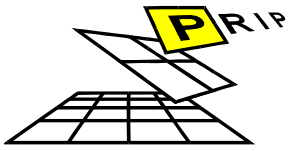
PRIP-TR-108

High Performance Profile Line Generation and Visualization using OpenGL

Julian Stoettinger

Pattern
Recognition &
Image
Processing
Group



Institute of
Computer Aided Automation

Technical Report

Pattern Recognition and Image Processing Group
Institute of Computer Aided Automation
Vienna University of Technology
Favoritenstr. 9/1832
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-18351
Fax: +43 (1) 58801-18392
E-mail: stoettij@prip.tuwien.ac.at
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-108 September 6, 2006

High Performance Profile Line Generation and Visualization using OpenGL

Julian Stoettinger

Abstract

On archaeological sites, thousands of fragments of ceramics are found. The classification of these findings is very time consuming and still done manually until today. To speed up this process, a 3d scanner for the data acquisition and software for the profile line generation is used.
This work provides an end-user friendly user interface for profile line generation of archaeological findings. For performance reasons, the visualization is done by the GPU using OpenGL. 3d scan data with several hundreds of thousands of vertices can be processed in real time.
The profile line generation is implemented in three levels: First, profile line generation is simulated by using OpenGL clipping planes and thus, the GPU only. Then, the nearest vertices of the intersection plane are marked and the intersection vertices are estimated. Finally, the profile segments of this diminished 3d scan data are decided.
Comparisons to a Matlab implementation and a commercial software have been carried out and these experiments with synthetic and real data are shown. Finally, an outlook towards future development is given.

Contents

1 Introduction

Motivated by the procedures and methologies of modern archaeology, this work provides an end-user application for estimating, choosing, generating and storing the profile of given scan data interactively. The application is part of the development of an fully automated, digital and portable system for acquisition and processing 3d scan data for archaeological documentation. It is motivated by the results of [10] and focuses on high performance, reusability, and the use of open source software only.
Due to the high hardware costs for 3d data acquisition devices (see Section 2.1), using free software is a way to decrease the costs of automatic systems for archaeological documentation. The following section shows the motivation for the developing of new systems for archaeological documentation and the implementation of the application *ArchiCut* (see Figure 1).

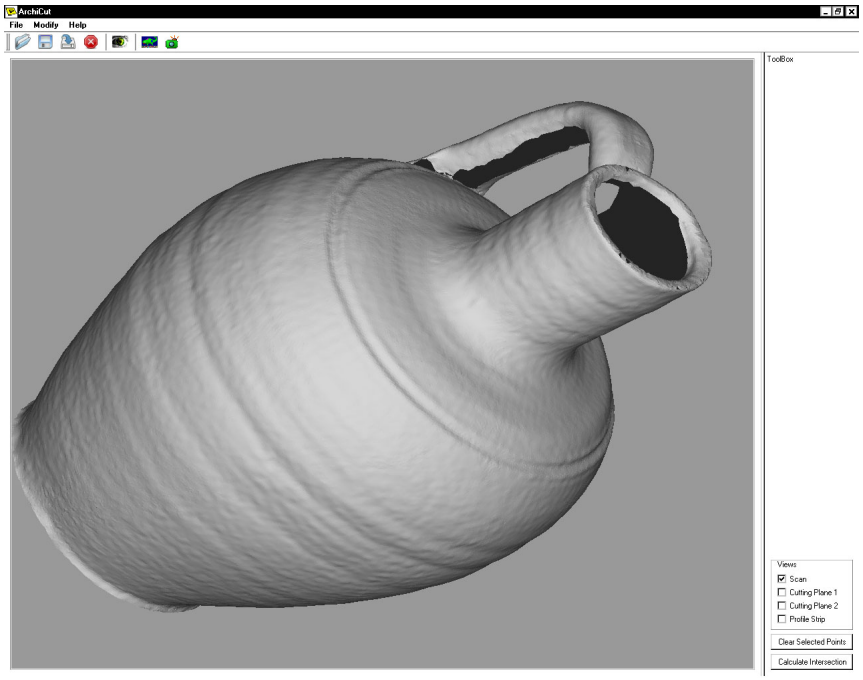


Figure 1: ArchiCut with loaded 3d scan data.

1.1 Archaeological Documentation

Beginning in the 19th century, the physical characteristics of archaeological pottery have been used to assess cultural groups, population movements, inter-regional contacts, production contexts, and technical or functional constraints [8]. In todays archaeology, there is still a lack of methology in analytical classification tools of artifacts [13]. Although, at every excavation site, thousands of ceramic fragments, called sherds, are found (see Figure 2).

Figure 2: Archive of ceramics stored in boxes. From: [9]

Traditionally, archaeological classification is based on the so-called profile of the object. It is the crossection of the fragment in the direction of the rotational axis of symmetry [9]. The resulting two dimensional plot holds all the information needed for the succeeding archaeological researches. The correct profile and the correct axis of rotation are thus essential to reconstruct and classify archaeological ceramics [10]. This is done by experts manually using different tools like a *profile comb*. This manual method is error prone and time consuming [11].
There are already two computer systems for sherd analysis: ARCOS (ARchaeological COmputer System) [4] and SAMOS (Statistical Analysis of Mathematical Object Structres) [18]. These systems are half-automated as explained in [15] and a resolution of approximately 2 mm. This is not sufficient for archaeological publications. As modern computers and data acquisition devices can process large amount of data and are able to scan data at a resolution at less then 0.5 mm, we are motivated to develop a complete, fully automated system at todays state of the art.

1.2 Implementation of ArchiCut

The application is part of the development of a fully automated, digital and portable system for acquisition and processing for archaeological documentation. The existing systems of the PRIP institute are implemented using Matlab[1]. This high level development kit is standard in scientific prototyping and allows rapid implementation of mathematic applications.
As a high level programming language, Matlab has performance issues for real-time applications. For example, a Matlab implementation is far beyond todays 3d visualization capabilities and allows only to display several ten thousand vertices, which is less than a typical 3d scan provides: As it is shown in Section 4.1, a single cleaned 3d scan contains about 50000 vertices. Using an automatic turntable as shown in Section 2.1, the object

[1]Matlab - High-level language and interactive environment - http://www.mathworks.de

is scanned from several view points and thus, the data is multiplied. Scanning a vessel with a turntable step of 60 degrees, then ends in registered 3d scan data containing about 250000 vertices.
On the test computer (as defined in Section 4.1), it is not possible to visualize more than 25000 vertices efficiently. Performance examples are given in Section 4.1.
As an commercial software, Matlab applications are only usable with a valid Matlab license, and it is not possible to include a convenient 3d scanner interface into the application.
Motivated by these drawbacks, the implementation of an rapid open source archaeological classification system has been done.
This work provides the first step of this project, the application *ArchiCut* (see Figure 1): It provides the possibility to load and visualize large 3d scan data, choose an intersection plane interactively and visualize and store the resulting profile in real time.
It focuses on high performance, reusability and the use of open source code only. For high performance, OpenGL[2] is used to take advantage of todays GPU power. An efficient 3d data set based on [1] is provided to load 3d scan data, calculate profile information and to be open for upcoming advancement (for example, texture information is already included). To achieve reusability as defined in [12], all the code is highly object oriented and, as a matter of fact, most of the time already reused. Using open source code only allows to have all possibilities for expanding the functional range of ArchiCut. All the software used is shown in Section 2.
the data structure and the developed algorithms are shown in Section 3.
Experiments an results of synthetic and real data are given in Section 4, and an outlook is given in Section 5.

[2]OpenGL - Open Graphics Library - http://www.opengl.org/

2 Theory, Methods and Fundamentals

This Section shows the theoretical fundamentals of the practical work. Starting with the data acquisition and showing properties of 3d scan data, the .OBJ file format is shown. This format is provided by the 3d scanner, and the application stores the profile data in this format. A short overview over the definition of the profile of an object is shown. Finally, a introduction to the main software tools used for this work is given: An overview over the software projects Qt and OpenGL.

2.1 Data Acquisition

The 3d scanner chosen for this application is the *Konica-Minolta Vi-9i*[3]. It is a non-contact laser triangulation scanner. The built in CCD sensor has a resoloution of 640 x 480 pixels, providing a preciseness of ±0.2mm in x,y-direction and ±0.1mm in z direction. The laser beam is splitted into a laser plane using a prism, the movement of the laser is achieved by a galvanometer-driven mirror [10].
For more than ten years, 3d scanners using structured light have been used [16]. As these devices are used for industrial and medical applications, the products became more practicable regarding size and weight and it is possible nowadays to join field trips with a 3d scanner [10]. As shown in figure 3, the laser beam is moved over or is projected onto

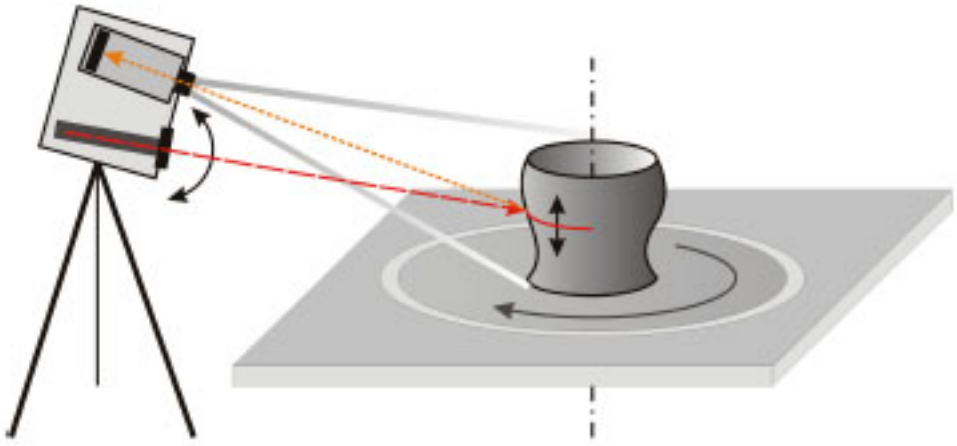


Figure 3: Scheme for 3D-acquisition based on a moving light plane using a laser and a prism. Left: 3D-scanner consisting of a camera (top) and laser enhanced by a prism (bottom) to a plane. On the right: The automatic turntable for automatic moving and registering of the 3d scan data with an acquired object. The rotational axis of the turntable is shown as dashed-dotted line. The movement of the laser plane (dashed line) and its intersection with the object is shown as double-arrows. The projection of the intersection to the cameras sensor is shown as dotted line. From: [10]

the surface of the given scan object. Triangulation is used to determine the depth of the object. The scanner has a built in auto-focus and a distance range from 0.6m to 2.0m.

