Optimisation of complex LIDAR & oblique imagery for real time visualisation of cityscapes.

Karl Jones
Cassidian Ltd
Karl.Jones@Cassidian.com

Abstract. 2 subsidiaries of EADS, CASSIDIAN and ASTRIUM have developed a system that can process captured LIDAR (Light Detection and Ranging) and oblique imagery into 3D meshes. This paper presents a system for optimising these complex meshes for real time visualisation, the techniques used to automate the system and how they were able to visualise the London cityscape, which consisted of over 30,000 buildings, totalling approximately 115 gigabytes of data. Finally a page-able Level of Detail (LOD) system is presented using the Unity game engine to display the optimised data and a summary of how the entire import process was automated.
Introduction

Today it is commonplace for industry and governments to use 3D landscapes and generate 3D visualisation that allow the user to “fly through” the landscape or place objects within the landscape. The generation of these landscapes and the buildings has traditionally been done by employing 3D modellers to build a bespoke landscape, and add significant buildings and features. The advantages of manually building up the landscape in this way are that the final object data base can be used to generate real-time views or flythrough. However, the process is labour intensive, and time consuming. This paper presents an automated process of capturing the landscape data from satellites and/or aircraft flybys using sensors to capture the raw 3D data and images of the landscape, through to the generation of an optimised object database that is able to run in real-time.

Through research and development activities that it was conducting during 2009 in providing real-time 3D simulation environments CASSIDIAN started to build competencies that would allow complex and high fidelity 3D terrain databases to be managed in real time in commercial computer game engines. At about the same time ASTRIUM was developing its SKAPE product line, which was intended as an architectural tool for assessing the impact of a development and resolving issues early in the development lifecycle, by allowing users to view a small area’s buildings and satellite imagery through a web based interface.

ASTRIUM can gather the initial data by performing a LIDAR flyby of the physical area in either an aircraft or satellite imagery, and capturing the landscape and buildings using LIDAR sensors. Depending upon the aerial platform chosen the resolution of detail can be from meters to mm. This initial flypast generates a vast quantity of 3D point data, which would not be possible to render as raw data in real-time visualisation with existing computing, network and storage technology. As such this data must be processed into an optimised object database in order to be able to view the landscape in real-time.

CASSIDIAN worked in collaboration with ASTRIUM to open the potential for the SKAPE product to support real time simulation systems. An automated processing pipeline was developed to allow for the process of capturing, processing and optimising large datasets for real time visual engines. This paper describes the processes involved and methodologies used in optimising large amounts of data in order to run at an acceptable frame rate.
1.1. Overview of the process
In this section the overall capture and optimisation is discussed from the initial capture using light aircraft to the final visualisation using the Unity Game Engine. Figure 1 shows the overview pipeline and the crossover between the 2 subsidiaries.

1.2. Data Capture & Initial Object Creation
The first stage of the pipeline is to capture both the imagery and the mesh/elevation data. The data capture is based on dual sensor collection. First an aircraft will overfly the area of interest equipped with a High Density LiDAR system which is capable of a capturing elevation data at a resolution of up to 12 points per metre.
In basic terms, LIDAR is a scanning laser range finder coupled to a digital storage device that can be used to capture a ‘point cloud’ of the 3D positions (and intensities) of laser pulses reflected from its surroundings. When properly coupled to a precision attitude and geo-referencing system, each point in the point clouds can be located to a high degree of accuracy within a world reference frame. This data must then be converted into a more manageable 3D mesh. The conversion process uses geometric modelling algorithms which examines the point-cloud data and replaces sections using simple 3D primitive shapes such as cubes, pyramids and cylinders where possible. Figure 2 shows the result of processing St Paul’s Cathedral using the geometric algorithms.

After the LIDAR flight is completed a second flight is conducted using a MIDAS (Multi-cameras Integrated Digital Acquisition System) oblique optical imagery system (TRACK’AIR) in order to capture the imagery data. Oblique optical imagery (or oblique imagery) is aerial photography that is captured at approximately a 45 degree angle to the ground. From this angle it is possible to see both the tops of buildings and the sides. Figure 3 show the MIDAS system on-board a light aircraft ready for data capture.
Once the LIDAR and imagery data is captured and processed they are fused together. An example of a fused image is shown in Figure 4. The oblique imagery is automatically rendered onto the LIDAR models faces; this process involves examining each building and selecting the most appropriate imagery data from the larger oblique imagery, then cutting the data and applying it to the corresponding face as a new texture map. The generated building and associated textures are then saved into standard 3D model/image formats.

