Workflow Optimization for a Graphic Artist working on large Texture Sets using Virtual Texturing

Michael Birsak[*]
Supervised by: Michael Wimmer[†]

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

Abstract

In this paper we present an approach to optimize the workflow for a graphic artist currently working on high resolution photographs of architectural and archaeological monuments. These photographs are used as texture maps to color the meshes, which are calculated from laser-scanned point clouds corresponding to exactly those monuments. Because a particular region belonging to a monument is covered from several photographs with different colorization, further manual processing of the photos is required. Therefore we developed an application, which generates masks to emphasize regions of the photographs that are used in the final model, to ease the work of the graphic artist. For fast rendering of the model, we took Virtual Texturing into account, and developed an application for fast generation of the Virtual Texture Atlas and the Tile Store. A fast and efficient update of the Tile Store, if a photograph is edited after the final generation of the Virtual Texture Atlas and the Tile Store, the application also provides. The used algorithms are stated as well as the speed-up compared to an existing implementation.

Keywords: Mask Generation, Virtual Texturing, Texture Atlas, Fast Tile Update

1 Introduction

At the Vienna University of Technology, the aim of the Terapoints-Project[1] is the preservation of important architectural and archaeological items. This preservation is done by laser-scanning these items, that yields huge point clouds. Further, photographs with registered cameras are taken, to make a colorization of the digitized model possible. To allow rendering of the model inside a standard tool like Meshlab [2], the point cloud is transformed into a triangle mesh using the algorithm from Abdelhafiz [1]. This has the advantage not to be constrained to applications implementing algorithms like Instant Points from Wimmer and Scheiblauer [8]. Further, a mesh allows a continuous mapping of the photographs onto the model. The different lighting situations depending on the different scanning positions lead to colorization differences between the photographs. For that reason, processing of these photographs is necessary. At the moment, all processing steps are done by a graphic artist, which means that every photograph contained in the final model has to be edited in a graphics editing application like Adobe Photoshop[2]. Without further processing, there would be visible artifacts in the model, where two regions, belonging to two different photographs, adjoin each other. In Figure 1 you can see such artifacts.

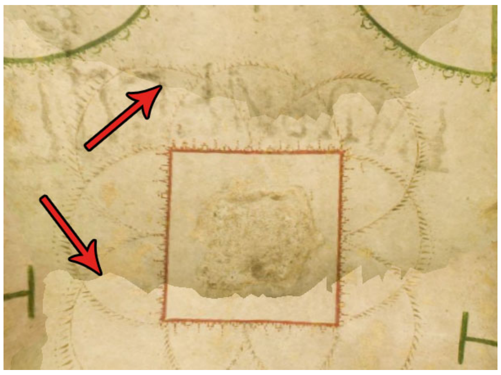


Figure 1: Visible artifacts in the Domitilla model without further processing of the photographs. The arrows show the regions, where different photographs adjoin each other.

The rendering of the whole model is important for the graphic artist to see all the photographs and the possible artifacts in action. Currently, the rendering is happening in MeshLab. The current workflow of the graphic artist is shown in Figure 3.

Because of the high resolution and the large quantity of photographs in a single 3D model, there is a need to accelerate the rendering of the whole model. Without this acceleration, there is an unnecessary amount of traffic between the CPU and the GPU during the rendering process, because not all photographs fit into the memory of the graph-

[*] michael.birsak@gmx.at
[†] wimmer@cg.tuwien.ac.at
[1] http://www.cg.tuwien.ac.at/research/projects/TERAPOINTS/
[2] http://www.adobe.com/de/products/photoshop/compare/