[3]Konica Minolta - Non contact 3d scanner - http://kmpi.konicaminolta.us

Due to different lenses, the resolution and therefore, the preciseness depends on the focal length: Using a lens with the longest focal length of 25 mm, the resolution is at the maximum as mentioned before. Lenses with shorter focal lengthes are resulting in a smaller resolution.
Figure 4 shows a standard scan procedure. Since the scanner is only able to provide 3d scan data from its view point only, it is necessary to combine multiple scans from different views into one 3d model. This process is called the registration process.
The process is done manually or, in simple cases, done automatically using an automatic turntable: The turntable is connected to the computer and allows to scan a object from 360 degrees with viewpoint information for the registration (see Figure 4).

Figure 4: Scanning of a vessel. Left: 3d scanner focussing the object on the object. The object is on an automatic turntable connected to the computer on the right. The scanner and the turntable are controlled by the computer. The 3d scan data is registered automatically based on the information from the turntable. From: [3]

2.2 3d Data

In standard scan procedure, 3d scan data is directly sent to a computer to store the gathered information. Konica-Minolta uses for this transmission its own proprietary binary format. It is used in Minolta products, but useless for external applications.
To work with 3d scan data in other applications, the provided Minolta Software and the VIVID software development kit is able to convert the given data in several open formats. Thus, the 3d scanner is able to provide an open and simple data format for its 3d scan data, the same format, the application stores generated profile lines.
Wavefront .OBJ (object) files are used by Alias Wavefront's Advanced Visualizer application to store geometric objects composed of lines, polygons, and free-form curves and surfaces. Wavefront is best known for its high-end computer graphics tools, including modeling, animation, and image compositing tools.
The file format is used and supported in other application (e.g. Raindrop Geomagic Stu-

dio[4], Autodesk 3ds Max[5], Autodesk Maya[6], ...), for its simplicity and lack of restriction. The data can be easily handled and the format is supported in those applications.
The file format is based on signal characters, followed by 3d data. These signal characters are called *tokens* in this work. The tokens used in this practial work are listed in listing 1. .OBJ files support more tokens, which are poorly supported by applications. Examples are: *curv*, *curv2* and *surf* for defining bezier curves, 2d bezier curves and surfaces.

The structure of .OBJ files is straightforward and all elements which are used in the application are shown in listing 1:
declares the beginning of an comment line. These are always ignored. Usually the first line of every OBJ file will be a comment that says which program wrote the file out. Also, it is common for comments to contain the number of vertices and/or faces an object contains of.
v specifies a vertex by its three coordinates. The vertex is implicitly named by the order it is found in the file. For example, the first vertex in the file is referenced as '1', the second as '2' and so on. None of the vertex commands actually specify any geometry, they are just points in space.
Konica-Minolta added further optional color information to the vertices: 3 floats after the position information define the color of the vertex in RGB values.
vt is the vertex texture command. It is defined as having a mandatory U and V and a optional W coordinate. They must be grouped in a face command. Texture coordinates are implemented in the application, but not visualized, yet. The application reads the arguments, but ignores it.
f specifies a polygon from the vertices listed. Reference of a vert is just the index as explained at the vertex command.
l specifies a line from the following arguments. The number of vertices after the command is not specified. This command is used for saving the profile lines.
g groups all following faces to an object.[7]

```
# some text
g name
v float float float (float float float)
vt float float
f v1[/vt1] v2[/vt2] v3[/vt3] ...
l v1[/vt1] v2[/vt2] v3[/vt3] v4[/vt4]...
```

Listing 1: .OBJ tokens used in the ArchiCut

[4]Raindrop Geomagic Studio - http://www.geomagic.com/de/products/studio/
[5]Autodesk 3ds Max - http://usa.autodesk.com/
[6]Autodesk Maya - http://usa.autodesk.com
[7].OBJ file format - http://www.royriggs.com/obj.html

2.3 Profile Line

As shown in Figure 5, a profile line is defined by the crossection of the 3d scan data and an intersection plane e_i [9]. In the presented application, the rotational axis is chosen by

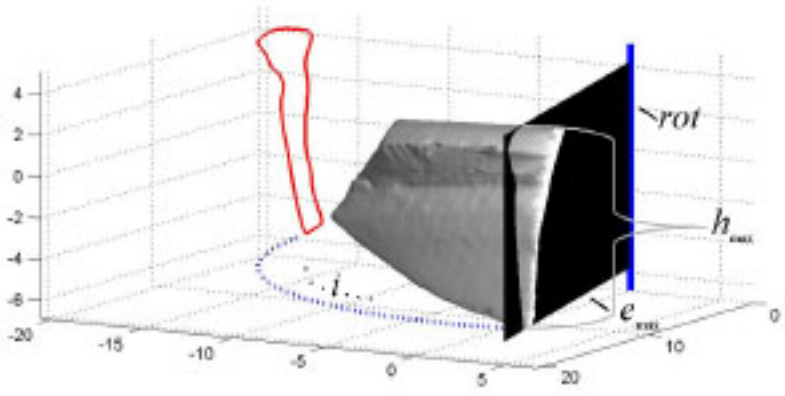


Figure 5: Sherd with orientation, rotational axis *rot*, intersection plane e_i, longest profile line $profile_{max}$, h_{max}. From: [9]

the user by selecting 3 vertices. Algorithms to calculate this automatically are added in further works. Several methods are discussed in [14].
As shown in Figure 6, if some parts of the objects are not visible from the scanner's point of view or the object covers parts of itself, *scan shadows*, and so, holes in the object occur(see Figure 6). To have one closed profile line as the whole profile of an object (see Figure 5)) is not granted: If their are holes in the object or scan shadows, one profile is divided in several *profile segments*. Profile segments, also called *strips* in this report, are line strips of adjacent intersection vertices and are part of the whole profile. Due to the properties of the 3d scan data and the processed data in the application, it is possible that *T crossing intersection vertices* occur.
T crossing vertices are intersection vertices with more than two adjacent vertices. These are introduced by scanner noise and special properties of the given object. With these vertices, furcations of the profile segments would occur. To avoid these, each T intersection vertex is treated as the beginning or end of the profile segments of all of its adjacent vertices.
Because of the structure of the 3d scan data, OpenGL triangle properties and scan errors like intersecting triangles, automatic profile line generation may result not in simple lines only. Therefore, lines with more than two endings must be separated: If one intersection vertex is part of more than one profile segment, it is the end point of each of these segments (at least three) and marked within the application for further investigations.
See Section 3.3 for detailed information about the profile generation shown in this work.

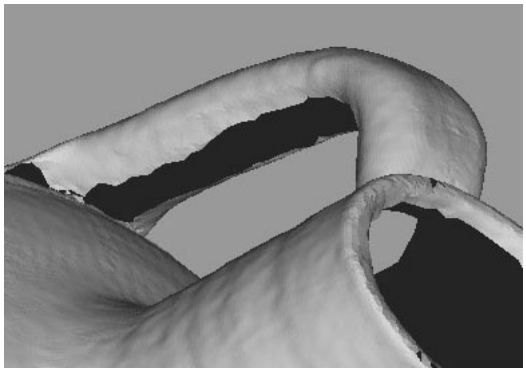


Figure 6: Scan shadows near the handle of a vessel. The back side of the surface is visualized as black faces.

2.4 Qt

Qt is a cross-platform graphical widget toolkit for the development of GUI programs. It became notably popular because of its use in the KDE[8] project. The name Qt [cu:te] came up in the 1991, when one of the developer found the letter "Q" so beautiful in his Emacs font. The "t"' is inspired by the X toolkit Xt. In 1998, KDE became one of the leading desktop environments for GNU/Linux, which is based on the Qt engine.
Until version 1.45, source code for Qt was released under the FreeQt license which was viewed as not compliant to the open source principle by the Open Source Initiative and Free Software Foundation because while the source was available it did not allow the redistribution of modified versions. With the release of version 2.0 of the toolkit, the license was changed to the Q Public License (QPL), a free software license but one regarded by the Free Software Foundation as incompatible with the GPL. Compromises were sought between KDE and Trolltech wherein Qt would not be able to fall under a more restrictive license than the QPL, even if Trolltech was bought out or went bankrupt. This led to the creation of the KDE Free Qt foundation, which guarantees that Qt would fall under a BSD license should no open source version of Qt be released during 12 months[6].