1.3. Real Time Optimisation

After the data has been captured and processed, the outputs are a large number of 3D model files and textures which have been sorted into multiple directories based on their locations. These files are of a complexity that is appropriate for viewing on a per model basis or in a small batch, however when attempting to display a large number of models visual performance, and frame rate quickly degrades and becomes an issue. For example the London test data which covered approximately 12km by 5km consisted of over 115GB of data which comprised of 142,056 files in 809 directories, of which 26,447 were 3D models and the remainder were associated texture files of varying size. It is simply not possible to display all this data simultaneously in one large 3D world. In order to load all this data for display, a significant amount of memory would be required however the main bottleneck is the memory on the graphics card. Modern graphics cards have on average between 1-2GB of memory onboard which can be used to store textures being displayed, the textures for London totalled over 114 GB uncompressed. In order to display all captured data a two stage process is used. The first stage involves optimising each of the single files, using this method it is now possible to display approximately 10% of the data simultaneously. The second stage consists of generating levels of detail (LOD) for logical groups of objects, using this method it
is now possible to display all of the captured London data.

1.3.1. Object/File Optimising

The first issue that was noted when examining each of the 3D model files was the structure of the objects and their geometry. In each file the object is not a single or small number of objects but is in fact hundreds of individual objects each having their own separate mesh. Figure 5 shows a sample object from the London dataset, it can be seen that some of the objects consist of a single triangle or quad face which is very inefficient, each of the individual mesh objects will result in a single draw call. Draw calls are the number of GPU calls the engine needs to make to render all the objects in the world. The lower the number of draw calls the better for performance. The sample object shown (Figure 5) consisted of 562 objects, most visual display engines would require 562 draw calls for this object to be rendered, one call per object. As a rule of thumb an object of this type should only need 1 draw call per texture, in this case the object has 14 textures so the target was to reduce the draw calls to 14. In order to achieve this reduction in the draw calls and improve the models performance the objects were combined into a single object/mesh, further optimising was then performed by combining any overlapping vertices so that the object now formed one single mesh. This now resulted in 14 draw calls.

![Textured Objects and Separate meshes](image)

*Figure 5: One of the captured London model files textured(a), showing the object is in fact composed of many separate meshes and not a single mesh(b).*

The next item to consider is the large number of textures that each object employs. Each model utilises several textures which quickly becomes an issue when attempting to display a greater number of models due to memory limitations on the GPU. By combining these textures into a single texture, also known as a texture atlas, it is possible to not only reduce the memory consumption of the model but also further reduce the number of required draw calls. The texture atlases technique is one tool that can reduce draw calls. While texture atlases have the stigma of producing lower image quality (NVIDIA 2004) in practise we have found that image quality is not significantly affected, it can be seen from Figure 6 that there is no noticeable quality loss, even when we combine 84 textures into a single atlas. Table 1 shows the reduction in the number of objects.
Figure 6: The London Museum before and after optimisation. Note 6 draw calls are not part of the object, but the background.

<table>
<thead>
<tr>
<th>London Museum</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects</td>
<td>3225</td>
<td>1</td>
</tr>
<tr>
<td>Textures</td>
<td>84</td>
<td>1</td>
</tr>
<tr>
<td>Vertices</td>
<td>16757</td>
<td>7043</td>
</tr>
</tbody>
</table>

