Volumetric models have become a popular representation for 3D scenes in recent years. While being used as early as [4], one of the breakthroughs leading to their popularity was KinectFusion [9, 7]. The focus of
KinectFusion and a number of related works [12, 1] is on 3D reconstruction using RGB-D sensors. However, monocular SLAM has since also been tackled with very similar approaches [8, 11]. In the monocular case, various dense stereo techniques are used to create depth images and then the same integration methods and volumetric representations are used to fuse the information into a consistent 3D world model. Storing the underlying truncated signed distance function (TSDF) volumetrically makes for most of the simplicity and efficiency that can be achieved with GPU implementations of these systems. However, this representation is also memory-intensive and limits the applicability to small scale reconstructions.

Several avenues have been explored for overcoming this limitation. A first line of works uses a moving volume [13, 16, 5] to follow the camera while a sparse point cloud or mesh representation is computed for the parts of the scene outside the active volume. In a second line of research, the volumetric representation is split into a set of blocks aligned with dominant planes [6] or even reduced to a set of bump maps along such planes [15]. A third category of approaches employs an octree representation of the 3D volume [17, 3, 14]. Finally, a hash lookup for a set of sparsely allocated subblocks of the volume is used in [10]. Some of these works, e.g. [5, 3, 10, 6], also provide methods for swapping data from the limited GPU memory to some larger host memory to further expand the scale of the reconstructions.

With the aim of providing for a fast and flexible 3D reconstruction pipeline, we propose a new, unifying framework called InfiniTAM. The core idea is that individual steps like camera tracking, scene representation and integration of new data can easily be replaced and adapted to the needs of the user.

Along with the framework we also provide a set of components for scalable reconstruction: two implementations of camera trackers, based on RGB data and on depth data, two representations of the 3D volumetric data, a dense volume and one based on hashes of subblocks [10], and an optional module for swapping subblocks in and out of the typically limited GPU memory. Given these components, a wide range of systems can be developed for specific purposes, ranging from very efficient reconstruction of small scale volumes with limited hardware resources up to full scale reconstruction of large-scale scenes. While such systems can be tailored for the development of higher level applications, the framework also allows users to focus on the development of individual new components, reusing only parts of the framework.

Although most of the ideas used in our implementation have already been presented in related works, there are a number of differences and novelties that went into the engineering of our framework. For example in Section 5 we define an engine for swapping data to and from the GPU in a way that can deal with slow read and write accesses on the host, e.g. on a disk. It also has a fixed maximum number of data transfers between GPU and host to ensure an interactive online framework. One general aim of the whole framework was to keep the implementation portable, adaptable and simple. As we show in Section 6, InfiniTAM has minimal dependencies and natively builds on Linux, Mac OS and Windows platforms.

The remainder of this report and in particular Sections 2 through 5 describe technical implementation details of the InfiniTAM framework. These sections are aimed at closing the gap between the theoretical description of volumetric depth map fusion and the actual software implementation in our InfiniTAM package. Some advice on compilation and practical usage of InfiniTAM is given in Section 6 and some concluding remarks follow in Section 7.

2 Architecture Overview

We first present an overview of the overall processing pipeline of our framework and discuss the cross device implementation architecture. These serve as the backbone of the framework throughout this report.

2.1 Processing Pipeline

The main processing steps are illustrated in Figure 1. As with the typical and well known KinectFusion pipeline [9], there is a tracking step for localizing the camera, an integration step for updating the 3D world model and a raycasting step for rendering the 3D data in preparation of the next tracking step. To accommodate the additional processing requirements for octrees [14], voxel block hashing [10] or other data structures, an allocation step is included, where the respective data structures are updated in preparation for integrating data from a new frame. Also an optional swapping step is included for transferring data between GPU and CPU.
A detailed discussion of each individual stage follows in Section 4. For now notice that our implementation follows the chain-of-responsibility design pattern, which means that the relevant data structures (e.g., ITMImage) are passed between several processing engines (e.g., ITMSceneReconstructionEngine). The engines are stateless and each engine is responsible for one specific aspect of the overall processing pipeline. The state is passed on in objects containing the processed information. Finally one central class (ITMMainEngine) holds instances of all objects and engines and controls the flow of information.

2.2 Cross Device Implementation Architecture

Each engine is further split into three layers as shown in Figure 2. The topmost, Abstract Layer is accessed by the library’s main engine and is in general just an blank interface, although some common code can be shared at this point. The interface is implemented in the Device Specific Layer, which will be very different depending on whether it runs on a CPU, a GPU, on OpenMP or other hardware acceleration architectures. In the Device Agnostic Layer there may be some inline code that is called from the higher layers and recompiled for the different architectures.

