feedback and comments.



Current coverage report

Standuply APP 12:01 PM

Here are the results of `Standup report` from **25 November**

7 replies Last reply 1 month

Matthew Brookes 6:33 PM
Can tests be disabled with `cmake`?

Matthew Brookes 7:56 PM
I think something in this `cmake` refactor has introduced something that means `dynfu` can only be built on lab machines, can people build on their own machines?

swarth100 11:00 PM
I cannot due to me

Standuply APP Nov 25th at 12:01 PM
in #general
Here are the results of `Standup report` from **25 November**
7 replies

Alberto Spina APP 1 month ago
What did you do yesterday?
Worked on test coverage reports and restructured the include paths
What do you plan on doing today?
Mainly have to work on robotics
Are any issues blocking your work?
unmerged MRs

Matthew Brookes APP 1 month ago
What did you do yesterday?
Some Docker stuff

Figure 2: *Slack* with *Standuply* used for daily standups

When meeting our product owner, we could confirm if we had met our goals for the last sprint, and reassess our goals for the next two-week sprint. On a more short term basis, we held daily standups online (Figure 2) where we had to answer three questions:

- What did you do yesterday?
- What do you plan to do today?
- Are any issues blocking your work?

These questions allowed us to keep track of the progress of the team as a whole, and address any individual problems or roadblocks that could have impeded us meeting our sprint objectives.

3.1.2 Planning

In order to decide which direction to take with our project, we put together a *Method and Plans* document at the beginning summarising how we would work together as a group and organise ourselves. Within the document we decided which project management methodology we would use and wrote down a list of preliminary aims for each of our four sprints following a meeting with our supervisor (reproduced below):

Checkpoint 1
We will aim to configure our system to be able to:

- Record RGBD video stream from the camera.
- Convert the video stream data to a *PCL* compatible format.
- Use an existing SLAM reconstruction framework to process (and render) the RGBD data.

Checkpoint 2
We will aim to extend the existing frameworks to support estimation of the volu-metric model-to-frame warp field parameters.

- Implement dual quaternion blending (DQB) to define the dense non-rigid warp function that transforms the frame-depth maps into the canonical model.
- Given a frame depth map, determine the values to assign to the warp function to obtain a model consistent with the canonical one.

Checkpoint 3
We will aim to achieve fusion of the live frame depth map into the canonical space via the estimated warp field.

- Merge the warped live frame depth maps with the canonical model.
- Use the new data to update the points captured originally for the canonical model.

Checkpoint 4
We will aim for adapting the warp-field structure to capture newly added geometry.

- Extend the warp field function to accommodate increases in the size of the canonical model.
- Insert newly mapped non-rigid warped nodes into the canonical model, extending it.

We expected these aims to change over time and did not treat them as an absolute, but they helped guide us through the project and gave us specific things to work towards.

We also analysed and anticipated potential risks which could cause problems during the project for example issues associated with working with unfamiliar software, as none of us had ever done any work with the camera or indeed any of the existing similar implementations, and compatibility among the various devices used by the group as our group comprised *Mac*, *Windows* and *Linux* machines.

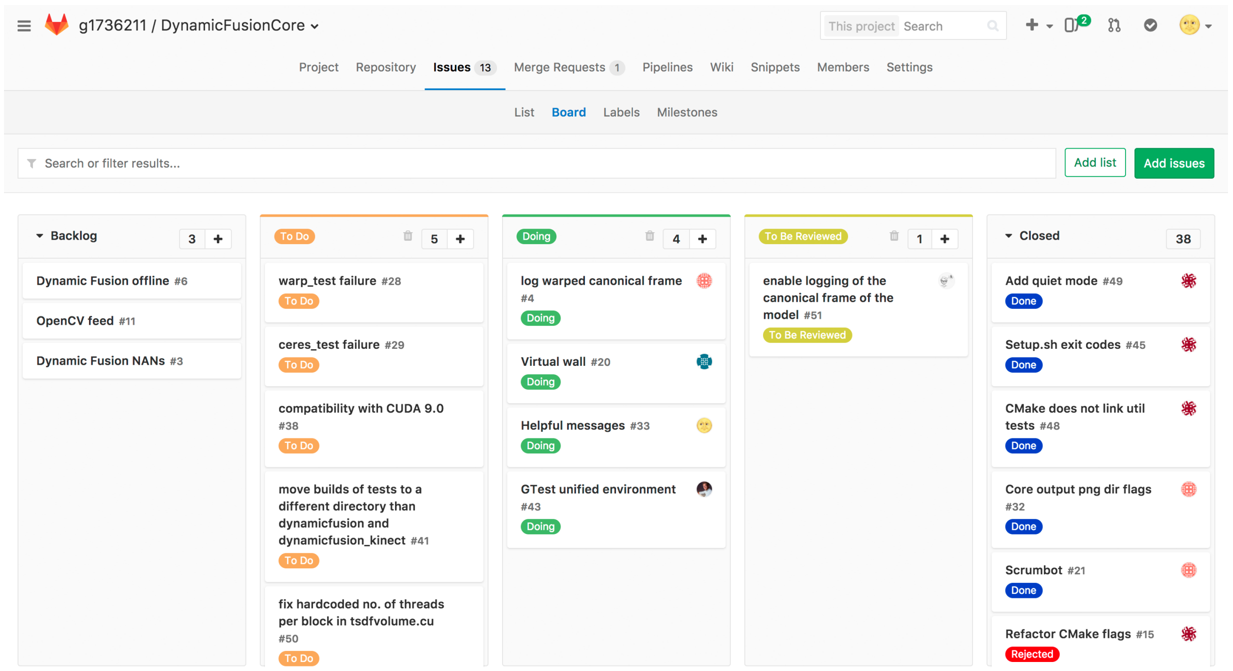
3.1.3 Organisation

By adhering to *Scrum*, it became clear that to work effectively as a group, we would need to take ownership of individual tasks to meet our expectations for each deadline. In order to do this without duplicating work or leaving somebody with nothing to do, which would be a gross waste of our limited resources, we made heavy use of *GitLab Issues* (Figure 3) which is an advanced and highly functional issue tracker built into *GitLab*. This was especially convenient because we used *GitLab* to store our source code and act as our version control system. Everything was intimately linked; commit messages, issues and merge requests could all be cross-referenced. We chose to use this over similar products such as *Trello* because of the advantages of the integration and not introducing yet another tool to check regularly.

The issue tracker features a board-like interface as well as a more traditional list-like interface, which sorts cards (or issues) into separate boards which can then be moved around. We introduced a series of five different tags to help us manage different tasks:

1. **Backlog:** Items which somebody will need to tackle eventually, but are not feasible, necessary, or important at the current time.
2. **To-Do:** Items which somebody needs to do, but currently not taken by anybody.
3. **Doing:** Items which somebody is working on currently.
4. **To Be Reviewed:** Items which have been completed but need to be checked by two other group members before they can be merged.
5. **Closed:** Items which have been completed, reviewed, and merged into the main branch of the project.

Each issue can be claimed by a member of the group and then completed - the tracker even allows to mark issues by their associated checkpoint, fitting in nicely with our agile methodology. Because we had a number of repositories for various aspects of our project, each repository featured its own set of issues and a unique board.



Backlog 3
- Dynamic Fusion offline #6
- OpenCV feed #11
- Dynamic Fusion NANs #3

To Do 5
- warp_test failure #28 — To Do
- ceres_test failure #29 — To Do
- compatibility with CUDA 9.0 #38 — To Do
- move builds of tests to a different directory than dynamicfusion and dynamicfusion_kinect #41 — To Do
- fix hardcoded no. of threads per block in tsdfvolume.cu #50 — To Do

Doing 4
- log warped canonical frame #4 — Doing
- Virtual wall #20 — Doing
- Helpful messages #33 — Doing
- GTest unified environment #43 — Doing

To Be Reviewed 1
- enable logging of the canonical frame of the model #51 — To Be Reviewed

Closed 38
- Add quiet mode #49 — Done
- Setup.sh exit codes #45 — Done
- CMake does not link util tests #48 — Done
- Core output png dir flags #32 — Done
- Scrumbot #21 — Done
- Refactor CMake flags #15 — Rejected

Figure 3: *GitLab Issues* allowed us to work effectively

3.2 Team Management

3.2.1 Allocation of Tasks

As mentioned earlier, tasks were assigned on a mostly first come, first served basis using *GitLab Issues*, however we generally stuck to working on areas that we were most comfortable and experienced in. We gradually split up into different areas:

- **`dynfu` Algorithm:** Alberto, Riku, Daniel, Alessandro.
- **Testing, Output, and Deployment:** Thomas, Matthew.

This worked well because we could all work on the areas that we knew the most about which allowed us to work more efficiently as a team. That being said, the entire team still worked on areas of particular importance and tasks were taken based on both length and difficulty to ensure work was split evenly.

3.2.2 Communication

As a group, we unanimously decided on using *Slack* as a means of communication for every aspect of our project. We set up our *Slack* team at the very beginning of the project and conducted all discussions through the app. We decided to keep all discussion within *Slack* as opposed to using alternative communication means such as *Facebook Messenger* so that all information was accessible by every member of the project, avoiding miscommunication, duplicated work, or any confusion.

We took full advantage of many of the features offered by *Slack*, including integrations:

- ***Standuply:*** Scrum integration to perform daily standups (Figure 2).

- ***GitLab:*** Integration to receive notifications of commits, updates to issues, pipeline failures, and other aspects of the *GitLab* workflow.

Initially, we also tried to use integrations such as *Kyber* (a task management and calendar system) and *Scrumbot* (similar to *Standuply*) but we quickly realised we didn't need many of the features offered, or other integrations did the same task better, so later removed them.

Within *Slack*, we utilised many different channels including *#doc* to provide important documentation (such as guidelines for commits and branching for consistency) and a record of our past submissions, *#meetings* to co-ordinate meetings with our supervisors and amongst ourselves, *#general* for general discussions about our project, and a variety of channels to get updates from *GitLab* such as *#issues*, *#pipelines*, and *#merge-requests*.

In order to organise meetings, we elected a member of our group to send emails to our supervisor and find a suitable date. We agreed on an appropriate time using a *#meeting* channel on *Slack*. We timed our meetings with Dr. Bernhard Kainz to coincide with the end of each checkpoint so that we could discuss our progress and get advice and comments for future checkpoints.

4 Background

4.1 Application Overview

dynfu is a C++ implementation of the DynamicFusion Paper[25]. The current implementation is mainly split into five different components: an input pre-processor, the actual implementation of the algorithm, an output handler, a web interface for testing and a *Docker* container for deployment.

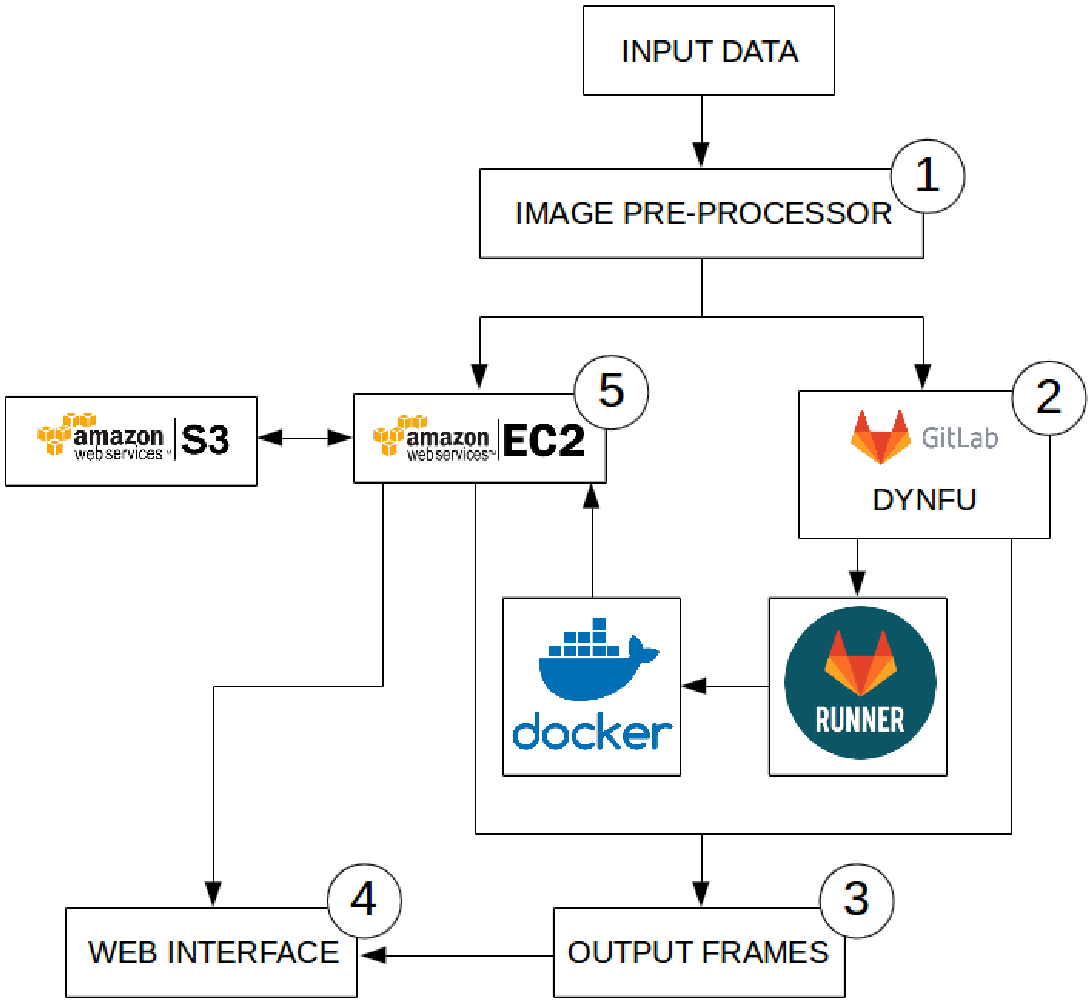


Figure 4: dynfu Overview

1. The **input pre-processor** is necessary to package the depth data we receive from multiple sources into the correct format and structure required by the algorithm.