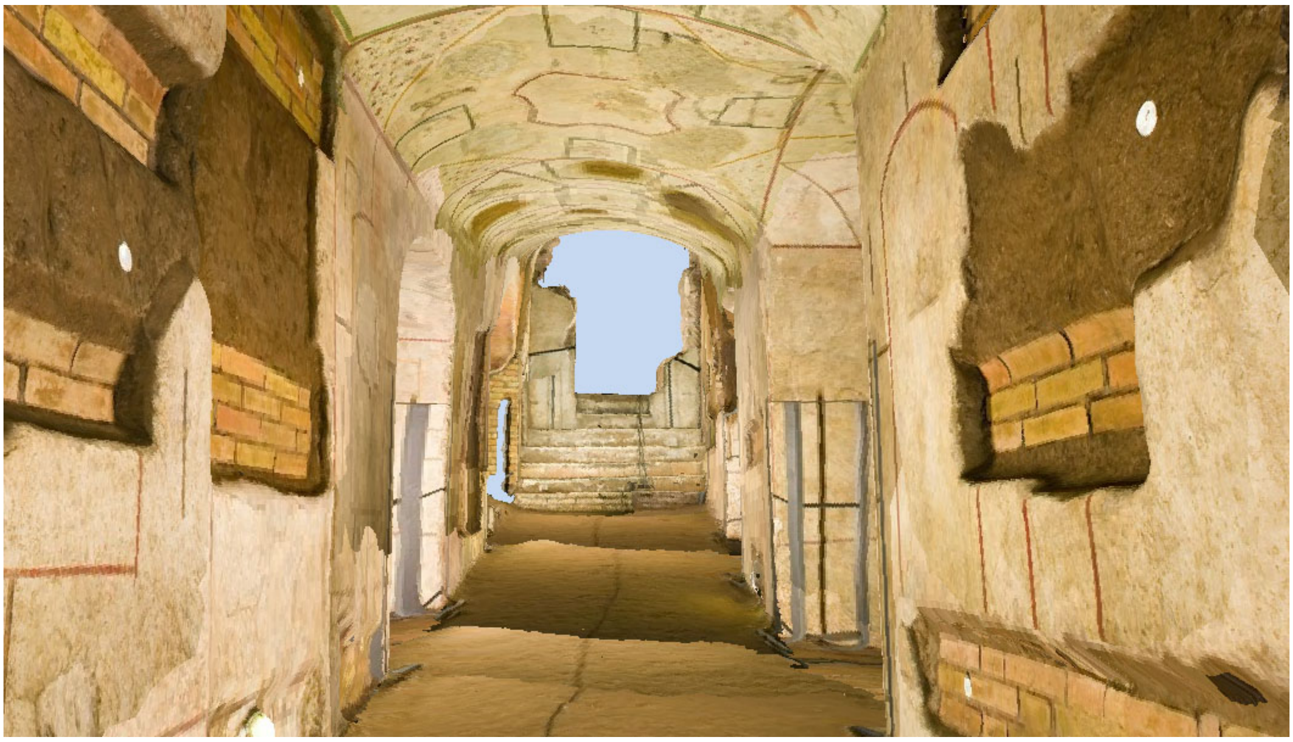
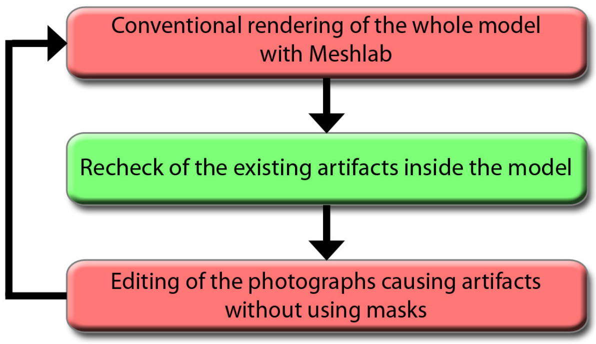
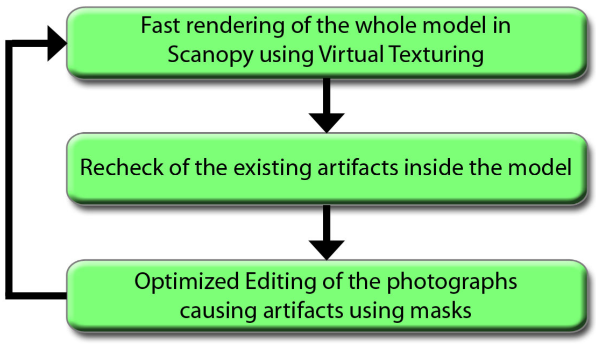


Figure 2: Part of the Domitilla Catacombs visualized using the unprocessed photographs. Especially on the floor the aforementioned artifacts are visible.



Conventional rendering of the whole model with Meshlab
Recheck of the existing artifacts inside the model
Editing of the photographs causing artifacts without using masks

Figure 3: Current workflow of the graphic artist. The first and third step in the cycle are the ones, we want to optimize.



