3D Authoring for Augmented Reality

Dominik Christian Widnig,
Dipl.-Ing. Peter Mohr,
Dipl.-Ing. Markus Tatzgern,
Dipl.-Ing. Dr.techn. Denis Kalkofen

Inst. for Computer Graphics and Vision
Graz University of Technology, Austria

Technical Report
ICG–TR–001
Graz, June 17, 2014
Abstract

Augmented Reality (AR) applications are often used to give users additional information to their current context. The problem is that those applications need a specific support of the real environment to place the AR models correctly. A wide-spread method is to use AR markers. A simple way is using 2D AR markers. These markers are registered in the system and consist of a unique pattern. For displaying a virtual object, the marker has to be placed in the scene.

The AR animations often refer to a real object in the scene, therefore the 2D marker has to be placed on a specific position to interact with these objects. To avoid placing such extra markers in the scene, we will use the object itself as a 3D marker. In this thesis, we will present a method which is based on [12] to find a real object in the scene based on a virtual model of itself. We will present a technique for creating 3D markers with the framework of [9], creating Virtual Reality (VR) scenes based on OpenSceneGraph (OSG), aligning a marker with the online scene and setting the created VR elements to the relative position in the real scene. The finished procedure will give us an AR scene with animated elements which refer to the 3D marker.

Keywords: Augmented Reality, Authoring tool, Registration
<table>
<thead>
<tr>
<th>Listings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Load OBJ File ........................................................................ 19</td>
</tr>
<tr>
<td>2  Playing Animations .................................................................. 25</td>
</tr>
<tr>
<td>3  Color mask ........................................................................... 26</td>
</tr>
</tbody>
</table>
1 Introduction

The usage of 2D markers in AR applications is a simple and fast method to display virtual objects in the real scene, but it needs some requirements. First of all, the marker must be in sight of the camera. If the marker cannot be recognized by the AR system, it will not place any objects in the environment. Secondly, a marker has no relationship with its AR objects. Therefore, a marker must be linked with a specific virtual scene, so that the AR system knows what to do when recognizing this marker. The motivation of using 3D markers is that we do not want to take care of some additional markers placed in the scene. The marker should be an object itself and all AR elements should be placed relatively to this element. If the system finds the real marker in the scene, it should display the VR elements. We also propose an OpenSceneGraph (OSG) \(^1\) editor which is used to quickly generate a VR scene for the 3D marker. In 5 we give an overview of the editors features. This editor is designed to be extended in an easy way in order to enable developers to quickly add new functionalities. Therefore, we will also give insight into the main design principles of the editor.

We will also adapt the registration algorithm of [12] to bring the virtual 3D marker in convergence with the real object. The result of [14] is a transformation matrix between the 3D marker and the corresponding captured marker in the online scene. We will make usage of the scene graph structure to apply this matrix to our generated VR elements which will be transformed to the right location in the scene. As a result, we will get an aligned AR scene that can play OSG animations.

\(^1\)http://www.openscenegraph.com, May 2014
2 Related Work

Since this thesis uses a wide range of technologies we classify this section into four parts.

The first part is about 2D tracking techniques. Vuforia presents a lot of different tracking techniques depending on the user’s application. It can be used on Android, iOS and Unity. Vuforia can create different types of trackable markers. An image target is an image that the Vuforia SDK can detect and track. Those image targets does not need special black and white codes to be recognized. Instead, they use sophisticated algorithms to detect features in the image. A target database is then compared with the detected features and if the image target is detected, it will track the image even if it is not fully displayed in the camera’s field of view.

Placing VR objects to the marker can be easily done with Unity \(^2\) and the Vuforia extension. They provide an editor to place targets with the VR elements (Figure 1).

![Figure 1: With Unity and the Vuforia extension an AR scene can be easily created. We can see an image target with a cube placed in the middle of the image. (Referenz)](image)

An extension of the image target are user-defined targets. Such a target is created at runtime from user-selected camera video frames. This type of tracking is often used in books, posters or magazines. There are more different tracking techniques for 2D targets, but we do not discuss them all here. We therefore refer to the Vuforia-Website \(^3\).

The second part will cover tracking techniques with 3D markers. \([6]\) uses

\(^2\)https://unity3d.com, May 2014
\(^3\)https://www.vuforia.com, May 2014
cylindrical targets, which needs a target image around the cylindrical objects (Figure 2).

![Figure 2](image)

Figure 2: Vuforia presents a sample for the usage of cylindrical targets. We can see a labeled can captured by a Smartphone. An additional object (football) was placed and animated around the can. With occlusion handling and their extended tracking feature the 3D marker can be transformed (e.g. rotated) online and the tracking will still work correctly. (Referenz)

In [7], Vuforia also provides a technique to track so called multi-targets. A multi-target consists of multiple target images. These images have a fixed spatial relationship to each other. These relationships are stored in a XML file. The file stores tracking attributes such as the image target itself, the size of the image target, the translation and the rotation of each side. Once one part of a multi-target is detected, all other parts can be tracked as their relative position and orientation is known.

