Extending an Existing VR Software Framework to support AR Applications - With an Example from Physics Classes

Florian Mannuß, Florian Bingel, André Hinkenjann
Institute of Visual Computing
Bonn-Rhein-Sieg University of Applied Sciences
Sankt Augustin, Germany
andre.hinkenjann@h-brs.de

***Abstract*—We present our approach to extend a Virtual Reality software framework towards the use for Augmented Reality applications. Although VR and AR applications have very similar requirements in terms of abstract components (like 6DOF input, stereoscopic output, simulation engines), the requirements in terms of hardware and software vary considerably. In this article we would like to share the experience gained from adapting our VR software framework for AR applications. We will address design issues for this task. The result is a VR/AR basic software that allows us to implement interactive applications without fixing their type (VR or AR) beforehand. Switching from VR to AR is a matter of changing the configuration file of the application. We also give an example of the use of the extended framework: Augmenting the magnetic field of bar magnets in physics classes. We describe the setup of the system and the real-time calculation of the magnetic field, using a GPU.**

***Keywords*-Augmented Reality; Virtual Reality; Software Framework; Software Architecture; Electromagnetic Fields**

I. Introduction

VIRTUAL Reality is an established technique in many industrial areas, like automotive, aerospace, plant engineering, and oil and gas industry. Recently, Augmented Reality has provided means to enhance reality with computer generated information. Application areas for AR are assembly training and support, gaming, and medicine. Both, VR and AR offer interactive environments where visual information is generated by a computer according to user input and modifications done by a simulation. This means similar components are needed in both types of environments:

- For user interaction mostly 6DOF (Degrees of freedom) tracking is utilized.
- Visual information is output stereoscopically.
- Some simulation engine manipulates the virtual scene.

The difference between AR and VR lies especially in how the visual information is presented to the user. While VR shows an entirely computer generated environment, AR mixes reality with computer graphics. This has technical consequences because other devices, like Head-Mounted-Displays (HMDs), are needed to implement an AR system. But it also influences the design of the software that drives the AR application.

When we were starting a new project on augmenting school experiments we decided to extend our existing VR software framework enabling it to cope with the new requirements. In this article we would like to present the extensions and changes necessary for this purpose.

In the following sections we present the status of the software before we started AR activities. We then describe the new requirements for AR in detail. Subsequently, we show the design changes on the way from VR to AR. Finally, we present our example application: Enhancing experiments on magnetic field explorations for physics classes. We also give some details about the real-time calculation of the magnetic field, using a GPU.

II. Related Work

To date, many Virtual and Augmented Reality software frameworks exist. Commercial frameworks for desktop / projector -based, immersive, stereoscopic environments with dedicated 6DOF input devices, like 3DVIA [1], are out of the scope of this article, because they can be used to implement AR applications, but information about their software design cannot be acquired by the public. The omni present AR apps on smart phones are based on AR toolkits, e.g Metaio's Mobile SDK [2]. These often only have AR functionality and are closed source.

Related to our work are VR frameworks that have a very modular design which can be modified towards AR usage and real VR/AR frameworks. In [3] an extension to the well known MVC paradigm is proposed, called MVCE(nvironment). While this is a valid abstract concept, no design details of the underlying software is given. DWARF [4] is a component based AR framework realized as a set of services. The latter, as well as the AMIRE framework [5] are focused on AR. VHD++ [6] contains the vhdRuntimeEngine which implements a vhdServiceManager. vhdServices are software components that are loaded as plug-ins. The vhdViewerService is the renderer component, capable of mixing the camera image with graphics. Although the modular, plug-in based design is similar to our approach and provides a high level of flexibility, it is not clear how migration from VR to AR applications is done. In [7] Ohlenburg et al. describe the MORGAN framework. It provides its own render engine, separating render data from other scene data. No details on

the design consequences from the difference of VR and AR applications is given. VARU [8] is a client-server framework for tangible space applications which is component based. Components are described by XML-files, as well as their configuration. AR and VR applications are handled by the VARU client by their respective "AR / VR managers". Both connect to the VARU server which supplies data needed by AR or VR applications. The "streaming manager" connects to clients, e.g. video cameras that supply real imagery. In the following we describe our own AR / VR software framework, called "basho".