2. The **dynfu algorithm** is the core aspect of this project, it performs the DynamicFusion algorithm, as specified in the paper[25], over the input data and generates an accurate canonical frame for the scene. This canonical frame is then fed to the output processor.

3. A number of **output frames** are produced by the algorithm (Canonical, Live and Warped frames). They are all logged into PCL format for storage.

4. The **web interface** (see Section 7.1.4), hosted in an *Amazon S3 Bucket*, combines the three previous components of `dynfu` together, as seen in Figure 4. Using the web interface, custom input data can be run on the latest deployed version of the algorithm and the output can then be compared with that of previous versions stored in *S3*.

5. The **testing and deployment** component (see Section 7.1) links the whole project together. All components of `dynfu` are unit tested on a *Gitlab Runner*. If successful then a `dynfu` *Docker* container is built and the project is deployed to *Docker Hub*. An *Amazon EC2* instance pulls the container and runs the `dynfu` algorithm on test data.

4.2 Input Pre-Processor

The input pre-processor allows us to record a scene and save it in the format required by the `dynfu` algorithm. By separating the input pre-processing from the main algorithm we remove the dependency between the `dynfu` algorithm and the camera hardware. The camera hardware can easily be changed by writing a custom input pre-processor for the relevant device.

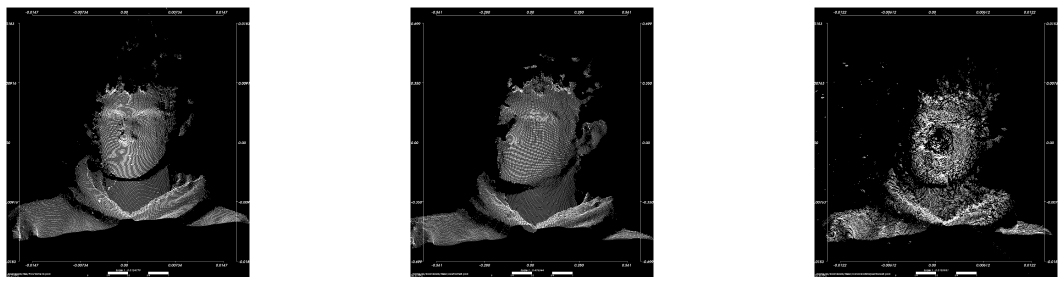
4.3 `dynfu` algorithm

The `dynfu` algorithm is the core aspect of the project, which aims to perform dynamic reconstruction of non-solid objects. For this reason it is analysed in detail within Section 6. Multiple background definitions can also be found within Section 5.

4.4 Output Frames

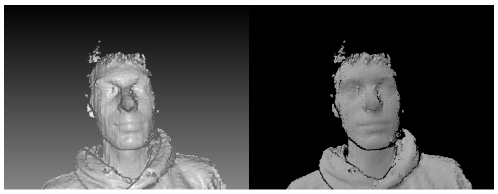
The output `dynfu` produces is mainly of two distinct types: PCL models and PNG images.

1. **PCL models**. `dynfu` outputs PCL models for the *canonical model* (Figure 5a), the *live frame* (Figure 5b) and the *canonical model* warped to the *live frame* (Figure 5c).



(a) Canonical model at t_9 (b) Live Frame at t_9 (c) Warped frame at t_9

Figure 5: PCL output from `dynfu`


(a) Canonical model at t_9

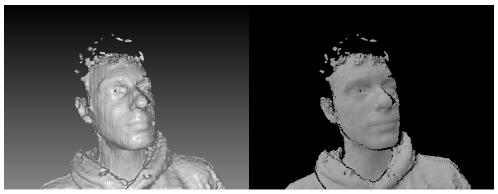

(b) Live Frame at t_9

Figure 6: PNG output from `dynfu`

2. **PNG images**. `dynfu` outputs PNG images for the raycasted *live frame* (Figure 6b) and the updated *canonical model* (Figure 6a).

Examples of `dynfu`'s output can be found in **Appendix A**.

4.5 Project Iterations

Over the course of the various checkpoints for the project, we performed numerous substantial changes to `dynfu`'s implementation, most of which were related to the library we were using as a foundation to the project.

1. **DynamicFusion Library**. We initially worked on an open source work-in-progress implementation of DynamicFusion[7]. However we soon realised this was a mistake. Even though the actual project was a fantastic experience to get us accustomed to what a hands on implementation of the paper[25] would look like, we unfortunately realised we needed complete knowledge and control over the code base. We do realise that implementing most of the functionalities from scratch is a lot more time consuming than building on top of someone else's implementation, but we just felt we had to get the basics right before dwelling into GPU optimised code.

2. **Kinetic Fusion Library**. Currently our implementation is heavily based on an open source implementation of Kinetic Fusion[17], *kinfu_remake*[3].

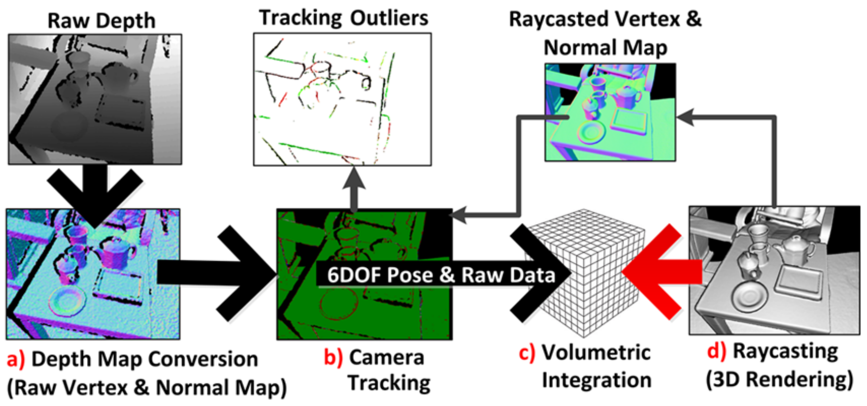

Raw Depth
Tracking Outliers
Raycasted Vertex & Normal Map
6DOF Pose & Raw Data
a) Depth Map Conversion (Raw Vertex & Normal Map)
b) Camera Tracking
c) Volumetric Integration
d) Raycasting (3D Rendering)

Figure 7: Kinetic Fusion pipeline diagram

Our project currently extends the kinetic fusion implementation (which can be seen in Figure 7) to handle non-rigid object using the steps outlined in Section 6.

5 Technical Overview

5.1 Dual Quaternions

In order to represent, in the most simple way, rotations around any given "axis" and translations, we decided not to stick to the more traditional rotational matrices and, instead, use quaternions. In particular we used dual quaternions, which "can be used to represent spatial rigid body displacements"[34].

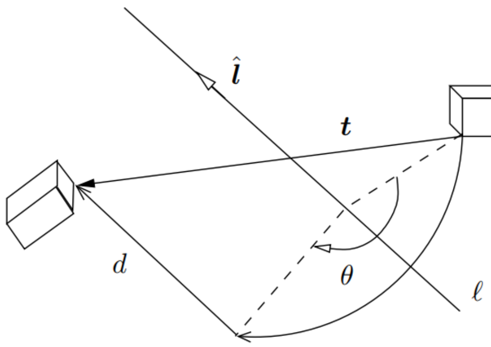


Figure 8: Dual Quaternion with rotation (l) and translation (d) visualized[18]

A dual quaternion transformation **t** (Figure 8) encapsulates a translation (d), which can be expressed by a vector `v = (x, y, z)`, and a rotation (l) over a given axis ($\hat{l}$). The rotation component is itself a quaternion, which is conventionally expressed as:

$$a + bi + cj + dk, \quad \text{where: } i^2 = j^2 = k^2 = ijk = -1$$

Dual quaternions, in particular, consist of eight elements which belong to two quaternions. These two quaternions which make up the dual quaternion (q) are called the real part (q_r) and the dual part (q_d). Where:

$$q = q_r + \epsilon q_d$$

And given a rotation l and a translation d both expressed by quaternions:

$$q_r = l \qquad q_d = \frac{1}{2}dl$$

Dual quaternions allow us to apply transformations in 6 Degrees of Freedom. They allow easy scaling of the transformation as well as supporting addition and multiplication of transformations. All of this allows us to implement Dual Quaternion Skinning, which will be discussed in Section 5.5.

5.2 Warp-field

The *warp-field* is, most simply, a collection of sparsely sampled *deformation nodes* (Section 5.3) which are necessary to compute the warp function (implementation details in Section 5.6).

5.3 Deformation Nodes

A *deformation node* $\mathcal{N}^t$ (at time t for a given *warp-field* `warp`) is defined by a set of three distinct parameters: its position, its radial basis weight and its transformation.

$$\mathcal{N}^t_{\texttt{warp}} = \{\mathbf{dg}_v, \mathbf{dg}_w, \mathbf{dg}_{se3}\}_t$$

- $\mathbf{dg}_v$: is a **position vector** which holds the `(x, y, x)` coordinates of the deformation node. In our implementation we decided to use `cv::Vec3f`, which is a 3D vector representation with floating point precision present within OpenCV[29].

- $\mathbf{dg}_w$: is the **radial basis weight**; a floating point value held by each *deformation node* that expresses the "influence" over neighbouring nodes. The radial basis weight is initialised to `2.0f` and is then updated for every new captured *live frame* (implementation details in Section 6.5).

- $\mathbf{dg}_{se3}$: is the **transformation** associated with a given *deformation node*. It is internally represented via a `DualQuaternion` instance with an associated translation and rotation component. The transformation should, thus, encapsulate the "movement" vectors associated with the *canonical frame* should perform to reach their targets in a given *live frame* at time t. Just like the radial basis weight, the transformation component is updated for every new captured *live frame*.

As hinted in the description, the inner fields of deformation nodes will be accessed during Dual Quaternion Blending (Section 5.5) and will be updated, for every newly captured *live frame*, during warp-field estimation stage (Section 6.5).

5.4 KD-Tree

A KD-tree (Figure 9) is a data structure optimised for efficient closest neighbour search. Within `dynfu`, KD-trees will prove especially useful for quick and fast nearest-neighbour searches. It is, in fact, with this information that we can effectively calculate the distance between the position ($\mathbf{dg}_v$) of each of the *deformation nodes* and an input vertex, when performing DQB (see Section 5.5).

5.5 Dual Quaternion Blending

Dual Quaternion Blending[19] is a core and fundamental aspect of `dynfu`, as it allows us to use the information stored in the *deformation nodes* of the *warp-field*, extracting all the information necessary to define the warp function's transformation (see Section 5.6). DQB, thus, solves the problem of being able to compute a transformation which is able to warp all the points present within the *canonical model* (see Section 6.3) into their respective targets in the *live frame* (see Section 6.2).

A naive approach to performing the warping would be, in fact, to approximate the transformation for each point x_c of the *canonical model*. However, even for a low resolution of 256^3 voxels we would need to estimate more than 100 million transformation variables. Instead, since in the real world the surface of the model will normally move smoothly, we create a *warp-field* full of deformation nodes, calculate the parameters for these nodes and

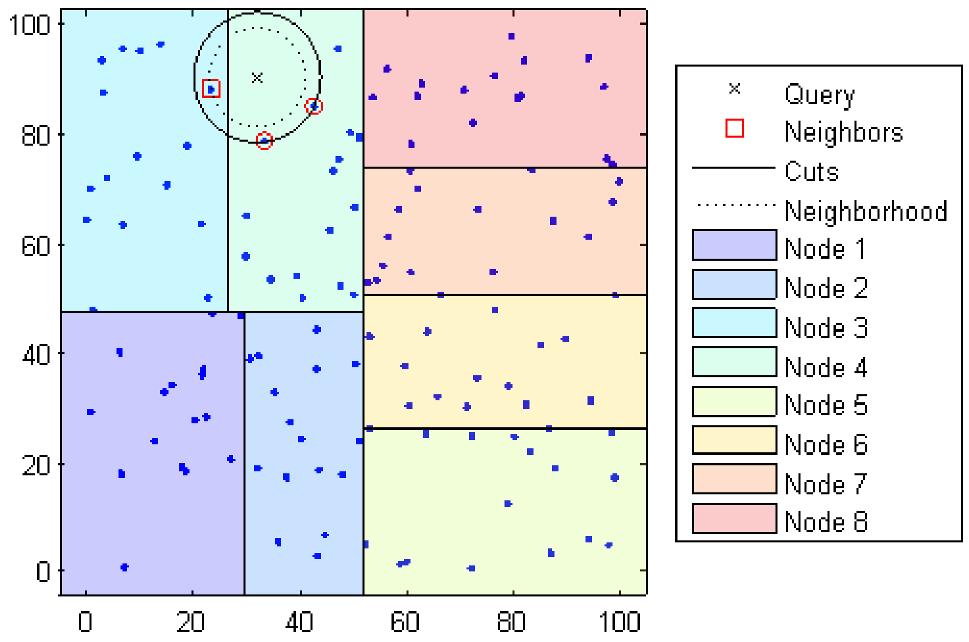


Figure 9: Visualisation of Nearest Neighbour Search within a KD-Tree[22]

apply the transformations (more information on warp-field initialisation can be found in Section 6.5.1).

Therefore given a *warp-field* $\mathcal{W}_t$ at time t and a *canonical model* $\mathcal{C}_t$ containing x_0 .. x_n vertices, for each $x_c \in \mathcal{C}_t$ we can define $\mathbf{DQB}(x_c)$ as[25]:

$$\mathbf{DQB}(x_c) \equiv \frac{\sum_{k \in N(x_c)} \mathbf{w}_k(x_c)\hat{q}_{kc}}{\| \sum_{k \in N(x_c)} \mathbf{w}_k(x_c)\hat{q}_{kc}\|}$$

- $N(x_c)$: is the set of the k nearest transformation nodes to the point x_c.
- $\mathbf{w}_k(x_c)$: is the weight (influence) of the *deformation node* k on the point x_c calculated by the equation again given in the DynamicFusion paper[25]:

$$\mathbf{w}_i(x_c) = exp\left(\frac{-\|\mathbf{dg}_v^i - x_c\|^2}{2(\mathbf{dg}_w^i)^2}\right) \tag{1}$$

  - $\mathbf{dg}_v^i$: is the position vector which holds the `(x, y, x)` coordinates of the *deformation node* i.
  - $\mathbf{dg}_w^i$ is: the radial basis weight of *deformation node* i.
- $\hat{q}_{kc}$: is the dual quaternion associated to the transformation ($\mathbf{dg}_{se3}$) of the kth nearest *deformation node* to x_c.

