

WEB-BASED VISUALISATION OF ON-SET POINT CLOUD DATA

Alun Evans^{*}

Interactive Technologies Group
Universitat Pompeu Fabra
138 Roc Boronat, Barcelona,
08018, Spain
alun.evans@upf.edu

Javi Agenjo

Interactive Technologies Group
Universitat Pompeu Fabra
138 Roc Boronat, Barcelona,
08018, Spain
javi.agenjo@upf.edu

Josep Blat[†]

Interactive Technologies Group
Universitat Pompeu Fabra
138 Roc Boronat, Barcelona,
08018, Spain
josep.blat@upf.edu

ABSTRACT

In this paper we present a system for progressive encoding, storage, transmission, and web based visualization of large point cloud datasets. Point cloud data is typically recorded on-set during a film production, and is later used to assist with various stages of the post-production process. The remote visualization of this data (on or off-set, either via desktop or mobile device) can be difficult, as the volume of data can take a long time to be transferred, and can easily overwhelm the memory of a typical 3D web or mobile client. Yet web-based visualization of this data opens up many possibilities for remote and collaborative workflow models. In order to facilitate this workflow, we present a system to progressively transfer point cloud data to a WebGL based client, updating the visualisation as more information is downloaded and maintaining a coherent structure at lower resolutions. Existing work on progressive transfer of 3D assets has focused on well-formed triangle meshes, and thus is unsuitable for use with raw LIDAR data. Our work addresses this challenge directly, and as such the principal contribution is that it is the first published method of progressive visualization of point cloud data via the web.

Categories and Subject Descriptors

I.3.4 [Computer Graphics]: Graphics Utilities; I.3.4 [Computer Graphics]: Computational Geometry and Object Modeling

Keywords

Visualisation, 3D, pointcloud, progressive, WebGL.

1. INTRODUCTION

Point clouds are frequently used to represent 3D surface data extracted using acquisition techniques such as laser range scans. For example, Light Detection and Ranging (LIDAR) is a remote sensing technique that uses the time difference between the transmis-

sion of a laser pulse and the detection of its reflection to calculate the position of observed point on the surface of an object. The information gathered from the pulses is then amalgamated and converted into a topological cloud of points in 3D space, which can be stored on disk in one of several formats (for example, LAS or OFF). Modern film production methods are now recording data in several different modalities, in order to assist and speed up the post-production process. The use of LIDAR scanning on-set is one example of this. LIDAR machines will be set up on-set to record the filming environment, which can then be viewed later to assist in post-production, whether as a visual aid to assist in the creation and/or lighting of digital assets, or a ground truth to assess registration techniques involving multiple cameras, or simply to provide better context of the set to the post-production staff. In parallel, modern business workflow is gradually moving towards a cloud-based, remote working model, which allows people to access data wherever they may be in the world, and regardless of the machine (or even platform) that they are using. Nevertheless, this move to cloud/web based workflow has been slower to reach the post-production world, mostly due to two factors: the first is the concern over security of the remote transfer of sensitive production data; and the second concerns the technical difficulties which arise in the remote transfer of digital assets which may reach terabytes in size.

In this paper we address the second issue, particularly with reference to point cloud data recorded on set. We create an efficient hierarchical representation of this data, designed for use within a server-client 3D context, with the goal of enabling a user to rapidly view a lower resolution visualisation of a large point cloud, whose resolution is then increased as more data is downloaded. We also present implementation details of the WebGL rendering techniques used to enable fast and smooth updating of the 3D scene, with two different modalities. Refining the data as it is downloaded is not trivial: simply 'drawing over' existing low-resolution data is not sufficient as it is visually unattractive and (more importantly) not representative of the detail of the scene. Low resolution data must be culled from the scene when and only when sufficient higher resolution data has been downloaded in the local area, a factor which requires constant monitoring of the current status of the visualization.

The principal contribution of this work is that it is the first published method of progressive visualization of point cloud data via the web. The technique is designed specifically to address the challenges presented above: we distinguish our work from established techniques for progressive mesh transfer [10, 8], as the work required to convert on-set LIDAR data to a high quality mesh can be slow and labour intensive; and we further distinguish our work from estab-

^{*}Corresponding author

[†]<http://gti.upf.edu>

lished methods of point cloud compression and visualization [16, 6], as they either require substantial offline pre-processing or are processor/memory intensive at the time of decompression, which is not suitable for a web context. We show results of applying our techniques to six different point cloud datasets, acquired using multiple LIDAR on-set scans, each comprising of between 4.5 million and 7.5 million points, with varying point densities. Particularly, we show what effect varying parameters, such as file size and bandwidth, has on the performance of the system.