Considering the example of a tracking engine, the Abstract Layer contains code for the generic optimisation of an error function, the Device Specific Layer contains a loop or GPU kernel call to evaluate the error function for all pixels in an image, and the Device Agnostic Layer contains a simple inline C-function to evaluate the error in a single pixel.

3 Volumetric Representation

The volumetric representation is the central data structure within the InfiniTAM framework. It is used to store a truncated signed distance function that implicitly represents the 3D geometry by indicating for each point in space how far in front or behind the scene surface it is. Being truncated, only a small band of values around the current surface estimate is actually being stored, while beyond the band the values are set to a maximum and minimum value, respectively.

The data structure used for this representation crucially defines the memory efficiency of the 3D scene model and the computational efficiency of interacting with this model. We therefore keep it modular and interchangeable in our framework and provide two implementations, namely for a fixed size dense volume (ITMPlainVoxelArray) and voxel block hashing (ITMVoxelBlockHash). Both are described in the following, but first we briefly discuss how the representation is kept interchangeable throughout the framework.
3.1 Framework Flexibility

Core data classes dealing with the volumetric signed distance function are templated on (i) the type of voxel information stored, and (ii) the data structure used for indexing and accessing the voxels. The relevant engine classes are equally templated to provide the specific implementations depending on the chosen data structures.

An example for a class representing the actual information at each voxel is given in the following listing:

```cpp
struct ITMVoxel_s_rgb
{
    .CPU_AND_GPU_CODE_ static short SDF_initialValue() { return 32767; }
    .CPU_AND_GPU_CODE_ static float SDF_valueToFloat(float x)
        { return (float)(x) / 32767.0f; }
    .CPU_AND_GPU_CODE_ static short SDF_floatToValue(float x)
        { return (short)((x) * 32767.0f); }

    static const bool hasColorInformation = true;

    /** Value of the truncated signed distance transformation. */
    short sdf;
    /** Number of fused observations that make up @p sdf. */
    uchar w_depth;
    /** RGB colour information stored for this voxel. */
    Vector3u clr;
    /** Number of observations that made up @p clr. */
    uchar w_color;

    .CPU_AND_GPU_CODE_ ITMVoxel_s_rgb()
    {
        sdf = SDF_initialValue();
        w_depth = 0;
        clr = (uchar)0;
        w_color = 0;
    }
};
```

where the macro `.CPU_AND_GPU_CODE_` identifies methods and functions that can be run both as host and as device code and is defined as:

```cpp
#if defined(_CUDACC__) && defined(_CUDA_ARCH__)
#define _CPU_AND_GPU_CODE_ __device__ // for CUDA device code
#else
#define _CPU_AND_GPU_CODE_
#endif
```

Alternative voxel types provided along with the current implementation are based on floating point values instead of short, or they do not contain colour information. Note that the member `hasColorInformation` is used in the provided integration methods to decide whether or not to gather colour information, which has an impact on the processing speed accordingly.

Two examples for the index data structures that allow accessing the voxels are given below. Again, note that the processing engines choose different implementations according to the selected indexing class. For example the allocation step for voxel block hashing has to modify a hash table, whereas for a dense voxel array it can return immediately and does not have to do anything.

3.2 Dense Volumes

We first discuss the naive way of using a dense volume of limited size, as presented in the earlier works of [9, 7, 8, 11]. This is very well suited for understanding the basic workings of the algorithms, it is trivial to parallelise on the GPU, and it is sufficient for small scale reconstruction tasks.

In the InfiniTAM framework the class `ITMPlainVoxelArray` is used to represent dense volumes and a simplified listing of this class is given below:

```cpp
class ITMPlainVoxelArray
{
```
public:
struct ITM_VoxelArrayInfo {
  /// Size in voxels
  Vector3i size;
  /// Offset of the lower left front corner of the volume in voxels
  Vector3i offset;

  ITM_VoxelArrayInfo(void)
  {
    size.x = size.y = size.z = 512;
    offset.x = -256;
    offset.y = -256;
    offset.z = 0;
  }
};

typedef ITM_VoxelArrayInfo IndexData;

private:
IndexData indexData_host;

public:
ITMPlainVoxelArray(bool allocateGPU)
...
˜ITMPlainVoxelArray(void)
...