Qt's Design Principles

The first versions of Qt had only two flavours: Qt/X11 for Unix and Qt/Windows for the Windows platform. The Windows platform was only available under the commercial license. In the end of 2001, Trolltech released Qt 3.0 which added support for the Mac OS X platform. Until June 2003, Mac OS X support was available in the commercial license only, when Trolltech released the version 3.2 with Mac OS X support available under the GPL license. Now, for Windows there is a dual license model: One closed source commercial license for use in the Microsoft Visual Studio .NET[9], one open source version under GPL for use with MinGW[10]. In this work, we use the open source version, developing in the Visual Studio but compiling with MinGW. The major drawback in the open-source version is the lack of auto-completion of Qt methods inside the IDE.

Complete abstraction of the GUI

[8]KDE - K Desktop Environment - http://www.kde.org/
[9]Microsoft Visual Studio Development Center - http://msdn.microsoft.com/vstudio/
[10]MinGW - Minimalist GNU for Windows - http://www.mingw.org/

Unlike other portable graphical toolkits, Qt uses its own engine on each platform. For example, wxWidgets and the Java based SWT[11] use the toolkit of the target platform for their implementation. Instead, Qt uses its own paint engine and controls. It just emulates the look of the different platforms it runs on. This made the porting work easier because very few classes in Qt depended really on the target platform. The drawback is that Qt had to emulate precisely the look of the different platforms. This drawback however no longer applies because the latest versions of Qt use the native styles API of the different platforms to draw the Qt controls.

Meta Object Compiler

Known as the moc, this is a tool that one must run on the sources of a Qt program prior to compiling it. This changes the development sequences and make the integration of a Qt project in an arbitrary software development environment tricky. The tool will generate "Meta Information" about the classes used in the program. This meta information is used by Qt to provide programming features not available in C++: Introspection, signal/slot system. The use of an additional tool has been criticised by part of the C++ community, stating that Qt programming is making a "moc"ery of C++. In particular, the choice of an implementation based on macros has been criticized for its absence of type safety and pollution of the namespace. This is viewed by Trolltech as a necessary trade-off to provide introspection and dynamically generated slots or signals. Further, when Qt 1.x was released, consistency between compiler template implementations could not be relied upon.[12]

Qt's Performance

Under Windows, Qt's C++ rendering engine, the performance of a Qt application is comparable to a native application. In this work, the main focus lies on the performance of the GlWidget object to integrate openGl application into Qt. There is definitely a loss of speed and a higher usage of memory than using standard OpenGL only, but not noticeable with the requirements of the application[2].

2.4.1 Qt's Tools

Qt Designer

The Qt Designer is a graphical layout and form designer. It allows to draw all sorts of widgets and create signals and slots. It is possible to generate custom widgets. The Layout is saved in .ui files. It integrates itself into MS Visual Studio and KDevelop.

Qt Assistant

Interactive help application for Qt. Is highly necessary, because of the lack of auto completion in the open source version.

qmake

For a Qt project, the developer has one single project file. Qmake is able to build the standard project file for a new project out of the box. Qmake then builds header files out of the existing .ui files and then an according make file for the defined operation system and it takes care of all the compiler and platform dependencies.

[11] SWT - The Standard Widget Toolkit - http://www.eclipse.org/swt/

[12] Qt (toolkit) - Wikipedia, the free encyclopedia - http://en.wikipedia.org

2.4.2 Developing with Qt

Installing of open source Qt is straightforward and on Windows, no manual configuration is necessary. Running under Linux is not considered, because the Konica-Minolta scanner API runs under Windows only. One common pitfall is the fact, that the debug libraries are not built automatically. If one build a project in debug mode without compiling them, weird errors will occur. In the Windows menu, there is a batch script "*Build Debug Libraries*". The process will take some minutes and will likely end in an "*access denied*" error. This happens on many machines and has no consequences at all.
The main idea of developing platform independent and with meta objects is having a project file as the basis to generate a makefile. ArchiCut's project file is called *AC.pro*. It defines all included locations, template project basis, linking targets and so on. Additionally, all source files of the applications and their .ui files (QtDesigner) are listed.
If a new software project is started, it is possible to generate this file using "*qmake -project*". In this project file generation mode, all files with known source extensions are interpreted as files to be built and included in the project file.
If this mode is used while improving ArchiCut, the adapted project file will be overwritten. So its highly recommended to add additional files into the project file manually. This procedure is proposed in lots of developer comments throughout the world wide web.
After adding new source files or changing project settings in the project file (again: manually!), one has to generate its platform specific make file. Using "*qmake*", a standard GNU make file is built. Using "*nmake*" or "*make*", the application can be built. Theoretically, it is possible to built for another platform. This mode is not suggested, by developers and Trolltech Inc. itself, as well.
Although the open source version of Qt is not supported by Microsoft Visual .Net, ArchiCut was developed in this development environment. So, the designer is not integrated into the GUI, and worse, no auto completion of Qt functions and methods are available, and even worse, no .NET debug information is built. Building and running is possible with defining an external tool to run the Microsoft compiler in the project directory and using run targets in the project properties. These settings are already defined in the Microsoft Visual Studio files in the source directory.
Debugging is done with the qDebug() command, as it does not lowers the performance of the application when started without a console.

2.5 OpenGL

Since its introduction in 1992, OpenGL is a 3d API that competes with Direct3D. OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. OpenGL differs from Direct3D in that OpenGL is not maintained by a corporation (as Direct3D is maintained by Microsoft) and, as such, updates and revisions to its code tend to come slower. All 3d video cards made today support both OpenGL and Direct3D.

2.5.1 The State Machine

OpenGL's basic operation is to accept primitives such as points, lines and polygons, and convert them into pixels. This is done by a graphics pipeline known as the OpenGL state machine. Most OpenGL commands either issue primitives to the graphics pipeline, or configure how the pipeline processes these primitives. Prior to the introduction of OpenGL 2.0, each stage of the pipeline performed a fixed function and was configurable only within tight limits but in OpenGL 2.0 several stages are fully programmable using GLSL[17].

OpenGL is a low-level, procedural API, requiring the programmer to dictate the ex-

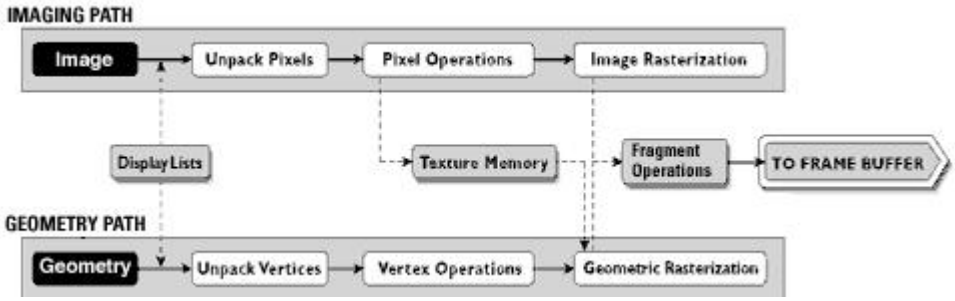


Figure 7: OpenGL operates on image data as well as geometric primitives. From: [13].

act steps required to render a scene. This contrasts with descriptive (aka scene graph or retained mode) APIs, where a programmer only needs to describe a scene and can let the library manage the details of rendering it. OpenGL's low-level design requires programmers to have a good knowledge of the graphics pipeline, but also gives a certain amount of freedom to implement novel rendering algorithms.[14] As visualised in Figure 7, the openGL architecture is able to handle both images and geometric primitves inside its state-machine. Commands may be accumulated in display lists or processed immediately, but are all processed through the pipeline. The openGL pipeline contains the following stages: The evaluator puts the incoming data into the corresponding vertex and attribute commands. Then, per-vertex operations including transformations, lighting, clipping, projection, and viewport mapping are processed. The rasterization stage produces fragments for the framebuffer, including bitmaps, pixels, and primitives.

[14]OpenGL - Wikipedia, the free encyclopedia - http://en.wikipedia.org/wiki/OpenGL

2.5.2 The Integration

OpenGL is a fully functional primitive-level API that allows the programmer to efficiently address and take advantage of graphics hardware. In this work, the main point of using OpenGL is the ability to take advantage of the GPU.
3d data can be loaded into the graphics memory and transformations and clippings of this data can be processed without using the CPU and the relatively slow AGP or PCIe interface.
Many high-level libraries and applications make use of OpenGL due to its performance, ease of programming, extensibility, and widespread support. It is well supported by Qt with several implemented "convenient" classes [2].

3 Data Processing

In the following Section, an overview of the implementation of the algorithms and software tools used for this practical work is given. The main functionality of the classes and structures are documented. For further and detailed information, refer to the inline documentation of the source code.
All the application's actions are triggered by user input.The OpenGL data is only drawn after user interaction, so no "game loop" at a fixed frame rate or something else common for OpenGL applications is used. This allows the application to gain full CPU time while calculating the algorithms shown in this Section. For a detailed description of the GlWidget methods, see the Qt examples and documentation[15] [2].

3.1 Data and Processing Layer

These objects are storing and processing the 3d data, and has been made as transparent from the user interface layer. The data structure is based on [1].