2. RELATED WORK

Although laser scanning can generate highly accurate point clouds of a surface, the resulting data can consist of millions of 3D points (and associated colour information). The large file sizes involved present considerable problems in terms of the storage and transmission of the data, particularly over the Internet. As a result, several research efforts have been made to compress pointcloud data. Mongus and Zalik [12] present an efficient method for loss-less compression of laser scan data coming from a LIDAR, applying three encoding steps to compress the data to 12% of its original size. Merry et al. [11] present a more generic compression approach by building a spanning tree and predicting point positions based on the location of their ancestors. Huang et al. [7] use an octree based approach to abstract the point cloud data, as do Schnabel and Klein [17]; both approaches built on seminal work by Gandoian and Devillers [6] and Peng and Kuo [14]. Despite these efforts, the challenge of real-time progressive visualisation of (very large) point clouds over the web has seen little direct research [15], perhaps due two key difficulties that need to be overcome; namely, the progressive transfer of 3D data, and the fast update of the 3D data in the browser. As Limper et al. [9] demonstrate, the balance between data compression rate (which decreases transfer time) and complexity (which increases decompression time) is by no means straightforward for browser-based contexts, which inherently rely on Javascript to deal with any received data. These challenges were addressed very effectively by recent research into progressive meshes over the web [10, 8], showing that it is possible for browser based 3D engines to effectively handle and present such data to the user. For more background on these issues and others relating to web-based 3D rendering, see the recent detailed survey by Evans et al. [5].

Visualisation of point cloud data at increasing levels of detail (in an offline context) is a subject that has seen much research, particularly with regards to the extremely high resolution laser scanning of physical artefacts. The Digital Michelangelo project resulted in a technique called QSplat [16] where extremely large points cloud datasets (130 million points) could be visualized offline at progressive levels of detail using a hierarchical technique based on bounding spheres. More information about the simplification of point cloud surfaces for rendering purposes can be found in [13]. Straightforward simplification of point cloud surfaces is not necessarily suitable for progressive visualization in a remote client (the task addressed in this paper), as it may not take into account issues regarding bandwidth and decompression speed. On the other hand, existing methods of asset transfer that do take into account these issues are restricted to progressive simplifying the topology of well-formed triangular meshes, and thus do not extend to other forms of 3D data such as point clouds. The novelty of our contribution in this paper is to directly address these issues by presenting a system of progressive transmission and visualisation of large point cloud data in a web based rendering environment.

3. GENERAL APPROACH

We aim at providing a system that allows progressive refinement of the visualisation of large point cloud datasets over the web. First, we use an offline process to store the point cloud data into hierarchical data structure, then save that hierarchical structure into a series of small files, which are made available for download by the browser client (and subsequent visualization). This approach opens up two possibilities. The first is to treat the scene holistically, downloading the octree in a breadth-first manner without any reference to camera position within the client render. By downloading the higher levels of the hierarchy first, the client is able to immediately display a low-resolution version of the point-cloud, and progressively refine it as files containing lower levels of the hierarchical structure are downloaded. The second possibility is to use the camera position within the octree to prioritise which data to download (with data closer to the camera being downloaded first). For this technique, a series of Level-of-Detail (LOD) spheres are projected from the camera position, and the octree is traversed depth-first to prioritise the download of the higher resolution data nearer the camera.

We have implemented both options, and sections 4 and 5 highlight the differences. Let us remark that, as usual with browser-based systems, once data are in the cache, the higher resolution data is provided much faster. To render the data in the browser, we treat each node of the octree as a single 3D point to be drawn to the screen, whose dimensions are proportional to the size of the octree node, and whose colour is equal to the average colour of all the nodes (and, ultimately, points) which lie lower down the hierarchy. The effect of this approach is that the initial lower-resolution data has a 'blocky' look, but this is quickly refined as higher resolution data is downloaded and the draw buffers is updated.

4. GENERAL APPROACH

4.1 Insertion of points

We use an octree to abstract the point cloud structure, as it has been previously demonstrated to be an effective method of storing such data [17, 1]. Our approach associates points to the nodes of an octree in an add, split and disperse manner, which minimises memory usage (and thus storage). Given the bounding cube of a point cloud P , our octree O is initialised as a single root node whose half-width HW and centre point C are equal to those of the bounding volume. The root node, and any potential child nodes, can store a maximum number of points which is equal to a threshold T (e.g. if $T = 20$, each node can store a maximum of 20 points). The points of P are added to O one-by-one, starting at the root node. Once the number of points stored by a node is greater than T , a split operation is executed; the child octant location for each of the points associated with the node is discovered (based on each point position) and a child node is created for each octant which contains a point. Finally, all the points associated with the parent node are dispersed and associated to the relevant newly created child nodes. Any further points which are added to the parent node are automatically dispersed to the child nodes, where the entire operation is repeated recursively. This top-down method of point insertion ensures two properties which are important for the future storage and distribution of the octree. First, each node has no more than T points associated with it (thus there are no overly 'heavy' nodes); and second, every node contains either at least one associated point, or at least one child node (thus there are no 'redundant' nodes, that waste memory and thus storage space).