Another approach is from [1]. With a given 3D CAD model, edges are extracted in their creator. It is also possible to place additional models next to the CAD model. If the marker is found in the scene, the additional objects are displayed. The workflow of such a tracking process is shown in Figure 3.
Figure 3: Metaio uses a 3D CAD model for their tracking technology. The CAD model is loaded into their application (a) and the edges are extracted in (b). We can also see in (b) that the user can select the level of details for the edges. After edge-detection is completed, we can add additional objects to the 3D marker (c). If we then capture the scene again, Metaio tries to find the model and displays, in case of success, the additional VR object immediately in scene (d).

In the third part we will mention some editors which are based on OSG. With [2] an OSG scene can be modified using the osgIntrospection framework ([4]). [3] is also an OSG editor to edit an existing OSG scene by visiting and displaying the OSG child nodes to the user. [5] allows to create different types of OSG elements and has something in common with [3], but it is easier to use for 3D graphic artists.

The fourth part is about the registration algorithm for 3D point clouds. [14] proposes a combination of feature extraction and Iterative Closest Point (ICP). In coarse registration the feature points are detected and feature point links are set up according to the relationship establishment among all detected feature points. In [8] a novel local invariant feature is proposed and used in the coarse registration stage. The precise registration is done with two new structural constraints combined with the ICP. [12] uses Fast Point Feature Histograms, which describe the local geometry around a point in a point cloud ([13]). They also introduce some optimization that reduces the computation time. A new sample consensus based method for bringing two data-sets into the convergence basin of a local non-linear optimizer is
proposed to validate the results: SAC-IA (SAmple Consensus Initial Alignment).
We will make usage of the provided framework of [9]. They demonstrate their approach with a SLAM (Simultaneous Localization and Mapping) based prototype [10] that allows them to capture a real world scene.

3 System Overview

The system consists of three main components: creation of a 3D marker, an editor for creating animations (OsgArEditor) and the registration process, which will align our generated 3D marker to the inherent real object and display the VR objects and animations in the real environment. Figure 4 illustrates the dependencies of each component.

Figure 4: The first step in this thesis is to create a 3D marker from a real object. This marker is then used in our OpenSceneGraph editor to create a VR scene. We load the 3D marker and the VR scene in our system and try to find the marker in the captured online scene. With the help of the registration algorithm of [12] we get the transformation between the 3D marker and the found object in the online scene. The resulting transformation matrix of the registration process is then used to align the VR scene from our editor to the online scene and we get an animated AR scene which was aligned by a 3D marker.

We make usage of the provided framework of [9] and enhance it to create 3D markers. The PointCloudLibrary (PCL) [11] combined with the Microsoft
Kinect for Xbox 360 \(^4\) is used to capture the scene as a point cloud. These point clouds are used to extract a 3D marker and also as an input for the registration algorithm from [12]. An implementation of the algorithm is already provided in the PCL.

OpenSceneGraph and Qt \(^5\) are the basis of the proposed editor, which will be used for supporting the 3D marker with additional VR elements and animations. Finally, we take the framework of [9] which already handles OSG and load our generated VR scene. The 3D marker will be transformed in a point cloud and the registration algorithm will be applied between this marker and the online scene. The result is a transformation matrix which is used to align the VR scene to the real scene correctly. Due to OSG properties, we can animate the aligned OSG elements. As a result, we receive the finished AR scene.

In section 4 we explain the creation of a 3D marker. Our authoring software for creating VR scenes and animations is described in section 5. In section 6 we illustrate the registration process in detail and finally we give some examples how our program can be used in practice.

### 4 Creating a 3D Marker

In this section we will explain the creation and the inherent process to create a 3D marker.

Starting our system will capture the whole scene. The scene is represented as a point cloud. In our case we capture the size of a cube of 1 metre length. We can export this point cloud to an OBJ file. Our file would contain all the vertices of the previous captured scene. The vertices would represent the new 3D marker. In some cases this will fulfill our requirements, if we really want to have a big marker and the rest of the environment will not change in future. However, in case the new marker is only a small part of the scene, we present a method in 4 to extract only the interesting points of our captured scene.

#### Enclosing the scene

We previously discussed the export of the whole scene. However, in some cases we prefer a marker which is only a part of the scene. Therefore, we have implemented a feature to only export a specific area. A simple but effective


\(^5\)http://qt-project.org, May 2014
way to minimize the scene is to enclose it. OpenSceneGraph provides a simple geometry, the so called osg::Geode which uses an osg::Shape. In our case the shape is an osg::Sphere which is displayed in the scene. We set the polygon mode to lines, otherwise we would get a sphere which covers all objects behind it. Our sphere should be placed in front of our camera, regardless of whether we move our camera or not. Therefore, we multiply the given view matrix with an offset in Z direction to place the center of the sphere in front of the camera. Otherwise the sphere would encase the camera and not the scene. Taking the inverse of this new matrix will result in the world coordinates of that point. Since we recalculate these steps every time the camera is moving, the center will always be in front of the camera. The osg::Sphere is placed as a child of an osg::MatrixTransform (sphere_pos). The matrix of the osg::MatrixTransform gets the updated position of the calculated center. As the system of [9] allows us to display the real environment as an OSG element, the polygon lines of the OSG sphere will intersect with the virtual scene. This gives us a good feedback on which area will be used for a marker export and which not. In Figure 6b we can see that the osg::Sphere intersects with the virtual scene.