3.1.1 Class C3dModel

C3dModel is the main class and implemented in 3dModel.[h/cpp]. The main functionality of the implementation is given in listing 2. It is defined statically in an own header file (g_3dModel.h), and there exists only one instance of the class, called g_3dModel. The class holds all 3d objects and invokes all actions with the data, including the intersection calculation described in Section 3.3.

```
class C3dModel {
public:
                //Load model from file into model structure
        bool LoadModel(FILE *m_filePointer, int modelType);
                //Let the model draw itself
        void DrawModel(bool bRandCols, bool bSmoothShading);
                // generating all intersection points and
                // storing it in totalStrip
        int generateTotalStrip(double eqn[4], Cvec3 planePoint);
                //Save Profile as OBJ file
        void saveProfile(QString filename);
                //This holds the 3d model that we loaded in
        t3DModel theModel;
                //The profile generation and storing class
        CProfile *profile;
};
```

Listing 2: Collection of most important methods and member objects of class C3dModel.

[15] Qt - Online Reference Documentation - http://doc.trolltech.com/

This nomenclature has been chosen because all the visualized 3d data is inserted into the OpenGL viewing matrix in one step and so handled as one model to be viewed. It is able to invoke the loading of the data from a file and the storing of given data into an .OBJ file.

3.1.2 Struct t3DModel

An instance of the t3dModel is given in C3dModel named *theModel*. The main functonality of the structure is the vector of pObjects (*t3DModel::pObject*) (see listing 3). The size of the vectors can be gathered easily, but for performance reasons and readability of the code, they are stored in integers *t3dModel::numOfObjects*, *t3DModel::numOfMaterials*). They are not always consistent, as they do not update by themselves. The elements of the vector of pObjects is shown in Section 3.1.4.

```
struct t3DModel
{
                // The number of objects in the model
        int numOfObjects;
                // The number of materials for the model
        int numOfMaterials;
                // The list of material information
                // (Textures and colors)
        vector<tMaterialInfo> pMaterials;
                // The object list for our model
        vector<t3DObject> pObject;
};
```

Listing 3: The structure to store multiple objects.

3.1.3 Class CProfile

The class CProfile provides the profile data for the 3d model in the application and the profile line generation methods. A simplified definition is given in listing 4. Additionally, it can insert the 3d data into the OpenGL matrix as well, to be able to draw itself autonomically (*CProfile::drawItself()*).
The profile data is stored in the vector *CProfile::totalStrip* and referenced with additional adjacency information in *CProfile::theProfile*.
The generation of *CProfile::totalStrip* is invoked by
C3dModel::generateTotalStrip(intersection plane information). The profile generation is finished by running *CProfile::createAdjStrips()*. This method is divided in several steps as described in 3.3, where the algorithm is shown.

```
class CProfile  {
public:
                //draws itself
        int drawItself();
                // creating the profile
        int createAdjStrips();
                // all the data for the algorithm
        vector<stripElem*> totalStrip;
                // the profile
        vector<strip*> theProfile;
private:
                //calculate intersection of given parameters
        Cvec3 calculateIntersection(int object, int v1Id,
              int v2Id, double eqn[4], Cvec3 planePoint);
                //test structure of the adjacency
        int testAdj(stripElem *vert);
                // results of testAdj. name the indices of the
                // neighbours
        vector<stripElem*> tempVisited;
        vector<stripElem*> tempCand;
```

Listing 4: Simplified implementation of the class CProfile.

3.1.4 Struct t3DObject

The data of vertices and its attributes is stored in pointer arrays. A shortened structure is given in listing 5.
When processing 3d data, the application has to iterate over all data in each object. So this iteration over several instances of the *t3DOject::pVerts* is time critical and has to be as efficient as possible.
For performance and simplicity in the code, the size of the arrays is stored separately as integers (*t3DObject::numOfVerts*). Here, the information of the length of these arrays is very important and must be consistent for each step of the processing, because the length of the vector array cannot be gathered by the data structure itself.
Texture and material information is not used in this implementation, yet. Of Course, this task will be achieved in future work. Especially, because there are researches about automatic calibration and using external cameras and calculating texture information from this picture information (see Section 5).

```
struct t3DObject
{
                // The number of verts in the model
        int  numOfVerts;
                // The number of faces in the model
        int  numOfFaces;
                // The number of texture coordinates
        int  numTexVertex;
                // The texture ID to use,
                //which is the index into our texture array
        int  materialID;
                // This is TRUE if there is a texture map
                // for this object
        bool bHasTexture;
                // The name of the object
        char strName[255];
                // The object's vertices
        Cvertex *pVerts;
                // The texture's UV coordinates
        Cvec2 *pTexVerts;
                // The faces information of the object
        Ctriangle *pFaces;
};
```

Listing 5: Simplified structure to store model and scene information.

3.1.5 Class CTriangle

CTriangle holds references to the three vertices defining itself, the polygon normal and color information. See Figure 6.

```
class Ctriangle
{
public:
                // points down to the vertices of our vertex list
        int v1, v2, v3;
                // points to the triangle's vertex objects
        Cvertex *vert1, *vert2, *vert3;
                // polygon normal
        Cvec3 normal;
                //vertex colors
        Cvec3 col1, col2, col3;
};
```

Listing 6: Overview over the implementation of CTriangle.

3.1.6 Class Cvertex

Cvertex defines a 3d vertex with all its needed properties, including its position in euclidean space, its normal, its id, and its attributes for the profile generation algorithm.
As seen in listing 7, each vertex has cross references to its adjacent vertices, and its adjacent triangles, as well. The two boolean flags used in the profile generation give the attribute for the first iteration over the vertices: Between an "isProfile" and an "isTargetProfile" vertex must be a crossection with the intersecting plane.

```
class Cvertex {
public:
                // location of point in euclidean space
        Cvec3 pos;
                // the vertex normal, required for shading
        Cvec3 normal;
                // adjacent triangles
        vector<Ctriangle *>     adjFaces;
                // adjacent vertices
        vector<Cvertex *> adjVerts;
                // flag if its part of profile line
        bool isProfile;
                //flag if its part of the profile line
                //but on the other side of the plane
        bool isTargetProfile;
};
```

Listing 7: Overview over the implementation of CVertex.

3.2 User Interface Layer

The user interface elements, action and event handlers are all built in the Qt Designer, as shown in Section 2.4. The Designer and qmake generate header files (ui*) which can be defined directly in the software project. The following classes are all *multiple heritaged*[16] from these header files as described in [2]. To change the graphical user interface, adaption of a .ui file file, and then the execution of qmake to generate an updated header file is mandatory.

3.2.1 Class ArchiCutWindow

Invoked from the main function, the implementation ACwin starts the application. As a multiple heritaged object, it is derived from both the class QMainWindow and the Designer's Mainwindow header file. In this object, all methods are implemented. Signal handlers are called after an user interaction in the menue bar or the toolbox on the right. It calls the GlWidget methods, or - less abtract - directly methods of the 3d model.

[16] Qt - Online Reference Documentation - http://doc.trolltech.com/

3.2.2 Class GlWidget

Defining the OpenGl Area as the main part of ArchiCut, it handles all user interaction of the mouse and keyboard input inside the 3d space (see listing 8). Derived from QGlWidget, it initializes the graphics engine, handles requests both from the data processing layer and the main window and realizes all the visualization procedures.
For a detailed documentation of the OpenGL implementation of QGlWidget using initializeGL(), paintGL(), and resizeGL(), refer to the QT documentation[17]. The events are called after a user interaction. After three vertices are selected, the profile generation is invoked.
For further developing: Handles are invoked from ArchiCutWindow, invoking a method in glWidget, invoking a method in g3dModel (abstract communication). Directly communicating from ArchiCutWindow to g3dModel is not recommended, as it provides wrong data.

```
class GLWidget : public QGLWidget {
  Q_OBJECT
public:
protected:
                //initialize graphics engine
        void initializeGL(void);
                // open file handler
  bool readInFile();
                //OpenGl methods for application loop
  void paintGL(void);
  void resizeGL(int width, int height);
                //events
  void mousePressEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
  void wheelEvent(QWheelEvent *event);
private:
                //Point Selection Methods
                //rays for mouse selction, Ps for storing
        Cvec3 rayP1, rayP2, vClickSlope, vSelSlope,
                                selP1, selP2, selP3;
                // invokes the profile generation
        int clipPlane(Cvec3 p1, Cvec3 p2, Cvec3 p3);
};
```

Listing 8: Overview over the Class GlWidget.

[17] Qt - Online Reference Documentation - http://doc.trolltech.com/

3.3 Profile Generation

The profile generation algorithm is shown and a pseudo code listing is given. The principle of the algorithm is that each vertex decides on its own where it belongs to and then inserts itself into the data structure on its own. After the calculation of the new vertices, no actual data is copied for the calculation, the intersection data is a reference. As a result, a vector of vectors is calculated, where the vertices are referenced in ascending order. Because of this, they can be drawn and analyzed linearly.
The algorithm works in three steps: First, all intersection points are estimated and stored in the vector totalStrip due to the method *CModel::generateTotalStrip(plane equitation, vertex on plane)*. It is invoked by CModel, because it is based on the whole 3d scan data. With the method *CProfile::createAdjStrips()*, all the properties of the vertices are estimated and then filled into the final data structure (see Figure 8). The optional method *CProfile::create2DProfile()* transforms the profile onto the x/y axis.