Therefore dual quaternion blending can be seen as a weighted average of the dual quaternion transformations associated to the k nearest neighbours of a given vertex x_c. Because the weight $\mathbf{w}(x_c)$ associated to each transformation ($\mathbf{dg}_{se3}$) is proportional to both the position ($\mathbf{dg}_v$) and the radial basis weight ($\mathbf{dg}_w$) of the *deformation node*, we can see how *deformation nodes* which are either further away from the vertex x_c or have lower radial basis weight, will play a much lower impact when calculating the dual quaternion associated to x_c.

84.9 FPS
197.4 FPS

(a) Log-matrix Blending (b) Dual Quaternions

Figure 10: A comparison of different rigid transformation blending algorithms[19]

Furthermore, as can be seen in Figure 10, amongst all rigid transformation blending algorithms, "blending based on dual quaternions not only eliminates artifacts, but is also much easier to implement and more than twice as fast as previous methods."[19].

5.6 Warp function

The warp function $\mathcal{W}(x_c)$ is a rigid body transformation that is able to transform all vertices $(x_0 \, .. \, x_n)$ of the canonical model $\mathcal{C}$ into those of the live frame. This transformation is able to take into account compression and expansion of space by moving neighbouring vertices in converging (or diverging) directions. More generally the warp function $\mathcal{W}(x_c)$ can be defined as[25]:

$$\mathcal{W}(x_c) \equiv SE3(\mathbf{DQB}(x_c))$$

Having seen how to compute $\mathbf{DQB}(x_c)$ in Section 5.5, to obtain the warp function $\mathcal{W}$ for vertex x_c we must convert the dual quaternions of **DQB** back to a transformation matrix. To achieve this it is important to remember that "any rigid transformation in 3D can be described by means of a 4 × 4 matrix P with the following structure"[4]:

$$P = \left[\begin{array}{ccc|c} & & & x \\ & \mathbf{R} & & y \\ & & & z \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

Where **R** is the 3-dimensional rotation matrix and the vector composed of (x, y, z) is a 3D translation.

All of this information, though, is already present within the dual quaternion q of $\mathbf{DQB}(x_c)$. If, in fact, we consider the rotation l and the translation d to both be expressed by quaternions:

- The **translation vector** (x, y, z) can be extracted from the dual component q_d of q. Knowing that:

$$q_d = \frac{1}{2} q_r \times d \quad \rightarrow \quad d = \frac{2 \times q_d}{q_r}$$

(x, y, z) are the values associated with the i, j, k components of the translation quaternion d.

- The **rotation matrix R** can be computed directly from the normalised real component q_r of the dual quaternion q as follows:

$$\mathbf{R} = \begin{bmatrix} 1-2(c^2+d^2) & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & 1-2(b^2+d^2) & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & 1-2(b^2+c^2) \end{bmatrix}$$

  Where $q_r = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$.

After we compute the $SE3$ transformation matrix that corresponds to dual quaternion given by $\mathbf{DQB}(x_c)$, for every point $x_c \in$ *canonical model* $\mathcal{C}$, we can then apply said transformation to the point x_c and achieve the results outlined in Figure 16.

5.7 Surface Fusion

Once we have defined the warp function, we can compute its inverse and apply it to the current *live frame*. Once we have warped the *live frame* into the *canonical model* we can then fuse the two together in order to update the canonical model with any newly captured geometry. But given the depth map, we noticed that projecting the depth straight to TSDF voxels will not fuse properly since the volume is not static.

In order to compensate for the non-rigid movement, we thus need to warp the centre of all the TSDF voxels to the current *live frame*'s space using the warp function which we have calculated. Similarly to *Kinect Fusion*'s implementation[17], we store two values per voxel: a distance to the surface and weight which encodes not only the uncertainty in depth (like in *Kinect Fusion*), but also the uncertainty of the warp function at that voxel. The weights are then scaled according to the distance to the k neighbouring *deformation nodes*.

6 Implementation

The core `dynfu` algorithm is fed PNG frames containing depth information encoded in 16-bit grayscale format and it subsequently performs the following operations in order:

1. Warp the *canonical model* into the *live frame* using the previously estimated warp field parameters (see Section 6.4).
2. Estimate new *warp-field* parameters given the previous *warp-field*, the warped *canonical model* and the new *live frame* (see Section 6.5).
3. Extend the *warp-field* to allow it to capture any newly added geometry (Section 6.6).
4. Given the *live frame* and the warp parameters, perform surface fusion and update the *canonical model*. Note that this part of the algorithm is currently not implemented as discussed in Section 6.7.

All of these will be analysed in detail within the following sections.

6.1 Input Handling

For our project we were given an *Intel Realsense SR300* camera to use for recording the data. Our input pre-processor is designed to work for this model, however it could easily be adapted to work with different hardware. The *Intel Realsense* camera sends a stream of RGBD frames to the computer. We separate this into a stream of RGB frames and a stream of depth frames using *librealsense*[28]. Any processing that we may wish to do on the frames can be done at this point. Further processing was required when working with the initial code base[7] (which can be seen in Section 4.5) as we found that for the algorithm to accept our input it was necessary to add a virtual wall to the image. This meant taking any depth values greater than a threshold distance and setting them to the distance of the virtual wall. We ended up finding that double the average distance of the non-zero pixels was a good distance at which to set the virtual wall. Since moving to writing our own implementation of the DynamicFusion algorithm we have found the virtual wall is no longer required.

Each frame from the RGB and depth streams are then converted to *OpenCV*[29] matrix form and are stored as RGB PNG files and 16 bit depth PNG files respectively; the file name can be specified, however the frame number is always appended to the file name. The file path of the output can be specified when running the program and color and depth folders are created in the specified directory to store the color and depth PNG files. We also use *OpenCV* visualisers to display the colour and depth image at each frame so the user can see frames as they are recorded.

By storing the frames as PNGs, it allows the `dynfu` algorithm to be run offline. This is useful for applications in which a detailed scene is required but it is not necessary to run in real time.

6.1.1 Input Frame

The *input frames* are the 16-bit grayscale depth PNGs generated by the input pre-processor, named `frame-no.depth.png` and present within the `depth` directory.

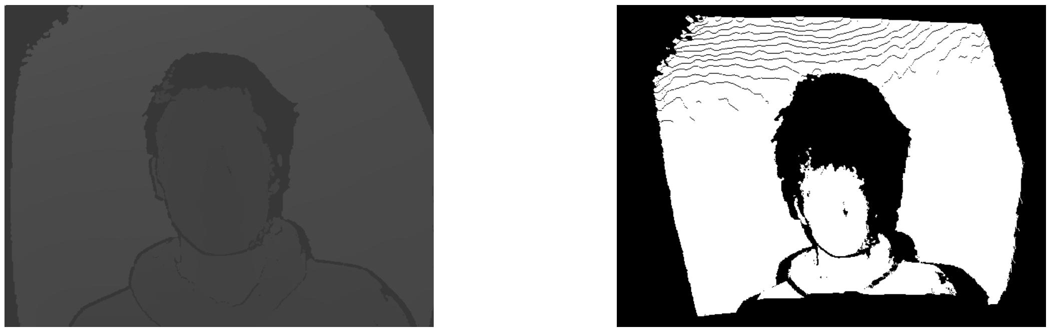

(a) Input Frame in PNG format (b) Correspondent Live Frame in PCL format

Figure 11: Input Frame and its correspondent Live Frame

6.2 Live Frame

Given an *input frame*, we can extract from it the current *live frame*. The *live frame* (as can be seen in Figure 11b) represents the volume, captured by the *input frame*, which we use when performing DynamicFusion.

We have over 60,000 vertices which is hard to reason about if we only have the data in PNG format. Instead, in our implementation, we allow the data to be visualised from the vertices and normals. Since all of the major operations on the point-clouds are done on the GPU, we modify the data on the GPU into two streams of RGB data. One is a normal texture and the other uses the surface of the volume and adds polygon texture, so that we can see the surface of the output volume.

Within `dynfu`, frames are represented via an array of the vectors, and are associated with a unique id:

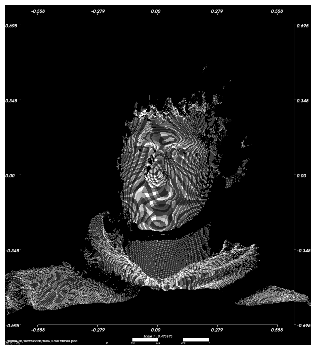
```
Frame(int id, std::vector<cv::Vec3f> vertices)
```

We decided to use `cv::Vec3f` within our implementation, which is a 3D vector representation with floating point precision present within *OpenCV*[29]; it allows us to easily perform vector algebra, normalization and many other supported operations.

6.3 Canonical Model

The *canonical model* is a point-cloud model of the currently reconstructed scene. The *canonical model*, thus, ideally holds a very accurate representation of all the "discovered" geometry of the scene. All *live frames* will continuously be warped (Section 5.2) back to the *canonical model*, updating it with newly sampled geometry.

In our implementation, we initially created the *canonical model* from the depth input of the first *live frame*, after passing it through the bilateral filter. However, this lead to lots of unnecessary points (which are present within the *live frame*) to be represented within the *canonical model*. Certain capture devices, such as the *Intel Realsense* camera detect data up to 1.5m[16], meaning that in most cases it will also capture segments of the background which we don't actually need. Not only would this mean we track motion for a much larger area, but it would also lead to an increase in number of *deformation nodes* we would have to solve for (see Section 5.3), slowing down the energy calculation (Figure 14).


(a) *Live frame* at t_0

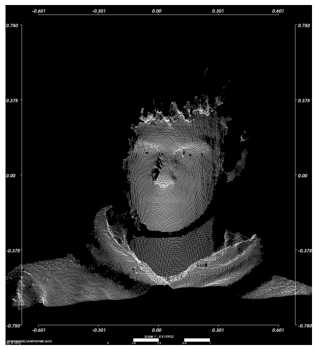
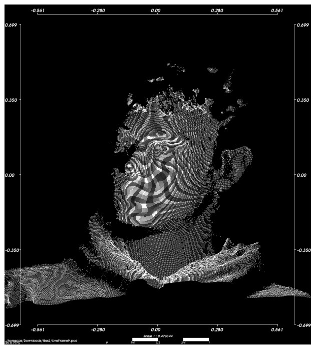

(b) *Canonical model* at t_0

Figure 12: Initialisation of the *canonical model* at time $t = t_0$


(a) *Live frame* at t_9

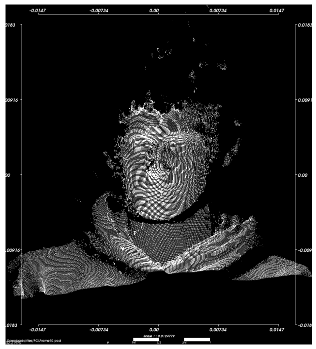
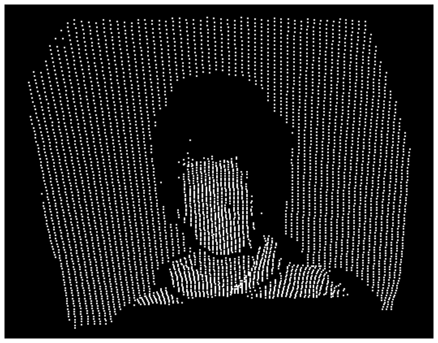

(b) *Canonical model* at t_9

Figure 13: *Canonical model* after time $t = t_9$


(a) Deformation Nodes with the background

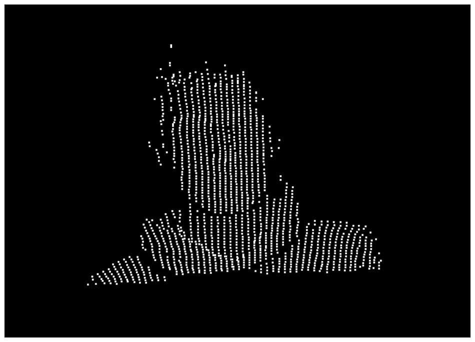
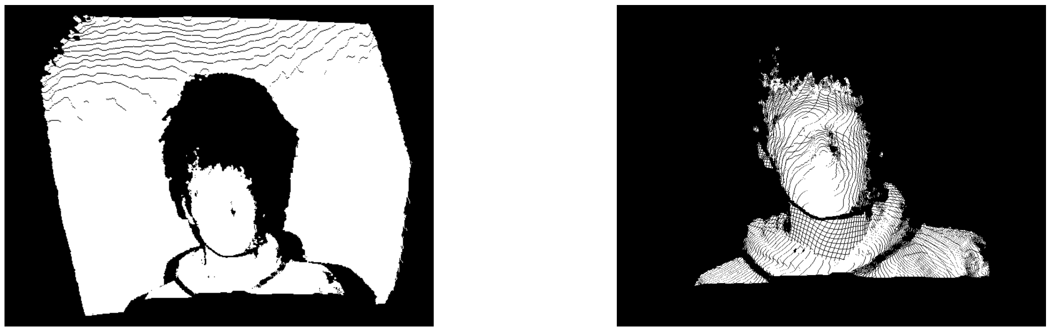

(b) Deformation Nodes without the background

Figure 14: Deformation Nodes corresponding to a given Canonical Model

As for the implementation, we store a frame's model using the Truncated Sign Distance Function (TSDF) for every voxel in the initial space spanned by the first *live frame*. The main difference between TSDF and the voxel grid is that we store, for each voxel within a given space, a signed value representing the distance to the surface, rather than a binary value for each voxel grid. This allows a smoother surface to be extracted using Marching Cubes as suggested within the DynamicFusion paper[25] or raycasting as suggested in *Kinect Fusion*[17]. Since raycasting was already implemented within the original codebase[3], we've decided to use raycasting for our implementation, as can be seen in

Figure 15.