Depending on the marker’s size the sphere can be increased and decreased by changing the scale values of its osg::MatrixTransform (sphere_size). If we are satisfied with size and position, we can save the circumscribed area. The system uses the scale factor of the sphere_size and center to keep only points of the point cloud where the distance is less than or equal to the radius. The radius is calculated by using the scale value of sphere_size. Figure 5 illustrates the OSG structure of our enclosed scene.
Figure 5: The OSG structure for enclosing the scene. The first child is an osg::Switch (sphere_switch) to toggle the scene on or off. The child of the osg::Switch is an osg::MatrixTransform (sphere_position) where we set the position of the sphere in front of our camera. We can increase or decrease the sphere by resizing the next osg::MatrixTransform (sphere_size). The model itself is an osg::Geode (sphere), which holds the osg::Drawable for an osg::Sphere.

The next step is to down sample the remaining points. We attach a voxel grid filter to reduce the high density of the point cloud. The more points a marker has, the longer the calculation of the alignment will take. Therefore, a level-of-detail (LOD) export is implemented to choose an appropriate marker for an appropriate use case. Figure 6c shows an exported marker with a high LOD. If there are no requirements in computation time, we can choose a high LOD, otherwise we should use a marker with a low LOD. Even a low LOD marker will perform well in registration process and computation time will radically decrease. The geometry form of the sphere will encase some vertices which we do not want to be part of the marker. In Figure 6c we unintendedly encased a part of the ground plane. We use common 3D Editors
like Meshlab ⁶ and Blender ⁷ to delete the rest of the useless vertices and reorient the marker. Figure 6d shows the finished 3D marker. After removing the unnecessary points we now have a good starting point for section 5 and 6.

Figure 6: (a) represents an object which will be used for a 3D marker. In (b) we see the captured scene as an OSG mesh generated out of the real environment and the displayed osg::Sphere which intersects with the virtual scene. Everything which is encased by our sphere will be exported. In (c) we see such an exported object (OBJ file format). Since we do not want unnecessary points like in this case the ground plane, we edited the element in Meshlab to remove those vertices. As a result, we can see our finished 3D marker in (d).

5 OsgArEditor - An OpenSceneGraph Editor

For this thesis we invented an editor to create VR scenes (objects and animations) for our 3D markers. The editor is based on OpenSceneGraph (OSG). OSG uses scene graph data structure to manage its group and leave nodes. We will exploit this structure to write an user friendly and extendable editor. We do not discuss further details about OSG because it would go beyond the

⁶http://meshlab.sourceforge.net, May 2014
⁷http://www.blender.org, May 2014
scope of this thesis. We therefore refer to the well-known OSG books [15] and [16], which we used to gain the provided knowledge to implement additional features to the editor.

The main functionality of this editor is to load the created 3D marker from section 4 and add additional elements and animations.

![Diagram](image)

Figure 7: The previously generated 3D marker is now loaded into our editor. We can add additional objects to the marker and animate them. As a result, we get an animated VR scene.

5.1 Requirements

The basic requirements of the editor are an user-friendly design and an easy management of the 3D elements. A 3D marker in OBJ file format should be loadable and the additional elements can be animated. Due to the plugin system of OSG, it is easy to import common 3D file formats. It is also possible to import already existing animations of an OSG file. The management of elements is improved by grouping them to animate or manipulate them together. The structure of the scene should be displayed in a tree structure which is shown to the user. Elements can be manipulated by re-sizig, rotating and translating. Due to the design, the editor is open to all OSG supplied animations. Standard geometries can be created as well as 2D and 3D text. We avoid misentries by using the undo/redo functionality. The created scene can be exported and used as an OSG File. Since we are using the editor primarily for the creation of AR scenes, the marker itself can be set to invisible but covers the other virtual elements placed behind the marker (phantom-rendering). New elements and animations can be integrated very easily by developers. The editor is working on Windows and Linux.
5.2 Chosen Components

Since OSG is based on C++, we have chosen the Qt framework. We developed a Qt version of 4.8.5 and an OSG version of 3.2.0. Due to this two components, it should be possible to start quickly with the development of the editor. Both, Qt and OSG, have a very good documentation and there are a lot of discussions since they are under active development. Moreover, Qt was chosen because it is cross-platform, which means that it can be used not only under Windows. OSG has an own library (osgQt) only for QT interface. Therefore, a osgViewer::Viewer can easily be integrated in a QtWidget and Qt supports all the user interface (UI) elements which are necessary for dealing with the editor.

5.3 Design Principles

The highest priority of such an authoring software is a well structured design for future enhancements. Therefore, we use a lot of design patterns which help us to fulfill the requirements of 5.1. Some basic design principles are now discussed in detail to help new developers dealing with the editor.

5.3.1 Command Management

The editor is based on commands. Each user interaction is performed through a command. Thus, we use the undo/redo functionality we have implemented as a simple command management. Due to this management all instructions can be saved and restored if necessary. Each command can be created in a factory (CommandFactory) and implements at least three functions:

- execute()
- undo()
- redo()

The execute() function is called when creating a command and handles the user interaction. undo() revokes the executed command and recovers the previous state before the command was executed. redo() restores the undo() execution. Each function only calls the intern components to do something, but never implements a feature itself. Only some error handling should be caught here to give feedback when a command could not be performed. The UndoRedoManager is the class which keeps the order and therefore the history of the commands. This class has two stacks. One stack is used for the already executed commands and the other one is used for the revoked
commands. If a command is created, the UndoRedoManager calls the \textit{execute()} function of the command. After a successful creation, the command is pushed on the stack of the executed commands (Figure 8). The latest command of the user is therefore always on top of this stack.