3.3.1 Intersection Calculation

For all vertices, their dot product with the intersection plane normal vector is calculated. If it is positive, the dot products of its adjacent vertices are compared. If we have two adjacent vertices, with one positive dot product and the other one with a negative, we mark one with the flag *isProfile* and the other one with the flag *isTargetProfile*, as it is clear that there is an intersection between them.
To calculate the intersection, the standard intersection of a plane and a line algorithm is used: Having two adjacent vertices v1 and v2 on both sides of the plane, the intersection vertex v is given by

$$v = v1 + s(v2 - v1)$$

and

$$s = \frac{n \cdot (pv - v1)}{n \cdot (v2 - 1)}$$

when n is the normal vector of the plane and pv on vertex on the plane itself.

3.3.2 Properties of Vertices

The data for the adjacency calculations is processed linearly through the array. Thus, if we compare with adjacent neighbors, we have always visited (already inserted) and unvisited ones. The adjacency of two vertices is decided in method *CProfile::isAdj(v1,v2)*: If they have the same neighbors on one side of the plane, and the neighbors on the other side are adjacent, we have adjacent intersection vertices.
The algorithm is shown in listing 9. For each of the 7 types of vertices, different insertion actions take place. These actions are basically to start a new strip, lengthen another strip, connecting two strips or finsihing one strip. Note that the insertion algorithm looks in unfinished strips at the end and the beginning only. This is a performance advancement regarding the alternative that each vertex is compared to all others. And it makes the correct and efficient processing of T intersection vertices possible as shown in the next Section.

```
for each vertex {
        switch (testAdj(vertex)) {
                type 0: //vertex without neighbors
                  fill it in an own strip alone;
                  mark as end, begin and finished;
                type 1: //vertex with 1 unvisited neighbor
                  start new strip;
                  mark as begin;
                type 2: //vertex with 2 unvisited neighbors
                  start new strip;
                type 3: //vertex with 2 visited neighbors
                  fill it into two existing, unfinished
                  strips where its neighbors are;
                  connect these two strips;
                  if they are the same,
                  mark as finished and loop;
                type 4: //vertex with 1 visited neighbor
                  fill it in an existing strip;
                  mark it as end or beginning;
                type 5: //vertex with 1 v. and 1 unv. neighbor
                  fill it in an existing strip;
                type 6: //T crossing, more than 2 neighbors
                  fill it in existing strips;
                  mark it as end or beginning
                  if no visited neighbors, just start new strip;
                  mark as begin;
        }
}
```

Listing 9: Pseudo code for profile generation algorithm.

3.3.3 T Intersection Vertices

As explained in 2.3, a T intersection vertex is a vertex with more than two neighbors. This happens in 3d data with intersecting triangles and other geometry errors as well as in well meshed 3d data. If the algorithm encounters such a vertex, the profile strip must be split. For each neighbor, the T crossing vertex is the begin or the end of a own split. This algorithm creates a new strip on its own, if a neighbor of a T crossing vertex is processed:
Regarding vertex 3 to 6, it can happen that the visited neighbor is a T crossing and therefore already in a closed strip. According to the procedure shown before, the vertex would not be inserted.
Therefore, each visited neighbor in a closed strip gets inserted at the beginning of a new strip another time, and the processed vertex is inserted as its new neighbor. This is the reason a T crossing with more unvisited neighbors gets inserted only one time and marked

as beginning or end: Whenever a adjacent vertex comes up, it inserts the T crossing by itself another time. The result of the algorithm for the cipa pot as shown in 4.1 is shown in Figure 8.

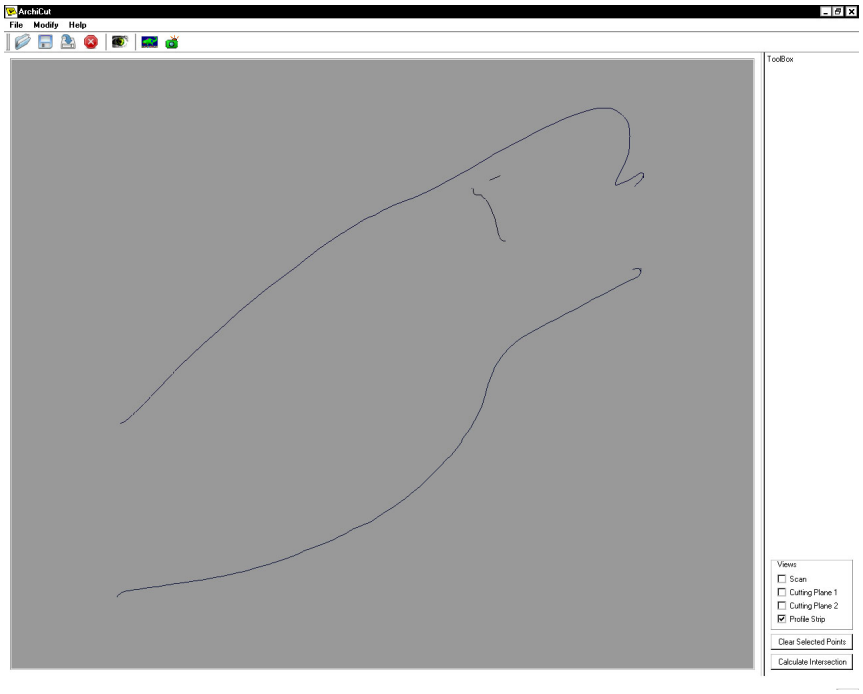


Figure 8: ArchiCut draws the 3d profile.

4 Experiments and Results

Experiments were done with synthetic data as well as with real antique ceramics 3d scan data. In this Section, the approach of the experiments are presented and some results and features of the piece of software are given.

4.1 Performance of ArchiCut

The main goal of the implementation of a real time 3d scan data application was the ability to handle large data sets fast and efficiently. In the following, the performance of loading and visualizing 3d scan data is evaluated. Test computer is a common desktop machine: Athlon XP 2400+, 2 GB RAM, 2 SATA hard discs (splitting data and application) and a Radeon 9500pro graphics cards with 128MB video memory on a 8xAGP interface.
As listed in Table 1 and 2, different kind of data are used for the evaluation: Two synthetic data set for the evaluation of regular meshed geometry. The unit sphere does contain 266 vertices and is so a very small set of data, whereas the havard bunny with its 34834 vertices is comparable with a small 3d scan.
The next object, the cipa pot, is a edited and smoothed object, having a comparable numbers of vertices like vessel nr. 3754 but due to the post processing a different geometry. It is a well known, modern vessel, but traditionally manufactured. For these reasons, it is called a *semi-synthetic* object, because it is scanned from a real life object, but edited to get a regular geometry.
A single scan of the vessel nr. 1054 of the Kunsthistorisches Museum Wien (KHM), with a number of vertices of 53388, which is a reasonable number for a average cleaned scan. Then, the 360 degree scan of this vessel is used: 4 scans taken from 90 degree differences of viewport is the candidate for 168795 vertices of 3d scan data, and so, a example for a relatively small 3d scan. The scan of vessel nr. 3754 with a 60 degree scan has 252325 vertices.
Reference applications are the Matlab implementation from [10], the Geomagic Studio 8 (GM Studio 8). As expected, ArchiCut showed better results in both the memory usage and the performance time compared to the Matlab implementation. As a rapid prototyping language, it has more overhead than a C++ implementation.
ArchiCut loaded given 3d scan data than the commercial Geomagic Studio 8, especially for small data and geometry with few adjacencies (see Table 1), where ArchiCut is more than twice as fast as the Geomagic Studio 8.
Notably, for the Matlab implementation and for ArchiCut is the geometry of the given 3d scan data a performance issue: For both applications, it took more time to load the edited geometry, whereas the Geomagic Studio 8 showed a constant performance. Supposed this is a matter adjacency density, Geomagic has probably developed a more scaleable algorithm than the one used in ArchiCut.
In Figures 9 to 26, the visual results of the evaluation are shown.

Data	File name	Nr. of verts	Matlab	Gm Studio 8	ArchiCut
synthetic	unitsphere.obj	266	0.7	0.0	0.0
synthetic	bunny.obj	34834	51.5	1.6	1,1
semi-synth	cipa-pot-02.obj	252551	2122	14	9.1
real	1054-3.obj	53388	926	2.03	0.7
real	1054.obj	168795	109	8.3	4.9
real	3754.obj	252325	197	14	7.2

Table 1: Loading and visualization time of given 3d data in seconds sorted by kind of data and number of vertices.

Data	File name	Nr. of Verts	Matlab	Gm Studio 8	ArchiCut
synthetic	unitsphere.obj	266	88	33	33
synthetic	bunny.obj	34834	95	56	47
semi-synth	cipa-pot-02.obj	252551	304	118	115
real	1054-3.obj	53388	734	81	40
real	1054.obj	168795	185	85	65
real	3754.obj	252325	196	140	122

Table 2: Memory use for loading and visualizing 3d scan data in rounding MB sorted by kind of data and number of vertices.

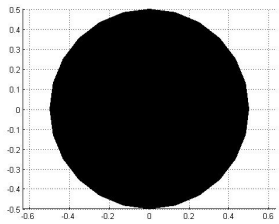


Figure 9: Matlab plot of unit sphere.

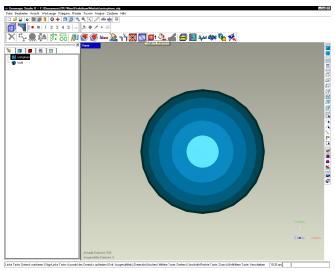


Figure 10: Geomagic Studio 8 loaded unit sphere.