Fast rendering of the whole model in Scanopy using Virtual Texturing
Recheck of the existing artifacts inside the model
Optimized Editing of the photographs causing artifacts using masks

Figure 4: Optimized workflow of the graphic artist. The rendering is accelerated using Virtual Texturing, the editing is eased by the generated masks.

ics card. The results are low frame rates and therefore smaller productivity of the graphic artist. Due to this we decided to use *Virtual Texturing*, where only those parts of all photographs reside in the memory of the graphics card, which are currently visible.

As application to visualize the virtual textured model, we chose Scanopy because it implements LibVT[3] (developed by J. Mayer, the author of [5]), a library which provides Virtual Texturing functionality. Scanopy is developed at the Vienna University of Technology and at the Imagination[4] in Vienna. We implemented additional functionality into Scanopy, to directly call our application via keystroke to update the *Virtual Texture Atlas* (in the following just referred to as *Atlas*) and *Tile Store* when a photograph has changed. So, in future the graphic artist will use Scanopy for faster rendering of the model. The optimized workflow of the graphic artist, introducing Virtual Texturing for faster rendering and masks for the photographs for easier editing, is shown in Figure 4. In Figure 2 you can see a part of the Domitilla Catacombs, on which the graphic artist is currently working on.

[3]http://sourceforge.net/projects/libvt/

[4]http://www.imagination.at/

Contribution. In this paper, we present two applications to optimize the current workflow of the graphic artist, who edits the photographs of the monuments. The first program generates masks for the photographs belonging to a model. These masks consist of black and white areas, where white areas correspond to areas in the photographs, which are visible in the model, whereas black areas correspond to invisible areas in the photographs. The generated masks can be used in every graphics editing program, to avoid editing areas of the image which are invisible in the model.

The second program is used for fast generation of the Atlas and the Tile Store. Although there already exists an implementation for this task, we found it too slow to meet the requirements of the graphic artist concerning time consumption. Our program is also used for updating the Atlas and the Tile Store, when a photograph is changed after the Atlas and the Tile Store are generated.

In Section 2 we give some background information about Virtual Texturing and the existing implementations for the generation of the Atlas and Tile Store. In Section 3 and 4 we will give detailed information about the development of the applications we have implemented. Finally, in Sections 5 and 6 we present the results concerning our applications as well as possible additions which can be done in future.

2 Related Work

Virtual Texturing, as it is described detailed by J. Mayer in [5], is a sophisticated technique to overcome the memory limitations of the graphics card, when rendering objects or scenes with a large quantity of textures. Rendering such scenes might work without Virtual Texturing as well, but the frame rate would probably suffer extremely under the high traffic rate between the CPU and GPU, resulting from continuous streaming of texture data. In contrast to Virtual Texturing, every needed texture would be loaded as a whole, regardless of the visible texture area. With Virtual Texturing, only those parts of the texture are streamed to the graphics card, which are actually visible. The smallest part, that can be streamed to the GPU, is called a *tile*. A tile is a small texture with a resolution of 64^2 up to 512^2 pixels, so its final side length must be of the form 2^n pixels for $n \in \{6, 7, 8, 9\}$. If just one pixel of a texture is needed to render the scene, at least the whole tile containing this pixel must be streamed.

The first step for Virtual Texturing is to produce one big texture, consisting of all the single textures in the scene. This texture is the Atlas. In Figure 5 an example of such an Atlas is shown. The Atlas must fulfill some requirements regarding its size (refer to [5] for details). Because the Atlas often has side lengths of 32k pixels and more, it would be very unhandy to store it in a single file. Therefore, it is stored in 4, 16 or more equally sized files.

Note, that it is important to adapt all texture coordinates

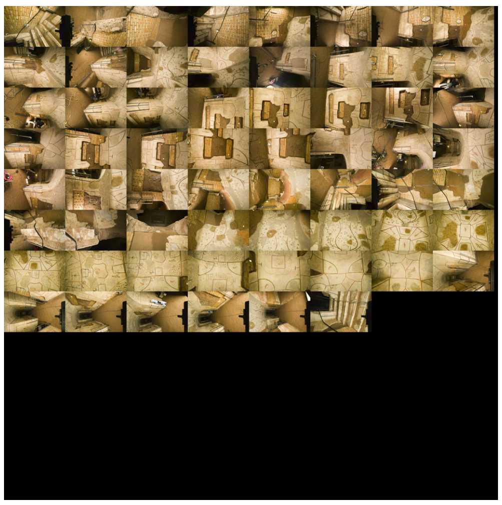


Figure 5: $32k^2$ Atlas consisting of 62 4064x2704 photographs of the Domitilla model.