![Diagram of command factory and execution](image)

Figure 8: The UndoRedoManager calls the \textit{execute()} function of the command, which was created in the command factory. If \textit{execute()} returns \textit{TRUE} it gets pushed on the stack of all executed commands.

If the user wants to revoke a command, the UndoRedoManager will pop it from the stack. After that, the \textit{undo()} function gets called and the UndoRedoManager will push it to the reverted commands. Should the user perform a redo of an revoked command, the UndoRedoManager will quickly pop the command from the reverted ones, perform its \textit{redo()} function and push it again on the stack of executed commands. Figure 9 shows the work flow of the command management in the UndoRedoManager. In case an \textit{undo()} or \textit{redo()} is not implemented, the return value has to be set to \textit{FALSE}. If a command returns \textit{FALSE}, the UndoRedoManager will not push it to the other stack.
Figure 9: When an executed command is revoked, the command gets popped from the stack and its undo() function is called. If the function has restored the previous state correctly, the command gets pushed to the reverted commands. In the opposite direction the redo() function is called when we want to restore a revoked command and the reverted command gets pushed on the stack with the executed commands again.

5.3.2 Managing the Elements

Since our editor should supply different OSG elements (osg::Node, osg::Geode, ...), we need an extendable element management. The base class of all concrete elements is called PresiObject and there are two classes which are inherited from PresiObject: Slide and Element. Each concrete element (e.g. ParticleElement, NodeElement,...) inherits from the Element class. Figure 10 gives an overview of the element structure in the editor.
Figure 10: The abstract base class of all concrete elements is the Element class. We can add different types of elements as long as it inherits from the base class. We also provide a GroupElement which will hold a list of elements.

**Slide**

A Slide is the root of all elements. The editor is designed like a presenter, therefore the name Slide. A Slide can hold all kind of elements and we can create as many Slide objects we want. Each Slide has its own scene. This means that if we switch between slides, our scene will change to the elements of the other slide. It is also possible to export one specific slide. The Slide class is very important for our animations. Each Slide holds an osgAnimation::BasicAnimationManager (BAM) which again holds a list of registered animations.

**Element**

As mentioned before, the Element class is the base class of all concrete elements. Since each child element needs common OSG components such as osg::Switch and osg::MatrixTransform, we define them in the base class as protected variables. Each child element can therefore connect to the OSG elements of its base class and only then our osgViewer::Viewer can display all the connected elements in a slide. In Figure 11 the OSG components and its connections to an Element object are shown. If we import or create a new Element, it will be connected to our Slide.
Figure 11: The OSG structure of an Element object. Base class PresiObject provides the root node. When creating a concrete element, the constructor of the base class will connect each component as depicted in the figure. The dotted components visualize the mounting points for the concrete elements.

**GroupElement**

A requirement of the editor is grouping of elements. This feature is useful for editing elements which belong together. Using a GroupElement, we can store all concrete elements in one element. This GroupElement will hold these added elements as its own childs and will connect the root of each child to its osg::MatrixTransform. If we want to ungroup a GroupElement, the GroupElement will be removed from the scene graph and the childs will be connected to the parent of the GroupElement.

**Action**

An Action object is used to supply different OSG animations. Since our editor is designed for presenting different animations we introduced the abstract class Action as the base class of all concrete Actions. An Action is nothing else than a holder of a specific OSG animation. These animations are reg-
istered to the slide’s osgAnimation::BasicAnimationManager (BAM). When
the user starts the animations, a new window with an osgViewer::Viewer will
appear and the BAM will play all registered animations until the animation
time is over or the viewer is closed.

5.3.3 Updating the graphical user interface (GUI)

Each created slide, element and action has to be displayed to the user. If we
execute a command which creates such an object, we have to make changes in
the OSG structure. Therefore, each command would have the responsibility
to update the GUI. We prevent developers to do this because each command
is unique and developers should not be asked to take care of GUI updating.
Hence, we decided to implement the observer pattern for this purpose.
Our Observer class is the abstract base class for all observers. All inherited
classes have to implement the notify(Manager* m) function. A concrete
class of Observer is the PresiTreeGenerator. This class has the responsibility
to update the QTree of our Editor. In this QTree the user is informed
about the current structure of its slides, elements and actions.
In the notify(Manager* m) function of the Observer class we have a class
called Manager as the parameter of the method. This class contains a list of
all registered Observer objects and a function called notifyObservers(), which
will tell all registered observers to call its notify(Manager* m) function. The
PresiManager is an inherited class from Manager and performs all actions
regarding to the structural alterations of the OSG elements. Therefore, a
Command object which is executed does not implement structural changes
of the scene graph. Instead it uses the PresiManager class. After each change
of the OSG structure in the functions of PresiManager we can call the no-
tifyObservers() function and our PresiTreeGenerator will then update our
QTree.

5.4 Functionalities

The functionalities of the editor will be described in this section. A user can
make usage of the functionalities to attain the desired VR scene. Due to the
command pattern and the effective management of the elements, the editor
can easily be extended with new functions.