Table 1: Optimisation results for London Museum.
One way to generate a texture atlas is to simply pack all the textures into a single texture, the problem with this however is increased texture size and the inefficient use of texture space. The atlas texture can be resized however it is the unused texture space that then causes quality loss. When performing a simple pack of the textures the unused texture space is also packed, ideally a texture should have none or very little unused texture space. Figure 7 demonstrates the inefficient use of texture space, over half is unused. If two textures of this layout were packed the size would be twice that of the original yet only half of the texture space would actually be used. An atlasing technique that takes into account the texture usage is far more efficient than simple packing. The following methodology is used to generate the texture atlases; this method takes into account the UV coordinates of the meshes and instead of packing the existing textures generates a new texture using the data from all the models textures, a method sometimes referred to as texture baking. This method allows for unused texture data to be excluded from the generated texture atlas. The entire process is automatic and requires no intervention other than to set the initial starting parameters (Max atlas size, texture output formats, model output format etc).

a) Create the texture atlas of the default size (2048² pixels) or smaller if the total texture coverage is less.
b) Unwrap the entire mesh onto the atlas texture and store the UV’s in the second UV layer, then pack the UV’s within the 0-1 range.
c) Perform a bake operation from UV layer 1 and its associated texture to the new atlas (Figure 8).
d) Remove UV layer 1 and all associated textures and materials.
e) Create a new material for the atlas and apply to the object.

In summary, the first phase of optimising involves processing each individual model file,
combing the models into single meshes and generating single texture atlases for each model. With file optimisation the London test data is now reduced from 115GB to 7.49GB (images are now compressed). On average 5-10% of the London test data can now be displayed before GPU memory becomes an issue.

1.3.2. Level of Detail Creation

It is clear that optimising single files will never be enough to display the entire dataset. The GPU memory soon becomes an issue once more. One solution is to reduce the size of the textures on the models from the default (2048²) to something much smaller (512²) and thus requiring less memory, however loss of visual quality then becomes an issue as well as the number of draw calls when displaying a large number of objects. The London test data would need over 27,000 draw calls if each object had one texture atlas and because 10,000 draw calls will bring even the most capable CPU to its knees (Dudash 2004) further optimisation is needed.

In order to maintain a low number of draw calls and memory footprint, a level of detail system is introduced. When viewing the data from a distance the objects can be combined into meshes with single texture atlases, as the viewer gets closer the object/s are then split into sub meshes, each containing a separate texture atlas, as the viewer gets even closer the sub objects are split again into further sub objects and so on until the original optimised file is displayed. As previously discussed the output capture data was stored in directories based on location, each of these directories had a further sub directory splitting the data into batches of no more than 100 models per directory. Each directory’s model files were loaded simultaneously and an algorithm was then run to generate a quad tree LOD structure (Figure 9).

The algorithm combines all the objects into a single mesh and generates a texture atlas; this becomes the lowest LOD level. The algorithm then splits the mesh into 4 sub meshes, one

---

Figure 9: The LOD generation algorithm.
per corner and then atlases each corner. The algorithm then continues the same process with each of the corner sub meshes until it reaches the maximum LOD depth defined (default 3) or it is deemed unnecessary to generate further LOD levels due to the number of materials/textures being only 1 on the current LOD level. Figure 10 shows the output of processing a single directory. At the highest LOD level an atlas is not generated (although this is an option in the tool) and instead the original objects from phase one can be found thus providing maximum quality.

During the LOD generation process the generated objects are kept in a logical hierarchical tree structure. The root node of the structure represents the lowest LOD, the child nodes are then the second lowest LOD and so on leaving the leaf nodes as the maximum LOD, this structure can then be stored in the output model format (where supported) for simple import into a visual engine such as a Synthetic Natural Environment (SNE).
In summary, the second phase of optimising involves loading batches of the optimised files from phase one then generating a LOD structure in which the lowest is a single mesh combined mesh of all the objects with a single atlas texture and the highest LOD is the original object. Finally outputting this new LOD structure and associated textures in standard 3d/texture formats. After phase two of optimising it was now possible to show the fully captured London dataset and maintain a low number of draw calls (no more than 1000) and thus a high frame rate.

1.3.3. Automating the process

Each phase of the optimisation pipeline is run as a separate process/application per model/batch. The first phase of the London test data required over 27,000 processes to be run, so it was not feasible for the optimisation to be a manual process. In order to automate the entire pipeline and thus require no intervention a management application was developed. The application was developed with a simple Plug-In based architecture allowing for new optimisation processes to be added in a modular fashion. It was capable of scaling the optimisation process by allowing the user to set the maximum number of simultaneous processes to be run (ideally 1 per CPU core).