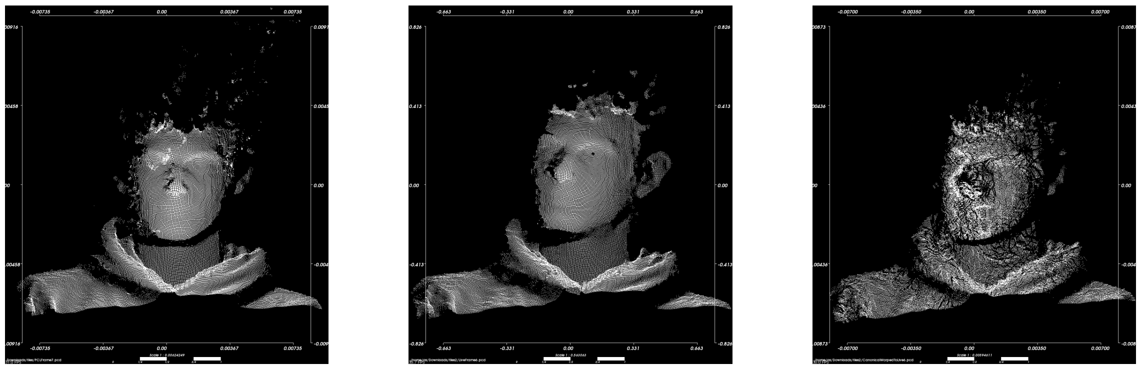
(a) Canonical Model with Background (b) Canonical Model without Background

Figure 15: Canonical Model before and after raycasting

Via raycasting we can extract the points which lay on the surface of our volume. In order to calculate the normals, *Kinect Fusion* simply, for each vertex in the volume, finds two neighbouring vertices, creates two vectors from the vertex to the neighbours and calculates the cross product between the three of them, normalising it. This process only happens to the initial *live frame*, since after we create the *canonical model*, we can ignore the background and the noise from the input using robust Tukey penalty.

6.4 Warping the Canonical Model to the Live Frame

The first step in DynamicFusion consists, at a given frame $t = t_n$, of warping the *canonical model* (last updated for frame t_{n-1}) to the *live frame* corresponding to t_n.



(a) Canonical Model at t_6 (b) Live Frame at t_6 (c) Warped Frame at t_6

Figure 16: Canonical Model Warped to Live Frame at $t = t_6$

A live frame can be warped into the canonical model by applying the warp function (see Section 5.6). It is important to note that there are two possible cases:

- $t = t_0$: we're currently processing the **first frame**. At this point the *canonical model* hasn't yet been initialised. The *live frame* corresponding to t_0 is stored as the *canonical model*, and the warp-field will be initialised from it (see Section 6.5.1). The warp function isn't applied and we can skip to t_1.

- $t = t_n, \quad n > 0$: from the **second frame onwards**, we will apply the warp function calculated from the warp-field corresponding to the previous frame t_{n-1}. We warp the *canonical model* to the *live frame* rather than other way around. This way, we guarantee the upper bound of the *warp-field* estimation to be the size of the *live frame*, which is constant. This differs from the size of canonical model which can potentially increase for every new processed frame.

When applying the warp function (Section 5.6) to every point x_c which belong to the *live frame*, we must:

1. Retrieve the k-nearest *deformation nodes* ($\mathcal{N}_{i_1}$ .. $\mathcal{N}_{i_k}$) to the given point x_c (see Section 6.4.1). Deformation nodes are stored within a *KD-Tree* data structure to allow for faster nearest neighbour searches.
2. Perform Dual Quaternion Blending (DQB) for each point x_c over the k-nearest *deformation nodes* (see Section 5.5). DQB is a means through which we can obtain a weighted average over the dual quaternion transformations ($\mathbf{dg}_{se3}$) held by the neighbouring *deformation nodes.*
3. Given the above steps we can define a warp function $\mathcal{W}(x_c)$ that we can apply to the canonical model in order to obtain the current live frame (see Section 5.6).

6.4.1 KD-Tree

To use the KD-Tree structure defined in Section 5.4 in our implementation, we used the *nanoflann* library[5], a header-file-only implementation of KD-tree for nearest neighbour search. We did consider other alternatives, such as *FLANN* provided by *OpenCV*[29], but we decided to use *nanoflann* instead as it is meant to execute faster than *FLANN* and it uses `size_t`, and not `int`, as the parameter for the node size in its implementation, which is a necessary feature were we to extend DynamicFusion to larger scenes.

In practice, we created an adapter for the *nanoflann* library, which allows us to retrieve the indexes of the k-nearest neighbours to a given input vertex x_c.

```
std::vector<size_t> findNeighborsIndex(int numNeighbor, cv::Vec3f vertex)
```

Internally we unpackage the `cv::Vec3f` data and query *nanoflann* to retrieve the indexes of the `numNeighbor` *deformation nodes* nearest to the input `vertex`.

6.4.2 Dual Quaternions

For `dynfu` we've implemented dual quaternions based on the Boost quaternion library[14] and a detailed guide[20] on how to implement and use dual quaternions. Our dual quaternion class is a header file only and is templated. We focused heavily on making the implementation as general purpose as we could.

Dual Quaternions can be constructed by providing `(x, y, z, roll, pitch, yaw)`, from, either 2 quaternions, or from a pair of a vector and a quaternion. The main mathematical operators are overloaded, allowing the dual quaternion objects to be used as if they were simple vectors in the code. $SE3$ transformation is calculated inside the dual quaternion class as getter functions.

6.4.3 Dual Quaternion Blending

When performing DQB (see Section 5.5) in our implementation we decided to consider the 8 nearest (*deformation node*) neighbours to the vertex x_c in the KD-Tree. We found this value allowed us to achieve reasonable speed for each point whilst not reducing the detail of the warped frame too much.

6.5 Estimating the new warp-field parameters

After warping the *canonical model* into the *live frame* (see Section 6.4) we must estimate the new parameters to assign to each *deformation node* of the *warp-field*. Just like the previous section, it is important to note that there are two possible cases:

- $t = t_0$: we're currently processing the **first frame**, and hence the warp-field hasn't yet been initialised. We must initialise the warp-field (see Section 6.5.1 and then skip to $t = t_1$.
- $t = t_n, \quad n > 0$: from the **second frame onwards**, after we've applied the warp function defined in Section 5.6, we must re-estimate the dual quaternion transformation fields ($\mathbf{dg}_{se3}$) for each of the *deformation nodes* in the *warp-field* $\mathcal{W}_t$ (see Section 6.5.2).

6.5.1 Warp-field initialisation

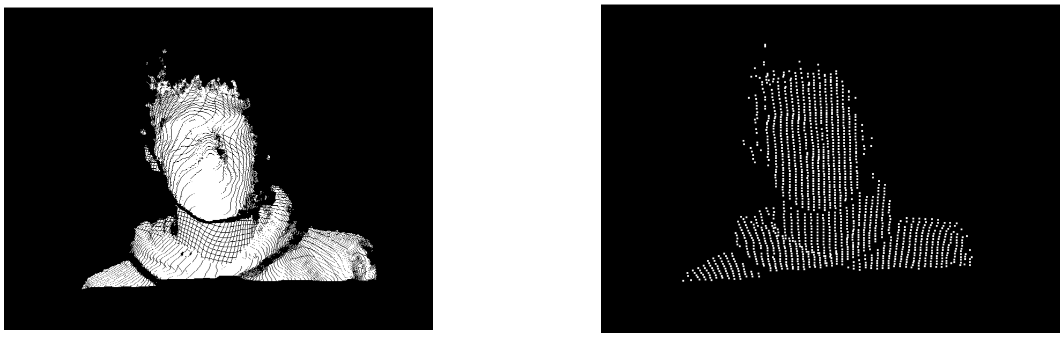
Within `dynfu`'s implementation, the *warp-field* is initialised with a sparsely sampled set of deformation nodes and is then updated for every new captured *live frame*. When at $t = t_0$, hence while processing the first frame, we initialise the *warp-field*. The *warp-field* will be initialised with a set of *deformation nodes*, which, in number, will be around $\frac{1}{50}$ of the total number of vertices in the canonical model. It is important to remember that at $t = t_0$, the *canonical model* is identical to the *live frame*.

In practice (if we recall Section 5.3) for each *deformation node* $\mathcal{N}$ in the *warp-field* $\mathcal{W}$ we will need to initialise the three fields:

$$\mathcal{N}_{\mathtt{warp}}^{t} = \{\mathbf{dg}_v, \mathbf{dg}_w, \mathbf{dg}_{se3}\}_t$$

- $\mathbf{dg}_v$: the position vector of the *deformation nodes* is initialised as a vertex in the *canonical model*. This is done by iterating through all the vertices in the point cloud with an arbitrary number as our step (as can be seen in Figure 17). We decided to use `50` as it was good trade off between speed and accuracy of the end result. To represent the $\mathbf{dg}_v$ field internally, we use a `cv::Vec3f`, which is a 3D vector representation with floating point precision present within OpenCV[29].
- $\mathbf{dg}_w$: the radial basis weight is initialised to `2.0f`.
- $\mathbf{dg}_{se3}$: the transformation associated with a given deformation node is initialised to a 0 dual quaternion. It has a 0-vector translation `(x, y, z)` component and a 0 `(pitch, yaw, roll)` rotation:

```
DualQuaternion<float>(0.f, 0.f, 0.f, 0.f, 0.f, 0.f)
```



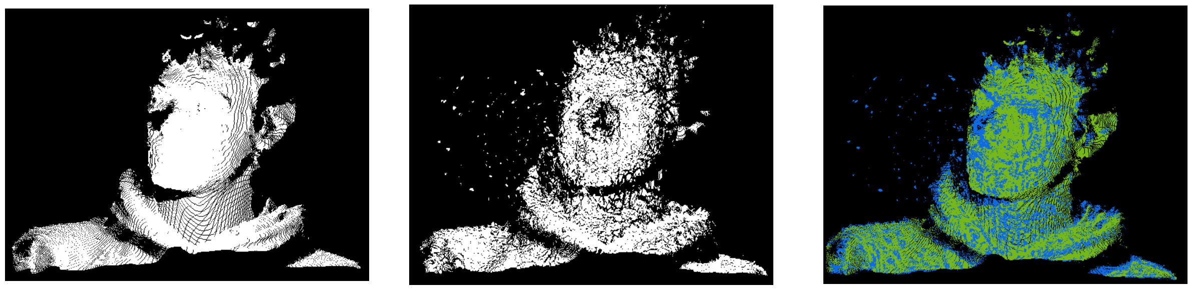
(a) Canonical Model at t_0 (b) Initialised warp-field at t_0

Figure 17: Visualization of *deformation node* positions w.r.t the *canonical model*

It is important to note that the position ($\mathbf{dg}_v$) of a *deformation node* will not change throughout the various iterations of `dynfu`. Newly sampled deformation nodes can be added during the steps outlines in Section 6.6, and during the estimation of the new warp-field state (Section 6.5.2) we will update both the radial basis weight ($\mathbf{dg}_w$) and the dual quaternion transformation ($\mathbf{dg}_{se3}$) with new values estimated by the Solver.

6.5.2 Estimating the warp-field state

Given, at time t, a *live frame* $\mathcal{D}_t$, a *canonical model* $\mathcal{C}$ and a *warp-field* $\mathcal{W}_t$, having already warped $\mathcal{C}$ into $\mathcal{D}_t$ (see Section 6.4), it is now time to estimate which new values should be assigned to the $\mathbf{dg}_w$ and $\mathbf{dg}_{se3}$ fields of the *deformation nodes* in the *warp-field* $\mathcal{W}_t$ to obtain a warped frame which better approximates the live frame.



(a) Live frame $\mathcal{D}_t$ (b) $\mathcal{C}$ warped into $\mathcal{D}_t$ using $\mathcal{W}_t$ (c) Model mismatch

Figure 18: Visualization of the Live Frame and Warped Frame at $t = t_9$

To estimate the $\mathbf{dg}_w$ and $\mathbf{dg}_{se3}$ fields for the *deformation nodes* in the *warp-field* $\mathcal{W}_t$, given the current *live frame* $\mathcal{D}_t$ and the updated *canonical model* $\mathcal{C}$ we use the following energy formula outlined in the DynamicFusion Paper[25]:

$$\mathrm{E}(\mathcal{W}_t, \mathcal{C}, \mathcal{D}_t, \mathcal{E}) = \mathbf{Data}(\mathcal{W}_t, \mathcal{C}, \mathcal{D}_t) + \lambda \mathbf{Reg}(\mathcal{W}_t, \mathcal{E})$$

Where $\mathbf{Data}(\mathcal{W}_t, \mathcal{C}, \mathcal{D}_t)$ is the energy cost and $\mathbf{Reg}(\mathcal{W}_t, \mathcal{E})$ is a term used to penalise non-smooth transformations. Figure 19 offers a visualization of how the energy formula E attempts to estimate better values for the *deformation node* fields of the *warp-field* to obtain a warped frame which better approximates the live frame.