/** Maximum number of total entries. */
int getNumVoxelBlocks(void) { return 1; }
int getVoxelBlockSize(void) { return indexData_host.size.x * indexData_host.size.y *
                               indexData_host.size.z; }

const Vector3i getVolumeSize(void) { return indexData_host.size; }

const IndexData* getIndexData(void) const
...
};

Note that the subtype IndexData as well as the methods getIndexData(), getNumVoxelBlocks() and getVoxelBlockSize() are used extensively within the processing engines and on top of that the methods getNumVoxelBlocks() and getVoxelBlockSize() are used to determine the overall number of voxels that have to be allocated.

Depending on the choice of voxel type, each voxel requires at least 3 bytes and, depending on the available memory a total volume of about $768 \times 768 \times 768$ voxels is typically the upper limit for this dense representation. To make the most of this limited size, the initial camera is typically placed towards the centre of the rear plane of this voxel cube, as identified by the offset in our implementation, so that there is maximum overlap between the view frustum and the volume.

3.3 Voxel Block Hashing

The key idea for scaling the SDF based representation to larger 3D environments is to drop the empty voxels from outside the truncation band and to represent only the relevant parts of the volume. In [10] this is achieved using a hash lookup of subblocks of the volume. Our framework provides an implementation of this method that we explain in the following.

Voxels are grouped in blocks of predefined size (currently $8 \times 8 \times 8$ voxels). All the voxel blocks are stored in a contiguous array, referred henceforth as the voxel block array or VBA. In the current implementation this has a defined size of $2^{18}$ elements.

To handle the voxel blocks we further need:

- Hash Table and Hashing Function: Enable fast access to voxel blocks in the voxel block array – details in Subsection 3.3.1.
3.3.1 Hash Table and Hashing Function

To quickly and efficiently find the position of a certain voxel block in the voxel block array, we use a hash table. This hash table is a contiguous array of ITMHashEntry objects of the following form:

```cpp
class ITMHashEntry {
  /** Position of the corner of the 8x8x8 volume, that identifies the entry. */
  Vector3s pos;
  /** Offset in the excess list. */
  int offset;
  /** Pointer to the voxel block array. */
  -1 identifies an actual allocated entry in the voxel block array
  -1 identifies an entry that has been removed (swapped out)
  -1 identifies an unallocated block

  int ptr;
};
```

The hash function hashIndex for locating entries in the hash table takes the corner coordinates vVoxPos of a 3D voxel block and computes an index as follows [10]:

```cpp
template<typename T> inline int hashIndex(const InfiniTAM::Vector3<T>& vVoxPos, const int hashMask)
{
  return ((uint)((uint)vVoxPos.x * 73856093) ^ ((uint)vVoxPos.y * 19349669) ^ ((uint)vVoxPos.z * 83492791)) & (uint)hashMask;
}
```

To deal with hash collisions, each hash index points to a bucket of fixed size (typically 2), which we consider the ordered part of the hash table. There is an additional unordered excess list that is used once an ordered bucket fills up. In either case, each ITMHashEntry in the hash table stores an offset in the voxel block array and can hence be used to localise the voxel data for this specific voxel block. This overall structure is illustrated in Figure 3.

3.3.2 Hash Table Operations

The three main operations used when working with a hash table are the insertion, retrieval and removal of entries. In the current version of InfiniTAM we support the former two, with removal not currently required or
implemented. The code used by the **retrieval** operation is shown below:

```cpp
template<class TVoxel>
_CPUTAND_GPU_CODE_ inline TVoxel readVoxel(const TVoxel *voxelData, const ITMVolumeBlockHash::IndexData *voxelIndex, const Vector3i & point, bool &isFound) {
    const ITMHashEntry *hashTable = voxelIndex->entries_all;
    const TVoxel *localVBA = voxelData;
    TVoxel result; Vector3i blockPos; int offsetExcess;
    int linearIdx = vVoxPosParse(point, blockPos);
    int hashIdx = hashIndex(blockPos, SDF_HASH_MASK) * SDF_ENTRY_NUM_PER_BUCKET;
    isFound = false;
    // check ordered list
    for (int inBucketIdx = 0; inBucketIdx < SDF_ENTRY_NUM_PER_BUCKET; inBucketIdx++)
    {
        const ITMHashEntry &hashEntry = hashTable[hashIdx + inBucketIdx];
        offsetExcess = hashEntry.offset - 1;
        if (hashEntry.pos == blockPos && hashEntry.ptr >= 0)
        {
            result = localVBA[(hashEntry.ptr * SDF_BLOCK_SIZE3) + linearIdx ];
            isFound = true;
            return result;
        }
    }
    // check excess list
    while (offsetExcess >= 0)
    {
        const ITMHashEntry &hashEntry = hashTable[SDF_BUCKET_NUM * SDF_ENTRY_NUM_PER_BUCKET + offsetExcess];
        if (hashEntry.pos == blockPos && hashEntry.ptr >= 0)
        {
            result = localVBA[(hashEntry.ptr * SDF_BLOCK_SIZE3) + linearIdx ];
            isFound = true;
            return result;
        }
        offsetExcess = hashEntry.offset - 1;
    }
    return result;
}
```