1.4. London Data Optimisation Results

Figure 11: LOD object structure. The root node is the lowest LOD; this node is then split into 4 for the next LOD and so on.

Figure 12: Comparison of London data. Percentage is the size compared to before data.
Phase 1

| Objects Removed | 1057477 |
| Vertices Removed | 7283721 |
| Faces Removed* | 7319 |
| Textures Removed | 36930 |

Phase 2

| Generated LOD’s | 334 |

Table 2: Results of optimisation for London test data.

*Face removal is not part of the optimization process but a fortunate by-product of the vertex combining, which removes duplicate faces.

Using the manager application the London test data was processed on a Dell R715 server (24 CPU cores). The total optimisation process (phase 1 & 2) took approximately 10 hours. Figure 14 and Table 2 show the before and after statistics for the data. The final output data was now able to run at an average of 334 draw calls at its lowest LOD with a high frame rate on an average range PC (Figure 135).

![Figure 13: Optimised London data running in Unity.](image)

Machine Specification: Intel Core 2 Quad CPU (2.66GHz), 8GB RAM, GeForce GT 430(1024MB).

### 1.5. Importing Into the Viewer – Unity

Once the optimisation is complete the processed data can then be imported into any visual system that is capable of supporting the LOD structures.

An automated system was developed to import and display the data using the Unity game engine.

![Statistics](image)
The majority of LOD systems including Unity’s own rely on having all LOD levels pre-loaded into memory and simply switching them on/off depending on the current LOD level that needs to be displayed. This method provides very fast switches between the LOD levels however when applied to the London data which was approximately 12GB in size it would be impossible on a 32 bit machine and even on a 64 bit machine would require a significant amount of memory. A method of dynamically loading and unloading or paging the LOD levels as and when they were needed was required.

### 1.5.1. Asset importing and LOD Setup

For the Unity game engine two possible technical solutions were identified. The first was to use the Unity Resources technique. In Unity all assets placed inside the Resources folder will be included in the final build, by default Unity would normally calculate what assets are being used and only include the used assets, but by using the Resources folder the assets can be referenced at runtime and Unity does not need to be aware of their usage beforehand. Two issues arose with the Resources method. As each LOD had been exported as a single file with all LOD data contained, using the Resource loading method the entire file would need to be loaded in order to extract a single LOD level. This was solved by splitting all the LOD’s out into separate files. The most significant issue was the time it would take to generate the resources file during a build. Every time a test build was made all 12GB of data would need to be compressed into a single resource file in Unity, which would take a significant time (hours) to perform and hence slow down development - particularly for the touch interface development which involved making changes, performing a test build and then repeating (many times), as the Unity editor was on a separate machine it was not possible to quickly test in the Unity editor.

The alternative solution and the chosen technique was to use AssetBundles. AssetBundles are compressed files exported from Unity that can be loaded on demand from the application at runtime. Any type of file/data can be contained inside the AssetBundles. AssetBundle building is a separate process from performing a build in Unity; once the AssetBundles are built they do not need to be rebuilt each time a build is performed. The AssetBundles are ‘streamed’ in during runtime allowing for none blocking loading and unloading of the LOD levels whereas the resource loading technique would often block whilst loading causing the application to intermittently freeze during heavy loads.

An advantage of using Unity AssetBundles is the ability for the data to be streamed over the internet or a network, allowing the potential for the data to be viewed in a web browser which would require very little further work due to Unity’s one click deployment system. However the large volume of data makes this approach impractical for internet streaming but streaming over a local network provides a powerful function, allowing for multiple clients to use a single dataset.

In order to automate the entire process in Unity including importing the entire London dataset and setting up the LOD structures for use in Unity an AssetPostProcessor script was used. Figure 14 shows an overview of the post processor script.
The entire dataset was imported into Unity using the standard method (import the directory into Unity) however during the import process the AssetPostProcessor script is called. The script is responsible for the following:

- Manages the texture size of the LOD atlases. By default all textures are 2048\(^2\) however it is not necessary for the lowest LOD to require such a large texture when it will only occupy a small amount of screen space, the post processor assigns new sizes based on the LOD level (see Table 3).