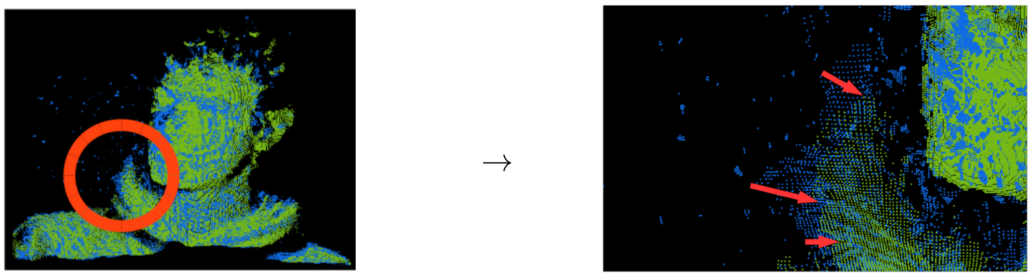

(a) Live frame and warped frame (b) Visualization of $\mathbf{Data}(\mathcal{W}_t, \mathcal{C}, \mathcal{D}_t)$

Figure 19: Visualization of the energy formula E

The actual minimization of the energy cost $\mathbf{Data}(\mathcal{W}_t, \mathcal{C}, \mathcal{D}_t)$ is done via non-linear optimizers:

- In our implementation, we initially used *Ceres solver*[1]. In particular we used the linear `SPARSE_NORMAL_CHOLESKY` solver which allowed Ceres to use all the available cores on the machine. However, since the solver was CPU based, it took us a day to process 9 input frames.

- A much needed speed-up was then found in the *Opt solver*[10]. *Opt* compiles the energy functions into GPU optimized solvers, and it seems like it can compete with application specific solvers. *Opt* tries different combinations of the Unknown parameters (namely $\mathbf{dg}_w$ and $\mathbf{dg}_{se3}$) until it finds the local optimum. Within our implementation we used the *Levenberg-Marquadt* solver option with *Opt*, as, compared to other alternatives such as the *Gauss-Newton* solver, although the LM solver is slower, it is more likely to find a local optimum even if it starts off very far from the minimum. Currently the solver solves one frame per 30 to 40 seconds, which is over 300 times improvement in speed compared to CPU based solver. *Opt* requires us to code in *Terra*[9] and it is compiled at runtime to *Lua*.

- An even more accurate *warp-field* was estimated when we introduced a penalisation term to the energy cost, implemented using the robust Tukey penalty function. In particular we used Tukey's biweight function[6]:

$$\rho'(r_i) = \begin{cases} r_i\{1 - (\frac{r_i}{c})^2\}^2 & \text{if } |r_i| \leq c \\ 0 & \text{otherwise} \end{cases}$$

  Where the cut-off value c is most conventionally set to the value 4.685. Tukey's biweight function is an "iteratively reweighed measure of central tendency"[12], which computes a Mean-estimator for the input data. The bi-weight function is considered to be robust as the bi-weight $\rho'(r_i)$ depends only on the intrinsic weight calculation for r_i and is not sensitive to outlier data. In our implementation r_i is the distance between point x_i in the warped *canonical model* and the point x_i in the *live frame*.

We currently, though, do not have an implementation for the $\mathbf{Reg}(\mathcal{W}_t, \mathcal{E})$ term as we do not have edge connection between the deformation nodes.

6.5.3 Data association

One of the main problems when warping, though, is that of data association. It is very important to match the vertices of the warped canonical model with those of the live frame, as by minimising their distance we will achieve a better approximation of the *warp-field*. And as for the data association between the warped canonical model to live frame and the *live frame*, the paper[25] suggests using Iterative Closest Point (ICP). *Kinfu remake* (see Section 4.5) already has an implementation of ICP, yet it is rather complicated as it is written in *CUDA* and is heavily optimised.

We, therefore, decided to take a different approach. For each vertex in the live frame, we calculate the closest neighbour to that point in the canonical warped to live frame using the KD-tree (see Section 5.4). This allows us to have a simple, efficient CPU implementation of data association.

6.6 Extending the warp-field

We increase the number of *deformation nodes* to make sure we will be able to warp correctly all the newly introduced geometry, such that all vertices in the volume will be influenced by at least one deformation node.

This is achieved after every frame by re-sampling the *deformation nodes* and finding a region where there are vertices with no close-enough *deformation nodes* to affect them. We perform this check using the radial basis weight ($\mathbf{dg}_w$) together with the **DQB** weight formula (Expression 1). The parameters of the newly introduced *deformation node* are set using the neighbouring *deformation nodes*, together with dual quaternion blending to update its warp parameters. This, thus, allows us to extend the *warp-field* of the *canonical model* as it gets updated with new geometry as the object moves around.

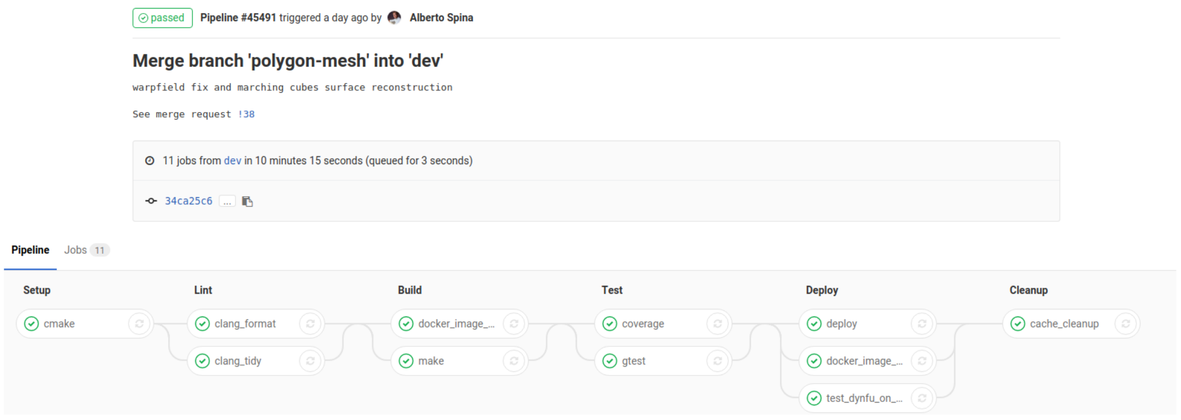
6.7 Surface Fusion

Unfortunately, we could not finish our implementation for the surface fusion, defined in Section 5.7. This is mostly due to the fact that we have a CPU implementation of the *warp-field* and in order to warp even for low resolution of 256^3 voxel centres it takes around an hour per frame. Whilst the algorithm itself is relatively simple, it will require a complete re-write of all the components of the algorithm in *CUDA* which was not feasible in the time we had left.

7 Evaluation

7.1 Testing and Deployment

Throughout its production, `dynfu` was backed up by a thorough pipeline (Figure 20) with several stages:


passed Pipeline #45491 triggered a day ago by Alberto Spina

Merge branch 'polygon-mesh' into 'dev'

```
warpfield fix and marching cubes surface reconstruction

See merge request !38
```

11 jobs from dev in 10 minutes 15 seconds (queued for 3 seconds)

`34ca25c6`

Figure 20: `dynfu` pipeline

1. **Setup** stage. The setup stage checks that all environment variables are set correctly (including the `CUDA_PATH`). Because our project relies on custom-modified *Terra* and *Opt* libraries, they are downloaded and installed locally during this stage. Finally during setup we run *cmake* for the project setting `CMAKE_EXPORT_COMPILE_COMMANDS=ON` to ensure the creation of the `compile_commands.json` required by the linters.

2. **Linter** stage. The linter stage was mainly split up between a *clang_format* and a *clang_tidy* stage.

   - *clang_format*: this task uses the *clang-format* linter[32] by LLVM and it ensures that our code fits the *Google C++ standard*.
   - *clang_tidy*: this task uses the *clang-tidy* linter[33] by LLVM. We tailored the flags to have coding conventions mostly based off the *Google C++ Standard*, the *High Integrity C++ Coding Standard* and the *LLVM Coding Standard*.

3. **Build** stage. The build stage mainly checks that `dynfu` builds successfully both on our *Gitlab Runner* machine and within the *Docker* container.

   - *make*: runs *make* on the *Gitlab Runner*.
   - *docker_image_build*: builds the *Docker* image (see Section 7.1.1).

4. **Test** stage. Testing was a core aspect of `dynfu`, for this reason it is split among two different tasks.

   - *gtest*: the core unit testing framework was written using *google-test*[13] (see Section 7.1.2).

- *coverage*: using *gtest*'s testing environment, a code coverage report is generated using *LCOV*[21] (see Section 7.1.2.1).

5. **Deploy** stage. The deploy stage handles the project's deployment both locally to lab machines and remotely to our *EC2* instance. A commit which both successfully builds and runs the tests on either our main `dev` branch or on `master` will be deployed.
   - *deploy*: deploys built project executables to a shared group directory.
   - *docker_image_deploy*: publishes the *Docker* image to *Docker Hub* (see Section 7.1.1)
   - *test_dynfu_on_ec2*: triggers testing on the *AWS EC2* instance (see Section 7.1.3).
6. **Cleanup** stage. The final stage of the pipeline cleans up all the cache and artefacts created by the pipeline.

7.1.1 Dockerising `dynfu`

We have produced a *Docker* container which includes the dependencies required to run `dynfu` and the built executable which can be run on custom data input via a host directory mounted into the container. The *Docker* container is built using the *CUDA 8.0 devel* base image provided by *Nvidia*[27] and uses *nvidia-docker*[26] to run on a *Linux*-based machine with a compatible GPU. The `dynfu` *Docker* image is published to *Docker Hub*[24].

The `dynfu` *Docker* image is published with three tags. The `latest` and `master` tags are built using a version of *OpenCV* which is compiled for every *Nvidia* GPU architecture. The `dev` tag, however, uses a version of *OpenCV* which is only compiled for the *Maxwell* GPU architecture, as the *EC2* instance used for testing contains this type of GPU. The reason to produce a separate tag for development was to reduce the image size of the *Docker* container as well as reduce the time required to build the *Docker* image (which, as stated in Section 7.1, is built as part of each CI pipeline).

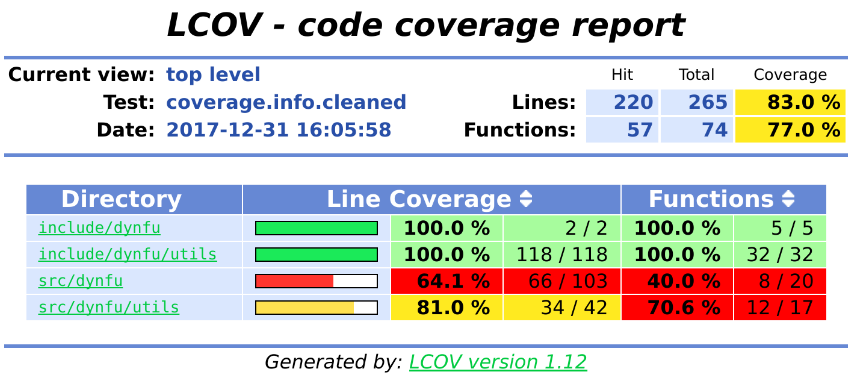
As part of our project's Continuous Development cycle the `dev` tag represents the container built using the latest source code on the `dev` branch in our version control repository, and the `latest/master` tags are built from the source code on the `master` branch which is considered ready for release.

7.1.2 Unit testing `dynfu`

We included Unit tests for every core part of `dynfu`, which guided us through and helped us verify the mathematics of the algorithm. We used *google-test*[13] as testing framework, and included the same tests in the CI pipeline. Tests can be built by setting `GTEST_BUILD_SAMPLES=ON` when running *cmake*.

7.1.2.1 Code Coverage

We used *gcov* with the graphical interface *LCOV*[21] to monitor the coverage of the Unit tests and ensure every component of `dynfu` was adequately tested. Whenever the testing flag (see Section 7.1.2) is set `dynfu` is also compiled to generate the *gcov* output and the *LCOV* HTML coverage report, easy and convenient to analyse. During the final iteration of the project we managed to achieve a code coverage of over 70% as can be seen in Figure 21.



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level				
Test:	coverage.info.cleaned	Lines:	220	265	83.0 %
Date:	2017-12-31 16:05:58	Functions:	57	74	77.0 %

Directory	Line Coverage			Functions	
include/dynfu		100.0 %	2 / 2	100.0 %	5 / 5
include/dynfu/utils		100.0 %	118 / 118	100.0 %	32 / 32
src/dynfu		64.1 %	66 / 103	40.0 %	8 / 20
src/dynfu/utils		81.0 %	34 / 42	70.6 %	12 / 17

Generated by: LCOV version 1.12

Figure 21: *LCOV* coverage report during the final iteration

7.1.3 Testing `dynfu` output

The Continuous Development pipeline includes an integration testing stage in which the `dynfu` algorithm is ran on test frames from the *Volume Deform* project[15]. *Amazon Web Services* is used for this testing process. A *Lambda* is triggered in the CI/CD pipeline to start an *EC2* instance equipped with a *Nvidia GRID K520* graphics card which runs the `dynfu` *Docker* container. The output is uploaded to an *S3* bucket so that it can be displayed in a web interface (outlined in Section 7.1.4) for comparison between different versions.

7.1.4 Web Interface

After we uploaded the images to the *S3* bucket (see Section 7.1.3), we needed a way to easily view and navigate the output of previous test runs in an accessible and user-friendly manner. The process we had to go through, in fact, involved having to log in to *S3* and look at each directory separately, or navigating through tricky file paths to load images directly into the browser.

To make the process of viewing test results easier, we decided to create a web interface (Figure 22) to streamline viewing historical data.

7.1.4.1 Usage

The interface can simply be accessed via the web[23] and displays a list of timestamps of all previous tests which have been run on the left. These timestamps can then be clicked to load all the processed frames from that test on the right-hand side.

Once a specific test result is loaded, the initial frame from the test run is displayed, and a slider can be used to view the other frames of the test. This can be adjusted manually via dragging the slider toggle (on the top of the page), or by pressing a *Play* button to view the frames as if it were a video, just like the original live input originally used to be. Frame meta data, such as its specific ID, for the currently viewed frame is also displayed to allow for easier debugging and to allow proper referencing of a specific image. The web interface, thus, allows users to view historical sets of test data, whilst granting granular control over each frame of the output.

