FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Visualizing Archaeological Excavations based on Unity3D

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Thomas Trautner
Matrikelnummer 1125421

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Univ.-Doz. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr. Gerd Hesina
Dipl.-Ing. Dr. Christoph Traxler

Wien, 3. Juli 2014

Thomas Trautner	Eduard Gröller

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Visualizing Archaeological Excavations based on Unity3D

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Thomas Trautner
Registration Number 1125421

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof.Univ.-Doz. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing. Dr. Gerd Hesina
Dipl.-Ing. Dr. Christoph Traxler

Vienna, 3rd July, 2014

Thomas Trautner

Eduard Gröller

Erklärung zur Verfassung der Arbeit

Thomas Trautner
Erdbrustgasse 58/1, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Juli 2014

Thomas Trautner

Abstract

As part of an archaeological excavation huge amounts of different types of data, for example laser scan point-clouds, triangulated surface meshes, pictures or drawings of finds, find attributes like location, age, condition and description or layers of excavated earth are collected. This detailed documentation is important to give archaeologists the possibility to analyze the collected data at a later date since the find spot might not be accessible anymore. Unfortunately all the accumulated data is separately saved and consequently complex to explore.
Therefore we present a novel solution that allows the user to digitally explore a virtual archaeological excavation in real-time. With our approach we can not only visualize different types of textured meshes and finds but allow the user to draw on surfaces to mark areas of certain interest that need further exploration, enable explosion views to investigate composition of different layers of earth and arbitrary slicing of the three-dimensional mesh structure to better visualize cross-sections and an easier tracing of accumulation points of finds. The result of this work is a new powerful tool that will support the analysis of future excavations. All results and the implementation itself will be presented as part of this thesis.

An example screenshot of the final 3D viewer can be seen in Figure 1:

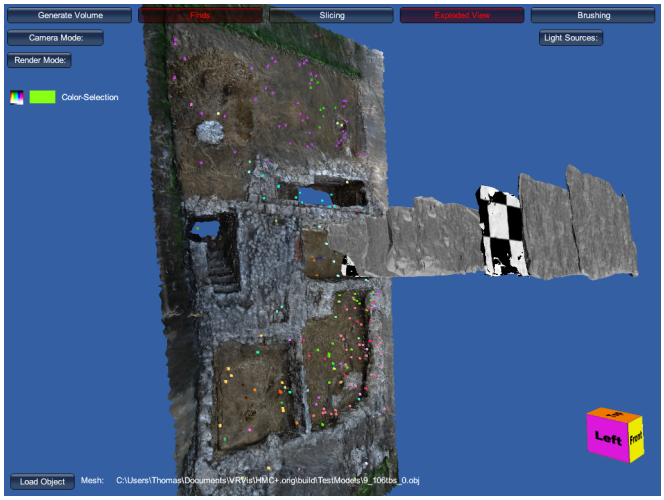

Figure 1: HMC+ 3D viewer in explosion view with finds enabled

Contents

CHAPTER 1

Introduction

1.1 General Information

This work complements the already implemented Harris Matrix Composer [5]. Our 3D viewer is an additional component of the new Harris Matrix Composer - Plus System. It is a real-time renderer with a GUI shown as a toolbox which allows the user to further explore and verify a given three-dimensional dataset. Therefore we allow the user to visualize selections and or combinations of all types of stratigraphic units within the Harris Matrix Composer. Figure 1.1. shows an example of a typical Harris Matrix.

1.2 Harris Matrix Composer

The Harris Matrix Composer (abbreviated below as "HMC") is an important tool archaeologists use for documentation of archaeological sites. It is based on the Harris Matrix which was invented in 1975 by Edward Harris. During an excavation this matrix is used to document the stratigraphic relations of dug out sediments. Every unit of stratification for example remains of wood or brick walls, a basement or inclusions is displayed as a single node in the hierarchical graph. A typical HMC matrix starts at the top with a circular green unit. It represents the top surface of the archaeological excavation, for example measured by a laser-scan. The lowest unit is as well a circular green unit, which stands for the lowest excavated stratigraphic layer. Every other unit in between consists of at least a top- and a bottom-surface or a hull of these two layers.

Additionally a HMC matrix enables an accurate assignment of the find locations. In Figure 1.1 we highlighted a subdivision which represents "room 1" of a ruin. Every path visualizes a different location and therefore a subdivision of the excavated area can be easily identified.

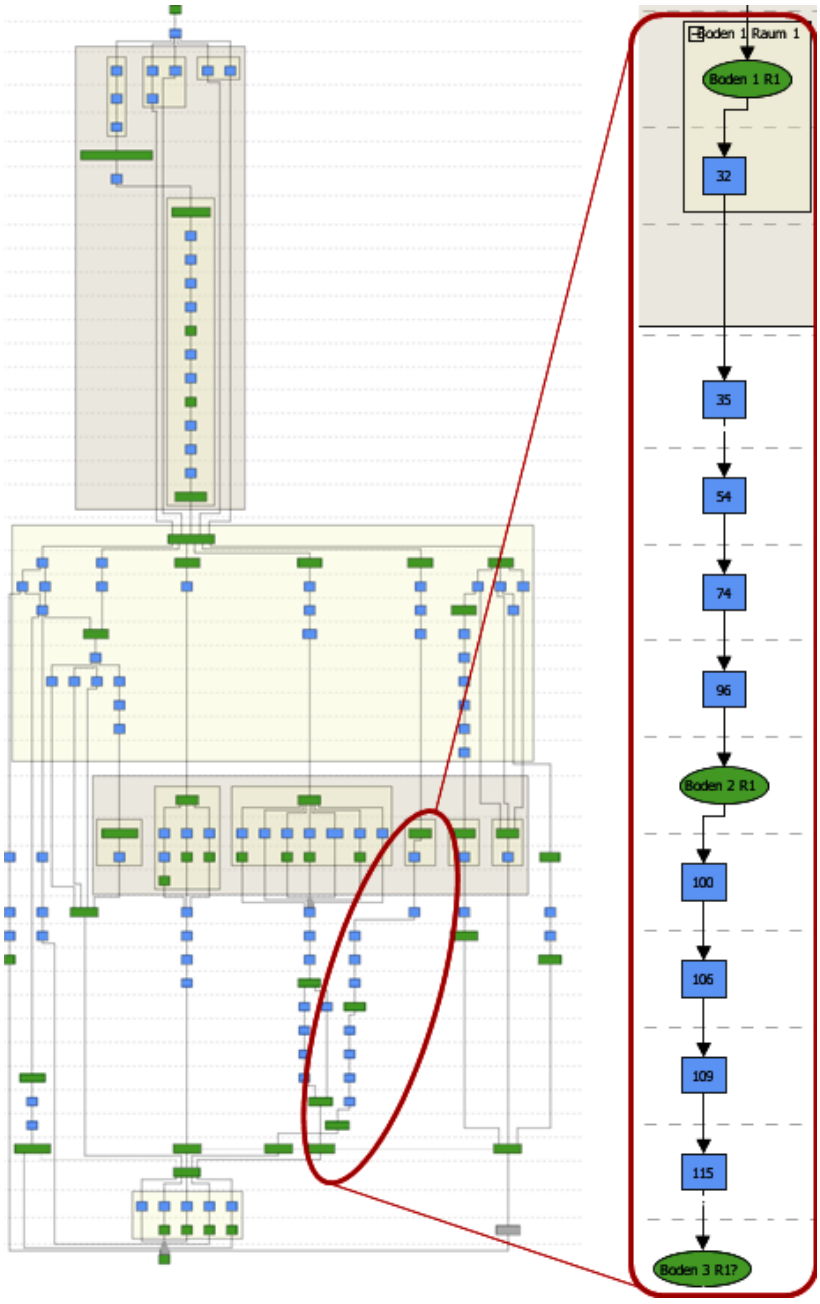

Figure 1.1: Example of the Harris Matrix Composer

1.3 Unity3D Game Engine

The resulting 3D-Viewer should be dynamically, customizable and easily extendable. Therefore we chose the Unity3D Game Engine [10] as an optimal framework for our implementation. Furthermore Unity 3D gives us the possibility to build the 3D-Viewer for different platforms.

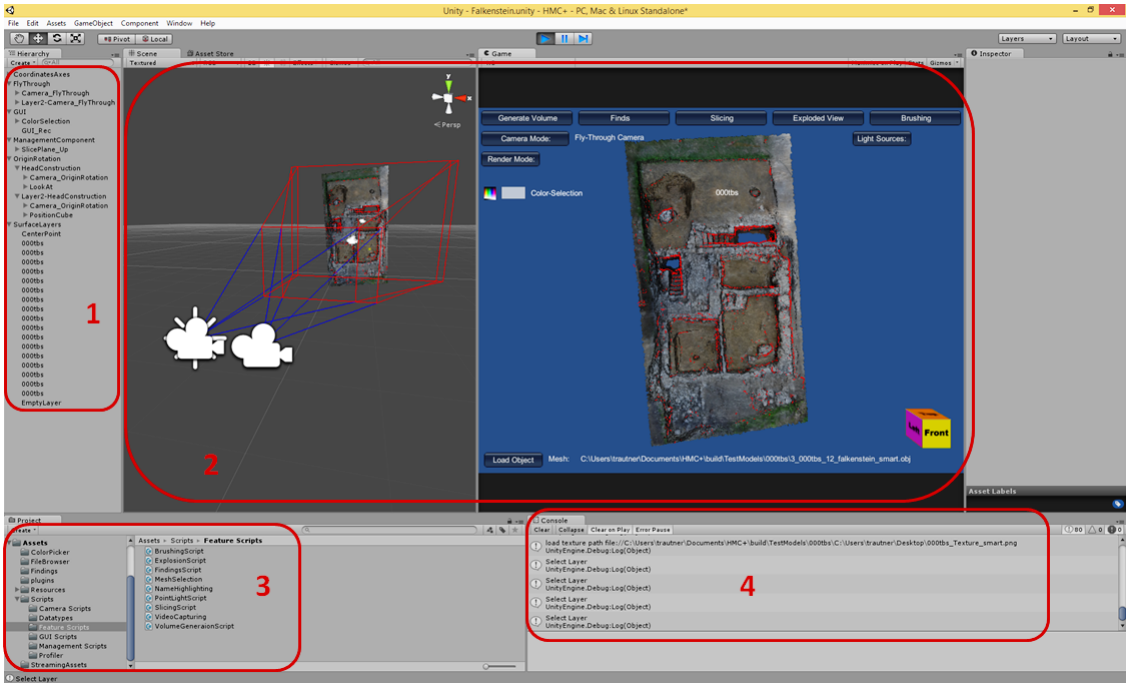

Figure 1.2: Unity3D 4.5.1 - Editor screenshot

Figure 1.2 shows an example scene developed with the Unity 3D Editor. On the left side, encircled in red and next to number 1 the current scene-graph is shown. It is a hierarchical list which includes every element of the current scene. This scene-graph for example consists of different types of cameras, several directional light-sources attached to cameras to illuminate the scene, GUI elements like the 3D Orientation Indicator, a color selection table, currently loaded and textured mesh surfaces and applied shaders.