4.2 Colour assignment

Once all the points of P are added to the octree, we employ a bottom-up approach to calculate the representative color of each node in the octree. For each bottom-level node (i.e. with points associated to it), the average colour of the associated points is calculated. Then, by recursively parsing the octree in a depth-first manner, we can calculate the colour of each node by averaging the stored colour for each of its child nodes (remembering that every node must have either child nodes or associated points). At the end of this process, every node in the octree comprises of a position value corresponding to the centroid of the node, and a colour value corresponding to the mean colour of all the points stored in all of the nodes below it. The halfwidth value of the node (which, at render time, is used to calculate the size of the point drawn to representing the node) is not stored directly, as the general halfwidth value for each depth in the tree can be easily calculated from the halfwidth of the root node only (stored separately, see below). Each node also has associated with it either a linked list of points or between one and eight pointers to child nodes.

4.3 Storage Structure

In order to store the octree and point cloud for progressive transfer, a breadth first traversal of the octree saves the data for each node or point to a sequence of files. The storage structure needs to be as efficient and data-light as possible, in order to maximise transfer speed and data management in the browser (via Javascript). A breadth-first traversal lead to two important results for data transfer: first, leaves are ordered by depth i.e. all leaves in one level of the octree are saved before descending to the next level; and second, the multiple files are saved with precise sizes: when saving the octree to disk, we can set a maximum size file-threshold for each file, and thus guarantee an array of files with constant size. For example, if we set the threshold at 5000 node/point entries per file, and each entry occupies 17 bytes (see below) then we know that each file will occupy 85 kilobytes of disk space. If a leaf does not have any children, we save the position and colour information of the points (of P) associated with that node. Beyond this common structure, the requirement to minimise the amount of data transferred means that the two different techniques as described in Section 3 require different approaches to storage. The holistic, progressive update of the entire scene (described in Section 5.1) does not, in fact, require any detail of the octree - it is never actually traversed by the client, the data is merely being downloaded and displayed. The significance of this is that we only transfer a pared-down amount of data for each node, in order to increase the speed of the visualisation. By contrast, the second technique, which prioritises download of data nearer the camera, does require the client to traverse the octree. This means that the octree data structure must be recreated in the browser client: parent-child relationships must be downloaded for each node, which means greater storage requirements. Thus, below we define two custom binary formats that are designed to optimise data transfer for each technique.

4.3.1 Progressive Transfer of entire scene

The data for each node is stored in a 17-byte binary chunk, as detailed in Figure 2. The first byte is an 8-bit signed integer which encodes the node 'depth' in the octree hierarchy (e.g. the root node depth is 0; its eight children depths are 0). It is used to ensure the lower resolution nodes are not rendered once data for higher resolution nodes are downloaded (see Section 4 below). The next 12 bytes represent the x, y and z coordinates of the central point of the node (each dimension stored as a 4-byte float value). The following

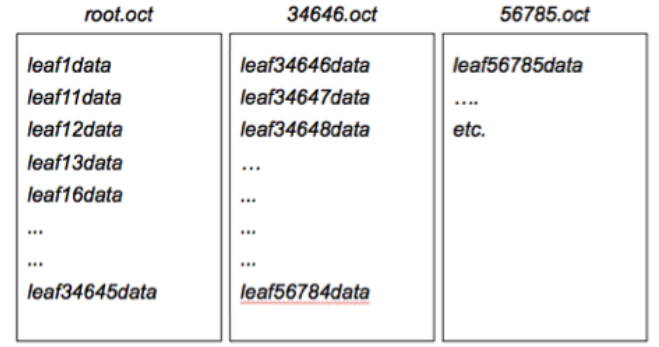


Figure 1: Representation of file storage structure. The figure contains representation of three files, each of which contains entries for the same number nodes in the octree, and each is the same size.

3 bytes store RGB colour information as a series of 8-bit unsigned integer values (alpha is not stored in our implementation, as all alpha values in our source files were set to 1.0). The final 8-bits are used as a bit mask (in Morton order [4]) to indicate whether the node has any child nodes and, if so, in which octant - again this information is used to decide whether to render a lower-resolution node, as described in Section 5.