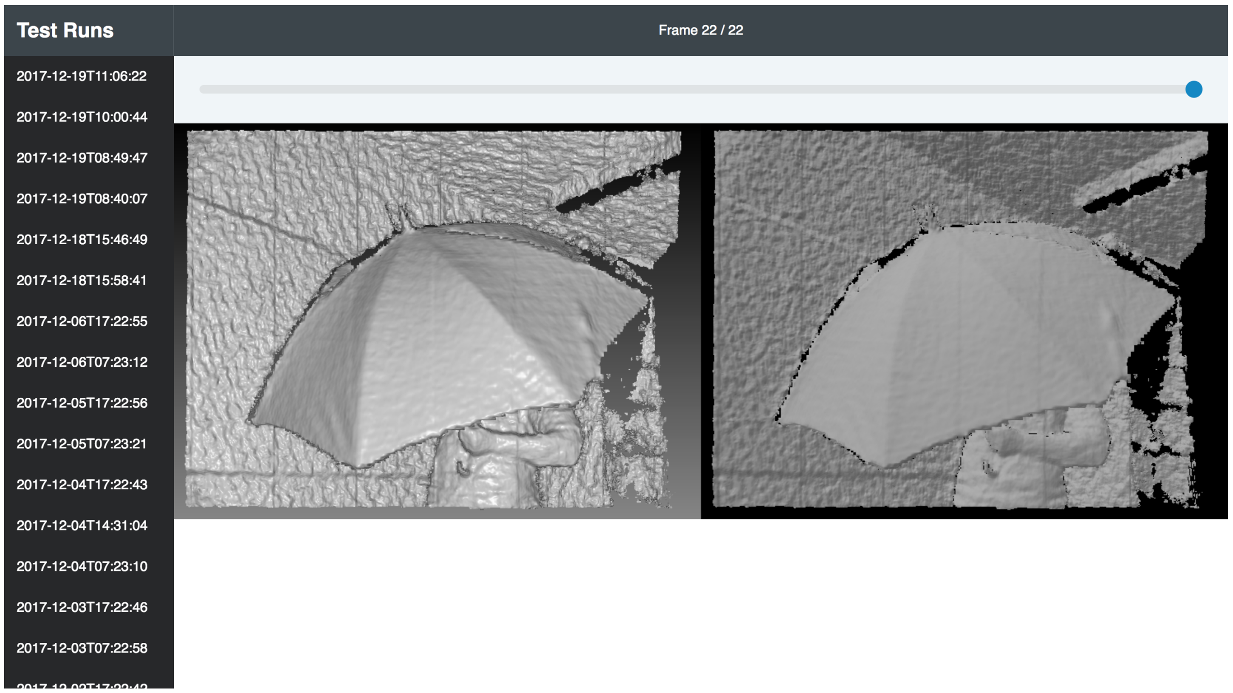


Figure 22: Example web interface session

7.1.4.2 Implementation

In order to make the interface platform-independent and to save time trying to allow compatibility on all of our different devices, we opted to make the test visualiser a web app that could be accessed via any web-enabled device – as such, the tool is written using JavaScript, HTML, and CSS.

Because the core functionality isn't extremely complex and the code is relatively lightweight, we determined that using vanilla JavaScript would be okay, and that using an entire library to write the app would not be necessary. We also used some new CSS and HTML features such as *CSS Grid* to speed up the prototyping phase of this feature. The design is influenced by other modern interfaces and colour schemes.

7.1.4.3 Reproducibility

As an intentional design decision, the web interface is entirely decoupled from the rest of the testing code – it can run independently of the *Docker* container and any other part of the test harness. The interface loads test runs by accessing a simple text file, `test-runs.txt`, from *S3*, which it then uses to determine which directories store images, then dynamically loads images from `1.png`, `2.png` etc. until no more images exist – this is required because test runs can be of a variable number of frames. In theory, the interface could be used to visualise *any* program which formatted its output in this way.

7.2 Results

7.2.1 Optimisation

Many optimisation were carried out in this project, even though we did not aim towards the real-time reconstruction. This was because we are dealing with a massive set of data (i.e. 512^3 voxels in TSDF). Any naive approach, especially on CPU, slows down our code, to a point where it takes almost half a day to process a frame. These are the performance gains which we got through different optimisations:

- With data association and background removal: `6426ms per frame` [1].
- Without data association: `13641ms per frame`.
- Without background removal: `109171ms per frame`.

As the results show, the most effective optimisation was to remove the background of the volume, as this introduced extra vertices and deformation nodes, which slowed down our solver.

7.2.2 Performance

As shown in the Table 1, we observe that our performance bottleneck is the warp function. This is due to the fact the warp field, and the dual quaternions, are implemented in CPU-bound code. Creating a GPU implementation of the *warp-field* and dual quaternions would not only improve our speed for the warp function, but, as discussed previously Section 6.7, it will allow us to implement surface fusion in reasonable time.

Frame N°	Add Live Frame	Data Association	Warp using DQB	Solver [2]
0	0.050199	0.528277	43.667182	15.791073
1	0.049438	0.544267	43.689484	16.573993
2	0.048886	0.530406	43.728054	16.425357
3	0.049466	0.549595	43.753973	17.128609
4	0.049651	0.56946	43.902804	17.173459
5	0.049348	0.560218	43.767318	18.670092
6	0.048962	0.598988	43.708631	17.629831
7	0.048981	0.607403	43.676454	21.661256
8	0.048972	0.642839	43.828769	19.647075
9	0.048837	0.660586	43.672136	17.006599
10	0.04882	0.697465	43.666541	17.686438

Table 1: Time [s] taken to run each stage of the `dynfu` algorithm

We also see from Figure 23 that the time for data association increases per iteration. This is due to the fact that we don't update the *canonical model* and, as new surfaces of the volume gets revealed, these points struggle to find the association with the original

[1]Calculated by taking the average of the time solver spent per frame on the first 5 frames.

[2]Includes data transfer.

canonical model warped the the previous *live frame*. It is also important to note that the fluctuation in the solver's time is due to the differing amount of motion made by the model between each frames.

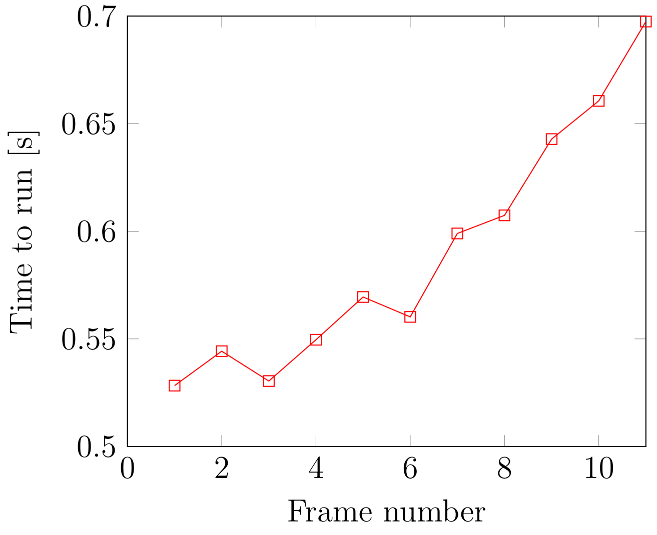


Figure 23: Time [s] to run the data association stage of `dynfu` at each frame

7.3 Challenges faced

7.3.1 Planning the implementation of DynamicFusion

For many of us this was the first project we had done that required thorough research to be done before we could start coding. The paper[25] we were suggested to use to start our research includes fields of mathematics that we were not familiar with, in particular dual quaternions, so we had to do research and understand this topic before we could start implementing it. Another challenge we found with implementing an algorithm described in a research paper was that in order to provide our customer with evidence of progress we had to dissect the paper and split it into deliverables. We found that our main reference paper[25] had a section on the features of DynamicFusion that could be easily translated in to initial milestones.

Initially, since there was a lot to understand before we could begin to implement the paper, we looked for a project on *GitHub* that was aiming for a similar goal, and which we could build upon. We found a project[7] trying to implement the algorithm described in the same paper[25]. Since there were a number of issues with the project that prevented it from properly working we set to work in correcting theses issues; however it became apparent over time that the poor coding practices used in this project were slowing progress and making it difficult to work on. After a couple of weeks we decided that progress was going so slow building on top of the existing code base that we would be better implementing DynamicFusion ourselves on top of a working *Kinetic Fusion* library[3]. Early on we were told by our supervisor that often it takes the same amount of time to build on top of somebody else's code base as it does to write the code yourself, since when adding to somebody else's code base you must first thoroughly understand it yourself. However we did find that by initially attempting to work on top of an existing code base we gained understanding of concepts talked about in the DynamicFusion paper[25] and ideas on good ways to structure our project.

7.3.2 Building `dynfu` without a dedicated build server

An issue we faced whilst developing `dynfu` was the necessity of running the algorithm on a machine with an *Nvidia* GPU. Not all team members had sufficient hardware so we had to decide on a environment which we could all access where we would develop and run `dynfu`. Using the *EC2* instance used for testing would have been too expensive, so we decided to use the *graphic* machines hosted by CSG[11]. These machines are *CUDA*-enabled, however we were affected by not having exclusive access to our reserved machine and not having permissions to install dependencies in standard locations. Several times other students were already using the GPU and we were unable to run `dynfu`. We resolved this by emailing these students and reminding them that we had reserved the machine. As we did not have *root* access to the computer and could not install pre-compiled binaries we had to compile dependencies and install them to a shared directory. This caused an issue when installing *Opt* which required *CUDA* to be installed in a standard location. We had to fork the *Opt* repository and change the location it was searching for *CUDA* in. We could have saved time on this project by having a dedicated machine with *root* access.

7.4 Deliverables

7.4.1 The `dynfu` algorithm

Even though it's far from being finished or complete, the current `dynfu` algorithm is a solid starting point for a correct implementation of the DynamicFusion Paper[25]. Currently `dynfu` is able to:

- **Provide a clean *cmake* project.** `dynfu` provides a new, clean, *cmake* project for the *kinfu_remake* library[3]. Dependencies are clearly listed (as can be seen in **Appendix D**) and the project can be built with ease on a multitude of machines.

- **Provide a refactored codebase for *kinfu_remake*.** Not only does `dynfu` extend *kinfu_remake* introducing features to construct a *canonical model* for non-rigid objects within a moving scene, but by wrapping the project entirely it introduces a stronger hierarchical class structure for easier and quicker access of the *live frames* or *warp-fields* processed by the library.

- **Provide a thoroughly tested codebase.** All components of `dynfu` are thoroughly tested with a code coverage greater than 70% (see Section 7.1.2.1). We provide a multitude of tests for the actual solver to assert correctness when warping *live scenes* to the *canonical model* and vice-versa. This differs from the standard set by most SLAM open sourced code bases, which would provide test data and sample outputs, yet fail to supply the tools to test the correctness of the various components.

- **Process (off-line) a new frame every 2 minutes.** Even though we process frames slower than the *kinfu_remake* library we built the project on, we are not excessively away from the real time processing an ideal implementation would be aiming for. All the processed frames are logged for output in PNG and PCL formats for visualization.

- **Use a GPU optimised solver.** Switching to the *Opt* Solver[10], which is GPU bound, from the original *Ceres* solver[1] not only allowed us to have more than a 100 times improvement in solving time, but it also enabled us to introduce GPU optimised code within `dynfu`, the performance of which can, hence, surely be improved in the future.

It is also true that our implementation has a number of limits, in fact `dynfu` is, currently, unable able to:

- **Provide a complete *canonical model*.** Even though we put our best efforts in, we aren't currently able to reliably perform surface fusion to update the *canonical model* with the new geometry captured by the warped *live frame*.
- **Solve correctly after around 20 frames.** Attempting to achieve real time performance we had to consider a number of trade-offs within our implementation. Unfortunately this means that our solver is unable to estimate *warp-field* parameters reliably after this point. A further complication might be given by a non-updated canonical model; should we unveil too much new geometry for a scene, the *warp-field* estimation phase will fail and the fields of the *deformation nodes* would be updated with garbage data.

7.4.2 Docker Container

A *Docker* container for users to run `dynfu` is an impactful product as it removes the cumbersome burden of downloading and compiling the various dependencies and ensuring that the versions installed are compatible. Running the `dynfu` executable requires only running the container with a mounted directory supplying the input data. However there are two limitations to the container's usability.

1. Size
2. *nvidia-docker* is required to run the container

As the container includes the dependencies for `dynfu`, for example the large libraries *OpenCV*, *PCL* & *FLANN*, the size varies from 12-14GB depending on the number of GPU architectures for which it is built. The container can be optimised to ensure that only runtime dependencies remain once the executable has been built however this was not considered to be a priority during the previous iterations as we focused on producing a runnable container and the `dynfu` algorithm.

nvidia-docker is still actively under development and is not a fully featured product. It currently suffers from the limitation of only supporting the *Linux* Operating System. Whilst this limitation did not have an effect on us for the purposes of testing on an *EC2* instance and can be used by our customer on his *Linux* machine; it reduces the number of future customers who will be able to use this product. We would like to be able to rely solely on *docker-engine* which is available in most popular Operating Systems however at this time there is insufficient support for a hypervisor above a GPU.

8 Future Extensions

In addition to the algorithm we have implemented, we plan to be able to update the canonical model using surface fusion in the short term (see Section 5.7). As discussed in Section 6.7, we could not implement this in time, yet given more time after the project's deadline, we would like to be able to fully implement it. Furthermore, the current implementation of *deformation nodes* does not have edges connecting the nodes, meaning that the implementation *Rigid as Possible*[30] was not able to be carried out for non-visible volume during warping. Implementing this feature should allow volume outside the visible scene to warp rigidly to the currently visible scene connecting to it; such that when the non-visible volume is visible again, there will be little error in its position.

Although there will always be a compromise between the speed of the algorithm and the detail of its output, given more time we would like to improve the performance of our algorithm through techniques such as performing calculations (i.e. dual quaternion blending[19]) in parallel on GPU and implementing ICP on GPU instead of using KD-tree for data association. Also the paper[25] suggests some further optimisation such as creating an application specific solver and pre-computing all the neighbouring deformation nodes per voxel, rather than computing them on a per-vertex basis at every iteration. The aim for this increased performance would be to achieve speeds such that the application could run in real time.