The center next to number 2 is divided into two different displays. The left side represents the scene with additional debug information and is called "Scene-Window". In this case two cameras and a single light-source is visible. According to the debug information we know that both cameras have a truncated pyramid frustum. Therefore we can easily tell that both use a perspective projection. A view-frustum represents the visible area of a camera and is defined by the near- and far-plane. In this example the

visible area is bounded by red and the invisible area between camera and near-plane by blue lines. The reason why we use two cameras in this scene is to support stereo display. Therefore the left camera renders images for the left eye and the right camera for the right eye. The distance between both cameras simulates the eye distance and can be changed dynamically. For more information about stereo rendering see Section 3.3.2

The right side of number 2 shows the current "Game-Window", which is only visible if the play-button in the top-center, above from number 2 is pressed. Then the scene is rendered by the cameras we saw in the Scene-Window before. Additionally we already see the GUI which was not visible in the scene mode including the yellow (front), orange (top) and purple (left) side of the 3D Orientation Indicator telling us from what side we are currently looking at the scene.

In the bottom left corner next to number 3, a file browser that shows all available files and directories is visible:

- **Scripts:** In this example the "Feature Script" folder which contains nine different C# scripts is displayed. If one of those scripts derivers from "MonoBehaviour" it can be added by drag-and-drop to one of the Game Objects of the scene-graph which is visible in number 1. Such a script consists of an update function which is called once per frame and for every scene object it is attached to.

- **Materials:** Apart from scripts, all materials including their main texture and attached shaders can be displayed. By default they are attached to spheres, to better visualize applied illumination models and shader properties.

- **Models:** Additionally the explorer allows us to search for 3D models. In this case the already mentioned 3D Orientation Indicator object (*.obj) file or a Collada (*.dae) coordinate-cross is shown.

- **Prefabrications:** Finally the explorer allows us to search for prefabrications which can be downloaded via the Unity 3D Asset Store. For example the renderer we developed uses a so-called "prefab" to simplify color selection. It allows the user to specify the RGB values for the red, green and blue channel. Additionally the overall alpha channel can be adjusted.

In the bottom right corner of Figure 1.2 next to number 4, a console that shows current debug outputs is visible. If a certain layer is selected or a mesh with additional texture was loaded it will be logged by the console. This information is marked as debug information and will therefore not be visible to the user in the final build-version. Furthermore thrown compiler errors or warnings are displayed. This is particularly advantageous during the development phase.

After loading a layer or scene during runtime or in the edit-mode, the "Inspector" becomes visible. Figure 1.3 shows an example output of the 000tbs ground layer.

Section 1: In the first section the name of the mesh and a checkbox to en- and disable the object is visible. Additionally a tag and layer can be chosen. Tags are important to find certain Game Objects during runtime. A script can for example search for all Game Objects with the tag "000tbs" which returns an array of all objects. The layer is used for the rendering process. Every camera has a list of visible layers attached to it and only layers from that list will be rendered. This is important to mask certain layers as in case of the 3D Orientation Indicator.

Section 2: Section two simply tells the user that the currently selected Game Object is a mesh which has a mesh renderer attached to it. If the mesh renderer is deselected it will not be rendered and is therefore not visible anymore. Additionally the user can select if the mesh casts and/ or receives shadows and set the number of materials applied to it.

Section 3: Section three shows a very important setting for our renderer. It states that the selected mesh has a collider in the form of the mesh itself. This is on one hand computationally expensive especially the calculation of intersections, on the other hand indispensable for visual analysis. If the user for example selects a mesh we simply trace a ray from the mouse cursor and check if this ray intersects a mesh collider.

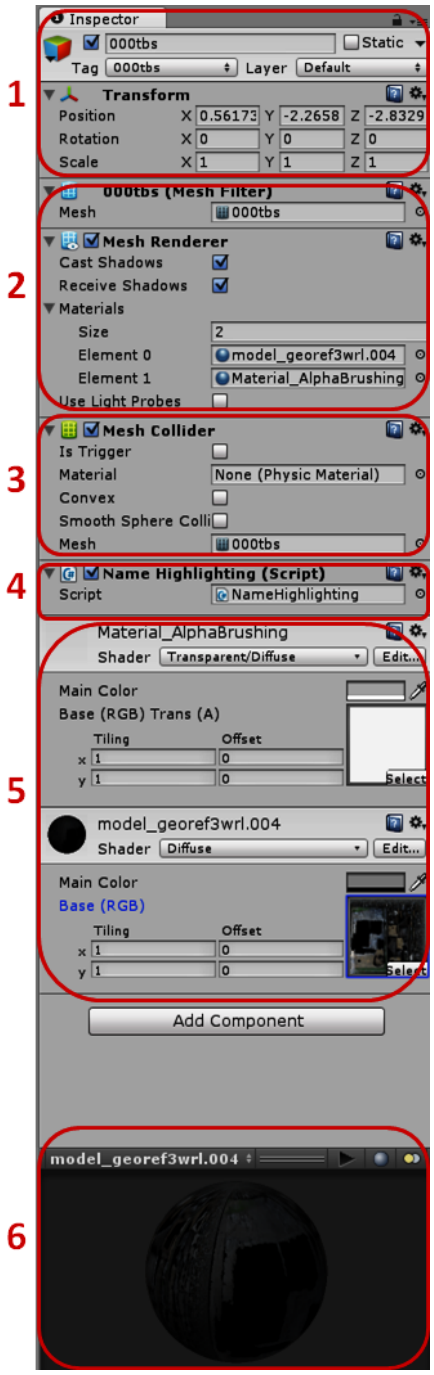

Figure 1.3: Inspector after loading a mesh during runtime

Section 4: The same tracing procedure is used to label a mesh with its name if the mouse cursor is moved over it. We achieve this by adding certain scripts such as "name highlighting" from section 4 to a Game Object.

Section 5 and 6: To provide surface brushing we need at least a main texture with diffuse illumination and a transparent alpha texture with a transparent shader and a diffuse illumination which can be seen in section 5.

An example of this is visible in section 6. As already mentioned they are attached to spheres to better visualize applied illumination models and shader properties.

CHAPTER 2

Related Work

A very similar approach of visualizing archaeological excavations is presented by [CIG+01]. They propose a tool to visualize building parts, finds, stratigraphical layers and textures in situ and furthermore provide advanced 3D reconstruction techniques. This is done by storing all different types of data in a multimedia database. The result is used for presentation or publication purposes.

Another approach presents a virtual reality tool [BdCTV03] which allows the user to measure stratigraphical units like curvature, length, thickness, height or volume to enhance archaeological fieldwork. Furthermore it allows the user to calculate volumes and simulate archaeological formations and deformation processes which is essential for understanding an excavation.

The work of Vote [Vot01] focuses on visualizing the Great Temple of Petra based on post-excavation archaeological analysis. Therefore four different prototypes were developed and analyzed. A final concluding survey shows that a combination of classic and new techniques is most promising for future excavation.

Dellepiane et al. [DDC+12] present amongst others a technique of interactive slicing where a slicing plane is moved further away toward the viewing direction to visualize disparity between two different time steps. A technique that is also very useful for archaeological visualizations.

Benko et al. [BIF03] focus on a mixed-reality approach implemented for multiple users. Furthermore it supports the tracking of see-through head mounted displays and multi-touch table surfaces. The final visualization is similar to our approach based on the Harris Matrix.

Additionally we considered a summary [Pat09] of five papers that propose different approaches to visualize seismic measurements. This is done to allow the user to further explore and analyze possible oil or gas reserves. For optimal use the author recommends the following expressive visualization and rapid interaction techniques: time of creation of such an illustration must be reduced, the program must independently recognize important stratigraphical layers and the exploration should be as simple as possible for the user.

Remondino and Campana [RC08] present image based approaches to capture detailed 3D information of archaeological excavations. This is done with cost-efficient approaches like taking pictures of finds from different angles and finally computing the three-dimensional structure. Therefore we implemented a run-time object loader for our 3D viewer which is able to load and finally visualize such models as well.

A similar low-cost approach is presented by Doneus et al. [DVF+11]. Instead of expensive techniques like measurements with laser scanners they use a technology called "Structure from Motion" and create a three-dimensional point cloud. Furthermore this approach requires only a minimal technical knowledge and user interaction and is therefore straightforward to use.

Allen et al. [AFT+04] introduce a 3D Modeling Pipeline for archaeological excavations and finds. First the excavation site is either scanned by a laser scanner or pictured with cameras. In the next step the site is represented as 3D model and additional context like background-images, videos or GIS data is added.

Santos et al. [SCBP05] focus on realistic rendering of archaeological excavations. Therefore they concentrate on illumination methods like global illumination. The paper presents different approaches that guarantee a frame-rate of at least 10 fps or more and the possibility to change the view point dynamically. Since lighting is such an important factor for a realistic perception of a scene, we allow the user to dynamically place light sources in the scene with our 3D renderer.