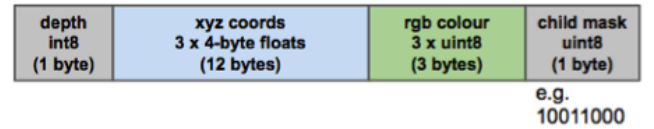


Figure 2: Structure of binary data representing a single node for whole scene visualisation

When storing information regarding a point of P, as opposed to a node, the exact same structure is used, except that the depth value is set to -1. This flag enables the browser client parsing the data to quickly identify this particular entry as a point. Each file is named in simple numerical order, as we intend to download each one in turn. Finally, we create an index file (in ascii text) which stores the total number of the node/point entries, and the half-width of the octree root node.

4.3.2 Progressive Transfer of Data Nearest Camera

As this technique requires transfer of the octree hierarchy, it involves transferring more information. Firstly, in the offline process the octree is transferred into a linear array (ordered breadth-first). The advantage of using a linear array is that it allows fast index based access to that node (i.e. without having to carry out depth-first traversal of octree), and this becomes particularly important when recreating the octree structure in the browser client. With each node is associated both its own index in the array, and the index of each of its child nodes. Our binary structure for each node is presented in Figure 3. The first 8-bits code the depth of the node in octree, as before. The following 4 bytes code a 32-bit integer with the node's index in the linear array. Position, colour, and child octant bitmask information are stored as before. The final 32 bytes are used to store the array index of each child of the node. There is a potentially some redundancy here, as our memory efficient octree is not guaranteed to have eight children for every node. Yet we have chosen this approach for simplicity with respect to parsing the

data the data in browser client (as we know that every single entry is a fixed number of bytes). In this case, as we intend to change the file download order based on camera position in the scene, a robust file-naming structure and index are required. We name each file according to the index of the first node stored within it (e.g. if the first entry corresponds to the node at index i in the linear array, we name that file 'i.oct'). As before, we store an index file with total number of entries and root node half-width. However, this index file also now stores a list of all filenames.

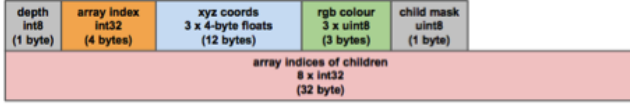


Figure 3: Structure of binary data representing a single node for progressive transfer of data nearest camera

4.4 Use of HTTP compression

As well abstracting the point cloud data, we can make use of pure compression methods that are built into the HTML specification. By ensuring that gzip compression is enabled on the web-server, and knowing that it is supported by default on all modern web browsers, we can apply fast (<10ms) compression and decompression of the files to be transferred. As an example, an 425kb binary file as specified in our format, when compressed using gzip to level 6, usually results in a 246kb file (individual file sizes vary depending on their content). As the gzip compression and decompression algorithms are coded at a very low-level into the core of the both the web server and browser client software, this provides us with a 40% compression in exchange for minimal processing time. Our tests show that applying a gzip compression level higher than 6 made no difference to the resulting file sizes.

5. DATA PARSING AND RENDERING

For rendering we use a custom WebGL engine created specifically to update the scene content as more data is downloaded. The challenge when creating such an engine in WebGL, which will have to deal with millions of nodes/points, is to be able to both effectively manage memory (both in the browser and on the GPU), and also update the draw buffers quickly enough so as not to provide any delay to the user. Last, but not least, the engine should attempt to render the lower-resolution levels of the octree in a coherent manner, such that the viewer is able to obtain some idea of the structure of the scene even while additional data is being downloaded (much in the same manner as progressive meshes).

This final challenge is not simply a case of drawing higher resolution data on top of the lower resolution data. Such an approach, while trivial to implement, would leave the undesirable effect of large, lower-resolution points nearer the camera obscuring the view of smaller, higher resolution points further away (due to the action of the z-buffer). Thus, we must construct our data buffers in order to:

1. Always draw the highest resolution which has been completely downloaded, in order to display the maximum amount of information to the user
2. Avoid drawing nodes at a lower level (lower resolution) so as not to occlude higher resolution data

3. Ensure there are no 'holes' in the scene in the situation where a node does not have any children

We know that the octree data is stored (and will be downloaded) breadth-first; thus, for each new node N of the octree which is read by the client, let $\text{currDepth} = (\text{depth of } N) - 1$.