of the models in the scene to refer to the layout of the Atlas. The Atlas is the base of the Tile Store. The Tile Store can be viewed as a mipmapped Atlas. In contrast to classic Mip Mapping [7], where the side lengths of the original textures are halved until only one pixel resides, the smallest part, representing the highest level of the Tile Store, is one single tile. If we have, for example, an Atlas with a resolution of $32k^2$, and a tile resolution of 128^2, there would be 65536 tiles at Level 0 of the Tile Store. At Level 1, there would be 16384 tiles and so on. At the highest Level with number 8, there would just be one tile representing the whole Atlas. In Figure 6 you can see exactly this scenario applied to the Atlas of Figure 5.

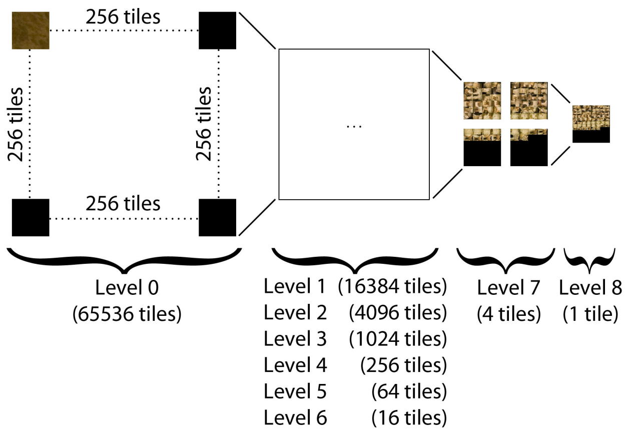


Figure 6: Tile Store generated with the Atlas shown in Figure 5 as its base. The tiles have a resolution of 128^2.

J. Mayer, the author of [5], has already implemented scripts to generate the Atlas and the Tile Store. Therefore he chose Python as scripting language. His script to generate an Atlas is based on ImageMagick[5], a command line

[5]http://www.imagemagick.org

based graphics editing program. The script to generate the Tile Store uses the Python Imaging Library. We will show, that our application, fully implemented in C++, accelerates the task significantly.

3 Mask Generation

The first step to ease the work of the graphic artist, was the development of an application to generate masks for every photograph contained in the model to prevent the editing of invisible image areas. The masks have exactly the same resolution as the photographs, so that every white pixel of a concrete mask corresponds to a visible pixel, and every black pixel corresponds to an invisible pixel in the underlying photograph. To generate a mask for a texture, the only information needed from the model are the face indices into the texture coordinates list, the texture coordinates themselves as well as the material information, to know which primitives belong to which photograph. The face indices are needed, to know which of the texture coordinates belong together to form a primitive, e.g. a triangle. The texture coordinates are values $\in [0, 1]^2$. Therefore it is important to choose a virtual camera (via the projection matrix), whose image plane corresponds to the whole area of possible texture coordinates. The camera, which fulfills exactly these requirements, is an orthographic camera placed in the world coordinate system at position $\mathbf{p} = (0.5, 0.5, 0.0)$. The width and height of the view frustum must both be 1. Since the simplest way to render 2D content is to use the XY-plane, the values for the near and far clipping plane can be set to an arbitrary negative and positive value respectively. In Figure 7 the view frustum of this virtual camera is shown.

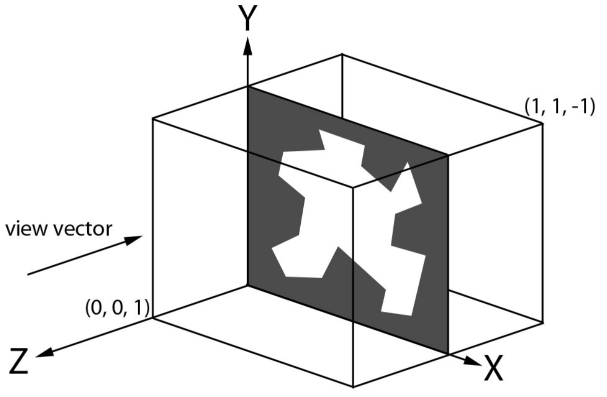


Figure 7: View frustum of the orthographic camera used to render the masks. All primitives are rendered into the XY-plane.