To provide stereoscopic rendering we implemented a technique called off-axis rendering which was described by Grasberger [Gra08]. It presents different approaches of stereo rendering and explains in detail their use, the advantages and disadvantages.

CHAPTER 3

Implementation

Our main goal was building a tool that allows the user to scientifically visualize archaeological excavations. The resulting renderer supports the import of high-resolution geometry and very large geometric models.

In the beginning of our implementation we started using Unity 3.5.7 but upgraded afterwards to Unity 4.5.1 to be able to use all the latest features. One important feature was wire-frame rendering. Before the upgrade we had to manually implement a line-rendering algorithm to provide wire-frame rendering. Unfortunately it required too much computation power and memory consumption.
Furthermore we chose C# as programming-language for our scripts.

3.1 Technical Architecture

All our implemented scripts can be divided into four categories which can be seen in Figure 3.1:

1. **Camera Scripts:** Scripts from this category are responsible for different types of cameras like fly-through or origin-rotation, stereo rendering using an off-axis projection matrix and slicing views.

2. **Feature Scripts:** This category is responsible for scripts that are used for implementing features and improving the overall data exploration. In an alphabetical order:

   - a brushing tool to paint on mesh surfaces
   - explosion algorithms to increase the offset between the layers
   - a finds visualization with location and detailed description
   - all types of mesh selections
   - highlighting names
   - adding further light-sources like point lights
   - providing slicing operations for cross-section views of the scenery
   - screenshot or video capturing
   - volume generation

   A detailed description of all mentioned features can be found under Section 3.2.

3. **GUI Scripts:** This category is responsible for the graphical user interface. Here we define the layout, color settings and button labeling. Additionally we use scripts to distribute clicked button events and enable and disabled feature selections.

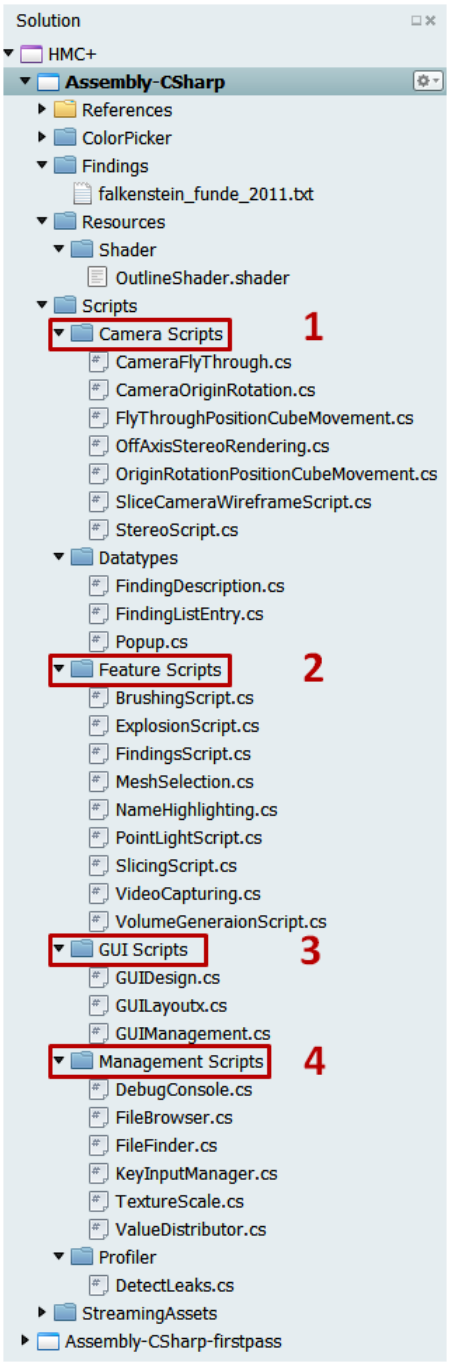

Figure 3.1: Architecture using MonoDevelop

4. **Management Scripts:** We used the management category to divide scripts in subcategories for overall management tasks and debug or profiler tasks. Overall management tasks include a file browser to search directories and files during runtime, a key-input manager to handle user input and change keyboard settings and a value-distributor to distribute current settings within the renderer. It includes for example current mesh selections, active cameras, selected render-modes, active light-sources or current camera movements. This is done by implementing a typical singleton pattern and only distributing a single instance. Therefore the constructor cannot be accessed from outside the class.

3.2 Features

In the following sections we want to discuss all mentioned features from Section 3.1.

3.2.1 Object import

We wanted the rendering software to be as independent and compact as possible. Therefore we implemented algorithms and partially extended them with already existing libraries, for example the file dialog [4]. It allows us to display different folder structures including all their files during runtime. Additionally we applied a filter function that allows the user to select only files with an (*.obj) extension. If a file is selected, we expect that the corresponding material (*.mtl) is located in the same folder. To select a file the user has to simply click the "Load Object" button in the lower left corner of the screen.

After clicking the button a new file dialog opens which can be seen in Figure 3.2. Additionally all object files are colored in light grey to mark them as selectable and all material and texture files are colored in dark grey. If the mouse cursor is moved over a name the whole line will be colored in light blue and if the user clicks on a file it will be locked and marked in dark blue. After choosing a file the user can either click the "Select" button in the lower right corner to load this file or "Cancel" to return to the rendering view.

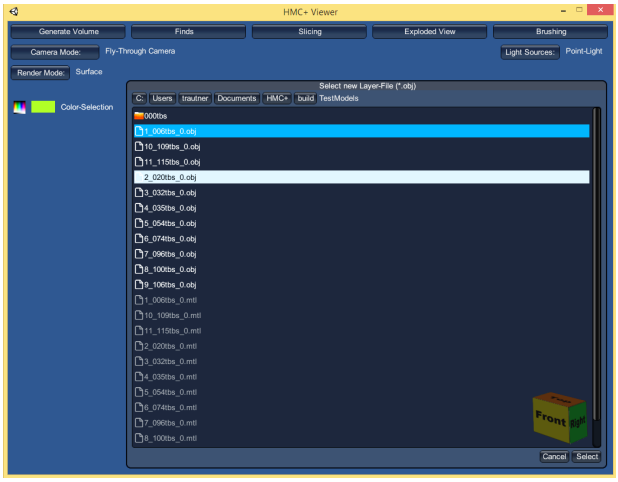

Figure 3.2: File dialog and current selection

Instead of using the "Select" button the user can simply double-click on the file to load it as well. Furthermore we expect the following naming restrictions to guarantee a successful mesh import:

[**0-9**] _ [**000-350**] tbs_ [**0-99**] _ [**A-Za-z0-9**]? .obj
[**Sub-Mesh Number**] _ [**Name**] _ [**Room Number**] _ [**Additional Information**].obj

Examples:
0_000tbs_0_falkenstein_smart.obj
0_000tbs_0_falkenstein_smart.obj
1_000tbs_0_falkenstein_smart.obj
1_000tbs_0_falkenstein_smart.obj
2_006tbs_1_falkenstein_smart.obj
3_032tbs_1_falkenstein_smart.obj

1. **Sub-mesh number:** We use this index to detect if a mesh consists out of smaller sub-meshes. In the given example the first two objects are named "000tbs" and belong to the same sub-mesh with index 0. If one of them is selected by the user during a load object operation, the other one will automatically be loaded as well.

2. **Name:** This name is displayed if the mouse cursor is moved over the mesh in our rendering window. The name consists out of three digits that represent the layer and the acronym "tbs" which stands for "top-bottom surface". All sub-meshes that belong to the same mesh should have the same name to avoid user distraction. The lowest layer of excavation should always be called 000tbs. All other layers can be called according to their HMC-label although the highest possible number is currently 350tbs. This limitation results from the current maximum number of tags that are assigned dynamically during runtime. Unfortunately tags must be created manually before building the renderer and cannot be created depending on the variable which is then passed during run-time.

3. **Room number:** This number is important to enable future versions of the renderer to perform manipulations separately depending on the room number of the currently selected mesh. For example only explode layers of a single room. The current version of the renderer explodes all layers equally independent from the selected mesh and room it belongs to.

4. **Additional information:** This part is not required by the renderer. It can be used to store additional information from the user. If there is no need for further information it can be left out.

Restrictions of external libraries: Unfortunately our external triangulation library [9] and Unity3D itself have a maximum vertex count limit of 65.534 vertices per mesh. If such a mesh is imported within the editor, Unity3D will automatically divide it into

default sub-meshes. After building the project or during runtime this is not possible anymore. Therefore we expect the user to either load smaller meshes or divide meshes which have more vertices into sub-meshes by hand in a preprocessing step. If a sub-mesh is selected during runtime our renderer will automatically look for corresponding meshes and will load all of them without any additional user interaction into the current scene. This is done by searching for meshes with the same name and sub-mesh number as described above. As soon as the mesh is loaded, the renderer will look for the texture that the materialfile is referring to. If the texture was already loaded it will not be imported again. To enable this we implemented a texture-collection that administrates all texture. By only using the external triangulation library [9] we were facing a memory consumption of more than 3GByte RAM after loading a couple of sub-meshes with less than 500.000 vertices. If the renderer does not find the texture or if the mesh has no texture coordinates, the mesh will be colored in light grey by default.

3.2.2 Mesh selection

If a layer is selected by clicking the left mouse button, the outline of the selected object is changed to red which can be seen in Figure 3.3. As we can see the mesh is still textured but the shader is changed from a simple diffuse shader to an outline-shader with red as its main color.