Both **insertion** and **retrieval** work by iterating through the elements of the list stored within the hash table. Given a target 3D voxel location in world coordinates, we first compute its corresponding voxel block location, by dividing the voxel location by the size of the voxel blocks. Next, we call the hashing function `hashIndex` to compute the index of the bucket from the ordered part of the hash table. All elements in the bucket are then checked, with **retrieval** looking for the target block location and **insertion** for an unallocated hash entry. If this is found, **retrieval** returns the voxel stored at the target location within the block addressed by the hash entry. 

**Insertion** (i) reserves a block inside the voxel block array and (ii) populates the hash table with a new entry containing the reserved voxel block array address and target block 3D world coordinate location. If all locations in the bucket are exhausted, the enumeration of the list moves to the linked list in the unordered part of the hash table, using the `offset` field to provide the location of the next hash entry. The enumeration finishes when `offset` is found to be smaller or equal to $-1$. At this point, if the target location still has not been found, **retrieval** returns an empty voxel. **Insertion** (i) reserves an unallocated entry in the unordered part of the hash table and a block inside the voxel block array, (ii) populates the hash table with a new entry containing the reserved voxel block array address and target block 3D world coordinate location and (iii) changes the `offset` field in the previous entry in the linked list to point to the newly populated one.
The reserve operations used for the unordered part of the hash table and for the voxel block array use prepopulated allocation lists and, in the GPU code, atomic operations.

All hash table operations are done through these functions and there is no direct memory access encouraged or indeed permitted by the current version of the code.

4 Individual Method Stages

A general outline of the InfiniTAM processing pipeline has already been given in Section 2.1 and Figure 1. Details of each processing stage will be discussed in the following. The distinct stages are implemented using individual engines and a simplified diagram of the corresponding classes and their collaborations is given in Figure 4. The stages are:

- **Tracking**: The camera pose for the new frame is obtained by fitting the current depth (or optionally colour) image to the projection of the world model from the previous frame. This is implemented using the ITMTracker, ITMColorTracker and ITMDepthTracker engines.

- **Allocation**: Based on the depth image, new voxel blocks are allocated as required and a list of all visible voxel blocks is built. This is implemented inside the ITMSceneReconstructionEngine.

- **Integration**: The current depth and colour frames are integrated within the map. This is implemented inside the ITMSceneReconstructionEngine.

- **Swapping In and Out**: If required, map data is swapped in from host memory to device memory and merged with the present data. Parts of the map that are not required are swapped out from device memory to host memory. This is implemented using the ITMSwappingEngine and ITMGlobalCache.

- **Raycasting**: The world model is rendered from the current pose (i) for visualisation purposes and (ii) to be used by the tracking stage at the next frame. This uses the ITMSceneReconstructionEngine.

As illustrated in Figure 4, the main processing engines are contained within the ITMLib namespace. Apart from these, the UI and image acquisition engines (UIEngine, ImageSourceEngine and OpenNIEngine) are contained in the InfiniTAM namespace.
The ITMLib namespace also contains the two additional engine classes ITMLowLevelEngine and ITMVisualisationEngine that we do not discuss in detail. These are used, respectively, for low level image processing (e.g. gradients and computation of a resolution hierarchy) and for visualisation (currently only a conversion of depth images to uchar4).

In this section we discuss the tracking, allocation, integration and raycasting stages in greater detail. We delay a discussion of the swapping until Section 5.

4.1 Tracking

In the tracking stage the camera pose consisting of the rotation matrix \( R \) and translation vector \( t \) has to be determined given the RGB-D image and the current 3D world model. Along with InfiniTAM we provide the two engines ITMDepthTracker, that performs tracking based on the new depth image, and ITMColorTracker, that is using the colour image. Both of these implement the abstract ITMTracker class and have implementations running on the CPU and on CUDA.