The files of the data structure are downloaded in the numerical order in which they were stored. Once downloaded, each node/point entry is stored (along with its relevant properties) as a Javascript object, and added to an array. Once a file is completely downloaded, the WebGL draw buffer is updated with the new data (quickly, using the BufferSubData command, and avoiding a full flush of the draw buffer). Each time currDepth is incremented, and in order to comply with points i. and ii. above, we reset the draw buffer and reparse the downloaded Javascript array of octree objects, adding only the nodes of the lowest fully-downloaded depth level. The octree design we employ means that it is perfectly possible (and indeed likely) to have a single low-resolution node containing data for a sparse set of points. This presents a potential issue if we wish to comply with point iii. above, as such a node does not have any children, and simply stores the data for these sparse points. To avoid any potential display problems, as soon as any point information (i.e. original data from the LIDAR) is downloaded (marked with depth value of -1) it is immediately and permanently added to the draw buffer. The process ends in one of two ways, either when the full point cloud is downloaded, or at some user-set termination level. In practice, if the end goal is to provide an attractive representation of the point cloud, then downloading the full dataset may not be necessary. This is especially the case for datasets comprising several millions of points, which results in poor performance due to the high memory requirements of the browser process (detailed in Section 6).

5.1 Update based on camera position

While the principal application of the work in this paper is to visualise the entire set, we have also implemented a system to prioritise the download of higher resolution data that is nearer the camera, and leave data that is further from the camera at a lower resolution. We define two spheres around the location of the camera, each sphere associated with a minimum display depth value (i.e. data within the sphere should be downloaded to a minimum resolution). The minimum depth values are correlated to sphere size, such that nodes/points found within the smaller sphere will be preferentially downloaded over nodes found within the larger sphere, which likewise are preferentially downloaded over all other data. Our system features a free moving camera, which is controlled by the user via a combination of keyboard and mouse input. The system periodically parses the octree recursively in a depth-first manner to find any cells intersecting with either of the spheres. It then cross-references the index of these cells in the linear array with the list stored in the index file. This cross referencing results in a list of files which contain the relevant depth information, ordered low-resolution to high-resolution. Note that, in this scenario, there is no concept of current depth (as in the previous scenario), as the depth of the nodes displayed in the scene is not uniform (as it depends on camera position). This means that we are unable to progressively update the draw buffers as in the previous (whole scene) technique, and must update the entire draw buffer after each file has been downloaded. Thus, after a file has been downloaded there is a brief (but noticeable) slow down while new buffers are uploaded to the GPU.

6. RESULTS

The work in this paper is designed to visualise large point cloud datasets in a web based environment. To test our methods, we used six datasets, each containing roughly between 4.5 and 7.5 million points. The datasets were constructed from LIDAR scans of different environments (occasionally several LIDAR scans of the same environment combined into a single dataset). Each set was presented to our system as a single text file in OFF format, where each line contained the position and colour of a single point. Table 1 lists the relevant details for each dataset. Each dataset was processed and inserted into an octree, via a C++ process, whose output was a series of files as described in Section 4. The threshold T of maximum points in any node was set to 20 (though this was varied in a future experiment, as described below). All results in the paper were obtained using Google Chrome v.33, running on a 2.6GHz Core i7 Macbook Pro, with 8GB of RAM, and an Nvidia Geforce 650M graphics card with 1GB of RAM.

Table 1: Details for input datasets

ID	# Points	Input File Size (MB)
1	7,325,251	290.5
2	4,647,968	181.2
3	5,952,681	235.0
4	6,757,348	340.6
5	4,837,404	249.2
6	6,641,100	334.6

Before embarking on any complete tests regarding download speed from a remote server, we wanted to assess the capability of the browser to manage and render any one of the datasets when stored locally (our hypothesis being that that the memory and data processing issues would mean that attempting to parse and render the full dataset would overwhelm the browser). We created a simple, offline, point cloud renderer application in OpenGL, and confirmed that each dataset could be opened and viewed at acceptable framerates. However, parsing and displaying the entire dataset in a WebGL version of that application, loading data from local storage, rendered the browser process inoperative. Further tests showed that progressively loading the data maintained a steady 60 fps average until reaching 3.5 million points, beyond which the framerate dropped suddenly to unusable levels (suggesting an issue relating to browser memory rather than GPU). Thus, for the results in this paper, we set a maximum number of points as 3.5 Million.