---

8http://qt-project.org/doc/qt-4.8/qtreewidget.html , May 2014
5.4.1 Manipulators

Elements can change its position in the scene. Since OSG uses a user-contrary osg::MatrixTransform for this, we use the osgManipulator::Dragger class. This class is the base class of all concrete draggers. A dragger is nothing else than a graphical representation to edit an osg::MatrixTransform. We have implemented a wide range of different draggers. Depending on the use case we can switch immediately between them. Figure 12, 13 and 14 are three of the implemented draggers.

Figure 12: The TabBox-Dragger is used to resize or move an element in two directions at the same time
Figure 13: With a TrackballDragger we can rotate the element.
Figure 14: A TranslateAxisDragger is used to move the element in one specific direction at a time.

5.4.2 Import Elements

If we want to work with existing 3D models, we have to load them from files. Due to the wide range of supported 3D model file formats in OSG, our editor can load nearly every 3D file. A command which is responsible for loading elements will create a NodeElement. This NodeElement holds an osg::Node, which is filled with the help of the osg::Node* osgDB::readNodeFile(const std::string &filename) function. After the import has been done, we add that node as a child of the osg::MatrixTransform of our Element class.

If we want to load an OBJ file, the OBJ model will automatically be rotated in the OSG scene. To avoid such rotations, we add an osgDB::Options to the readNodeFile function (Listing 1).

Listing 1: Load OBJ File

```cpp
osg::ref_ptr<osgDB::Options> opt = new osgDB::Options
("noRotation\nnoTriStripPolygons");
osg::ref_ptr<osg::Node> loaded_node =
osgDB::readNodeFile(obj_name, opt);
```
Otherwise our OBJ 3D marker would be rotated and the registration result would transform our scene incorrectly.

5.4.3 Creating Elements

In our editor we can also create new elements. We can implement any element, the only requirement is that we have to wrap this OSG object in an inherited class of Element. For this thesis we have extended our editor with some elements which can be useful for an AR application. In the following paragraphs we will shortly explain the implemented elements:

Standard Geometries

OSG provides the class osg::Geode. This is a mixture of nodes and geometries, therefore the name "Geode". With an osg::Drawable and an osg::Shape we can create different standard geometries for our editor. Three of them are already implemented and shown in Figure 15.

![Standard Geometries](image)

Figure 15: We have implemented an osg::Sphere, an osg::Box and an osg::Cylinder. All of them can have custom sizes and colors.

Billboard Text

Billboards are used to display the front side of an image or text even if the camera moves or rotates. Since we are creating virtual scenes for our AR application, we have decided to implement a 2D billboard text to label the objects in the scene. In OSG we are using the osgText::Text class. With
such an object we can set the font, fontsize, orientation and of course the text. Because a pure 2D text in a 3D scene would only be clearly visible from one side, we use an osg::Billboard geometry. This geometry is an osg::Geode which orients its front side always towards the view point. Therefore, we can read the text clearly although we view our scene from different angles. In Figure 16a we can see such a billboard text in the scene with a TabPlane-Dragger. The camera is now in front position. If we move our camera to the left side, our dragger will not rotate, but instead the billboard will rotate the text to the new camera position (Figure 16b).

![2D Billboard Text](image1)

![2D Billboard Text](image2)

(a) (b)

Figure 16: We can see a 2D billboard text in the scene with a TabPlaneDragger. In (a) the camera displays the text from the front side. Altough we view our scene from a different angle (b) our manipulator is viewed from that angle, but our billboard will rotate the text to the viewers eye.

3D Text

Since our AR scene perhaps needs a headline, we introduced a 3D text. The 3D text is created by an osgText::Text3D drawable. The osg::Geode gets the drawable and can therefore display the text. We can set font, fontsize, the orientation and the text of the Element. Figure 17 is an example what a 3D text can look like.

![Hello World!](image3)

Figure 17: This figure shows an example what a 3D text geode can look like.
**Line Segment**

Labels (2D Text) can not always be set next to their models. Therefore, we have implemented line segments to connect the models to their labels. Line segments are defined by their amount of edges and their order. Giving the user feedback on the positions of the edges we show a numbered osg::Sphere for each edge. These spheres can be placed like normal elements in the scene with standard manipulators (5.4.1).

When the user accepts the position of the spheres, a line segment will be created and the spheres will be deleted from scene. As a result, the user has now a colored line segment of which each edge is located at the former spheres center. Figure 18 illustrates the work flow of a line segment creation.

![Figure 18](image)

Figure 18: At the beginning of a line segment creation we display the amount of chosen edges in spheres shifted in X-axis (a). In (b) the user can now move the numbered spheres to the position the line edges should be located. The user can submit the translations of the spheres and will get a finished colored line segment where the center of the spheres will have been transformed to the edges of the line (c).

**Particle Systems**

Particle systems are often used in 3D applications to simulate smoke, dust, explosions, fluid, fire and rain. Due to the osgParticle library we can easily create and manage particle systems.

The editor is extended by a ParticleElement which can be configured to simulate different kinds of effects. The osgParticle::Particle has some properties such as size-, alpha- and color range.

The placer (osgParticle::Placer) sets the initial position of every particle. The shooter (osgParticle::Shooter) sets the initial velocities of particles. Plus, the counter (osgParticle::Counter) determines how many particles should be created [15].