III. BASHO AS IS

The VR framework *basho* was designed with the aim to build a modular and extensible system. This includes

- Small kernel for data management and system control
- Separation of scene data and processing (rendering, simulation)
- Independance of input/output devices and scene representation
- Easy extendability through components, e.g. sound, physics, ray-tracing

Further on, all plug-ins have to implement a predefined configuration interface allowing a plug-in independent processing of the configuration file. This is needed to make known all plug-ins that have to be loaded to the application. A central design principal was the principle of locality which forbids distribution of source code to complete one task over different classes that are responsible for different tasks. For example, registration of a new data component should not need changes in different already existing source files. Instead, one method call should be sufficient.

In our terminology the kernel is the framework responsible for system control and high level data management. System control includes the system's main-loop, loading and managing run-time loadable components. These run-time loadable components represent input-devices, scene data, scene manipulators (actions), rendering and simulation. Components that have to be loaded at run-time as well as their configuration are specified inside the application's configuration file.

Our VR framework is divided into five main components drawn in solid lines as seen in figure 1.

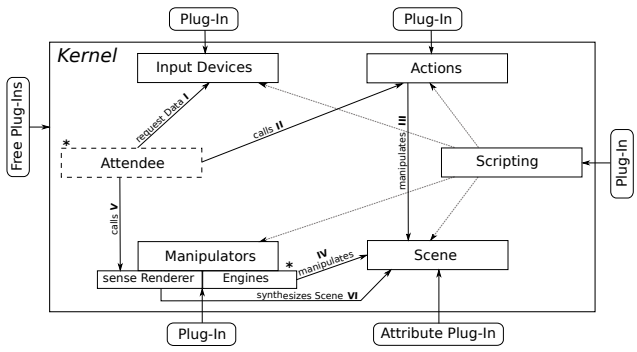

Fig. 1. Kernel sketch with the plug-in interfaces. The numbered lines show the calling order of the single components during one main-loop cycle. Objects marked with (*) are called through the main loop directly.

The connecting lines characterize the information flow within the system. The *Scene* component is responsible for storing and representing the virtual world. The *Attendee* has a special purpose within this virtual world, since it is the representation of the actual user. The attendee is part of the virtual world and it consequently belongs to the *Scene* component.

The *Manipulator* component is responsible for managing *Engines*, that manipulate the virtual world and *Sense Renderers* for stimulation of the user's senses. Figure 3 shows three example engines for manipulating the virtual world.

Input Devices handles all known physical input devices. Input devices do not manipulate the scene. They only convert the received input data into a generic format. *Actions* manages all scene manipulators the user can use to interact with the scene or manipulate it. These actions are called from the input devices after pre-processing the input data in order to be able to implement non input device specific actions. Some sample actions are "create new object", "delete object", "move object" or "interact with object".

The *Scripting* component provides an interface for programmers to extend the framework using a scripting language or to develop applications using a scripting language with the benefits of run-time manipulation and rapid prototyping.

All components only provide the interfaces and all input devices, actions, scene element attributes, renderers, and scripting support are implemented as plug-ins that are loaded during run-time.

One key design strategy was the separation between scene data storage and the data processing. Data processing is on one hand the rendering, this is producing the input for the human senses, and on the other hand scene manipulation through simulation (Engines). In order to gain this separation an abstract scene entity concept has been introduced.

The objective of this concept was to avoid class hierarchies that would have to be embedded into the kernel if the need arises to make this class hierarchy visible to any VR application. Another design requirement was that all scene components, like transformation, geometry, shader, physics, or sound attributes are loaded at run-time and therefore all classes in this class hierarchy would be abstract classes.