In the ITMDepthTracker we follow the original alignment process as described in [9, 7]:

- Render a map \( V \) of surface points and a map \( N \) of surface normals from the viewpoint of an initial guess for \( R \) and \( t \) – details in Section 4.4
- Project all points \( p \) from the depth image onto points \( \bar{p} \) in \( V \) and \( N \) and compute their distances from the planar approximation of the surface, i.e. \( d = (Rp + t - \mathcal{V}(\bar{p}))^T N(\bar{p}) \)
- Find \( R \) and \( t \) minimising the linearised sum of the squared distances by solving a linear equation system
- Iterate the previous two steps until convergence

A resolution hierarchy of the depth image is used in our implementation to improve the convergence behaviour. Alternatively the colour image can be used within an ITMColorTracker. In this case the alignment process is as follows:

- Create a list \( V \) of surface points and a corresponding list \( C \) of colours from the viewpoint of an initial guess – details in Section 4.4
- Project all points from \( V \) into the current colour image \( I \) and compute the Euclidean norm of the difference in colours, i.e. \( d = \| I(\pi (R\mathcal{V}(i) + t)) - C(i) \|_2 \)
- Find \( R \) and \( t \) minimising the sum of the squared differences using the Levenberg-Marquardt optimisation algorithm

Again a resolution hierarchy in the colour image is used and the list of surface points is subsampled by a factor of 4. A flag in ITMLibSettings allows to select which tracker is used and the default is the ITMDepthTracker.

The two main classes ITMDepthTracker and ITMColorTracker actually only implement a shared optimisation framework, including e.g. the Levenberg-Marquardt algorithm and solving the linear equation systems. These are always running on the CPU. Meanwhile the evaluation of the error function value, gradient and Hessian is implemented in derived, CPU and CUDA specific classes and makes use of parallelisation.

4.2 Allocation

In the allocation stage the underlying data structure for the representation of the truncated signed distance function is prepared and updated, so that a new depth image can be integrated. In the simple case of a dense voxel grid, the allocation stage does nothing. In contrast, for voxel block hashing the goal is to find all the voxel blocks affected by the current depth image and to make sure that they are allocated in the hash table [10].

In our implementation of voxel block hashing, the aim was to minimise the use of blocking operations (e.g. atomics) and to completely avoid the use of critical sections. This has led us to doing the processing three separate stages, as we explain in the following.

In the first stage we project each pixel from the depth image to 3D space and create a line segment along the ray from depth \( d - \mu \) to \( d + \mu \), where \( d \) is the measured depth at the pixel and \( \mu \) is the width of the truncation
band of the signed distance function. This line segment intersects a number of voxel blocks, and we search for these voxel blocks in the hash table. If none of the blocks is not allocated yet, we find a free hash bucket space for it. As a result for the next stage we create two arrays, each of the same size as the number of elements in the hash table. The first array contains a bool indicating the visibility of the voxel block referenced by the hash table entry, the second contains information about new allocations that have to be performed. Note that this requires only simple, non-atomic writes and if more than one new block has to be allocated with the same hash index, only the most recently written allocation will actually be performed. We tolerate such artefacts from intra-frame hash collisions, as they will be corrected in the next frame automatically for small intra-frame camera motions.

In the second stage we allocate voxel blocks for each non-zero entry in the allocation array that we built previously. This is done using a single atomic subtraction on a stack of free voxel block indices i.e. we decrease the number of remaining blocks by one and add the previous head of the stack to the hash entry.

In the third stage we build a list of live voxel blocks, i.e. a list of the blocks that project inside the visible view frustum. This is later going to be used by the integration and swapping stages.

4.3 Integration

In the integration stage, the information from the most recent RGB-D image is incorporated into the 3D world model. In case of a dense voxel grid this is identical to the integration in the original KinectFusion algorithm [9, 7] and for voxel block hashing the changes are minimal after the list of visible voxel blocks has been created in the allocation step.

For each voxel in any of the visible voxel blocks, or for each voxel in the whole volume for dense grids, the function `computeUpdatedVoxelDepthInfo` is called. If a voxel is behind the surface observed in the new depth image by more than the truncation band of the signed distance function, the image does not contain any new information about this voxel, and the function returns without writing any changes. If the voxel is close to or in front of the observed surface, a corresponding observation is added to the accumulated sum. This is illustrated in the listing of the function `computeUpdatedVoxelDepthInfo` below, and there is a similar function in the code that additionally updates the colour information of the voxels.

```cpp
// In-line float computeUpdatedVoxelDepthInfo(TVoxel &voxel, Vector4f pt_model, Matrix4f M_d, Vector4f projParams_d,
          float mu, int maxW, float *depth, Vector2f imgSize)
{
    Vector4f pt_camera; Vector2f pt_image;
    float depth_measure, eta, oldF, newF;
    int oldW, newW;

    // project point into image
    pt_camera = M_d * pt_model;
    if (pt_camera.z <= 0) return -1;

    pt_image.x = projParams_d.x * pt_camera.x / pt_camera.z + projParams_d.z;
    pt_image.y = projParams_d.y * pt_camera.y / pt_camera.z + projParams_d.w;
    if ((pt_image.x < 1) || (pt_image.x > imgSize.x - 2) || (pt_image.y < 1) || (pt_image.y > imgSize.y - 2)) return -1;

    // get measured depth from image
    depth_measure = depth[(int)(pt_image.x + 0.5f) + (int)(pt_image.y + 0.5f) * imgSize.x];
    if (depth_measure <= 0.0) return -1;

    // check whether voxel needs updating
    eta = depth_measure - pt_camera.z;
    if (eta < -mu) return eta;

    // compute updated SDF value and reliability
    oldF = TVoxel::SDF_valueToFloat(voxel.sdf); oldW = voxel.w_depth;
    newF = MIN(1.0f, eta / mu);
    newW = 1;
}
```
newF = oldW * oldF + newW * newF;
newW = oldW + newW;
newF /= newW;
newW = MIN(newW, maxW);