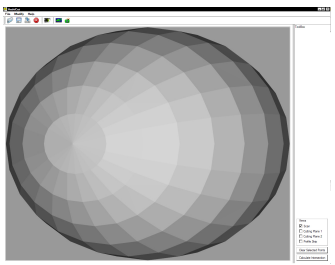


Figure 11: ArchiCut loaded unit sphere.

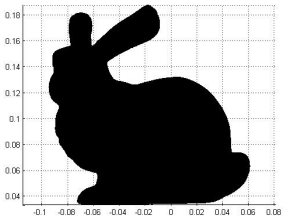


Figure 12: Matlab plot of harvard bunny.

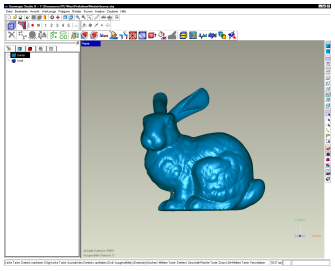


Figure 13: Geomagic Studio 8 loaded harvard bunny.

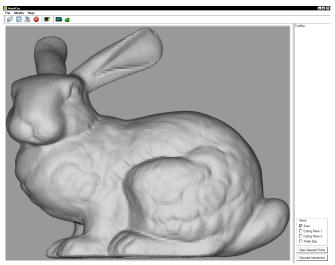


Figure 14: ArchiCut loaded harvard bunny.

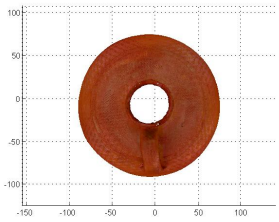


Figure 15: Matlab plot of cipa pot.

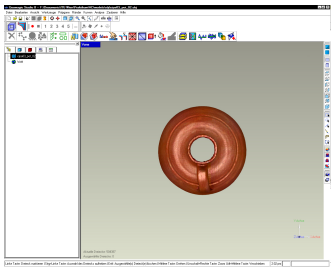


Figure 16: Geomagic Studio 8 loaded cipa pot.

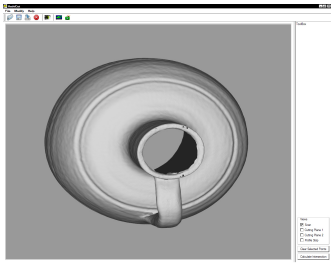


Figure 17: ArchiCut loaded cipa pot.

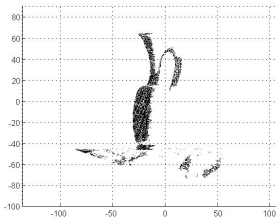


Figure 18: Matlab plot of single scan of KHM's vessel nr. 1054.

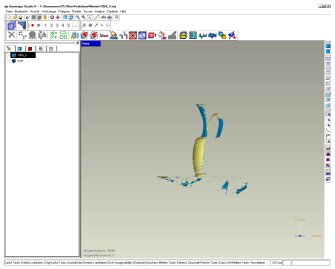


Figure 19: Geomagic Studio 8 loaded single scan of KHM's vessel nr. 1054.

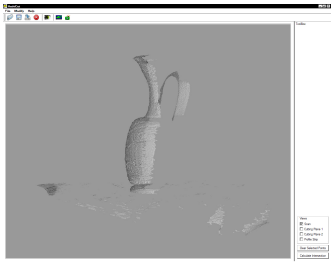


Figure 20: ArchiCut loaded single scan of KHM's vessel nr. 1054.

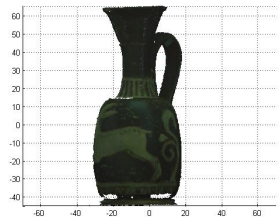


Figure 21: Matlab plot of 90 degree turntable scan of KHM's vessel nr. 1054.

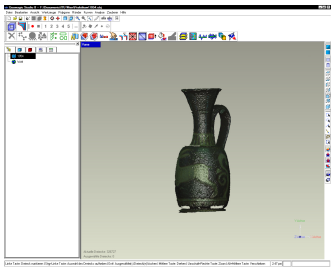


Figure 22: Geomagic Studio 8 loaded 90 degree turntable scan of KHM's vessel nr. 1054.

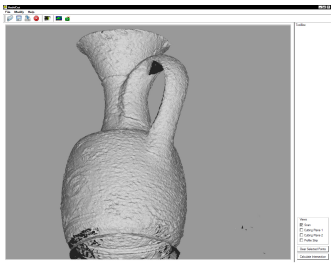


Figure 23: ArchiCut loaded 90 degree turntable scan of KHM's vessel nr. 1054.

Figure 24: Matlab plot of 60 degree turntable scan of KHM's vessel nr. 3754.

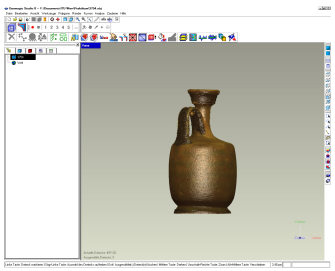


Figure 25: Geomagic Studio 8 loaded 60 degree turntable scan of KHM's vessel nr. 3754.

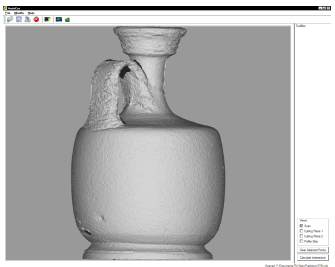


Figure 26: ArchiCut loaded 60 degree turntable scan of KHM's vessel nr. 3754.

4.2 Profiles of Synthetic Data

For verification of the visualizations and the profile line algorithm, synthetic data was used. Having simple 3d input like the unity cube allowed the testing and verification of coordinate normalization and shading issues. Using spheres of different resolution (10 to 1000 vertices) where used for testing prior versions of the profile generation algorithm. As such objects tend to have no T intersection vertices in profiles, it is possible to analyze the performance of the algorithm without this issue and see how it deals with different amount of 3d data. An OpenGL visualization of the trivial profile is shown in Figure 27, the according estimated profile is shown in Figure 28.
Complex models were used, including the famous harvard bunny, the 3ds detail cow, and the unity teapot (as it is ceramics as well) for further testing. The size of these 3d data is comparable to real 3d scan data and therefore has been adequate for first tests (see Figures 29 and 30).
As shown in Figures 31 and 32, for further testing of the applications, semi-synthetic data of real objects has been used.

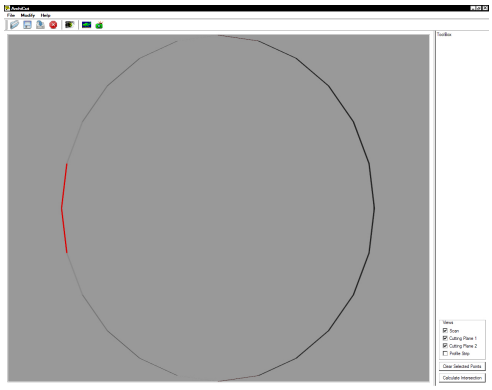


Figure 27: OpenGL visualization of profile of the unit sphere.

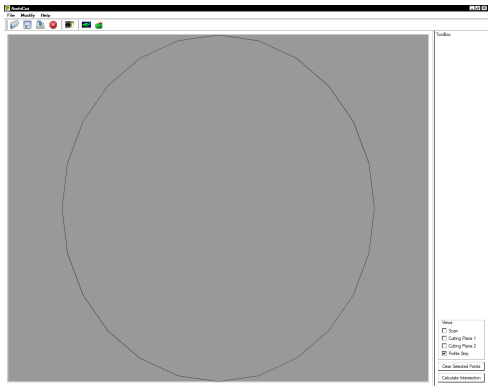


Figure 28: Profile of unit sphere

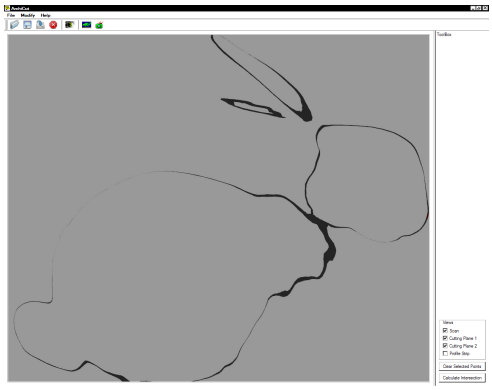


Figure 29: OpenGL visualization of profile of harvard bunny.

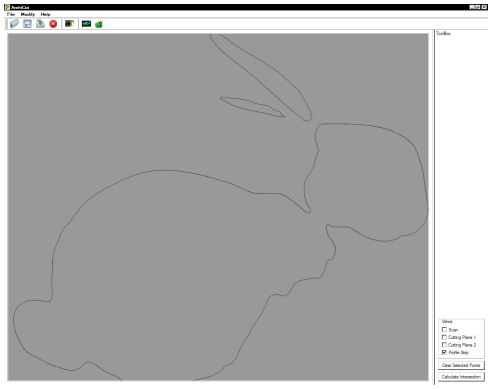


Figure 30: Profile of harvard bunny.

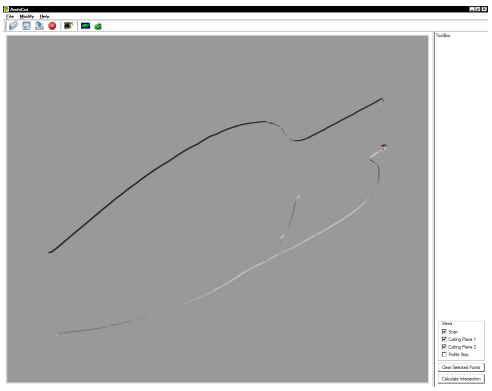


Figure 31: OpenGL visualization of profile of semi-synthetic cipa pot.