In our editor we have implemented an osgParticle::SectorPlacer. With such a placer we can set the position of the particles to a circle and configure angle, inner and outer radius. In our case the osgParticle::Shooter is an osgParticle::RadialShooter where we set the velocity of our particles. With the osgParticle::RandomRateCounter we set the range of generated particles. An
example of a particle system in our editor is shown in Figure 19.

Figure 19: This generated particle system symbolizes colored smoke.

5.4.4 Outline Effect

The outline effect was introduced to give the user feedback which element is selected at the moment. When a user selects another element in the GUI, the osgFX::Outline effect is set to invisible at the previous selected element and appears on the currently selected element. Figure 20 shows two planes (cessna.osg \(^9\)), one of which is selected in the GUI. The scene graph displays a green border (the outline effect) over the selected plane.

Figure 20: The outline effect is displayed on the selected plane.

5.4.5 Group/Ungroup

As already discussed in 5.3.2, a requirement of the editor was the group and ungroup functionality. A GroupElement can only be created if the selected elements (which are the children of the GroupElement) have the same parent or have no parents (root elements under slide). If we group elements, we remove the previous connections in the scene graph and create a new connection to the newly generated GroupElement. In Figure 21 we have grouped two nodes and re-sized them together with a TabBoxDragger.

**Note:** We use an editor based on a scene graph structure. If we deleted the GroupElement, the osg::MatrixTransform would also be removed from the scene graph. Therefore, our manipulated scene would return to its original transformations.

![Figure 21: We re-size a GroupElement with two children. With our manipulator we can achieve this in one step now.](image)

5.4.6 Create Actions/Animations

In 5.3.2 we talked about the design of actions in our editor. We are now going to discuss these actions in detail. The osgAnimation::BasicAnimationManager manages all the animations of one slide. Each Element has a list of its actions. Each concrete Action has to implement the execute() function of its abstract base class. This function has two responsibilities. First, it has to create the OSG animation. Second, the animation has to be registered at the BasicAnimationManager of the slide.

As part of this thesis we have implemented simple key-frame animations:
Keyframe Animation

Keyframe animations are used to move elements from one position to another. With the usage of our manipulators, translation, rotation and scale can be changed in each keyframe. The user moves the element to the desired position, rotates and scales it. Each keyframe has also a duration time. The whole keyframe animation has a play mode:

PlayModes

Play modes are implemented through the given osgAnimation::Animation::PlayMode. The editor supports 3 different types:

- **ONCE**: The animation is played as a single event.
- **PPONG**: The animation is played permanently. It plays the animation from start to end and vice versa.
- **LOOP**: The animation is played permanently similar to the PPONG mode with the difference that it always starts with the first keyframe.

5.4.7 Play Animations

The editor can play the animations by exporting all the elements and their animations of a slide. The export is done by using the slide’s root node. Since we have a connected scene graph, we take advantage of the `bool writeNodeFile (const osg::Node &node, const std::string &filename)` function of the osgDB library.

After the export is completed we read the temporarily exported file again in our editor and extract the osgAnimation::BasicAnimationManager (BAM). The BAM has a list of all the animations and can play all the animations of the scene. Listing 2 shows the implementation details of getting a BAM of an imported file.

Listing 2: Playing Animations

```cpp
osg::NodeCallback* cb =
    ar_scene->getUpdateCallback ();
anim_manager_ = dynamic_cast<
    osgAnimation::BasicAnimationManager*>(cb);

osgAnimation::AnimationList a_list =
    anim_manager_->getAnimationList ();

for (unsigned int i = 0; i < a_list.size (); i++)
    anim_manager_->playAnimation (a_list [i]);
```
5.4.8 Undo/Redo

In 5.3.1 we discussed the command management of the editor. Each user interaction is executed as a command and pushed on a stack. If we want to revoke the desired action, we execute the `undo()` function of the latest pushed command and it will be pushed on the stack with the revoked commands.

One of the most important user interactions is the transformation of elements. Due to our manipulators an unwanted transformation of the model can easily occur. Trying to fix this with the manipulator again is not really user-friendly. Therefore, we have implemented an own osgGA::GUIEventHandler which creates a command whenever a transformation occurs. The command creation is triggered by the mouse clicks of the user and saves the initial position of the model. When we revoke the command, we call the `undo()` function, save the current matrix of the element and set the osg::MatrixTransform to the matrix with the initial position. Thus, we have both matrices and can therefore call `redo()` again which only exchanges the matrices of the element’s osg::MatrixTransform.

5.4.9 Edit Colormask

This feature was implemented for our registered AR scene. Due to the reason that the virtual 3D marker is placed on the position of the real object, we have to set the marker to invisible. If we only switched off the virtual marker (with an osg::Switch) or deleted it from the scene graph, all virtual objects behind the marker would be displayed.

We avoid such a behavior by setting the color mask of the marker of each RGB value to off. OSG provides the osg::Colormask class. By setting the color mask to off we get an invisible 3D marker which still covers the other virtual elements behind it. In Listing 3 we switch each color and alpha (RGBA) off. Figure 22 shows the result of a colormask which is set to off.