Instead, our scene description is very high level and scene elements managed through the kernel represent whole entities, like a chair or table as whole. With the fact that all scene components are loaded at run-time, these scene elements themselves are only containers storing all attributes needed to define some entity. In our terminology *attributes* are the building blocks of a scene element, like transformation, geometry, shader or physics attributes. In order to keep a VR application independent of a certain data plug-in, direct access to attributes describing a scene element must be prohibited. To realize this, the scene element acts as mediator to access the attributes. A special command object has to be used to specify the attributes' method with its parameters that, in turn, has to be called through the scene element. This ensures that data plug-ins can be exchanged. For example, if we want to replace the currently used scene graph only the plug-ins using

the scene graph have to be replaced, but no application has to be changed.

Figure 2 sketches this construct. The *Door knob* scene element is represented through the all enclosing rectangle, that stores all attributes (geometry, material, position, ...) specified in the top of the rectangle. The mediator requirement of the scene element is symbolized through the S/G (Set/Get) boxes. The virtual world managed via the kernel consists only out of scene elements. The management of the attributes is the scene elements' task. All entities in the virtual world sketched in figure 3 are therefore only scene elements.

These scene element attributes describe only the static appearance of an entity. However, a scene element may have a dynamic behaviour. This dynamic behaviour describes how the user can interact with the scene element and is called *functionality*. Like attributes, functionalities are user defined and not part of the kernel. Further on, functionalities are again not called directly, they are called through messages.

Often, functionalities are bound by constraints, so every functionality can hold one or more constraint function and only if all of them return true the request is processed. An example constraint is shown in figure 2.

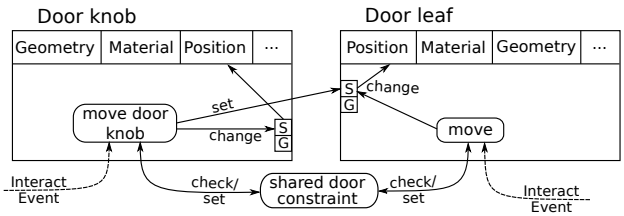

Fig. 2. Objects and functionalities needed to represent a door. The S/G boxes symbolize the mediators to call set/get-methods in order to access the attributes shown in the top of the objects.

The constraint needed here is to ensure the correct functionality of a door. Its task is to guarantee that the user can only open the door up to a certain position and that if the door is closed it has to be first unlocked using the door knob.

The *attendee* represents the user and is derived from *scene element*. As sketched in figure 3 the attendee is the linkage between the virtual and the real world. Consequently, an attendee knows all input devices associated with it and is responsible to create the sensual output for the user. For every input device a scene element can be added as a representation. The attendee's task is to query its input devices and to call the active or configured action. As the last call during the main loop cycle the attendee calls its sense renderers.

Putting the input devices and the sense renderers into the attendee is a logical choice. Especially if there are different attendees in a collaborative environment, the mapping of the input devices and the sense renderers would be spread over the different system components and thus violate our principle of locality.

The display configuration is done through a special plug-in, the display abstraction. This is an active component that knows its associated renderer and is responsible to call them. The output buffer is always an OpenGL framebuffer.

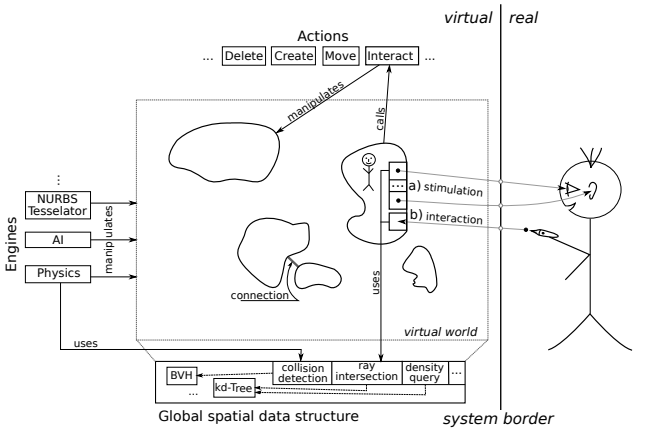

Fig. 3. Illustration of the coupling of the real and virtual world. The attendee is the central connection. It is responsible for the sensual stimulation of the user and the processing of user interaction. The blobs inside the virtual world represent scene elements.