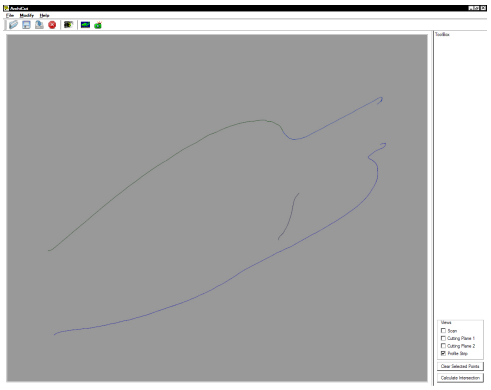


Figure 32: Profile of semi-synthetic cipa pot.

4.3 Profiles of Real Data - Antique Ceramics at KHM

Figure 33: Room of vessels at the KHM. From: [5]

From March, 31st to April, 14th the scan of 121 antique greek vessels (see Figure 33) took place. 6 vessels have been acquired later due to the combination of a spectral analysis [7]. Due to security restrictions, the technicians must not touch any of these precious vessels and sherds. Therefore, the work of storing the ceramics was done by the staff of the Kunsthistorisches Museum Vienna (KHM), the placing on the turntable by an archaeologist and the actual scanning by a technician. The 121 pieces of ceramics took eight working days to scan, doing averaged 15 vessels per day. Transport and installation of the equipment into the KHM and back not included.
As seen in Figure 34, the experiment arrangement can easily be installed in an office, as done at the KHM.

[18]Photo by Dr. Elisabeth Trinkl.

Figure 34: Scanning of antique greek ceramics[18].

One new task of the experiment was the fact, that a ordinary digital camera was mounted on the top of the 3d scanner. For 24 of extraordinary pieces of ceramics, each scan position has been photographed (see Figure 42). The fotos have been indexed and can be assigned to the according scan for future research of texture mapping and visualization enhancements of ArchiCut (see Section5).
Manual cleaning and registering of the gathered 3d scan data took another 3 days to get reasonable data for the automatic profile generation[10]. Because of the valuable objects, the objects have to saved and stabilized on the turntable with foamed material. Therefore, this stabilizing objects have been scanned with the object.
This endurance is convincing compared to the traditional way of documentation of archaeological finds and will be verified after having generated all profiles.
Converting the 3d data manually to .OBJ files, it is possible to test the application with real 3d scan data. The original 3d scan data is larger than the cleaned and registered data, often larger by a factor of ten or more. The reason for this increase of amount of data is that the scanner often gathers data not only from the scan object, but from the turntable, the surrounding or security mountings of the scan objects, as well. Next, using turntable steps of 60 degrees, result in having six scans as one input file, which happen to be more, if there is a need to scan in several smaller steps for reducing scan shadows.
The given data was loaded into the application without any difficulties. Rotation and viewing of the data was possible in real time. If the 3d data size exceeds the memory size of the graphic card, the performance decreases notably, depending on the speed of the AGP or PCIe slot and the memory of the computer. Using the scan computer with a Pentium 4 HT with SCSI drive, 4 GB of RAM and a decent GeForce graphics card, no problems occurred.
Profiles of the data used for the performance test in Section 4.1 are shown in Figure 35 to 40.

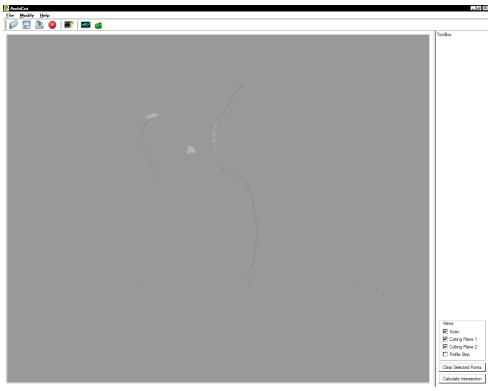


Figure 35: OpenGL visualization of profile of single scan of vessel nr. 1054.

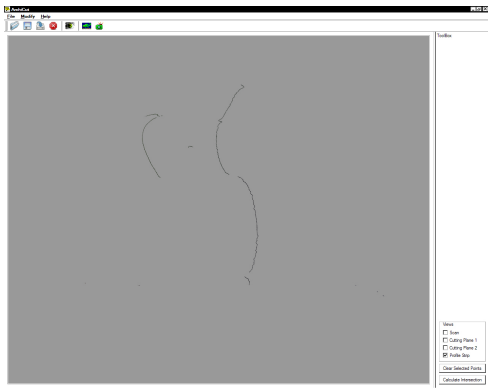


Figure 36: Profile of single scan of vessel nr. 1054.

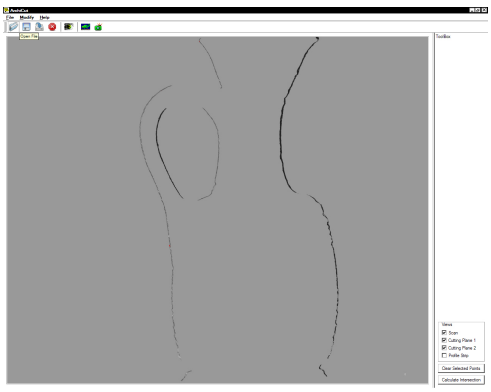


Figure 37: OpenGL visualization of profile of 4 scans of vessel nr. 1054.

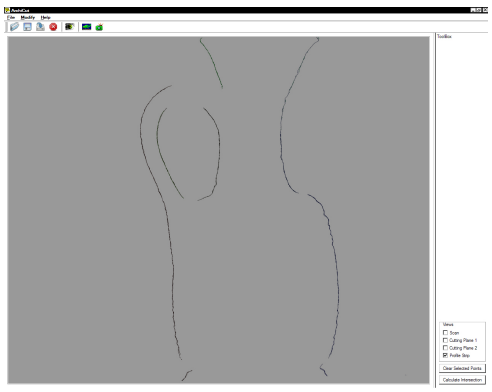


Figure 38: Profile of scan of 4 scans of vessel nr. 1054.

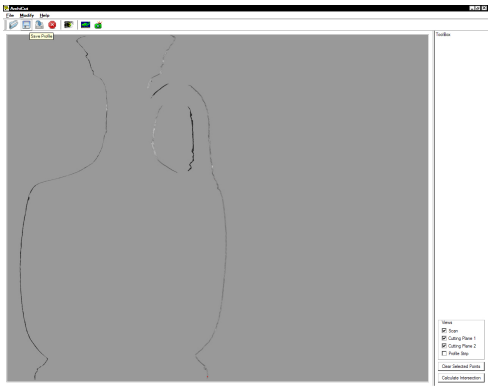


Figure 39: OpenGL visualization of profile of 6 scans of vessel nr. 3754.

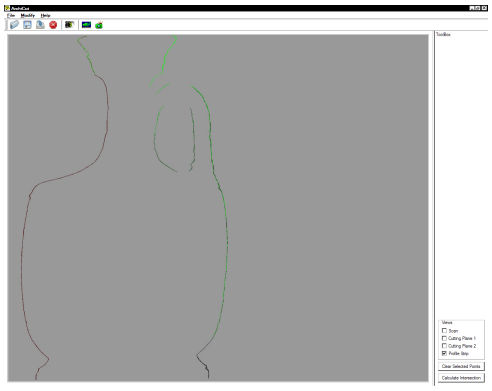


Figure 40: Profile of scan of 6 scans of vessel nr. 3754.

5 Outlook and Conclusion

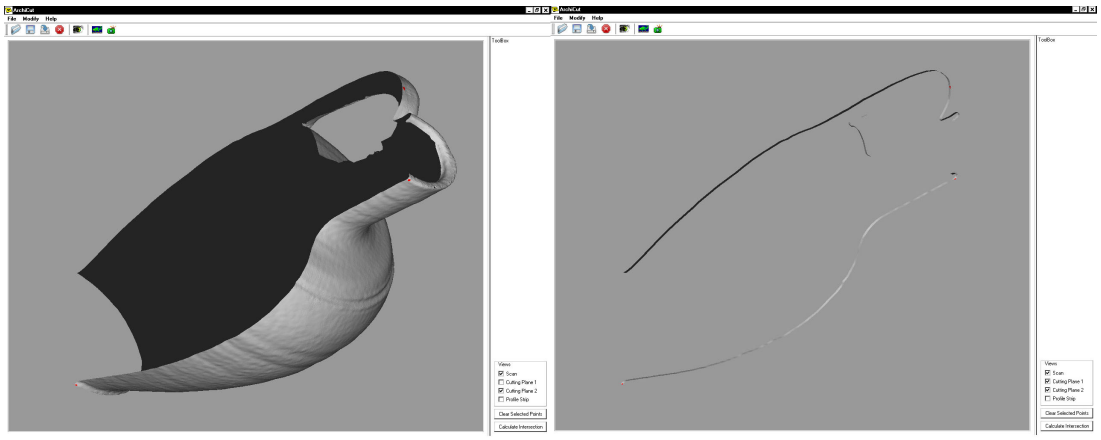


Figure 41: ArchiCut with OpenGL profile visualization.