Listing 3: Color mask

```cpp
osg::ref_ptr<osg::ColorMask> color_mask_ =
new osg::ColorMask(false, false, false, false);
// (RGBA - values)

element_ -> setColorMask(color_mask_)
```

26
5.4.10 Save

Each Slide represents its own OSG scene. Therefore, we have implemented an export for each of them. Through the scene graph structure of OSG we take the root node (osg::Switch) of a slide and save the whole graph to one OSG file. Thus, all elements with their animations are saved in that file. We have also implemented an export function for one specific element. In that case the root node of the Element is taken and saved. If the Element is a GroupElement, all children will be included in the saved file.

6 Aligning AR-Animations to the Online Scene

In this section we will discuss how to use the output of our OsgArEditor to playback it in a real scene. On the one hand, we have the output of our editor in an OSG file which holds the 3D marker with additional elements and animations. On the other hand, we have the original OBJ file of our marker. Our system, which uses the Microsoft Kinect for XBox 360 with the PointCloudLibrary (PCL), loads both files. The OBJ file is used to get the vertices of the 3D marker to parse it in a point cloud and the OSG file is needed for displaying the virtual elements. The registration algorithm of [12] uses the point cloud of the marker and the point cloud of the online scene to find the real marker’s position in the scene. As a result of the algorithm we get a rotation and translation matrix which will be applied to the elements of

Figure 22: We can see two avatars in this scene. The right one’s color mask is set to off. It is displayed invisible but still covers the blue wall and some parts of the left avatar.
our OSG file. The OSG marker with all OSG elements is then placed at the position of the real marker and we gain our AR scene. Figure 23 illustrates the sequence of a registration process.

![Workflow Diagram](image)

Figure 23: This figure represents the workflow of our registration process. Our VR elements which are previously generated in our editor are loaded in our system. The system then captures the online scene. We also load our 3D marker and apply a registration algorithm. As a result of the registration algorithm we get a transformation matrix. The matrix represents the transformation between the 3D marker and the real object in the scene. We then transform the VR elements with this matrix, play the OSG animations and therefore get an aligned online scene with AR elements and animations.

6.1 3D Marker Registration

Due to the implementation of the SAC-IA (SAMple Consensus - Initial Alignment) algorithm of [12] in the PointCloudLibrary (PCL) we need an input and at least one target point cloud. The input cloud represents our current tracked scene and the target cloud is our 3D marker, which was generated from an OBJ file. Since we want to achieve our registration process in real time we apply a voxel grid filter to our input cloud to reduce computation time of the algorithm. We do not need to set a voxel grid filter to our target cloud because as previously discussed in 4, the system exports different LOD markers and the user can choose one of them. Plus, this LOD objects are already generated by applying a voxel grid filter.

Furthermore, we estimate the surface normals and the local features with Fast Point Feature Histograms (FPFH) ([12]) for the point clouds. Afterwards we apply the SAC-IA algorithm ([12]) by calling the align() function. As a result, the algorithm transfers the two data-sets into the convergence basin. The output of the algorithm is a matrix which represents the transformation between the marker’s point cloud and the real object marker in the input cloud. In Figure 24 we see the input cloud and the original position of our 3D marker in point cloud format on the left-hand side. After applying the SAC-IA algorithm our marker is transformed to the position of the real marker (right-hand side).
Figure 24: On the left image we see our online scene with our 3D marker in original position. In this example we use a human liver as a marker. After applying the SAC-IA algorithm, the marker is transformed to the position of the real object (right image).

The highest priority in the implementation is to try to get an aligned marker in real time. Each extra point in the marker or in the input cloud raises the computation time of the algorithm. Therefore, we use an additional improvement to minimize this time. We have already discussed in 4 how to enclose a marker out of a given scene. This method is also used to cut out all points which are outside of our grid sphere. As in 4, the user can display the sphere if a quick result is wanted. Otherwise the whole scene would act as the input cloud and the computation time of the registration would increase.

6.2 Transforming OSG Elements to the Online Scene

Since the provided system of [9] can also deal with OSG, we load our created scene from our OsgArEditor (5). The provided OSG standard function from the osgDB library is used to import the OSG scene (Listing 1). Since we will get a transformation matrix as a result of the registration process, we can now prepare our OSG scene to transform it to the same position as the point cloud marker in Figure 24. Our loaded scene, henceforth called ar_scene, is now in our system and we add an osg::MatrixTransform as a parent of it. This osg::MatrixTransform is responsible for the correct rotation of the scene. Let us call it arScene_rotation. We also have to apply the translation to our OSG scene. Therefore, we create a further osg::MatrixTransform which is the parent of arScene_rotation. This osg::MatrixTransform is called arScene_translation. Due to the fact that we will attach this scene graph to
the root node of the system when the application starts, we have to hide the scene before registration is done. Otherwise we would have a disturbing scene in the origin of the coordinate system. Therefore, we set an osg::Switch as a parent of the arScene_translation. Thus, we can set all children off until registration is completed. Figure 25 visualizes the OSG structure of the scene.

Figure 25: The AR scene visualized in a scene graph structure. The first child is an osg::Switch (arScene_switch) to toggle the AR elements on if registration is completed. arScene_translation is used to set the translation of the registration result. For rotating we apply a second osg::MatrixTransform (arScene_rotation) to set the rotation. arScene is the loaded node with all VR elements and their animations.