<table>
<thead>
<tr>
<th>LOD</th>
<th>Texture Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>512 x 512</td>
</tr>
<tr>
<td>2</td>
<td>1024 x 1024</td>
</tr>
<tr>
<td>1</td>
<td>2048 x 2048</td>
</tr>
<tr>
<td>0</td>
<td>2048 x 2048</td>
</tr>
</tbody>
</table>

Table 3: Default assigned texture sizes based on LOD level.

- Split the LOD levels into single objects and save as AssetBundles, one per mesh/object. After this process \textbf{14,647} AssetBundles totalling \textbf{13.7GB} in disk space.

- Create the LOD structure using GameObjects (Figure 15). The LOD structure is replicated using GameObjects, each GameObject contains a reference to the location of its relevant AssetBundle file for loading at runtime.
1.5.2. Runtime LOD/Paging system

The LOD paging system was required to support the following operations:

- Load AssetBundles (LOD Levels) as and when required in an efficient manner. The process should not ‘block’ other operations, cause freezes in the application whilst the data was being loaded.
- Allow for cancelling/aborting the loading of an AssetBundle due to a further LOD switch.
- Delete/Remove LOD levels when they were no longer being displayed due to an LOD change.

The system was divided into three logical states of functionality (Figure 16).

1. **AssetBundle Loading**: *AssetBundleLoader* class.
   Responsible for loading AssetBundles and notifying LOD levels when the load was completed. Capable of loading simultaneous AssetBundles and cancelling LOD load requests, implemented as a singleton class.

2. **LOD Calculation**: *LODSwitchAB* class.
   Performs LOD calculations, issues load/abort requests to the *AssetBundleLoader*. Each LOD level (GameObject) is responsible for performing LOD calculations when it is active. (Figure 19).

3. **Memory Management**: *MemorySweeper* class.
   Clean up memory periodically when required.
Figure 16: UML class diagram for Unity LOD system.

Figure 17: UML Activity Diagram method for loading and switching LOD.
In summary, it was possible to import the processed data into Unity which would then automatically set up the LOD AssetBundles and associated LOD structures which were saved as Unity prefabs. It was then a simple case of importing in each LOD prefab (334 in total) into a Unity scene, the prefabs had a specific naming convention (‘_SKP’) which allowed for a quick lookup in the Unity editor so they could be easily found and added to the scene in one operation.

Whilst the simulation was running it was possible to view the LOD status in the Unity editor (Figure 18), the current LOD levels and loading status were displayed through the use of different coloured lines and icons (gizmos). An icon of a house indicated the lowest LOD was active and also showed the root of the LOD tree, grey lines represented inactive LOD’s, yellow active LOD’s and blue those waiting to be loaded/paged, this allowed for simple debugging of the system and for further tweaking of LOD distances (if required).

![Gizmo Key](image)

**Gizmo Key**

- Lowest LOD Active
- Active LOD
- Inactive LOD
- Currently Paging/Loading LOD

**1.5.3. Screenshots of London Data Running In Unity**

**Future Work and Conclusions**

This paper has presented in detail the processes involved in capturing, processing and optimising large amounts of LIDAR and oblique imagery data for use in real time systems. Finally software architecture for displaying the data using the Unity game engine was presented, using this architecture the London data performed at a steady frame rate (~60fps) on a Middle of the range PC (Intel Core 2 Quad CPU (2.66GHz), 8GB RAM, GeForce GT 430). An innovative technique for using Unity AssetBundles was presented, showing it is feasible for Unity and the developed system to manage thousands of AssetBundles with no issues.
A number of areas of further work have arisen during the entire process:

- **ASTRIUM** have improved the capture process which now only requires a single flight.
- Further optimisation of the mesh and LOD system through more aggressive reduction parameters.
- Integration into simulation systems such as Rockwell Collins EPX IG systems which are used to power state of the art flight training devices such as the EADS Combat Air Systems (CAS) Compact Flight Training Device (CFTD).
- Taking advantage of the new Unity memory features.

**References**