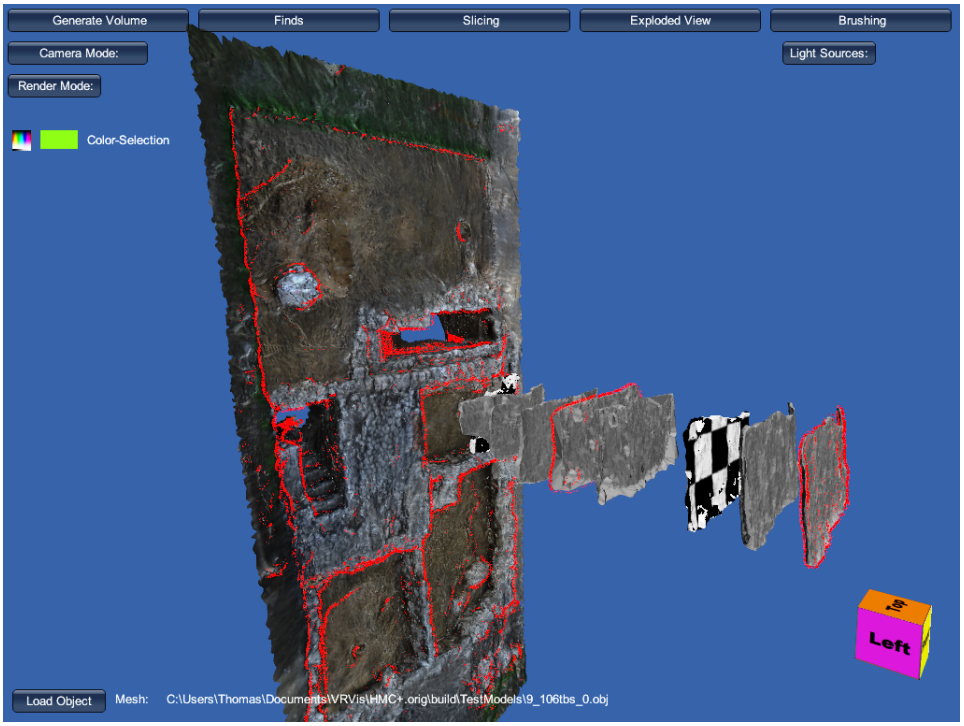

Figure 3.3: Mesh selection with three selected layers

If the "Ctrl" button is pressed and another object is selected, both objects are selected and their shader is changed to an outline-shader. In Figure 3.3. three different layers were selected. First of all the 000tbs ground layer then layer 074tbs in the middle and the highest and therefore latest layer 006tbs on the most right side of the screen. If the "Ctrl" button is not clicked the first selected object will be deselected after another element is selected. By clicking on the background all currently selected meshes will be deselected.

Functioning of the outline shader

The dot product of two vectors $\vec{a} = [a_1, a_2, ..., a_n]$ and $\vec{b} = [b_1, b_2, ..., b_n]$ is defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_n b_n$$

The result is a scalar which represents the angle between two vectors. If the dot product of the current viewing vector and the surface normal is zero , we simply change the pixel color to the outline color. If the dot product is not zero we do a texture look up for the current pixel and calculate the dot product of the surface normal and the light direction to illuminate the pixel with a diffuse illumination model.

Note: *Usually a triangle is not visible if its normal and the viewing vector have a dot product of 0. Therefore we would need a dot product very close to 0 to still see the red color of the surface. We chose a tolerance range of [87°, 93°] for reasons of simplicity and better illustration.*

An example of this algorithm can be seen in Figure 3.4. The black polygon line represents an example layer which has an outline-shader attached to it. Every triangle of this mesh has a surface normal which is represented by the vectors called **n**. The current camera position is represented by the eye with the corresponding viewing vectors called **v**. If the dot product is zero, the surface color will be changed to red, which is represented by the red angle in the middle of Figure 3.4. In this case vector **n** is orthogonal to vector **v**.

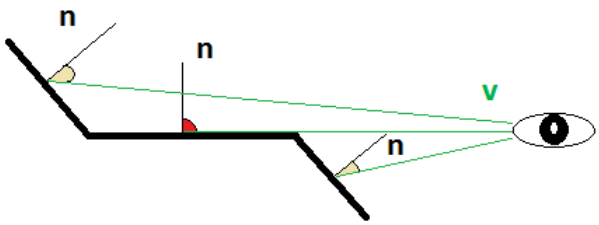

Figure 3.4: Example of an outline-shader

The reason why the border of the mesh which can be seen in Figure 3.3. is not always red is simple to explain. Due to the fact that the layer is more or less a simple rough and bumpy plane the dot product can be larger than zero at the border. If we always want the border to be an outline we need to load a very high resolution convex hull instead of a plane, in the simplest case a sphere. This guarantees that the boundary gets an outline.
If the hull is concave in certain cases no outline will be drawn. An example of this can be seen in Figure 3.5. Here we visualize a plane with a missing border and a concave polygon with two viewpoints from which a correct outline is drawn.

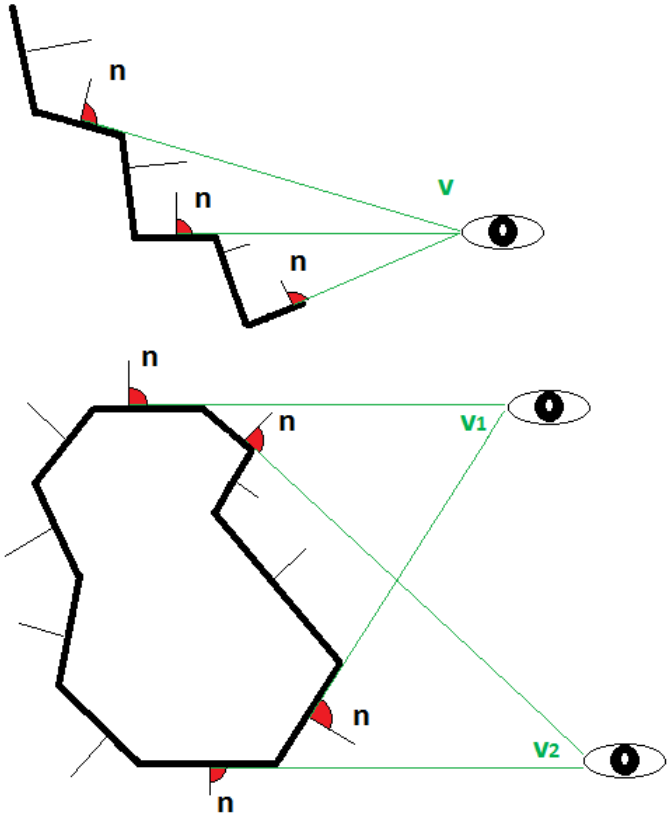

Figure 3.5: Example of low resolution meshes with an outline-shader

3.2.3 Show finds

During the excavation process usually different items or materials like coins, glass, wood, porcelain, iron or others are found. To provide interactive exploration of find locations our renderer is capable of visualizing them as well. If the user selects the "Finds" button in the top left corner of the screen, all documented finds are imported. An example of this can be seen in Figure 3.6.

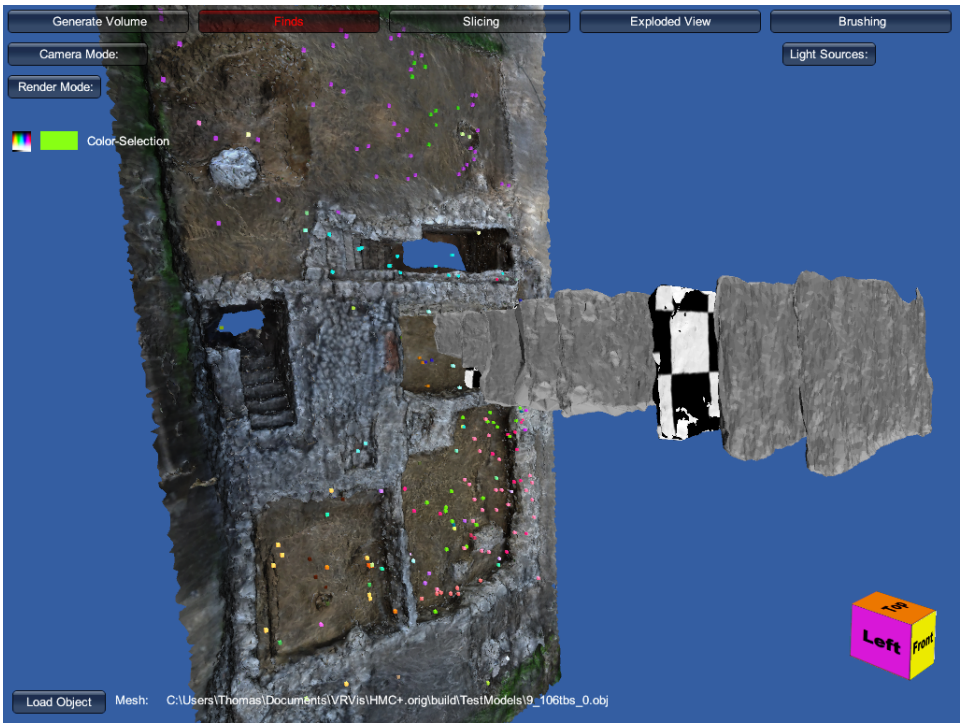

Figure 3.6: Show finds

Finds are usually represented by a point-cloud file which stores all X-, Y- and Z-coordinates (*.obj) and an additional description file (*.txt), which includes the layer-number where it was found, the material and an additional description for every find.

As soon as the renderer has imported all finds, colored boxes will be displayed. Every box represents a single find. The color is chosen randomly but finds with similar materials are always colored the same. For example every silver coin is colored red and every copper coin blue. If the user presses the C-key on the keyboard, the color distribution will be changed. Then all finds of a certain layer will get the same color. This feature allows the user to easily see which finds were found in which layer.

