November 29, 2018

DEFINING CONTAINERS OF PART VOXELIZATIONS IN A 3D PRINTER

HP INC

Defining containers of part voxelizations in a 3D Printer

This disclosure defines a method for reducing the number of files generated during the voxelization of a 3D Part by means of volumetric slices. The solution consists of encapsulating all the generated slices in a single container.

A 3D printer prints content received by means of 3MF (3D Manufacturing Format). A 3MF file consists of several parts positioned in a build bed which are sent to the 3D printer for their production. Before being able to print a job, the content must be rendered to some print-ready voxelization. In the case of some 3D printers, the render process consists of slicing each part, and generating a volumetric slice for up to 4 different properties (SHAPE, SHELL, NORMALS and SHELL). Every slice property is represented individually, generating thus 4 files per part slice. Considering that the slicing occurs at the print layer thickness (for example, 80 microns), the number of files generated for a multi-part full-bed job can be in the order of millions. From the printer FW Operating System perspective, handling a directory containing such a huge number of files can cause problems in some printer workflows.

Among these problems, the most important one may occur while printing, causing a sudden cancel of the job. While printing, we need to provide the 4 properties for the whole print bed in order to decide where to place the print agents, but if there are a large number of parts in a certain layer, getting all the individual part properties and composing them into a single bed image is computationally expensive. In this situation, the Operating System policies may slow down the retrieval of the files when dealing with a large amount of files, causing the layer to exceed its maximum execution time in some situations. If the maximum time is exceeded, the job will be aborted for safety reasons and/or major part quality defects.

To overcome these issues, we propose a method to encapsulate the slice properties in a single container per part, reducing thus the number of produced files to the order of the number of parts in the job.

When processing the job to generate the internal render (voxelization), the printer slices each part independently. In the usual approach, the 4 different properties per slice generated during the slicing phase are written directly to a directory owned by the job. The slicing is done at the same layer thickness that will be used at printing time to avoid aliasing problems. The most commonly used layer thickness is 80 microns. This means that for a mid-sized part with a height of 15 cm, there will be about 7500 files generated for just a single part. If we consider a full bed job containing about 300 of these parts in the bucket, this gives more than 2 million files on disk. Having such a big number of files can cause different problems to the FW. These are the most relevant issues that we detected:

- The deletion of a job with 2 million files takes up to 1 minute. This action might be performed by a user using the Front Panel, so a long deletion time will cause a bad user experience.
- When the user selects to reprint a job, the job directory structure is replicated and every single file is hard-linked to the new destination. When all the files are hard-linked, then the original job is removed. This means that, since the job selects to print a stored job till the new job actually appears on the Front Panel, it may take up to 2 minutes, leading to a poor user experience.

- While printing, loading part-slices which usually takes few milliseconds, may increase to up to 3 seconds due to OS buffering of open files. Considering that the time to print a layer is around 7 seconds, this time increase may lead to a print job cancellation, with the subsequent loss of material, agents and time to the customer.
- Having 2 million files on disk requires using 2 million iNodes from the hard disk in the embedded linux Operating Systems. Being iNodes a limited resources, having several jobs with such a big amount of files may exhaust the resources from the OS, causing unexpected behavior on the system.

Container of Voxel Slices

We have designed a solution based on ZIP containers to overcome all the limitations when having such a large number of files in the hard disk. The solution consists on appending the slices into a ZIP container, so when a part slice for a given property is generated, instead of writing the property in disk, we rather encode the content in memory using the same format and append this bitstream to a ZIP file. A ZIP file has a structure defined in the PKWare ZIP spec [2]. Mainly, the ZIP file begins with all the files included in the ZIP, each one starting with a local header followed by the content of the file (see Figure 1). Then, after all the files, there is a magic number which identifies the start of the *Central Directory*. This *Central Directory* contains a record for each of the files inside the ZIP, describing information such as:

- File name
- File compressed size
- File uncompressed size
- CRC32 of the file content
- Offset to the start of the file from the position of the start of the *Central Directory*
- Compression methods (NONE, DEFLATED, IMPLODING, LZMA, LZ77, …)
- Other metadata

After all this information there is another magic number identifying the *End of Central Directory*, which may contain additional metadata. When using the ZIP64 extension, there will be also a couple of records more between the *Central Directory* headers and the *End of Central Directory*. There are some different ways to consume a ZIP file; one of them consists in opening the file, searching for the magic number which identifies the start of *End of Central Directory*, get from there where the *Central Directory* start, and then loading the information about all the files from the headers. These file headers provide offsets to the location of the actual files in the zip, so we have direct access to the files, being able to locate and extract/consume every file inside the ZIP.