The DynamicFusion algorithm allows us to map moving scenes in detail. This is useful in many fields of work where analysis of a scene is required. In particularly throughout the project we have been working alongside Daniel Grzech, a PhD student, whose project is investigating using DynamicFusion to map the movements of babies immediately after birth to detect early stage motor problems.

References

[1] Agarwal, S., Mierle, K. and Others. 2016. *Ceres Solver*. [ONLINE] Available at: `http://ceres-solver.org`. [Accessed 30 December 2017].

[2] Amazon. 2017. *EC2 Instance Types*. [ONLINE] Available at: `https://aws.amazon.com/ec2/instance-types/`. [Accessed 28 December 2017].

[3] Baksheev, A. 2014. GitHub repository. *kinfu_remake*. [ONLINE] Available at: `https://github.com/Nerei/kinfu_remake`. [Accessed 30 December 2017].

[4] Blanco, J. 2010. *A tutorial on SE(3) transformation parameterizations and on-manifold optimization*. [ONLINE] Available at: `https://pixhawk.org/_media/dev/know-how/jlblanco2010geometry3d_techrep.pdf`. [Accessed 1 January 2018].

[5] Blanco, J., Rai, P. 2014. GitHub repository. *nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees*. [ONLINE] Available at: `https://github.com/jlblancoc/nanoflann`. [Accessed 30 December 2017].

[6] Breheny, P. 2017. *Robust regression*. Lecture notes distributed in the unit, BST 764: Applied Statistical Modeling for Medicine and Public Health, University of Kentucky, Kentucky on 1 December 2017. [ONLINE] Available at: `https://web.as.uky.edu/statistics/users/pbreheny/764-F11/notes/12-1.pdf`. [Accessed 28 December 2017].

[7] Bujanca, M. 2017. GitHub repository. *Dynamic Fusion Implementation*. [ONLINE] Available at: `https://github.com/mihaibujanca/dynamicfusion`. [Accessed 27 December 2017].

[8] Cadena, C., Carlone, L., Carrillo, H., Latif, Y., Scaramuzza, D., Neira, J., Reid, I., Leonard, J. *Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age*. IEEE Transactions on Robotics, (Volume: 32, Issue: 6), 1309 - 1332.

[9] DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., Vitek, J. 2013. *Terra: A Multi-Stage Language for High-Performance Computing. In ACM 2013*. Seattle, Washington, June 16-19, 2013. 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. 105 - 116.

[10] DeVito, Z., Mara, M., Zollöfer, M., Bernstein, G., Theobalt, C. Hanrahan, P., Fisher, M., Nießner, M. *Opt: A Domain Specific Language for Non-linear Least Squares Optimization in Graphics and Imaging*. ACM Transactions on Graphics (TOG), (Volume: 36, Issue: 5), Article 171.

[11] DoC Computing Support Group. 2017. *Teaching Labs: Workstations*. [ONLINE] Available at: `https://www.doc.ic.ac.uk/csg/facilities/lab/workstations`. [Accessed 27 December 2017].

[12] Dombi, G. 2006. *Using Biweights for Handling Outliers. In MWSUG 2006*. Detroit, Michigan, October 22-24, 2006. MidWest SAS Users Group 2016. SD01.

[13] Google Inc. 2017. GitHub repository. *Google Test.* [ONLINE] Available at: `https://github.com/google/googletest`. [Accessed 29 December 2017].

[14] Holin, H. 2006. *Boost.Quaternions.* [ONLINE] Available at: `http://www.boost.org/doc/libs/1_40_0/libs/math/doc/quaternion/html/index.html`. [Accessed 30 December 2017].

[15] Innmann, M., Zollhöfer, M., Nießner, M., Theobalt, C., Stamminger, M. 2016. *VolumeDeform: Real-Time Volumetric Non-rigid Reconstruction. In ECCV 2016.* Amsterdam, Netherlands, October 11-14, 2016. The 14th European Conference on Computer Vision. Proceedings, Part VIII (362-379).

[16] Intel Corporation. 2017. *Intel RealSense Technology.* [ONLINE] Available at: `https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html`. [Accessed 31 December 2017].

[17] Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A., Fitzgibbon, A. 2011. *KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In UIST '11.* Santa Barbra, CA, October 16-19, 2011. UIST '11 Proceedings of the 24th annual ACM symposium on User interface software and technology: ACM. 559-568.

[18] Jia, Y. 2017. *Dual Quaternion.* Lecture notes distributed in the unit, CS577 Problem Solving Techniques for Advanced Computer Science, Iowa State University, Iowa on 5 September 2017. [ONLINE] Available at: `http://web.cs.iastate.edu/~cs577/handouts/dual-quaternion.pdf`. [Accessed 30 December 2017].

[19] Kavan, L., Collins, S., O'Sullivan, C., Zara, J.. 2006. *Dual Quaternions for Rigid Transformation Blending.* Trinity College Dublin, Tech. Rep. TCD-CS-2006-46. [ONLINE] Available at: `https://www.scss.tcd.ie/publications/tech-reports/reports.06/TCD-CS-2006-46.pdf`. [Accessed 1 January 2018].

[20] Kenwright, B. 2012. *A Beginners Guide to Dual-Quaternions: What They Are, How They Work, and How to Use Them for 3D Character Hierarchies. In WSCG 2012.* Pilsen, Czech Republic, June 26-28, 2012. 20th International Conference on Computer Graphics, Visualization and Computer Vision 2012. A29.

[21] Linux Test Project (LTP). 2016. *LCOV - the LTP GCOV extension.* [ONLINE] Available at: `http://ltp.sourceforge.net/coverage/lcov.php`. [Accessed 27 December 2017].

[22] MathWorks. 2017. *Classification Using Nearest Neighbors.* [ONLINE] Available at: `https://uk.mathworks.com/help/stats/classification-using-nearest-neighbors.html`. [Accessed 30 December 2017].

[23] Murai, R., Spina, A., Brookes, M., Boulby, D., Bower, T., Bonardi, A. 2017. *Dynamic Fusion Test Interface.* [ONLINE] Available at: `https://s3.eu-west-1.amazonaws.com/dynamic-fusion/index.html`. [Accessed 24 December 2017].

[24] Murai, R., Spina, A., Brookes, M., Boulby, D., Bower, T., Bonardi, A. 2017. *dynfu Docker Image*. [ONLINE] Available at: `https://hub.docker.com/r/g1736211/dynfu`. [Accessed 1 January 2018].

[25] Newcombe, R., Fox, D., Seitz, S. 2015. *DynamicFusion: Reconstruction and Tracking of Non-rigid Scenes in Real-Time*. [ONLINE] Available at: `http://grail.cs.washington.edu/projects/dynamicfusion/papers/DynamicFusion.pdf`. [Accessed 3 January 2018].

[26] Nvidia. 2017. GitHub repository. *Docker Engine Utility for NVIDIA GPUs*. [ONLINE] Available at: `https://github.com/NVIDIA/nvidia-docker`. [Accessed 28 December 2017].

[27] Nvidia. 2017. *Cuda 8 Development Docker Image*. [ONLINE] Available at: `https://hub.docker.com/r/nvidia/cuda`. [Accessed 28 December 2017].

[28] Orsten, S., Dorodnicov, S., Diakopoulos, D. and Others. 2017. GitHub repository. *Intel® RealSense^{TM} SDK 2.0*. [ONLINE] Available at: `https://github.com/IntelRealSense/librealsense`. [Accessed 30 December 2017].

[29] Pavlenko, A., Kurtaev, D. and Others. 2015. GitHub repository. *Open Source Computer Vision Library*. [ONLINE] Available at: `https://github.com/itseez/opencv`. [Accessed 28 December 2017].

[30] Sorkine, O., Alexa, M. 2007. *As-Rigid-As-Possible Surface Modeling. In SGP '07*. Barcelona, Spain, July 4-6, 2007. The fifth Eurographics symposium on Geometry processing. 109-116. [ONLINE] Available at: `https://igl.ethz.ch/projects/ARAP/arap_web.pdf`. [Accessed 29 December 2017].

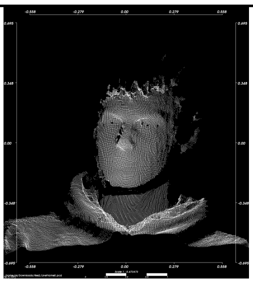
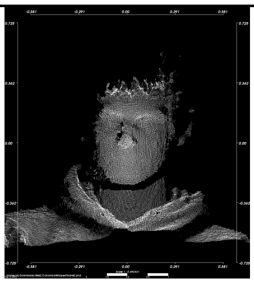
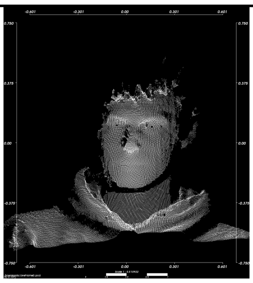
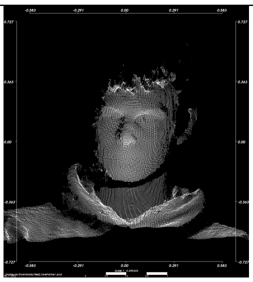
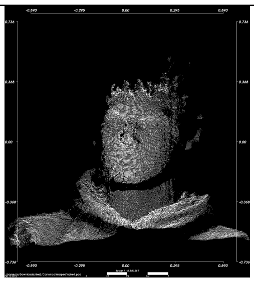
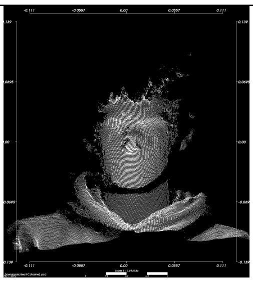
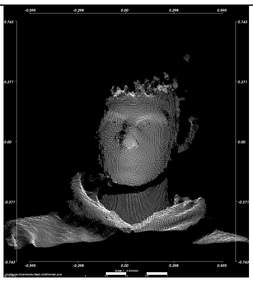
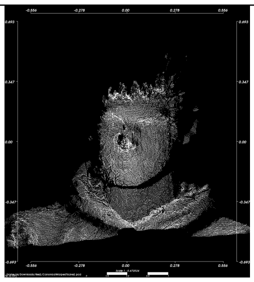
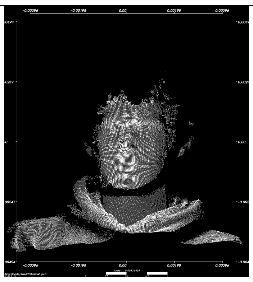
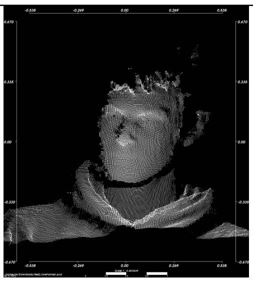
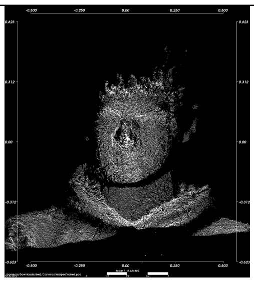
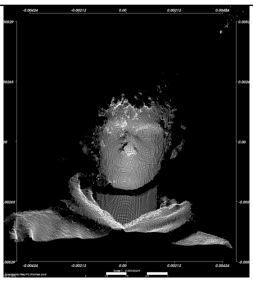
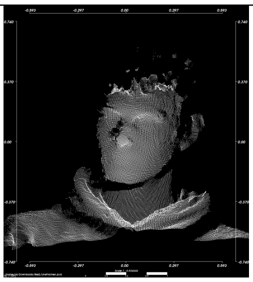
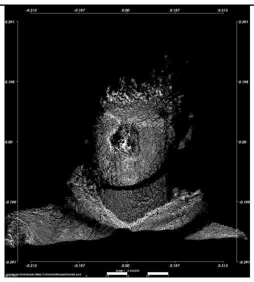
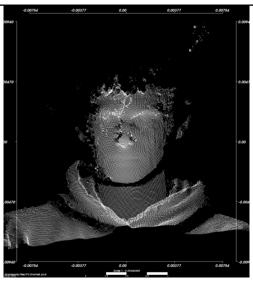
[31] sp4cerat. 2016. *[Hiring] Graphics / GPU Developer to implement Dynamic Fusion*. [ONLINE] Available at: `https://www.reddit.com/r/jobbit/comments/46lemv/hiring_graphics_gpu_developer_to_implement/`. [Accessed 3 January 2018].