Example of the description file:

11	**Glas**	**Fragment Glas**
14	**Keramik**	**Teller 4 Teile**
11	**Keramik**	**Perle**
11	**Metall**	**Münze**
11	**Metall**	**Haken**
13	**Metall**	**Kellengriff**
7	**Stein**	**Flintenstein**
7	**Glas**	**Fläschchen Boden**
7	**Metall**	**Zange**

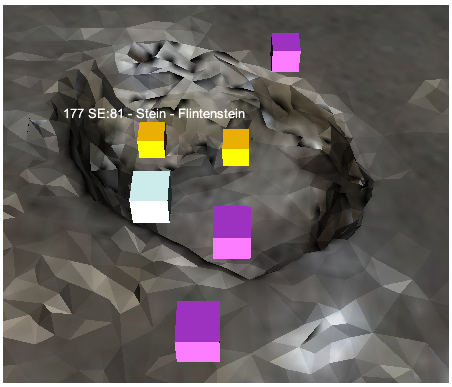

Figure 3.7: Example finds

Figure 3.7 shows a hole in which different finds were found.

Figure 3.8: Example finds of an excavation

Figure 3.8 shows an overview of an excavation site. If the mouse cursor moves over a find, its description is shown. This can be seen in the lower left corner. The description says "203 SE: 75 - Silber - 3 Kreuzer, Leopold I, 1699". It represents a silver coin with sequence number 203 which was found in surface layer 75. Additionally it has a dating of 1699, the time of Leopold the First.

3.2.4 Slice surface-layers

After the user has selected a layer, the "Slicing" button can be used to enable the slicing mode. The renderer will automatically calculate the bounding-box of the selected mesh and a slicing-plane is shown. This can be seen in Figure 3.9 which shows the debug output of our renderer. The dimension of the slicing plane depends on the dimensions of the bounding box. To guarantee that the slicing-plane is always large enough to cut through the whole currently selected layer, the greatest dimension of the bounding-box determines the width of the slicing-plane.

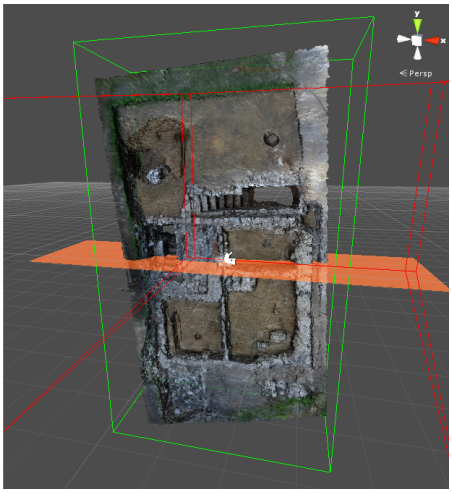

Figure 3.9: Bounding-box of the selected mesh

By using the num-pad the user can change the position and orientation of the slicing plane. This can be seen in Figure 3.10 Additionally we want the user to be able to place the plane exactly at the desired location. To simplify this task the slicing plane is semitransparent and its color is very different from ordinary earth colors to further strengthen the contrast.

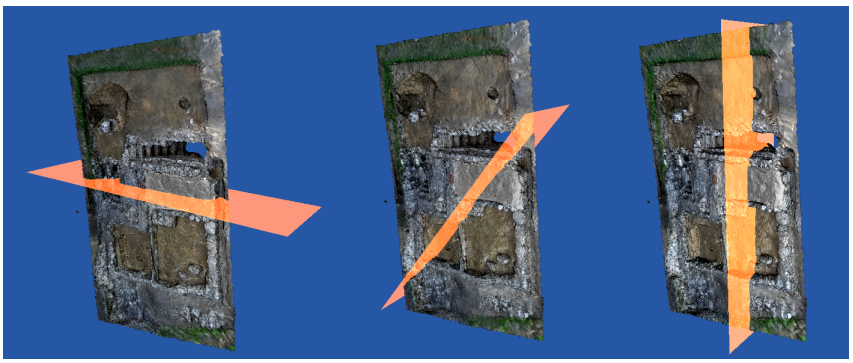

Figure 3.10: Different positions of the slicing plane

If the "Space"-Key is pressed, the renderer changes to the slicing view. The slicing view uses two orthographic cameras to visualize cross-sections of surfaces. An example of the sliced model can be seen in Figure 3.11 Additionally all the finds are visible and room 1 is currently in explosion mode. Even if the slicing view was already enabled it is still possible to move the slicing plane with the num-pad and cut through the model in real-time. This allows the user to analyze cross-sections of all layers until certain finds or artifacts are visible.

Figure 3.11: Two different slicing views

Figure 3.12 shows that both camera near-planes are almost equal to the slicing plane. We did this to automatically slice the selected mesh at the point of interest and used a minimal offset to avoid z-fighting if both planes are too close to each other. If the slicing plane is moved both cameras are moved perpendicularly and therefore their near-planes, which are used to render the cross-section of the mesh, change. Figure 3.12 shows an illustration of how we achieved the slicing.

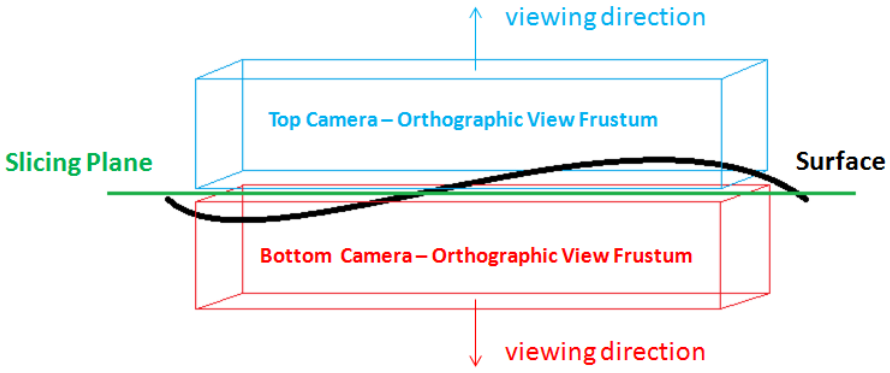

Figure 3.12: Explanation of the slicing mode

3.2.5 Explosion view

Another important feature we implemented allows the user to explore the structure of all stratigraphical layers of an excavation in explosion view. Using this type of visualization the hierarchical order and composition of all layers are better visible and therefore easier to analyze.

The explosion view is enabled by either pressing the "Explosion View" button or holding the "E" key on the keyboard. As soon as the explosion view is enabled, the offset of the layers to each other can be changed with the mouse-wheel. The resulting animation steps can be seen in Figure 3.13.

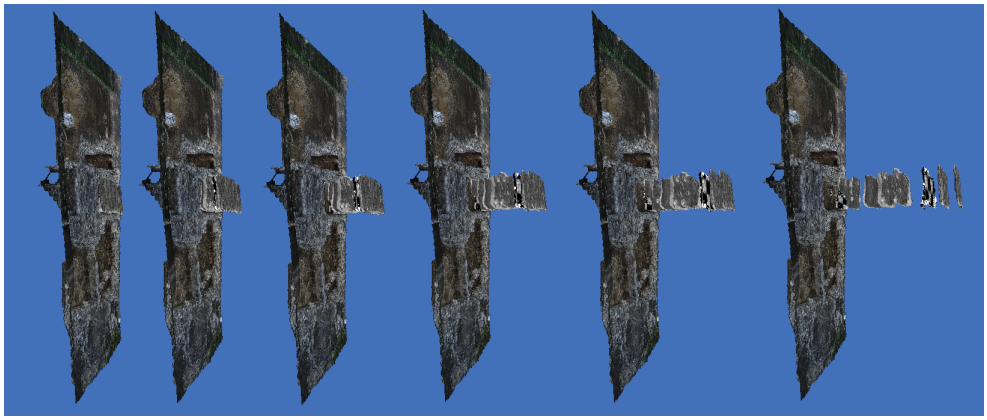

Figure 3.13: Explosion view animation

Figure 3.14 shows a graphical explanation of our algorithm. The expander variable is a float value which controls the factor of explosion. Depending on the number of layers it is multiplied with k minus current layer index. Therefore the collection of expandable layers is always sorted in descending order. The sign factor depends on the direction in which the mouse wheel was turned. With this adaption the same algorithm can either perform an explosion to enlarge the offset or an inverse explosion and move all layers back to their original position.

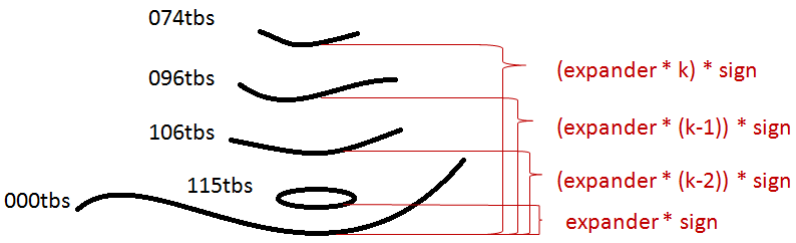

Figure 3.14: Simplified explosion view explanation

Explosion - Source code

The following source-code is used to perform the explosion. The variable valueDistributor is an instance of the already mentioned management script which distributes values within the renderer. It is instanced with ValueDistributor.getInstance() and has a private constructor. Input.GetAxis("Mouse ScrollWheel") returns a float value that represents the angle of the mouse-wheel rotation. Depending on the direction the mouse-wheel is turned we determine the sign which influences the direction of the explosion.

```
public void PerformExplosion(){

        // delta movement that happend in the last frame
        mWheelDelta = Input.GetAxis("Mouse ScrollWheel");

        if(mWheelDelta != 0){

                //decide if explosion or inverse explosion
                if(mWheelDelta < 0){
                        sign = -1f;

                        if(vd.GetExplosionFactor() > 0){

                                vd.SetExplosionFactor(vd.GetExplosionFactor()-1);

                                oncePerWheel = true;
                        }
                }else{
                        sign = 1f;

                        vd.SetExplosionFactor(vd.GetExplosionFactor()+1);
                }

                //wheel was changed perform action
                if(vd.GetExplosionFactor() > 0) oncePerWheel = true;
        }

        if(oncePerWheel == true){

                //Update Current list
                UpdateSurfaceDictionary();

                accumulatedSum = expander * sign * Time.deltaTime;

                foreach(string s in surfaceDictionary.Keys){

                        //Move every layer besider ground level --> 000tbs
                        if(s != "000tbs"){

                                GameObject[] currentSubmeshes = new GameObject[25];
                                surfaceDictionary.TryGetValue(s, out currentSubmeshes);

                                foreach(GameObject surface in currentSubmeshes){

                                        surface.transform.position = new Vector3(
                                                surface.transform.position.x,
                                                surface.transform.position.y,
                                                surface.transform.position.z-accumulatedSum);
                                }

                                accumulatedSum += expander*sign;
                        }
                }
        }

        oncePerWheel = false;
}
```