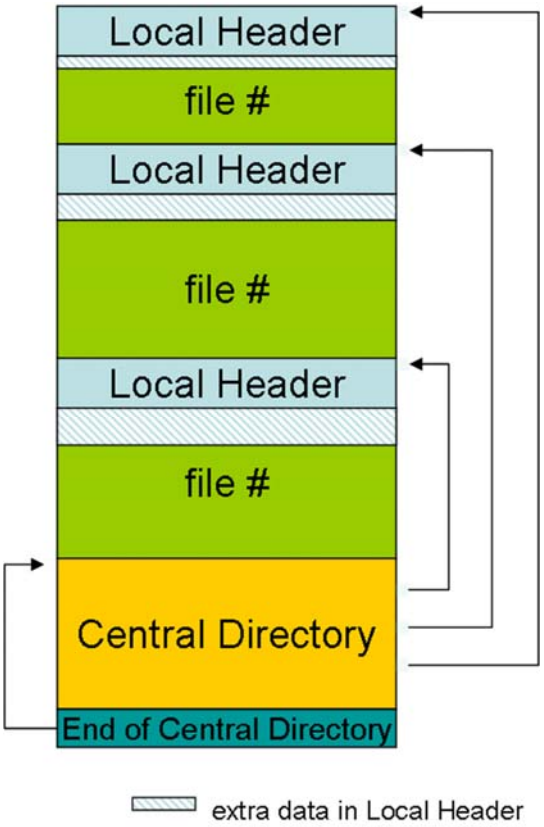

Figure 1. High level structure of a ZIP file

Due to the characteristic of the ZIP file, we cannot "close" the file till we have included all the files, because the headers which contains the information about all the files contained on the ZIP file is located at the end of the file. For that reason, while we are creating the ZIP file, we open an auxiliary file where we write all the *Central Directory* headers for the files included in the ZIP. This way, when we want to close the file, we just load the information about all the files previously included in the ZIP container and use it to write the *End of Central Directory*. This allows our solution to recover the state after a reboot, so that when processing a complex file we can start from the point it was executing after a reboot. It is worth mentioning that ZIP supports different compression methods. By default we selected the NONE compression method, as it reduced the time to generated the ZIP file and some printers are less limited on disk space. However, only changing a configuration parameter we can configure the FW to generate ZIP files with DEFLATED compression method, which uses the zlib library for compressing the individual files bitstream. In this case, when we are going to append the bitstream with the encoded raster image, we first compress this bitstream using zlib and append the compressed bitstream which usually has a smaller size than the original one. We have tested this approach and only had a penalty of about 50% of increase in the time to process a job. However, when consuming files it made almost no difference in time as we only require to decompress a small amount of bytes when trying to get a slice from the ZIP container.

The proposed solution has several advantages. First, we reduce the total amount of files generated when processing a job. This overcomes known limitations in different workflows like print layer generation, reprint of a job or deletion of a job, which become slower (in some cases even unusable)

when working with folders containing millions of files. Moreover, we generate the job rendered content in a portable format, and only changing a configuration parameter we can get additional compression, which is very useful for scenarios in which there is a reduced amount of hard disk memory.

Another benefit, is that we keep the ZIP structure backed up in an auxiliary file in disk, so we can recover the render generation process at the same point it was left if a crash were to occur. Finally, this solution provides an important benefit during the printing process, which is opening only a file per part instead of a large number of files per part. Then, a slice can be retrieved by extracting the corresponding bytes from the respective portion of the ZIP file. This protects the printing process from unexpected delays due to the necessity of opening a different file for every property and for every part of a layer.

Disclosed by Sergio Gonzalez, Alex Carruesco Llorens and Jordi Gonzalez Rogel, HP Inc.