// write back
voxel.sdf = TVoxel::SDF_floatToValue(newF);
voxel.w_depth = newW;

return eta;
}

The main difference between the dense voxel grid and the voxel block hashing representations is that the aforementioned update function is called for a different number of voxels and from within different loop constructs.

4.4 Raycast

As the last step in the pipeline, an image is computed from the updated 3D world model to provide input for the tracking at the next frame. This image can also be used for visualisation. The main process underlying this rendering is raycasting, i.e. for each pixel in the image a ray is being cast from the camera up until an intersection with the surface is found. This essentially means checking the value of the truncated signed distance function at each voxel along the ray until a zero-crossing is found, and the same raycasting engine can be used for a range of different representations, as long as an appropriate readVoxel() function is called for reading values from the SDF.

As noted in the original KinectFusion paper [9], the performance of the raycasting can be improved significantly by taking larger steps along the ray. The value of the truncated signed distance function can serve as a conservative estimate for the distance to the nearest surface, hence this value can be used as step size. To additionally handle empty space in the volumetric representation, where no corresponding voxel block has been allocated, we introduce a state machine with the following states:

```cpp
enum {
    SEARCH_BLOCK_COARSE,
    SEARCH_BLOCK_FINE,
    SEARCH_SURFACE,
    BEHIND_SURFACE,
    WRONG_SIDE
} state;
```

Starting from SEARCH_BLOCK_COARSE, we take steps of the size of each block, i.e. 8 voxels, until an actually allocated block is encountered. Once the ray enters an allocated block, we take a step back and enter state SEARCH_BLOCK_FINE, indicating that the step length is now limited by the truncation band \( \mu \) of the signed distance function. Once we enter a valid block and the values in that block indicate we are still in front of the surface, the state is changed to SEARCH_SURFACE until a negative value is read from the signed distance function, which indicates we are now in state BEHIND_SURFACE. This terminates the raycasting iteration and the exact location of the surface is now found using two trilinear interpolation steps. The state WRONG_SIDE is entered if we are searching for a valid block in state SEARCH_BLOCK_FINE and encounter negative SDF values, indicating we are behind the surface as soon as we enter a block. In this case the ray hits the surface from behind for whichever reason, and we do not want to count the boundary between the unallocated, empty space and the block with the negative values as an object surface.

Another measure for improving the performance of the raycasting is to select a plausible search range. If a sparse voxel block representation is used, then we are given a list of visible blocks from the allocation step, and we can render these blocks by forward projection to give us an idea of the maximum and minimum depth values to expect at each pixel. Within InfiniTAM this can be done using an ITMBlockProjectionEngine. A naive implementation on the CPU computes the 2D bounding box of the projection of each voxel block into the image and fills this area with the maximum and minimum depth values of the corresponding 3D bounding box of the voxel block, correctly handling overlapping bounding boxes, of course.

To parallelise this process on the GPU we split it into two steps. First we project each block down into the image, compute the bounding box, and create a list of 16 \( \times \) 16 pixel fragments, that are to be filled with
specific minimum and maximum depth values. Apart from a prefix sum to count the number of fragments, this is trivially parallelisable. Second we go through the list of fragments and actually render them. Updating the minimum and maximum depth for a pixel requires atomic operations, but by splitting the process into fragments we reduce the number of collisions to typically a few hundreds or thousands of pixels in a 640 × 480 image and achieve an efficiently parallelised overall process.

5 Swapping

Voxel hashing already enables much larger maps to be created, compared to the much simpler dense 3D volumes. Video card memory capacity however is often quite limited. Practically an off-the-shelf video card can roughly hold the map of a single room at 4mm voxel resolution in active memory, even with voxel hashing. This problem can be mitigated using a traditional method from the graphics community, that is also employed e.g. in [5, 3, 10, 6]. We only hold the active part of the map in video card memory, i.e. only parts that are inside or close to the current view frustum. The remainder of the map is swapped out to host memory and swapped back in as needed.