3.2.6 Brushing

The brushing tool can be used to mark certain regions of interest. To enable this we implemented multilayer-texturing using an additional alpha texture. If the renderer imports a mesh it will have two texture layers. The fist layer is the texture that the material file was pointing at. The second layer will be added by the renderer. It is a simple alpha-texture that is used for all the brushing operations. After selecting a color and enabling the brushing tool, the user can use the left mouse button to draw and the right button to erase drawings.
The brush size is calculated dynamically whereas k stands for the kernel size. We only support quadratic kernels so an example for a 3x3 kernel could look like as shown in Figure 3.15:

1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))
1 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))
1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))

Figure 3.15: Proceeding for a 3x3 kernel

The value inside a cell represents the percentage of the brush-color that is added to the alpha texture. For example 2 * (1/(Floor(kernelSize/2) + 1)) can be simplified to 2 * (1/(1+1)) which is equal to 1. With this formula the center will always be 1, therefore 100% of the new color is added. If we increase the distance to the center, the original color intensity decreases. In this case the outer circle will only add 50% of the selected color (see Figure 3.16).

50%	50%	50%
50%	100%	50%
50%	50%	50%

Figure 3.16: Exact values of the 3x3 kernel

If we apply a 5x5 kernel the color gradient will be smoother. An example of the different brushing kernels can be seen in Figure 3.17.

1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))
1 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))
1 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	3 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))
1 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	2 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))
1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))	1 * (1/(Floor(kernelSize/2) + 1))

33.33 %	33.33 %	33.33 %	33.33 %	33.33 %
33.33 %	66.66 %	66.66 %	66.66 %	33.33 %
33.33 %	66.66 %	100%	66.66 %	33.33 %
33.33 %	66.66 %	66.66 %	66.66 %	33.33 %
33.33 %	33.33 %	33.33 %	33.33 %	33.33 %

Figure 3.17: Example of a 5x5 kernel

Another characteristic can be observed in Figure 3.18. If the brushing is repeatedly performed on the same spot the opacity of the color increases.

In the top left corner of Figure 3.18 the color selection is visible. It can be used to select RGBA colors either by simply clicking on a color or by inserting values from 0 to 255 for every red, green, blue and alpha channel. After the brushing the alpha texture can be exported by using the "T" key of the keyboard. It will be saved as (*.png) in the folder for screenshots of the opened scene. In addition, the date and time is recorded in the name of the file.

For example:

6-18-2014_10-42-50_AMScreenshot_44

[**month**]-[**day**]-[**year**]_[**hour**]-[**min**]-[**sec**]_[**AM/PM**]Screenshot_[**sequence number**].png

An example of how marked regions could look like can be seen in Figure 3.19.

To remove drawings, they are changed back to the original alpha value. For ideal results we recommend a erase-kernel size that has at least two or three times the size of the brushing kernel.

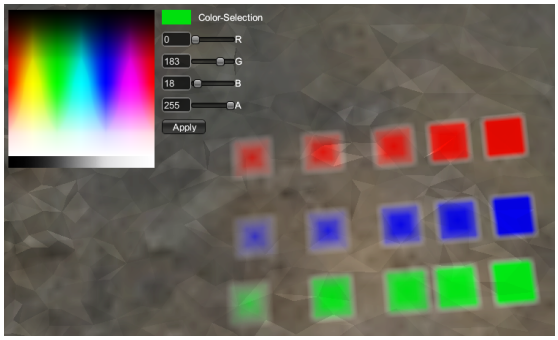

Figure 3.18: Color selection and kernel with different opacities

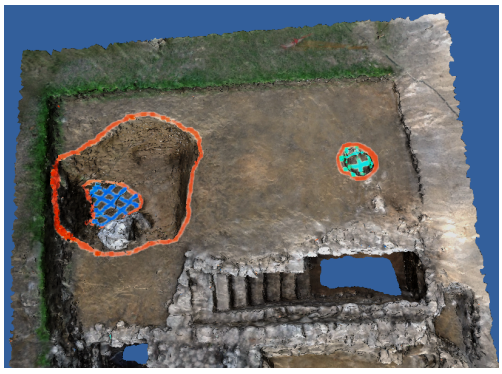

Figure 3.19: Example of marked regions

3.2.7 Adding light sources

To enhance the perception of the structure of the reconstructed geometry the scene can be illuminated manually during runtime. After pressing the "P" key, an additional light source is added to the scene. The current version of the renderer supports point-light, however future versions will include different types of light sources for example spot-lights or directional-lights.

If a light source was added to the scene, its intensity can be changed by using the mouse-wheel. An example of how different intensities could look like is shown in Figure 3.20. To manipulate the light source position three arrows representing the X-, Y- and Z-axis are displayed which can be seen in Figure 3.21. By clicking on one of them with either the left or right mouse button their position can be moved up or down along this axis respectively. Thereby we guarantee that even after a light-source was added to the scene its intensity and position can be changed.

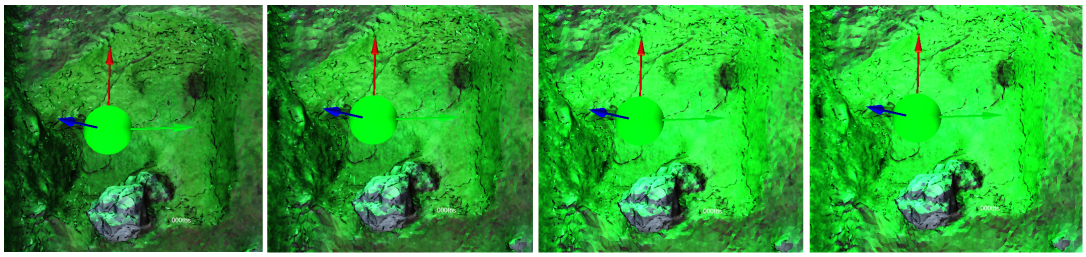

Figure 3.20: Controlling the intensity of the point-light source using the mouse-wheel

An example of the scene using different light-source colors and intensities can be seen in Figure 3.22. It shows the scene illuminated by four different light-sources. An orange light-source in the top left corner, a turquoise one in the top right corner, a purple one in the left center and a green light-source in the bottom right corner.

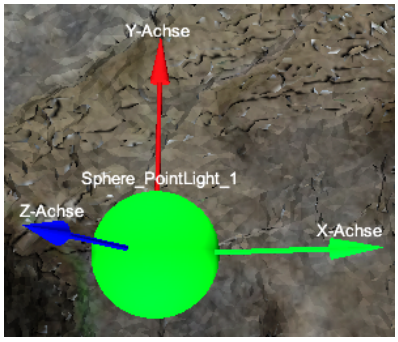

Figure 3.21: Controlling of the point-light source

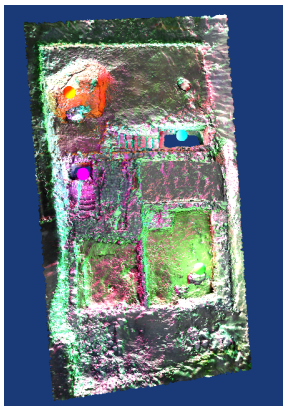

Figure 3.22: Different light-sources

3.2.8 Name highlighting

If the mouse cursor moves over a mesh we automatically display the name of the mesh. We do this by continuously emitting rays from the mouse position into viewing-direction. If such a ray hits a mesh-collider we print a label with the corresponding mesh name next to the cursor. Thereby we allow the user to interactively explore the given dataset. The same highlighting technique is used to label light-sources and coordinate axes which was shown in Figure 3.21.

3.3 Additional GUI-Settings

Apart from the already mentioned exploration features we allow the user to change some basic settings of the rendering system as well. These adjustments include the actual type of camera and the used render mode.

3.3.1 Camera modes

Currently we support two different camera modes. The first mode which can be seen in Figure 3.23 is called "Origin-Rotation-Mode" and the second mode which is visible in Figure 3.24 is called "Fly-Through-Mode".

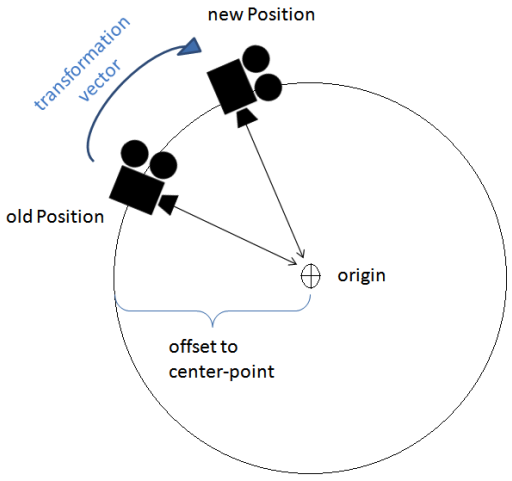

Figure 3.23: Origin-Rotation-Mode explanation

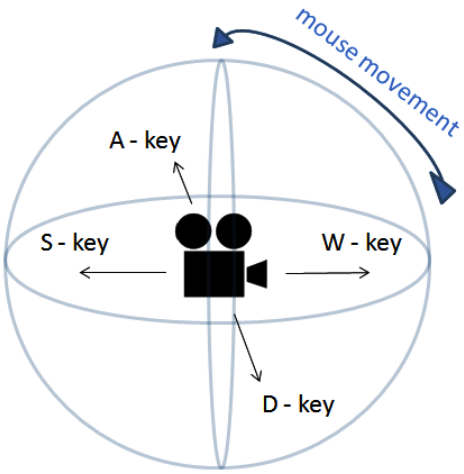

Figure 3.24: Fly-Through-Mode explanation

1. ***Fly-Mode:***

   If the "Shift" button is pressed the viewing direction can be changed by moving the mouse. The camera transformation is then calculated according to the difference of the old and new mouse position. Therefore we subtract both, the old X- and Y-axis from the new coordinate position. To move the camera its position can be changed by using the W, A, S and D keys. With the resulting vector we transform the camera to obtain the new camera position and viewing direction.