The masks are generated on the GPU using OpenGL as the graphics API. Therefore a buffer with the same dimensions as the current photograph is created. The background color is set to black, which corresponds to invisible image areas. After that, all primitives of the model are rendered in white color into the buffer. The texture coordinates of the primitives are used, as if they were vertex positions. Because texture coordinates are $\in [0, 1]^2$, the z-value is set to 0 to use the XY-plane as the plane to render into.

Our application to generate masks is fully implemented in C++, currently only available for Microsoft Windows. To make it small and simple, we omitted a graphical user interface. Per default, the mask generation application copies a shortcut to itself into the Windows SendTo directory during the installation. Due to this, it can simply be started by a right click onto the model file followed by a "Send To" to our application. After that, all masks will be generated in a sub directory called masks. Of course, all textures belonging to the model must be at the position referenced in the model file.

4 Virtual Texturing

The second step to optimize the workflow of the graphic artist was the introduction of Virtual Texturing for faster rendering of the models. Due to this, we had to improve the already existing scripts, developed by J. Mayer, the Author of [5], to generate the Atlas and the Tile Store. The existing scripts are implemented in Python. Because these scripts need a very long time, to produce the Atlas and the Tile Store (see Section 5 for concrete values), we decided to implement a new application, which should do the same task in much less time. In contrast to the existing scripts, which first generate an own Tile Store for every part of the Atlas, and then merge these to come to the final Tile Store, we decided to implement it in that way, that the final Tile Store is generated out of all the parts of the Atlas in a single run. Our application is also used for the update of the Atlas and the Tile Store, when the graphic artist has changed one or more of the photographs. Because this is the most time-critical function of our application, since it is used after every editing step of a photograph, we will explain it in detail in the following subsection.

4.1 Atlas- and Tile Store-Update

Because the generation of the Atlas and the Tile Store is, also when done with our fast application, a relatively time consuming task, we decided not to regenerate the Atlas and the Tile Store for an update, but to reproduce just those parts, which are concerned with the changed photographs.

To make an update of the Atlas and the Tile Store possible, it is important to know which photographs inside the Atlas have changed and where every photograph is positioned in the Atlas. To accomplish this, a text file is produced during the generation of the Atlas. This text file contains the time of last change, which can be queried from the operating system, of all produced Atlas files. If a photograph is changed and the update routine is run, the time stamp in the text file and the current time stamp would differ, indicating that all parts of the Atlas and the Tile Store belonging to this concrete photograph must be updated. The exact position of the photograph which has changed is so important to know, because otherwise it would not be possible (at least not without much effort) to find the areas in the Atlas and Tile Store the photograph belongs to.

When the changed areas of the Atlas have been found, it is easy to calculate the concrete coordinates of the parts of the Atlas, that are concerned. During the update all levels of the Tile Store have to be considered. Because every level corresponds to a mipmap level of the Atlas, the Atlas must be scaled down to the half after one particular level has been updated. As already mentioned in Section 2, the Atlas is of course not stored in one single file, but in 4, 16 or more parts. Therefore, there may be parts of the Atlas, which are not concerned by the update process. After the update of one particular level of the Tile Store, the Atlas must be scaled down to deliver the necessary information for the next level of the Tile Store. This would be redundant work for every update, if unchanged parts would be scaled every time. Due to this, our application not only stores the Atlas itself, but also every mipmap level needed. Certainly, this scaled parts of the Atlas must also be updated, if they consist areas of a photograph that was changed.

Tile Cache and VRAM. When updating the Atlas and the Tile Store, it is also important to update the dedicated memory region on the graphics card (*VRAM*), which is used to store the currently used tiles. Further, it is necessary to update the so called *Tile Cache*, which is the dedicated region inside the main memory, to hold a finite number of tiles to prevent another time consuming streaming from hard disk. Another streaming of a particular tile can be necessary when it has been overwritten inside the VRAM by another tile because of a certain time period without usage. A problem that arises, if such an update is not executed, is the simultaneous usage of the old and the new version of tiles corresponding to a particular photograph. This happens, because currently just those tiles are streamed from hard disk, cached, and then streamed to the graphics card, that are needed to render the next frame, but are not already stored inside the Tile Cache. When a particular tile is needed again, but is already stored in the Tile Cache, this version is streamed to VRAM, no matter if it has changed on hard disk.