6.1 Progressive visualisation of whole scene

Table 2: Mean time taken (in milliseconds) to download and render the point clouds at three resolution levels, and at three different bandwidths (megabits per second)

Display Level (Av. points)	1mbps	8mbps	64mbps
6 (1958)	2263	886	565
8 (48182)	10912	3203	2517
Final (3.5M)	393287	95072	61590

Figure 4 presents data on the average time (in milliseconds) required to download and render all the datasets at different resolution levels. Table 2 shows the raw figures for the graph. Figure 5 shows visual results of the different resolutions for a single dataset. For all datasets, we consider an octree depth level of 6 (i.e six subdivisions of initial bounding volume) to be acceptable resolution

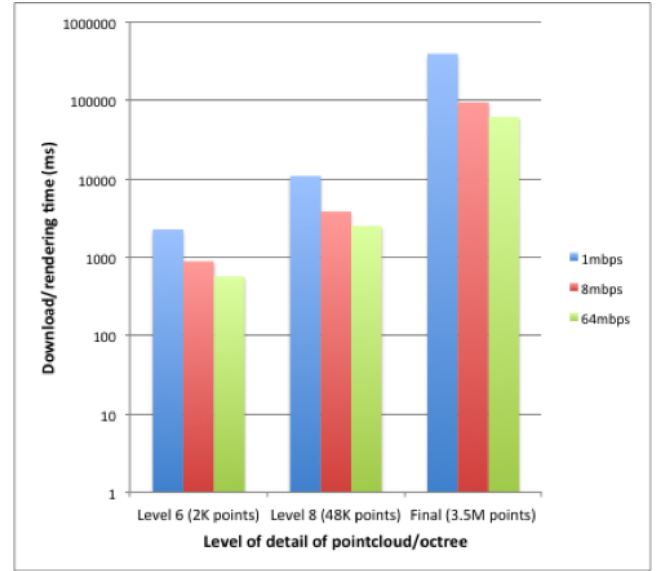


Figure 5: Mean time taken (logarithmic scale, in milliseconds) to download and render the point clouds at three resolution levels, and at three different bandwidths (megabits per second). Chunk size for all data was 425kB. Figure 5 shows the visual results of the different levels on one dataset.

for use as a 'preview' of the scene; level 8 to be acceptable as an approximation of the scene, and the final level of detail is when the number of points $\leq 3.5M$.

Table 3: Effect of changing the chunk (or file) size on the mean time taken (in milliseconds) to download and render the point clouds at three resolution levels, at a bandwidth of 8mbps

Display Level (Av. points)	85kb	425kb	1700kb
6 (1958)	452	886	2397
8 (48182)	5262	3203	5599
Final (3.5M)	154581	95072	85746

The results show that, with a moderate bandwidth of 8 megabits per second, an initial approximation of the point cloud is visible on screen within less than a second, and in fewer than four seconds it has been refined to a level that we consider acceptable for usage as an approximate visualisation of the scene. The full resolution pointcloud, capped at 3.5 million points, appears after 95 seconds, however the scene is not static during that time, it is being constantly refined. One of the main variables in the system is the file-threshold value discussed in Section 4.3 above. This value essentially determines the size in bytes of each file (each chunk of data) that is downloaded. Table 3 presents data showing the effect of varying the file size, from 85 kilobytes (KB) (5000 node entries per file) to 425KB (25000 entries per file) to 1700KB (100000 entries per file), at a bandwidth of 8mbps. As might be expected, a smaller file size displays the lower resolution data faster (as the first 6 levels of information of the octree are present in the first one or two files). However, as more files are downloaded, the overhead involved in making multiple HTTP requests begins to become more of a factor, and when using a larger file size, the 3.5M point limit is reached in almost half the time. The data in the table suggest that a file size in the region of 425 kilobytes provides a balance between being small



Figure 4: Examples of a dataset being progressively rendered. The far left image shows the dataset at a depth of 6, the centre image at a depth of 8, and the far right image once the dataset has finished downloading (with 3.5M points). All images are screenshots from Google Chrome v.33

enough to quickly display lower resolution data and large enough to not slow down performance due to unnecessary parsing and updating the draw buffer. Viewing the results, one natural conclusion would be to gradually increase the file size as the depth of the octree increases, this is discussed as part of future work below.



Figure 6: Progressive loading of higher resolution data based on distance from camera

6.2 Update based on camera position

Figure 6 shows screenshots of the system progressively loading higher resolution data as the camera moves through the scene. One particular difficulty in this implementation is the speed at which the camera is permitted to move (as discussed above). In this implementation we simply limit the movement speed of the camera (to simulate as if the user were walking through the environment).