If registration is completed, the resulting transformation matrix will be set to arScene_rotation and arScene_translation. We then turn the osg::Switch on and can therefore see our virtual elements next to the real 3d marker object. The virtual marker is not displayed in the scene because we set its osg::ColorMask of all colors to FALSE. Nevertheless, the "invisible" marker will hide all virtual elements which are placed behind.

The AR scene is now at the right position but the elements are not animated yet. OSG animations are only played if the osg::BasicAnimationManager
(BAM) starts the registered animations of its list. Due to the fact that we only have a loaded osg::Node from the file system, we have to fetch the BAM out of the node. We now refer to Listing 2 in 5.4.7. The system performs the same process as the osgArEditor to animate a loaded OSG file. As a result, all virtual elements with animations then act out in the AR scene.

7 Results

Combining the exported OSG scene from our editor with the aligned online scene through the 3D marker gives us a new animated AR scene. In Figure 26, 27, 28 and 29 we give some examples to show how our technique can be applied for different purposes.

Figure 26 is a general example for an AR scene. We can see the exported OSG scene with a white background (26a). The marker is represented by a wooden candle stand. In Figure 26b and 26c the registration process is already completed with the difference that in Figure 26b the captured virtual scene is also displayed. The finished AR scene is then shown in Figure 26c.

![Figure 26](image)

Figure 26: In this example the 3D marker is a candle stand. The dancing avatar was placed at the left-hand side of the marker in the editor with a billboard text and a linesegment. On top we placed a heading (a). We also set additional real objects in the scene to make it more difficult to find the actual marker. In (b) we see the registered scene with the aligned virtual objects and the captured scene as a mesh build through OSG ([9]). The finished AR scene with the animated OSG elements can be seen in (c).

In the result shown in Figure 27 we applied key-frame animated car parts to the 3D marker. In Figure 27a we see the exported 3D marker of our system. Since we also have a high detailed model of this marker, we replaced our 3D marker with that model. In Figure 27b we can see the detailed model. In our editor we loaded both models, the low-detailed 3D marker and the high-detailed model. Both models have a different position and size in the scene. Therefore, we resized and transformed the detailed model to the same
position as the 3D marker. Then we deleted the 3D marker and added some single car components and applied them to the new "marker". Finally, we set the osg::ColorMask of the detailed model to FALSE for phantom rendering. However, the registration is still done with the original low-detailed marker due to performance reasons. As a result, we get an animation which represents a disassembling of that model (Figure 27c). The key-frame animations of each part take four seconds and when disassembling is finished we play back the animation again.

![Figure 27: This result represents a car model. Our created marker in (a) is not representing a good overlapping to the real object. We replaced it by using an already existing OBJ model (b) in the editor. Therefore, we transformed the new model to the same position, rotation and scale as the original marker. The low LOD marker was then deleted and the osg::ColorMask of the new model was set to FALSE for each color. The components were attached with key frame animations which present a simple disassembling of a car (c). Nevertheless, the registration process was still executed with the original LOD marker due to computation time.](image)

In the next example we put some objects "inside" the 3D marker. We used a LEGO\(^\text{10}\) tower as a 3D marker. The tower has an entrance and some planes. We put a few defense units in front of the entry and on the different floors. In Figure 28a we see the captured scene as an OSG mesh with the already aligned marker and units. In Figure 28b we set the osg::ColorMask of the OSG mesh to FALSE and get the finished AR scene.

\(^{10}\text{http://www.lego.com/en-us , May 2014}\)
Figure 28: The LEGO tower is our 3D marker. We have placed some skeletons and dragons as VR elements as a defense units to our tower. In (a) we can see the captured scene in an OSG mesh with the units. In (b) we set the osg::ColorMask of the mesh to FALSE and see the skeletons and dragons defending the tower in an AR scene.

A further result of this thesis is a model of a human head (Figure 29). We want to show that we can easily describe a 3D model with the OsgArEditor. In Figure 6 we see how the human head was generated out of the scene. In our editor we applied some line segments and 2D billboard texts to describe the model. In Figure 29a we see the real object before registration is done. After registration we set the transformation matrix of the alignment algorithm to the OSG scene. As a result, we get a 3D object which is described with texts and lines (Figure 29b).
Figure 29: In this example of our results we want to show how easy it is to label 3D markers. In our editor we loaded the 3D marker (the head) and added billboard texts and lines. As a result, we get a human head which is labeled with some of its components. In (a) we can see the captured scene before the registration is done. (b) represents the aligned AR scene.

8 Conclusion and Future Work

In this thesis we have proposed a fast and easy method to create 3D markers out of a given scene. We have also demonstrated our authoring software to create a virtual 3D scene with our extracted marker. This VR scene can also be animated with key-frame animations. The generated VR scene can then be loaded in our system which captures the real scene. With a registration algorithm we can align the 3D marker to the respective real object. We use the returned transformation matrix of the registration algorithm and apply it to our loaded virtual scene. Finally, we start the animations of the OSG scene and get a nice AR scene with animated OSG elements.

We hope that through the design of the editor developers will extend our OSG editor with new features. The creation of 3D markers should also be improved in future. For example, the ground plane should be directly removed so that an additional 3D Editor is not needed. The registration process could perhaps be improved by splitting the 3D marker in several different views (e.g. front, back, left,...). The registration result would be better if we captured the scene from only one side.
References