2. ***Origin-Rotation-Mode:***

   Using this mode the camera moves circular around its center-point. After pressing the "Shift" key the distance to the center-point can be changed by using the mouse-wheel. Additionally this mode consists out of two cameras to provide stereoscopic rendering. To render these images we implemented off-axis-projection by using different projection matrices for the left-eye camera as well as for the right eye camera. Both images are then displayed side-by-side. With the "G" (=GUI) keyboard button, all GUI-buttons are set invisible so that they do not disturb the 3D effect.

3.3.2 Off-axis rendering

For our implementation of stereo rendering we chose the off-axis technique. Therefore we had to calculate a new projection matrix that computes a new projection plane. This plane is defined by 5 green spheres which can be seen in Figure 3.25. The four green outer spheres define the border and the middle sphere the center of the projection plane. This technique is most commonly used and additionally ensures that the user will not feel uncomfortable even after longer experience.

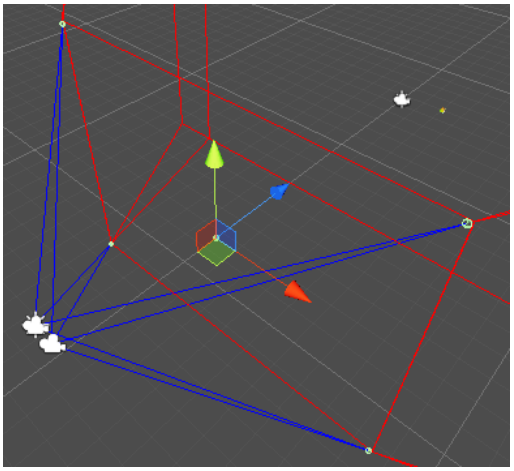

Figure 3.25: Off-axis rendering

An overview of different approaches can be seen in Figure 3.26.

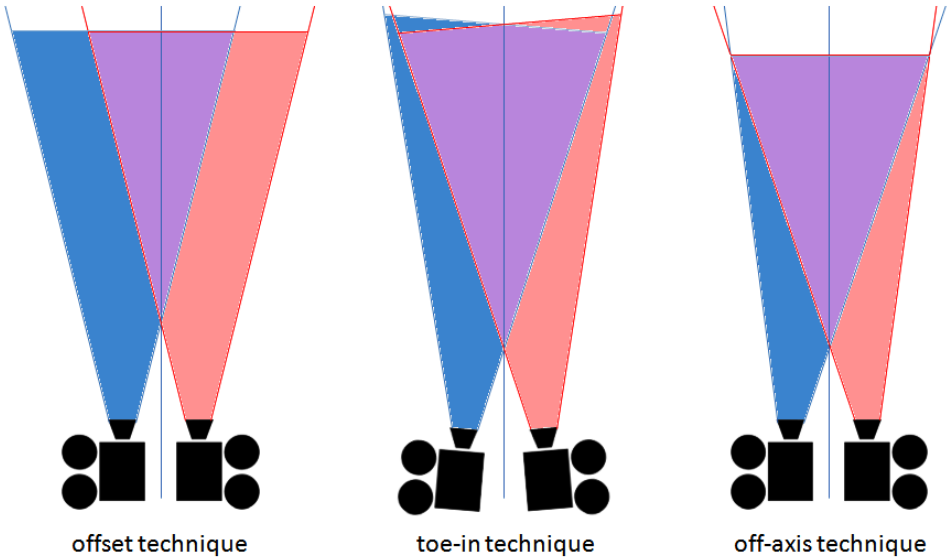

Figure 3.26: Stereo techniques

The offset technique is very simple. In this case the same projection matrix can be used for both cameras. The only thing that needs to be considered is the eye-distance which is usually about 6,3 cm. Irritating for the user in this technique is that parts of the image are only visible in either the left or the right eye. This might make the user feel uncomfortable depending on the time of use.

The toe-in technique is very similar to the offset technique. The only thing that needs to be changed is an additional rotation towards the center of the projection plane. Although this approach seems to be slightly better the projection planes in this approach are still not equal and therefore the result will make the user feel uncomfortable as well.
The last technique which is called off-axis rendering is the best approach although it requires two different projection matrices.

3.3.3 3D Orientation Indicator

We add a 3D Orientation Indicator to the scene to simplify the navigation for the user. With this approach it is always immediately clear from which direction the excavation is observed.

To avoid irritations like flickering of the cube while it intersects with another surface, it is rendered in a separate layer, which can be seen in Figure 3.27 (image 1 and 2). Subsequently these two separate images are combined. With this approach the cube is always visible even though geometry was already in front of it. This is not possible by using a single camera with a cube attached to it and only considering the Z-buffer. The result of final composition can be seen in Figure 3.27 image 4.

We do not consider any light source except ambient light in the separate cube layer. This is important because otherwise disturbing light artifacts could be seen after adding further light-sources. Finally we always render the cube in surface render-mode even if the rest of the scene is rendered in wire-frame mode which can be seen in Figure 3.27 image 3. Otherwise the texture of the cube would not be readable and the cube would get useless.

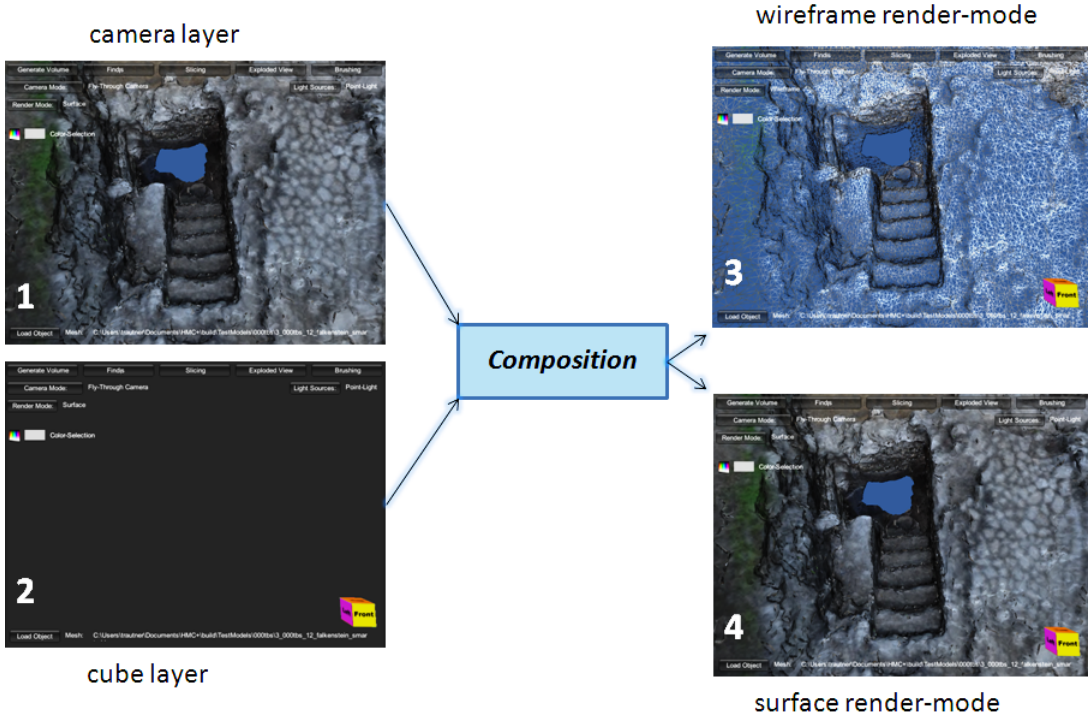

Figure 3.27: Composition of rendered images

3.3.4 Render modes

In the beginning of our implementation we were using Unity 3.5.7 and therefore it was not possible to render a scene in wire-frame mode without manually implementing a line renderer. The main disadvantage of this implementation was the performance. With this approach we had to iterate through all vertices per mesh and manually draw lines connecting vertices with triangles.

Since Unity 4 this is possible. Therefore we implemented functions like OnPreRender() and OnPostRender() and either set GL.wireframe to true or false depending on the current camera layer. The user can change this setting in our GUI by selecting either the "surface" or "wireframe" render mode.

An example of this can be seen in Figure 3.28. In this image we additionally selected a layer to visualize the function of the outline shader while the wireframe mode was enabled.

3.3.5 Color selection

Furthermore we added a color selection to our renderer so that the user can select any combination of RGBA colors. RGBA colors are chosen by either simply clicking on a color or by inserting values from 0 to 255 for the red, green, blue and alpha channel. An example of this can be seen in Figure 3.29.

These colors can then be used for either brushing or illuminating the scene. The library we are using to add this feature was developed by Sergey Taraban [3] and can be found in the Unity Community Wiki [2].