6.3 Effect of number of points per node

As proposed in Section 4.1, the maximum of number of points that can be added to each node before they are dispersed to child nodes is controlled by a threshold T . Varying this threshold naturally con-

trols the depth of octree, and the effect of this can be seen in Figures 7 and 8. If T is set too high, then the finer details of the point cloud might never be downloaded, due to the restriction on the maximum number of points. Figure 7 shows how increasing T to a value of 100 means that the final rendering of the scene loses final detail, as the octree did not create nodes of sufficient resolution. Figure 7 shows the same view with $T = 20$, showing the finer detail better represented. The rendering of both versions of the dataset was capped to 3.5M points, as described above.



Figure 7: Effect of setting $T=100$ (final visualisation).

6.4 Buffer update performance

One of the most important characteristics of our system is the time required to update the draw buffer. This operation blocks the GPU and thus, if it takes too long and/or happens too frequently, then the user experience is unpleasant. We measured the average time taken (for all datasets) for the update function to upload new data to the GPU once each file had finished downloading. For accuracy, measurements were made using the `window.performance.now()` function, as specified by the W3C Web Performance working group [18]. Table 4 presents the average update time for the three dif-

Table 4: Draw buffer update time for different file sizes

File Size (kb)	Average update time (ms)
85	11.73
425	46.83
1700	214.53

ferent file sizes tested in Section 6.1. Unsurprisingly, the update times for the larger file sizes are greater than those for the smaller file sizes. For the smallest file size, and if the application is running at 60 frames a second, the update time is less than the time to render a single frame, and in practice is barely noticeable. On the other hand, the average update time for the largest file size is nearly a quarter of a second, which is impossible to ignore. The update time for 425kb represents a middle ground, but 46ms is the equivalent of several dropped frames and is clearly noticeable.

6.5 Summary of results

Table 5 presents a summary of all the timings involved in our technique, for moderate bandwidth of 8mbps and using a 425kB file size. In the table, we break down the timings for for each step in order to provide an overview of the bottlenecks in the system.

7. DISCUSSION AND FUTURE WORK

It is difficult to directly compare this work with that previously published for two reasons. Firstly, our goal is not the pure compression (and subsequent decompression and display) of point cloud data; rather it is the fragmentation of that data in manner that permits its fast transfer over a network to a remote visualisation client. Secondly, the majority of the work carried out in this particular field has concerned progressive transfer of well-formed mesh data, which can be time-consuming to create directly from LIDAR data, as previously stated. Nevertheless, we can make some comparisons based on published results. Table 6 shows comparative statistics between the Progressive Mesh transfer of Lavoué et al [8], using the same bandwidth (5mbps). The results show that our technique is equivalent (or marginally faster) to download and render both low resolution and medium resolution data (high resolution data for Lavoué et al. is not available). While Lavoué et al. must deal with the correct insertion of vertices during this time, our technique must equally deal with the culling of higher level (lower resolution) nodes of the octree. As mentioned above, the use of progressive meshes over the web is perhaps the most well-established method of progressively transferring 3D assets for web visualisation [10, 8]. Yet the core technique of progressive meshes cannot be applied to point clouds. Addressing this issue, while still retaining the benefits of progressive visualisation of 3D data in a browser based context, is the principal contribution of the paper. The work particularly addresses, and is suitable for, situations where laser scans of environments are recorded using a LIDAR scanner or similar (for example, in the case of a film production set, or mapping or geographical features).

Table 6: Summary of mean results for all datasets (with sample images for each level). Bandwidth was clamped to 8mbps and the file size used was 425kB

	Low Resolution			Medium Resolution		
	# verts	time (ms)	Rate (verts/ms)	# verts	time (ms)	Rate (verts/ms)
Lavoué13	1500	800	1.875	38K	3000	12.666
Our work	1958	938	2.087	48K	3781	12.743

The results show that, unsurprisingly, a smaller file size (chunk of data to be transferred) results in faster appearance of a low-resolution representation of the point cloud, particularly at low bandwidths. This highlights two main benefit of this work: the fast visualisation of large point cloud data in a web environment, without having to wait for the entire data set to download; and the ability to simplify a large point cloud to a level where it can be rendered in a WebGL context without performance issues. Furthermore, a smaller file size provides a better user experience as there is only minimal delay when updating the draw buffers. Nevertheless, the increased number of petitions to the server that are required when using a smaller file size means that the total time to download the dataset is considerably longer. One possibility for future research is to take the best of both worlds and gradually increase the file size as the depth of the octree increases, thus the lower resolution data is displayed very quickly, while higher resolution data is transferred more efficiently.