We have designed our swapping framework aiming for the following three objectives: (O1) the transfers between host and device should be minimised and have guaranteed maximum bounds, (O2) host processing time should be kept to a minimum and (O3) no assumptions should be made about the type and speed of the host memory, i.e. it could be a hard drive. These objectives lead to the following design considerations:

- O1: All memory transfers use a host/device buffer of fixed user-defined size.
- O2: The host map memory is configured as a voxel block array of size equal to the number of entries in the hash table. Therefore, to check if a hash entry has a corresponding voxel block in the host memory, only the hash table index needs to be transferred and checked. The host does not need to perform any further computations, e.g. as it would have to do if a separate host hash table were used. Furthermore, whenever a voxel block is deallocated from device memory, its corresponding hash entry is not deleted but rather marked as unavailable in device memory, and, implicitly, available in host memory. This (i) helps maintain consistency between device hash table and host voxel block storage and (ii) enables a fast visibility check for the parts of the map stored only in host memory.
- O3: Global memory speed is a function of the type of storage device used, e.g. faster for RAM and slower for flash or hard drive storage. This means that, for certain configurations, host memory operations can be considerably slower than the device reconstruction. To account for this behaviour and to enable stable tracking, the device is constantly integrating new live depth data even for parts of the scene that are known to have host data that is not yet in device memory. This might mean that, by the time all visible parts of the scene have been swapped into the device memory, some voxel blocks might hold large amounts of new data integrated by the device. We could replace the newly fused data with the old one from the host stored map, but this would mean disregarding perfectly fine map data. Instead, after the swapping operation, we run a secondary integration that fuses the host voxel block data with the newly fused device map.

The design considerations have led us to the swapping in/out pipeline shown in Figure 5. We use the allocation stage to establish which parts of the map need to be swapped in, and the integration stage to mark which parts need to swapped out. A voxel needs to be swapped (i) from host once it projects within a small (tunable) distance from the boundaries of live visible frame and (ii) to disk after data has been integrated from the depth camera.

The swapping in stage is exemplified for a single block in Figure 6. The indices of the hash entries that need to be swapped in are copied into the device transfer buffer, up to its capacity. Next, this is transferred to the host transfer buffer. There the indices are used as addresses inside the host voxel block array and the target blocks are copied to the host transfer buffer. Finally, the host transfer buffer is copied to the device where a single kernel integrates directly from the transfer buffer into the device voxel block memory.

An example for the swapping out stage is shown in Figure 7 for a single block. Both indices and voxel blocks that need to be swapped out are copied to the device transfer buffer. This is then copied to the host transfer buffer memory and again to host voxel memory.
All swapping related variables and memory is kept inside the ITMGlobalCache object and all swapping related operations are done by the ITMSwappingEngine.

6 Compilation, UI, Usage and Examples

As indicated in the class diagram in Figure 4, the InfiniTAM software package is split into two major parts. The bulk of the implementation is grouped in the namespace ITMLib, which contains a stand alone 3D reconstruction library for use in other applications. The InfiniTAM namespace contains further supporting classes for image acquisition and a GUI – these parts would almost certainly be replaced in user applications. Finally there is a single sample application included allowing the user to quickly test the framework.

The project comes with a Microsoft® Visual Studio® solution file as well as with a cmake build file and has been tested on Microsoft® Windows® 8, openSUSE Linux 12.3 and Apple® Mac® OS X® 10.9 platforms. Apart from a basic C++ software development environment, InfiniTAM depends on the following external third party libraries:

- **OpenGL / GLUT** (e.g. freeglut 2.8.0): This is required for the visualisation in the InfiniTAM names-
Figure 7: Swapping Out. The hash table entry at location 1 and the voxel block at location 3 are copied into the device transfer buffer at locations 2 and 4, respectively. The entire used transfer buffer is then copied to the host, at locations 5, and the hash entry index is used to copy the voxel block into location 7 inside the host voxel block array.

\[ \begin{array}{c}
\text{Device Voxel Memory} \\
\hline
\text{Hash Table} \\
\hline
\text{Transfer Buffer} \\
\hline
\text{Host Voxel Memory}
\end{array} \]

Finally the framework comes with a Doxygen reference documentation, that can be built separately. More details on the build process can be found in the README file provided alongside the framework.