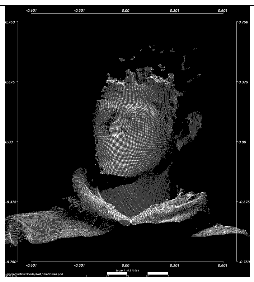
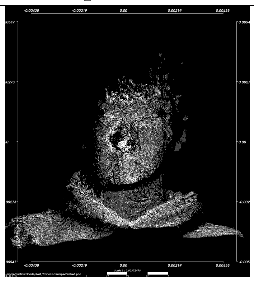
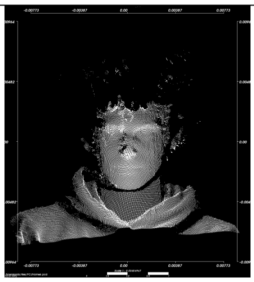
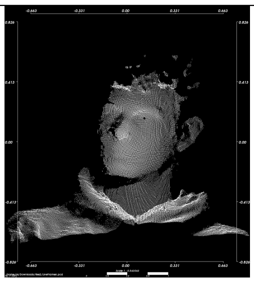
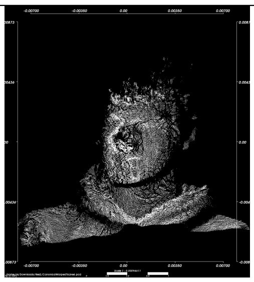
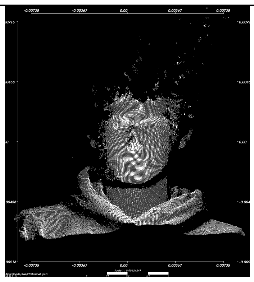
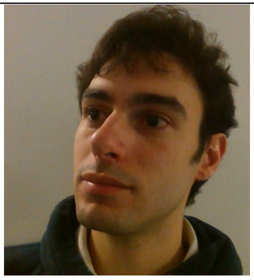
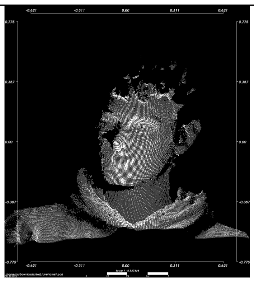
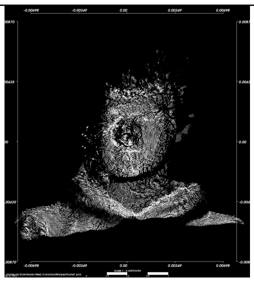
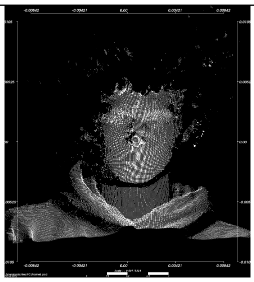
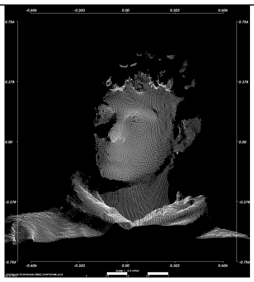
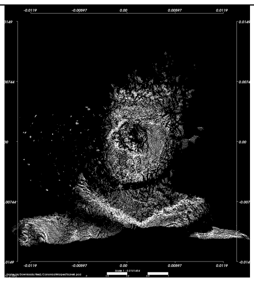
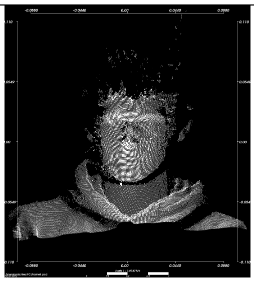
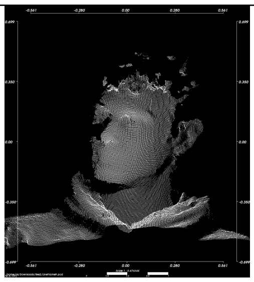
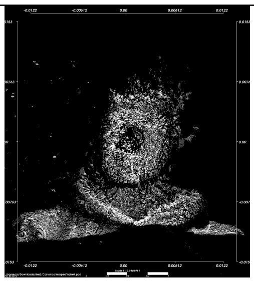
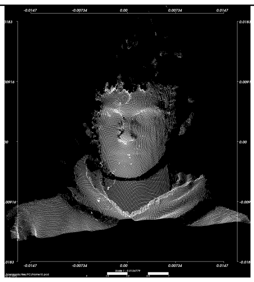
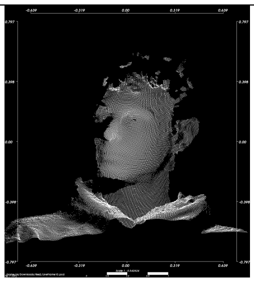
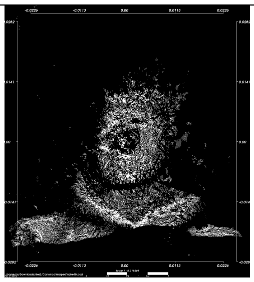
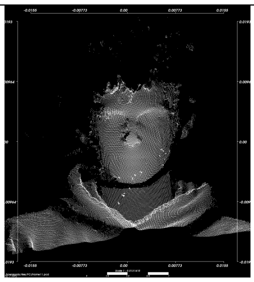
[32] The Clang Team. 2017. *Clang 6 Documentation: ClangFormat*. [ONLINE] Available at: `https://clang.llvm.org/docs/ClangFormat.html`. [Accessed 29 December 2017].

[33] The Clang Team. 2017. *Extra Clang Tools 6 documentation: ClangTidy*. [ONLINE] Available at: `http://clang.llvm.org/extra/clang-tidy/`. [Accessed 29 December 2017].

[34] Yang, A. 1963. *Application of Quaternion Algebra and Dual Numbers to the Analysis of Spatial Mechanisms*. PhD. New York, NY: Columbia University.

Appendix A: Output

Frame N°	Input Frame	Live Frame	Warped Frame	Canonical Frame
0				
1				
2				
3				
4				

Frame N°	Input Frame	Live Frame	Warped Frame	Canonical Frame
5				
6				
7				
8				
9				
10				

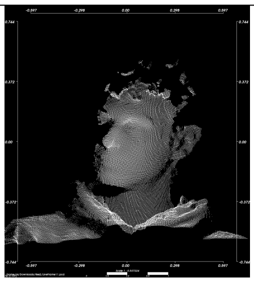
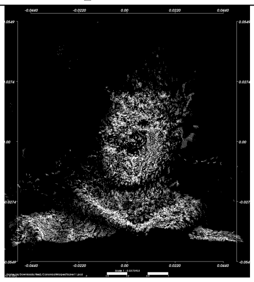
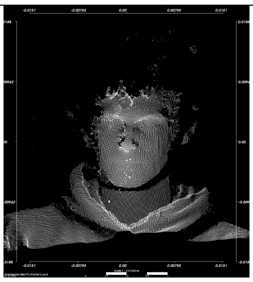
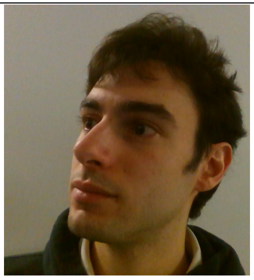
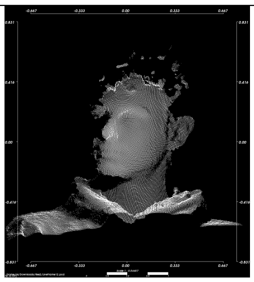
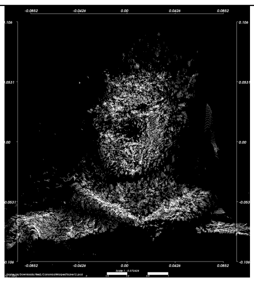
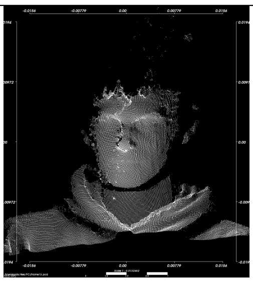
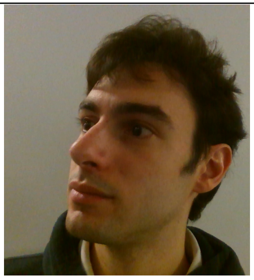
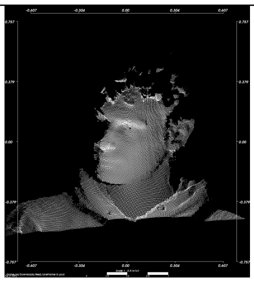
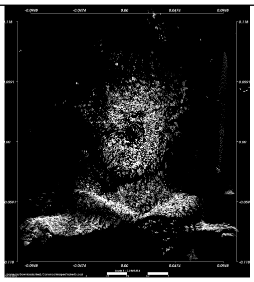
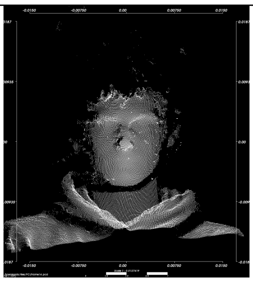
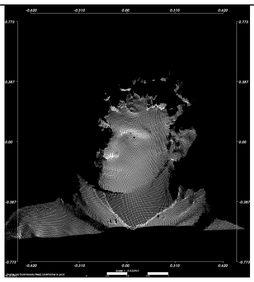
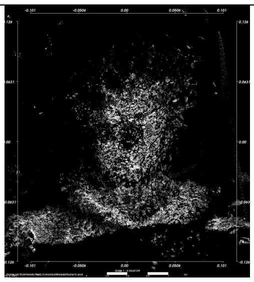
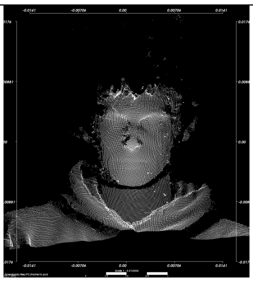
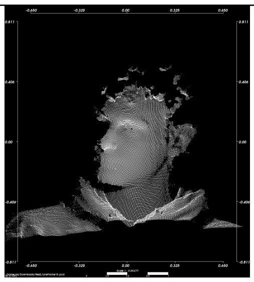
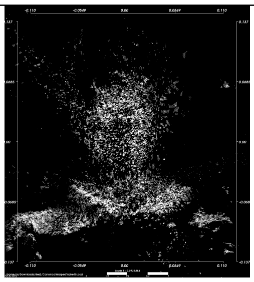
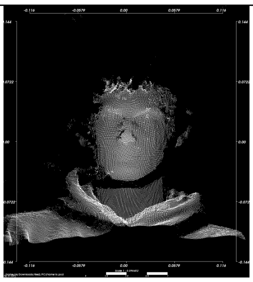
Frame N°	Input Frame	Live Frame	Warped Frame	Canonical Frame
11				
12				
13				
14				
15				

Table 2: Sample `dynfu` output

Appendix B: Opt Logs

Table 3 shows the costs computed by OPT when solving for the Warp function (See Equation 1 in Section 5.6) for four different inputs. [3]

Frame N°	Head Rotation [4]	Umbrella [5]	Advent Calendar [6]	Shirt [7]
1	1.6354E+00	1.9921E+00	5.0629E+00	2.2465E+00
2	2.5871E+00	3.3493E+00	3.6980E+00	2.4935E+00
3	2.6296E+00	2.7533E+00	2.2968E+01	4.8559E+00
4	3.4551E+00	8.8889E+00	1.7260E+01	1.6693E+01
5	5.6324E+00	1.0502E+01	4.8031E+00	4.1517E+01
6	5.9188E+00	4.5500E+00	3.1451E+01	8.6532E+00
7	6.5153E+00	2.6888E+00	2.8551E+01	8.3181E+01
8	8.7943E+00	4.7961E+01	9.3787E+01	3.6795E+02
9	8.5737E+00	2.6836E+01	1.6675E+02	2.5980E+02
10	1.3416E+01	8.6223E+01	8.8296E+02	6.2861E+02
11	8.9125E+00	1.7327E+03	2.8146E+03	2.7466E+03
12	8.8376E+00	3.9147E+03	2.6078E+04	1.0958E+04
13	8.3382E+00	1.6521E+04	1.1146E+05	3.2844E+05
14	9.2546E+00	4.8476E+04	4.6650E+05	4.9891E+04
15	1.0463E+01	3.6219E+05	1.9645E+06	1.6947E+06
16	1.0127E+01	1.1092E+07	4.5290E+06	8.3515E+05
17	9.5811E+00	1.4518E+07	3.2840E+07	9.9260E+07

Table 3: Logged `dynfu` Opt costs

[3]Volume parameters are calibrated for the Head Rotation, hence the background of the scene for rest of the inputs are included as the volume, increasing the cost over time.

[4]The Head Rotation input data can be seen in **Appendix A**.

[5]The Umbrella input data can be downloaded from the VolumeDeform project[15].

[6]The Advent Calendar input data can be downloaded from the VolumeDeform project[15].

[7]The Shirt input data can be downloaded from the VolumeDeform project[15].

Appendix C: Running dynfu on an Amazon EC2 instance

The supported method of running `dynfu` is to run the Docker container on an *EC2* instance, running an *Ubuntu* based distribution, which is equipped with an *Nvidia* GPU such as the *P3* and *G3* type instances[2]. There is online documentation available for help on launching an *EC2* instance so this guide assumes that the reader has a running instance and is connected via SSH.

Both *Docker* and *nvidia-docker* must be installed (see Listing 1) to run the container with the connected GPU.

```
# Install Docker and give ubuntu user permissions to run Docker
   containers
$ sudo apt-get update
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
   apt-key add -
$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
$ sudo apt-get update
$ sudo apt-get install docker-ce
$ sudo usermod -a -G docker ubuntu

# Install nvidia-docker and nvidia-docker-plugin
$ wget -P /tmp https://github.com/NVIDIA/nvidia-docker/releases/
   download/v1.0.1/nvidia-docker_1.0.1-1_amd64.deb
$ sudo dpkg -i /tmp/nvidia-docker*.deb && rm /tmp/nvidia-docker*.
   deb
```

Listing 1: Set up *Docker* and *nvidia-docker*

Log out and log back in before attempting to run the container so that the group membership for *ubuntu* is updated. The container must be pulled from *Docker Hub* and be run with the input data mounted as a volume to the container's */data* directory. Input data must be stored in two folders *color* and *depth* (matching format from the *Volume Deform* project[15]). The enclosing folder is mounted into the container and the output will be in the *out* sub-directory. Listing 2 shows how to run `dynfu` on the contents of the */test-data* directory.

```
# Pull Docker container
$ docker pull g1736211/dynfu
# Run on custom input
$ nvidia-docker run -v /test-data:/data g1736211/dynfu
```

Listing 2: Run `dynfu` on custom input

Appendix D: Dependencies

Dependency	Components	Version	Licence
Boost	Filesystem Math System	1.36.0	Boost Software License
Ceres	*	1.13	New BSD License
CUDA	*	8	NVIDIA Software License Agreement
OpenCV	cudev core cudaarithm flann imgproc ml viz cudafilters cudaimgproc cudawarping imgcodecs highgui features2 calib3d cudafeatures2d cudastereo	3.2.0	BSD License
OpenMesh	*	6.3	BSD 3 Clause License
Opt	*	0.2.0	Creative Commons Public License
PCL	common octree kdtree search sample_consensus filters io 2d features geometry visualization surface tracking	1.8.1	BSD License
Terra	*	release-2016-03-25	MIT License

Table 4: `dynfu` dependencies