Despite the contribution and promising results, we note that our current solution, as proposed in this paper, is far from optimal. Apart from visualisation, one of the primary reasons to record data with a point cloud is to extract topological information; which is notably lacking from our approach. We also have made little effort to truly understand the limits on the rendering of points in a browser engine. We intend to explore these avenues in our future work. Creating mesh topology from point clouds it is an issue which has seen much research [2]. Such meshes could be used in a progressive meshes approach to remote visualisation, bypassing the need for point cloud visualisation. On the other hand, we carried out several tests using the popular meshing/visualisation software MeshLab [3] and found that creating high quality meshes of our input point cloud data (specifically with regards to their topology) required a considerable amount of manual intervention. Our future work will focus on creating high quality reference meshes for our input data, which will then enable us to carry out a critical comparison between progressive transfer of point clouds and meshes. We also plan to investigate the possibility of using node information to create meshes of the data: by carrying out Principal Component Analysis on the points stored within a node, the eigenvectors can be used to estimate a normal vector for a putative plane representing those points. This plane could be used to visualise the node (instead of a single point), which would then allow us to essentially create a mesh from the octree.




8. ACKNOWLEDGEMENTS

This work has been partially funded by the Spanish Ministry of Science and Innovation (TIN2011-28308-C03-03), and the IMPART FP7 European Research project (<http://impart.upf.edu>).

9. REFERENCES

- [1] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proceedings of the 13th Eurographics Workshop on Rendering*, EGRW '02, pages 53–64, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.
- [2] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3D objects with radial basis functions. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, pages 67–76, New York, New York, USA, Aug. 2001. ACM Press.

Table 5: Summary of mean results for all datasets (with sample images for each level). Bandwidth was clamped to 8mbps and the file size used was 425kB

			
	Level 6	Level 8	Final
Mean dataset size (MB)	255.18 MB		
Mean dataset pointsize	6,026,958		
Mean preprocessing time (offline, full datasets)	12 seconds		
Mean bytes transferred (compressed, approx)	13.6kB	335.6kB	23.8MB
Mean number of points	1958	48128	3,500,000
Mean Transfer time (8mbps)	886ms	3.2s	95.1s
Mean framerate (60fps limit)	57fps		
Mean buffer update time	46.83ms		

- [3] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia. Meshlab: an open-source mesh processing tool. In *Eurographics Italian Chapter Conference*, pages 129–136. The Eurographics Association, 2008.
- [4] C. Ericson. *Real-time collision detection*. CRC Press, 2004.
- [5] A. Evans, M. Romeo, A. Bahrehmand, J. Agenjo, and J. Blat. 3D graphics on the web: A survey. *Computers & Graphics*, 41:43–61, Feb. 2014.
- [6] P.-M. Gandoi and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. *ACM Transactions on Graphics*, 21(3):372–372–379–379, July 2002.
- [7] Y. Huang, J. Peng, C.-C. J. Kuo, and M. Gopi. Octree-based progressive geometry coding of point clouds. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG’06, pages 103–110, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.
- [8] G. Lavoué, L. Chevalier, and F. Dupont. Streaming Compressed 3D Data on the Web using JavaScript and WebGL. In *ACM International Conference on 3D Web Technology (Web3D)*, San Sebastian, Spain, pages 19–27, 2013.
- [9] M. Limper, S. Wagner, C. Stein, Y. Jung, and A. Stork. Fast delivery of 3D web content: a case study. In *Proceedings of the 18th International Conference on 3D Web Technology*, pages 11–17. ACM, 2013.
- [10] A. Maglo, C. Courbet, P. Alliez, and C. Hudelot. Progressive compression of manifold polygon meshes. *Computers & Graphics*, 36(5):349–359, 2012.
- [11] B. Merry, P. Marais, and J. Gain. Compression of Dense and Regular Point Clouds. *Computer Graphics Forum*, 25(4):709–716, Dec. 2006.
- [12] D. Mongus and B. Žalik. Efficient method for lossless LIDAR data compression. *International Journal of Remote Sensing*, 32(9):2507–2518, May 2011.
- [13] M. Pauly, M. Gross, and L. P. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings of the Conference on Visualization ’02, VIS ’02*, pages 163–170, Washington, DC, USA, 2002. IEEE Computer Society.
- [14] J. Peng and C.-C. J. Kuo. Octree-based progressive geometry encoder. In *ITCom 2003*, pages 301–311. International Society for Optics and Photonics, 2003.
- [15] I. Prieto, J. L. Izgara, and F. J. Delgado. From point cloud to web 3D through CityGML. In *Virtual Systems and Multimedia (VSMM)*, 2012 18th International Conference on, pages 405–412, 2012.
- [16] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’00*, pages 343–352, 2000.
- [17] R. Schnabel and R. Klein. Octree-based Point-Cloud Compression. In *SPBG*, pages 111–120, 2006.
- [18] W3C. Web Performance Working Group, 2014.