The UI for our InfiniTAM sample application is shown in Figure 8. We display the raycasted reconstruction rendering, live depth image and live colour image from the camera. Furthermore the processing time per frame and the keyboard shortcuts available for the UI are displayed near the bottom. The keyboard shortcuts allow the user to process the next frame, to process continuously and to exit. Other functionalities such as exporting a 3D model from the current reconstruction and rendering the reconstruction from arbitrary viewpoints are not currently implemented. The UI window and interactivity is implemented in UIEngine and depends on the OpenGL and GLUT libraries.

Running InfiniTAM requires the intrinsic calibration data for the depth (and optionally colour) camera. The calibration is specified through an external file, an example of which is shown below.

```
640 480
504.261 503.905
352.457 272.202

640 480
573.71 574.394
346.471 249.031

0.999749 0.00518867 0.0217975 0.0243073
−0.0051649 0.999986 −0.0011465 −0.000166518
−0.0218031 0.00103363 0.999762 0.0151706

1135.09 0.0819141
```

This includes (i) for each camera (RGB and depth) the image size, focal length and principal point in pixels, as per the Matlab Calibration Toolbox [2] (ii) the Euclidean transformation matrix mapping points in the RGB camera coordinate system to the depth camera and (iii) the calibration converting Kinect-like disparity values...
to depths. If the depth tracker is used, the calibration for the RGB camera and the Euclidean transformation between the two are ignored. If live data from OpenNI is used, the disparity calibration is ignored.

We also provide two example command lines using different data sources, OpenNI and image files. These data sources are selected based on the number of arguments passed to InfiniTAM and for example in the bash shell with a Linux environment these are:

```
$ ./InfiniTAM Files/Teddy/calib.txt
$ ./InfiniTAM Files/Teddy/calib.txt Files/Teddy/Frames/(video,depth)%04i.ppm
```

The first line starts InfiniTAM with the specified calibration file and live input from OpenNI, while the second uses the given calibration file and the RGB and depth images specified in the other two arguments. We tested the OpenNI input with a Microsoft Kinect for XBOX 360, with a PrimeSense Carmine 1.08 and with the Occipital Structure Sensor. For offline use with images from files, these have to be in PPM/PGM format with the RGB images being standard PPM files and the depth images being 16bit big-endian raw Kinect disparities.

All internal library settings are defined inside the ITMLibSettings class, and they are:

```cpp
/// Use GPU or run the code on the CPU instead.
bool useGPU;

/// Enables swapping between host and device.
bool useSwapping;

/// Tracker types
typedef enum {
   /// Identifies a tracker based on colour image
   TRACKER_COLOR,
   /// Identifies a tracker based on depth image
   TRACKER_ICP
} TrackerType;
/// Select the type of tracker to use
TrackerType trackerType;
/// Number of resolution levels for the tracker.
```
The ITMSceneParams further contain settings for the voxel size in millimetres and the truncation band of the signed distance function. Furthermore the file ITMLibDefines.h contains definitions that select the type of voxels and the voxel index used in the compilation of ITMMainEngine:

```cpp
typedef ITMVoxel_s ITMVoxel;
```

For using InfiniTAM as a 3D reconstruction library in other applications, the class ITMMainEngine is recommended as the main entry point to the ITMLib library. It performs the whole 3D reconstruction algorithm. Internally it stores the latest image as well as the 3D world model and it keeps track of the camera pose. The intended use is as follows:

1. Create an ITMMainEngine specifying the internal settings, camera parameters and image sizes.
2. Get the pointer to the internally stored images with the method `GetView()` and write new image information to that memory.
3. Call the method `ProcessFrame()` to track the camera and integrate the new information into the world model.
4. Optionally access the rendered reconstruction or another image for visualisation using `GetImage()`.
5. Iterate the above three steps for each image in the sequence.

The internally stored information can be accessed through member variables `trackingState` and `scene`.

## 7 Conclusions

We tried to keep the InfiniTAM system as simple, portable and usable as possible. We hope that the external dependencies on CUDA, OpenNI, OpenGL and GLUT are easy to meet and that the cmake and MSVC project files allow users to build the framework on a wide range of platforms. While the user interface and surroundings are fairly minimalistic, the underlying library should be reliable, robust and well tested.

We also tried to keep an eye on expandability. For example it is fairly easy to integrate a different form of tracking into the pipeline by reimplementing the ITMTracker interface. Also different media for swapping can be made accessible by replacing the ITMGlobalCache class. Finally it is still a manageable process to port InfiniTAM to different kinds of hardware platforms by reimplementing the Device Specific Layer of the engines.

While we hope that InfiniTAM provides a reliable basis for further research, no system is perfect. We will try to fix any problems we hear of and provide future updates to InfiniTAM at the project website.
References