In Figure 8 you can see a part of the Domitilla model rendered with Virtual Texturing inside Scanopy, showing visual artifacts when the Atlas and the Tile Store change, but an update of the Tile Cache and VRAM is not executed. To emphasize the artifacts, the old version of the photograph was patterned before the creation of the Atlas.

LibVT originally was not designed for a modification of the Atlas after it was generated. Therefore, the LibVT has been modified by the implementation of two new functions, one function to delete a particular tile from VRAM, and one function to delete it from Tile Cache. Now, the graphic artist can edit a particular photograph, while

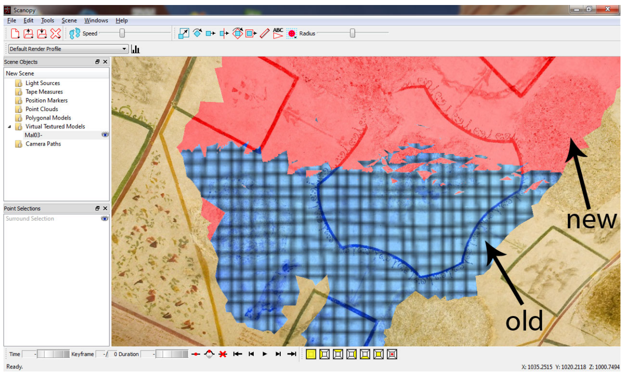


Figure 8: Visible artifacts while rendering in Scanopy resulting from simultaneous usage of the old (patterned) and the new version of tiles corresponding to a particular photograph.

Scanopy is running, and can start the update procedure by keystroke to see the changes inside the 3D model immediately.

We chose C++ as programming language to use a library J. Mayer proposed in his thesis [5]. This library, called libjpeg-turbo[6], produced the best results regarding loading JPEG-images from hard drive. Because of the large quantity of textures, the Atlas and the Tile Store provide, JPEG is because of its high compression rate the image format of choice. Like the application for mask generation, our program for the generation of the Atlas and the Tile Store can be called via ”SendTo”. The only file the application expects is a small text file with all configuration parameters. The most important of these parameters are the maximum side length of the Atlas parts, the paths where the Atlas and the Tile Store should be stored, the side length of the tiles as well as the output format of the Atlas and the Tile Store. The configuration file must be placed in the directory of the images, which should be contained in the Atlas. The application will consider all available image files in the base path of this text file. If all parameters are valid, the generation of the Atlas and the Tile Store starts. The results are the Atlas, split into as many parts, so that the desired maximum side length is not exceeded and the Tile Store, consisting of tiles with the desired side length.

5 Results

Our first application for the mask generation is already in use by the graphic artist and provides considerable ease of his work. In Figure 9 you can see the result of the mask generation for a single photograph, in Figure 10 the usage of the mask inside Adobe Photoshop is shown.

Our second application for the generation of the Atlas and Tile Store is significantly faster regarding Atlas gener-

[6]http://libjpeg-turbo.virtualgl.org/

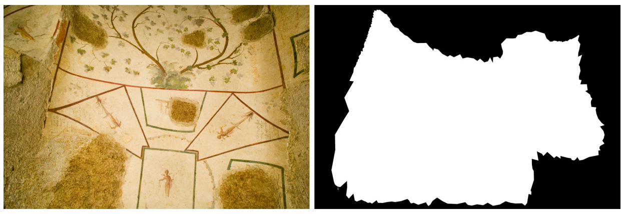


Figure 9: Photograph used as texture (left side) with its corresponding mask (right side).