A real time application for profile generation is presented. Motivated by the needs of modern archaeology to automatically document archaeologic findings, the emphasis was put into modern development and use of decent hardware. The manual classification is a time consuming process to get a systematic view on the excavation finds by being photographed, measured, and drawn.

- Viewing capabilities of 3d data are faster than the possibilities of the similar Matlab application. It takes advantage of today's GPU performance with doing the visualization in OpenGL only.
- The implementation of an efficient vertex selection algorithm provides interactive intersection plane determination.
- Visualization of profile lines are implemented by using the GPU only. No calculations are done for testing intersecting planes by the CPU (see Figure 41).
- A capsuled profile line generation algorithm is implemented.
- Storing of the profile in normalized 3d coordinates as well as in transformed 2d data is implemented.
- Profiles can be stored in convenient image file formats. ArchiCut provides the exports functionality in the common file formats .jpg, .bmp, and .png. The info file provides information about the origin of the picture, the resolution, the scale factor and the projection of the image.

The existing system for documentation of rotationally symmetric archaeological finds of [10] showed outstanding results in all experiments, but has problems in visualizing and processing large 3d scan data because of its MATLAB implementation. Thus, the future goal of this project is to implement all the project with a reliable real time capable programming language.

ArchiCut provides the basis of a longer time project: Using well supported and active open source tools only, future development will be possible due to extended documentation and development of the tools itself. Having now a graphic engine to visualize, process and work with clean 3d scan data, more and more of the automatic documentation systems may be added. Using this code is not only a performance and quality of visualization issue, but last but not least the lack of an expensive license agreement like the one from Mathworks' MATLAB.
Data acquisition is abstracted from the application. Using file handlers, the data acquisition implementation has to provide a temporary file of the 3d scan data. Using Qt and its Designer, new interaction windows may be integrated easily and so, new functionalities and even completely capsuled programs in the field of data acquisition may be attained in acceptable time.
Thanks to the former application as a ground work, ArchiCut is able to handle three open source ASCII 3d data files. For now, .OBJ files are used only. As they are well provided by the Minolta Software Development Kit, this functionality is enough for now. If other scanners using other format, additional file parser may be included.
A profile of ceramics as a major part of its documentation can be generated by this application as well. Using an own selection routine, it is possible to interactively select vertices through several different 3d data objects. This selection functionality should provide editing capabilities in future works.
Visualizing profiles by not calculating any of the intersection vertices but just clip data on the GPU, is a major performance success in the development of a documentation system. Calculating the real intersection vertices and their adjacency, a highly object oriented approach is used: After calculating the intersections, this data is used for the algorithm only. Then, to each vertex, several attributes are given to determine its properties in the final profile and finally, inserted in the profile data structure. Having an interactive solution is an advancement in experimenting with different approaches, and finally, for the use of experts by having a saving of time compared to drawing of a profile.
The main emphasis on implementing this piece of software was the abstract layering of the code. Using the heritage system from Qt, data processing, and user interface is capsuled on so, open for further development. One proof for the reusability for the code is the fact, that the majority of the data parser and data structure is already reused from former applications.
The profile generation is implemented in one abstract object. It may be changed and other algorithms added with no change in the overall structure of the 3d data, assumed to have the specifications of the resulting profile met. The algorithm was designed for the future use in the rotational axis estimation. Thus, it is possible to use previews and estimations of the profile for the rotational axis estimation and thus have a performance advantage.
The experiments with given 3d scan data were all positive and using the given 3d scan hardware, no problem for both the GPU and the CPU, to process all tasks in real time. Assuming future development in the field of graphics cards, having more graphics memory than the 128MB and using a PCIe instead of AGP interface on the test computer will be highly appreciated and will resume in a major performance advance in the future.
Future developments will include:

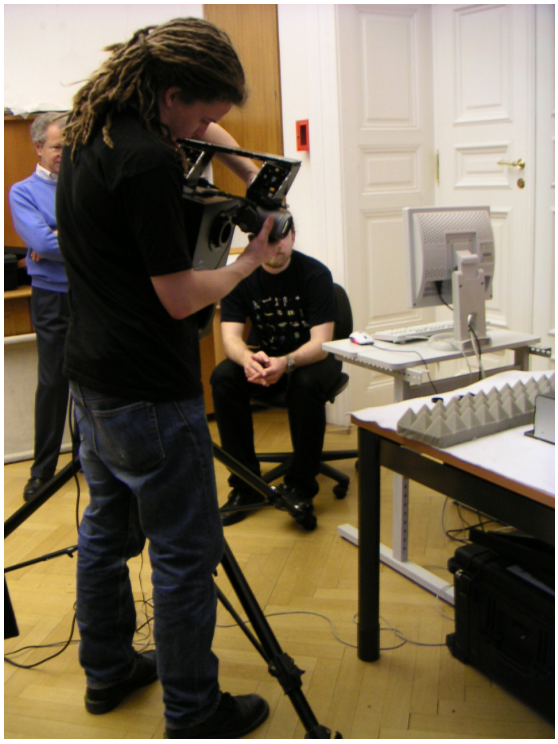


Figure 42: Taking a high resolution texture photo for further analysis[19].

- The implementation of the axis estimation using circle fitting or the future ellipse fitting (in process) to provide a fully automated documentation system as proposed in [10].
- The implementation for blinning high resolution textures on the model using advanced projection models. The provided UV map of the 3d scan data may be a basis of this work. Probably, the abstracted methods of Qt could be used as well for standard projections.
- Archaeologists want to rework generated profiles with certain hatchings, colors and estimated inward profile segments. This can be done in external applications as the data can be exported both in 3d data and image information. For further development, it is reasonable to add this functionalities for building a all in one solution.

As shown, the application assists and increases the speed of archaeological documentation. The high performance implementation of a documentation system will be of interest of archaeologists and for other user-groups working in the field of restoration and conservation, which are mostly museums, and for archaeological research regarding interpretation of finds.

References

[1] Werner Bruckner. Comparison and Evaluation of Mesh Simplification Techniques. Technical report, Institut fuer Computergraphik Algorithmen der Technischen Universitt Wien, 2004.

[2] Matthias K. Dalheimer. *Practical Qt.* DPunkt Verlag, 1. Auflage, 2004.

[3] Peter Fuerreder. Genauigkeitsabschaetzung von 3d-Scannern Praktikumsbericht. Technical Report PRIP-TR-094, Vienna University of Technology, Inst. of Computer Aided Automation, Pattern Recognition and Image Processing Group, 2005.

[4] I. Gathmann. ARCOS, ein Gerät zur automatischen bildhaften Erfassung der Form von Keramik. *FhG - Berichte*, (2):30–33, 1984.

[5] K. Gschwantler. Zur Wiedereroeffnung der Antikensammlung des Kunsthistorischen Museums in Wien. Technical report, Forum Archaeologiae 38/III/2006.

[6] Helmut Herold. *Das QT-Buch. Portable GUI-Programmierung unter Linux/Unix/Windows.* Milin, 2. Auflage, 2004.

[7] P. Kammerer, H. Mara, B. Kratzmueller, and E. Zolda. Detection and Analysis of Lines on the Surface of Archaeological Pottery. In Vito Cappelini and James Hemsley, editors, *Proc. of EVA05: Electronic Imaging and the Visual Arts.* Pitagora Editrice Bologna, March 2005.

[8] U. Leute. *Archaeometry: An Introduction to Physical Methods in Archaeology and the History of Art.* John Wiley & Sons, December 1987.

[9] H. Mara. Automated profile extraction of archaeological fragments. Technical Report PRIP-TR-083, Vienna University of Technology, Inst. of Computer Aided Automation, Pattern Recognition and Image Processing Group, 2003.

[10] Hubert Mara. Documentation of Rotationally Symmetrical Archaeological Finds by 3d Shape Estimation. Technical Report PRIP-TR-103, Vienna University of Technology, Inst. of Computer Aided Automation, Pattern Recognition and Image Processing Group, 2003.

[11] Hubert Mara and Robert Sablatnig. Semiautomatic and Automatic Profile Generation for Archaeological Fragments. In Allan Hanbury and Horst Bischof, editors, *Proc. of 10th Computer Vision Winter Workshop*, pages 123–134, Zell an der Pram, Austria, February 2005.

[12] Institute of Electrical and Electronics Engineers. Standard computer dictionary: A compilation of ieee standard computer glossaries, 1990.

[13] C. Orton, P. Tyers, and A. Vince. *Pottery in Archaeology*, 1993.

[14] H. Pottmann and J. Wallner. *Computational Line Geometry.* Springer Verlag, 2001.

[15] R. Sablatnig, C. Menard, and P. Dintsis. A Preliminary Study on Methods for a Pictorial Acquisition of Archaeological Finds. Technical Report PRIP-TR-010, Vienna University of Technology, Institute of Computer Aided Automation, Pattern Recognition and Image Processing Group, 1991.

[16] R. Sablatnig, C. Menard, and Petros Dintsis. A preliminary study on methods for a pictorial acquisition of archaeological finds. Technical Report PRIP-TR-010, Vienna University of Technology, Inst. of Computer Aided Automation, Pattern Recognition and Image Processing Group, 1991.

[17] Dave Shreiner. *OpenGL Programming Guide. The Official Guide to Learning OpenGL, Version 2.* Addison-Wesley Professional, 5. Auflage, 2005.

[18] C. Steckner. SAMOS: Dokumentation, Vermessung, Bestimmung und Rekonstruktion von Keramik. *Akten des 13. internationalen Kongresses für klassische Archäologie - Berlin*, pages 631–635, 1988.