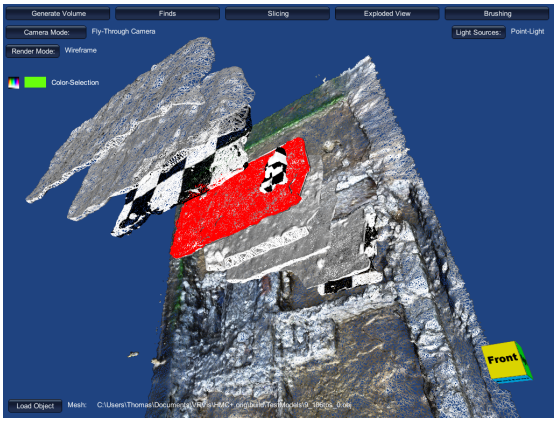

Figure 3.28: Wireframe rendering with mesh selection

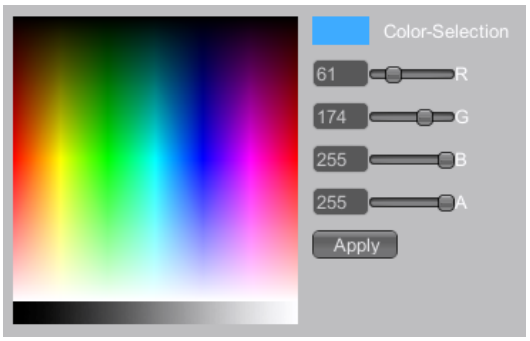

Figure 3.29: Color Picker

CHAPTER 4

Results

The most important performance criteria for us was the accessibility in real-time to allow the user to interactively explore our visualization. Therefore we focused mainly on GPU- and CPU consumption and the average number of frames per second.

The following hardware settings were used on the testing computer:

Operating system:	64-bit Windows 7 Home
RAM:	16 GB
CPU:	Intel(R) Core(TM) i7 -3930K CPU @ 3.20Ghz
Graphic card:	NVIDIA GeForce GTX 690 with 4096 MB memory

The scene we used for testing consists of twelve different stratigraphic layers. Every layer is a triangulated and textured high-resolution point-cloud (*.obj). The biggest layer we used is the 000tbs ground layer which consists of 314.347 vertices and has an overall size of 20,17 MB.

Furthermore we used a 1024x1024 picture (*.png) taken during the excavation to texture the ground layer and a 128x128 checkerboard pattern (*.png) to texture all other sub-meshes.

An overview of all object and texture sizes can be seen in Table 4.1:

Number of layers	1	2	3	4	5	6	7	8	9	10	11	12
Added layer	000tbs	006tbs	020tbs	032tbs	035tbs	054tbs	074tbs	096tbs	100tbs	106tbs	109tbs	115tbs
Vertices in layer	314347	6070	10931	10543	7734	10589	10393	8793	3067	10403	5731	3465
Vertices in scene	314347	320417	331348	341891	349625	360214	370607	379400	382467	392870	398601	402066
Size of *.obj file(s) (kB)	20.170	329	618	800	706	616	612	519	162	593	395	311
Texturesize of layer (kB)	2.370	4,74	4,74	4,74	4,74	4,74	4,74	4,74	4,74	4,74	4,74	4,74

Table 4.1: Overview of the example scene "Falkenstein"

4.1 Benchmarks

Usually an average frame rate of at least 35 frames per second is needed to ensure interactivity in real-time. The following Table 4.2 shows a minimum average frame rate of 40 FPS. Thereby we can guarantee that our viewer is able to deal with large data sets with a high vertex count.

Number of layers	0	1	2	3	4	5	6	7	8	9	10	11	12
Added layer	-	000tbs	006tbs	020tbs	032tbs	035tbs	054tbs	074tbs	096tbs	100tbs	106tbs	109tbs	115tbs
lowest FPS	140	133	120	103	88	80	73	69	43	35	31	22	19
highest FPS	173	168	165	152	149	148	132	113	109	91	80	75	60
average FPS	156	148	142	135	134	125	115	95	73	48	45	41	40

Table 4.2: Overview of the lowest, highest and average frame rate

4.2 Project demonstration

The renderer was already presented to the potential end-user group of archaeologist of the "Ludwig Boltzmann Institut für Archäologische Prospektion und Virtuelle Archäologie" [1] which is led by PD ao. Univ.-Prof. Mag. Dr. Wolfgang Neubauer. They found the 3D viewer to be especially helpful and easy to use, on one hand to watch cross-sections and explosion views to analyze structures and their relations and on the other hand the stereo rendering for public presentations such as press conferences.

CHAPTER 5

Conclusion

Although we built a stand-alone renderer and could therefore not totally benefit of a fully developed game engine like in our case Unity3D, we were able to achieve excellent results. We presented the development of a powerful tool, which is designed to simplify the future work of archaeologists. With this solution it is possible to not only analyze a graph-representation of an archaeological excavation, but visually represent all measured data at run time. This provides a virtual exploration for archaeologists who were not able to physically visit the excavation site. In addition, it allows archaeologists to research different stratigraphic layers at the same time. This is usually difficult as younger layers, which are mostly located above older layers must be irretrievably removed to reach lower surfaces. With our approach no information is lost and even younger surface layers can be studied easily.

5.1 Future Work

The aim of our future work will be to develop an interface between this viewer and the Harris Matrix Composer - Plus. It will be possible to not only investigate the three-dimensional data-set but also determine its position in the Harris Matrix and vice versa. As result we want to create a single even more powerful tool out of two already developed components. Additionally we want to improve our current volume generation and replace the current octree-like representation with a real volumetric computation. Finally we plan to implement selection based manipulations like exploded views which only affect certain areas or rooms of an excavation.

CHAPTER 6

Acknowledgement

We want to thank "Ludwig Boltzmann Institut für Archäologische Prospektion und Virtuelle Archäologie" [1] which generously provided us with actual data from their excavation in "Falkenstein".

At this point I also want to thank my supervisors that always had helpful suggestions and useful tips. Through this work I got an excellent insight in real-world applications in the field of computer graphics. Thank you!

CHAPTER 7

References

[**1**] Ludwig Boltzmann Institut für Archäologische Prospektion und Virtuelle Archäologie
June, 25th, 2014 - 3:15 pm
http://archpro.lbg.ac.at/

[**2**] Unity3D - unify community Wiki
June, 25th, 2014 - 3:45 pm
http://wiki.unity3d.com/index.php/Main_Page

[**3**] GUI Element - Color Picker by Sergey Taraban
May, 23th, 2014 - 10:35 am
http://staraban.com/en/simple-color-picker-control-for-unity/

[**4**] GUI Element - File Browser by Daniel Brauer
May, 26th, 2014 - 2:20 pm
http://wiki.unity3d.com/index.php?title=ImprovedFileBrowser

[**5**] Harris Matrix Composer
June, 25th, 2014 - 3:40 pm
http://www.harrismatrixcomposer.com/

[**6**] Off-axis-projection with Unity3D
June, 16th, 2014 - 9:45 am
http://forum.unity3d.com/threads/192409-Off-axis-projection-with-Unity

[**7**] Texture Scale in real-time
June, 10th, 2014 - 5:00 pm
http://wiki.unity3d.com/index.php/TextureScale

[**8**] Additional Debug Console
June, 2nd, 2014 - 6:45 pm
http://wiki.unity3d.com/index.php?title=DebugConsole

[**9**] Object Loader by Jon Martin
May, 30th, 2014 - 3:45 pm
http://www.jon-martin.com
http://www.fusedworks.com

[**10**] Unity3D Game Engine
July, 31st, 2014 - 10:30 am
http://unity3d.com//

Bibliography

[AFT+04] P. Allen, S. Feiner, A. Troccoli, H. Benko, E. Ishak, and B. Smith. Seeing into the past: Creating a 3d modeling pipeline for archaeological visualization. *Proceedings of the 2nd International Symposium on 3D Data Processing*, 2004.

[BdCTV03] Juan A. Barcelo, Oscar de Castro, David Travet, and Oriol Vicente. A 3d model of an archaeological excavation. *The Digital Heritage of Archaeology*, Computer Applications and Quantitative methods in Archaeology, 2003.

[BIF03] Hrvoje Benko, Edward W. Ishak, and Steven Feiner. Collaborative mixed reality visualization of an archaeological excavation. *Workshop on Collaborative Virtual Reality and Visualization (CVRV)*, 2003.

[CIG+01] John Cosmas, Take Itegaki, Damain Green, Edward Grabczewski, Fred Weimer, Luc Van Gool, Alexy Zalesny, Desi Vanrintel, Franz Leberl, Markus Grabner, Konrad Schindler, Konrad Karner, Michael Gervautz, Stefan Hynst, Marc Waelkens, Marc Pollefeys, Roland DeGeest, Robert Sablatnig, and Martin Kampel. 3d murale: A multimedia system for archaeology. *Proceedings of the 2001 conference on Virtual Reality, archaeology, and cultural heritage*, pages 297–306, 2001.

[DDC+12] Matteo Dellepiane, Nicolo DellUnto, Marco Callieri, Stefan Lindgren, and Roberto Scopigno. Archeological excavation monitoring using dense stereo matching techniques. *Journal of Cultural Heritage*, 2012.

[DVF+11] M. Doneus, G. Verhoeven, M. Fera, Ch. Briese, M. Kucera, and W. Neubauer. From deposit to point cloud - a study of low-cost computer vision approaches for the straightforward documentation of archaeological excavation. *Geoinformatics CTU FCE 2011*, 2011.

[Gra08] Herbert Grasberger. Introduction to stereo rendering. *Student Project, Institute of Computer Graphics and Algorithms - Vienna University of Technology*, 2008.

[Pat09] Daniel Patel. Expressive visualization and rapid interpretation of seismic volumes. *Thesis for the degree of Philosophiae Doctor (PhD) at the University of Bergen - Norway*, 2009.

[RC08] Fabio Remondino and Stefano Campana. Fast and detailed digital documentation of archaeological excavations and heritage artifacts. *Proceedings of the 35th International Conference on Computer Applications and Quantitative Methods in Archaeology (CAA)*, 2008.

[SCBP05] Luis Paulo Santos, Vitor Coelho, Paulo Bernardes, and Alberto Proenca. High fidelity walkthroughs in archaeology sites. *6th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST*, 2005.

[Vot01] Eileen Louise Vote. A new methodology for archaeological analysis - using visualization and interaction to explore spatial links in excavation data. *Thesis for the degree of Philosophiae Doctor (PhD) at Brown Computer Science - Rhode Island*, 2001.