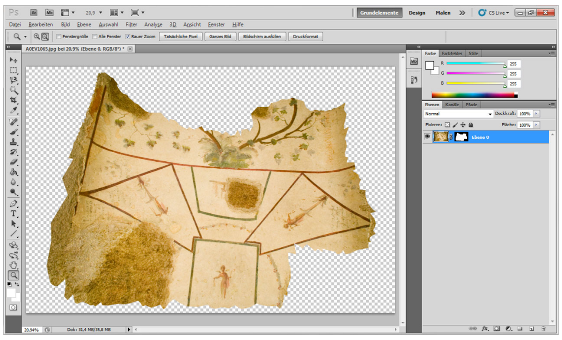


Figure 10: Usage of a generated mask inside Adobe Photoshop.

ation than the existing Python implementation, as you can see in Figures 11 and 12. Both diagrams show, that our application is at least four times faster than the Python script with ImageMagick. While our application produces nearly consistent results, the Python script needs the longer the more parts are produced. This increase can be explained by the nature of the script, which calls ImageMagick for every single image it produces. So when 64 Atlas parts are desired, ImageMagick must be called 64 times. This also means, that it has to read a particular photograph for every part, the photograph belongs to, again. In contrast to this behavior, our application holds as many photographs in memory as possible, to read them only once.

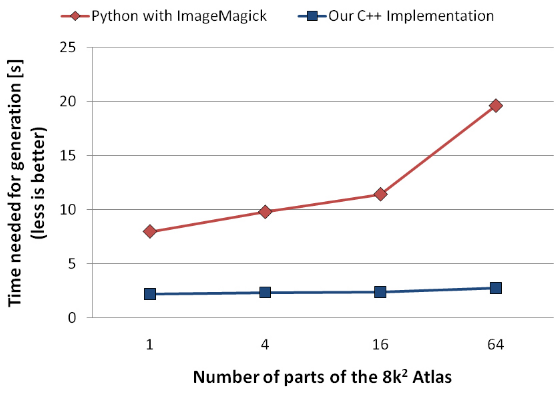


Figure 11: Performance of the generation of an 8k^2 Atlas using our C++ implementation compared to the existing one.

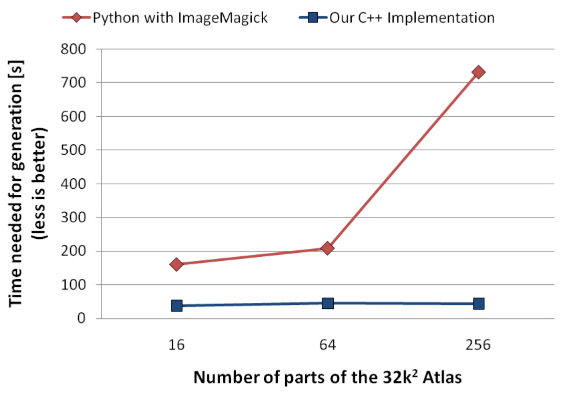


Figure 12: Performance of the generation of a 32k^2 Atlas using our C++ application compared to the existing one.

The generation of the Tile Store delivers even more dramatic results. As you can see in Figures 13 and 14, our application is at least ten times faster than the Python script. To be consistent, the python script has been altered to use ImageMagick instead of the Python Imaging Library. Because of the huge time consumption of the Python script, the generation of the Tile Store with a tile resolution of 128^2 has been omitted for the 32k^2 Atlas. With our application, this takes only 158.1 seconds. We also tested the generation of a 128k^2 Atlas with its corresponding Tile Store with a tile resolution of 128^2. The Atlas was generated in 9min 11s, the Tile Store (11 levels with 1,398,101 tiles) in 2h 58min 21s.

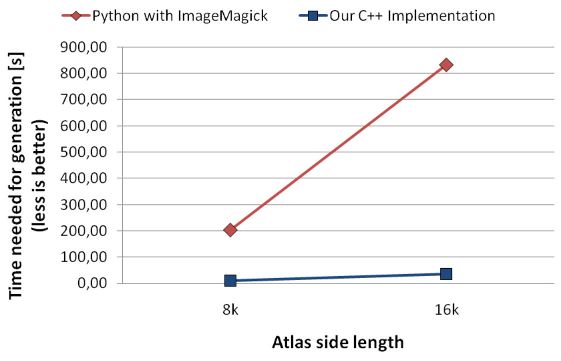


Figure 13: Performance of the generation of a Tile Store using our C++ implementation compared to the existing one. The tiles have a resolution of 128^2.

The update procedure again shows the speed-up when our application is used instead of a Python script. The Python script doing the update procedure was implemented to show a comparison. In Figure 15 you can see the times needed for an update.

The times for Atlas generation have been measured on a Hewlett Packard Pavilion dv6599eg notebook with an Intel Core2Duo T7300 processor with 2.0 GHz, 2 GB RAM and an nVidia GeForce 8400M GS graphics card. The times

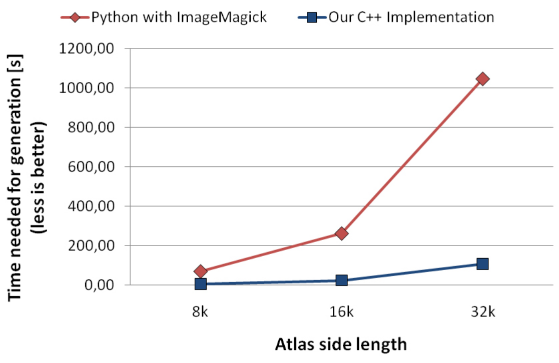


Figure 14: Performance of the generation of a Tile Store using our C++ implementation compared to the existing one. The tiles have a resolution of 256^2.

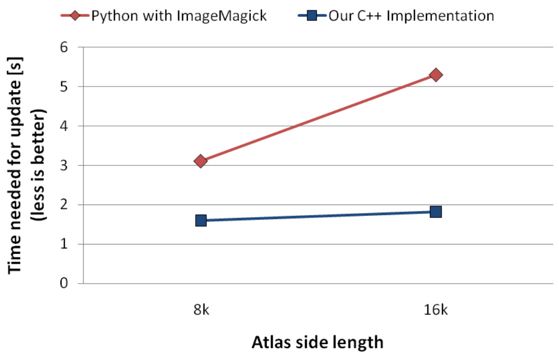


Figure 15: Performance of the update process using our C++ implementation compared to a Python script.

for Tile Store generation and update procedure have been measured on an Intel i7 2600k processor with 3.4 GHz, 8 GB RAM and an nVidia GeForce GTX 570. Only the 128k Atlas was also generated on the i7.

6 Conclusions and Future Work

We have presented an optimization of the workflow for a graphic artist, who is currently editing a large quantity of photographs used as textures for laser-scanned models. Our approach is based on the development of two applications and the introduction of Scanopy implementing Virtual Texturing into his workflow to ease his work. The first application is a mask generation program to visibly emphasize image areas corresponding to visible areas in the final model. Our second application is used for fast Atlas and Tile Store generation and update.

In future, the workflow could be further improved by transformation of the surrounding photographs into the plane of the actually edited photo. One approach to do this is the generation of a list with all photos and the corresponding neighbors, to know which of them have to be opened inside the editing program as a reference. This could decrease the number of checks inside the rendered 3D model.

Further, usage of the mask information for the Atlas and Tile Store generation might be useful to reduce the final size of the produced JPEG images, because of the higher compression rate when using input material with big homogeneous areas. An even smaller Atlas could be achieved by tightly packing of the visible image material. Therefore a transformation of the texture coordinates would be necessary, to reference the right areas inside this new Atlas.

A fully automated editing of the photographs corresponding to a model is desirable. This could be done with an approach based on Poisson Image Editing by Pérez, Gangnet and Blake [6]. There are already approaches for stitch-less image composition, that could be introduced to further ease the work of the graphic artist: [4], [3].

References

[1] Ahmed Abdelhafiz. *Integrating Digital Photogrammetry and Terrestrial Laser Scanning*. PhD thesis, Technical University Braunschweig, 2009.

[2] Paolo Cignoni, Massimiliano Corsini, and Guido Ranzuglia. *MeshLab: an open-source 3D mesh processing system*, April 2008.

[3] Ran Gal, Yonatan Wexler, Eyal Ofek, Hugues Hoppe, and Daniel Cohen-Or. Seamless montage for texturing models. *Comput. Graph. Forum*, pages 479–486, 2010.

[4] Anat Levin, Assaf Zomet, Shmuel Peleg, and Yair Weiss. Seamless image stitching in the gradient domain. In *ECCV (4)*, pages 377–389, 2004.

[5] Albert Julian Mayer. Virtual texturing. Master's thesis, Vienna University of Technology, 2010.

[6] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 313–318, New York, NY, USA, 2003. ACM.

[7] Lance Williams. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '83, pages 1–11, New York, NY, USA, 1983. ACM.

[8] Michael Wimmer and Claus Scheiblauer. Instant points. In *Proceedings Symposium on Point-Based Graphics 2006*, pages 129–136. Eurographics, Eurographics Association, July 2006.