IV. From VR to AR

Our goal by extending the VR framework with AR components was to create a framework that enables us to write applications that can be used as pure VR application or as AR application by changing the configuration, e.g. loading VR Input/Output devices or AR Input/Output devices.

Having a modular design, many extensions needed to enable AR with our framework can be implemented as independent plug-ins, but changes inside the kernel are unavoidable. Both will be described in the following paragraphs.

A. Requirements of AR

In order to work out the key differences between AR and VR we can use Azumas' [9] three key characteristics of AR. These are the combination of the real and virtual world, registration in 3D, and interactivity in real time. The first two characteristics represent the major differences to a virtual environment.

In order to merge the real and the virtual world the real world has to be captured. This can be done using a video-camera, e.g. a web cam or a head mounted display (HMD), either with video or optical see through. Registration, the alignment of the video coordinate frame with the virtual camera coordinate frame, is an essential component in such a system.

In VRs, registration is needed for aligning tracking devices working inside a local coordinate system and the virtual world. This registration is done inside the input device plug-in and does not affect other plug-ins. For the registration of the visual data the display abstraction needs an interface to set the view and projection matrix for the left and right eye since the stereoscopic matrix pairs are usually calculated inside the display abstraction using a user defined monoscopic view position and view plane.

In an AR setup the system often needs knowledge about the appearance of the real world in order to correctly place virtual objects into the real world's image. However, those "shadow" scene elements representing the real world must be hidden and

must not be rendered as color image but as depth image. This is needed to stitch the real and virtual world images together. In order to do so, the display abstraction needs to support rendering into different buffers and a mechanism to distinguish between hidden scene elements and scene elements that have to be rendered.

B. Input/Output devices

In AR two types of input devices can be distinguished. First the video input to capture the real world and second the tracking of the interaction devices or objects we are interested in. For the video input we can distinguish between monoscopic desktop AR using a web-cam for example and stereoscopic AR using HMDs. For every different video input device a new input device plug-in has to be created. The task of such an input device is to capture the actual frame and store the video data.

In order to be able to pass the received video data to a tracking or the compositing component a *Video* attribute was added. Data stored in the attribute is the video data and the image resolution. Since the video data is a central part of the rendering its appropriate storage is inside the attendee.

Further information the video input has to provide are the view and projection matrices, either for mono or stereo, that are needed for registration of the real with the virtual world. In case of an HMD, the LCDs are one or two separate screens and the rendering output will be directed to those screens. This configuration is part of the display configuration component.

C. Layers

In order to subdivide the scene into the two needed categories, scene elements and shadow scene elements of real objects, a layer concept as known from vector drawing applications is integrated into the kernel. Since the kernel by itself only knows the scene elements forming the scene, but no attributes, layers are not used to organize graphical output. They are more or less a generic concept of organization. Every scene element can be assigned to one or more layers and every system component, like engines or renderers, can request all objects of the desired layers. A consequence of the usage of the layer concept is being able to identify engines that can be executed in parallel. Since engines change the scene only engines without common scene elements can be run in parallel without risking a corrupted scene state.

A related layer concept is used by Kaufmann et al. [10] in their Construct3D application in order to separate the geometry of different users and the construction process into different layers. Their approach seems to be bound to the application and is only used on a visual level.

For an AR application two layers are needed. One *AR layer* stores the shadow scene elements and the other *VR layer* stores the scene elements that have to be integrated into the video stream of the real world. When using this application in a VR setup both layers need to be rendered into the color buffer to create an appropriate output.

These two layers as well as the initial assignment of the scene elements to the different layers are both created inside the AR application. However, these layers will be also used in the configuration file to direct the graphical rendering.

D. Display configuration

In order to combine the video stream and the virtual scene, different intermediate canvases are required next to the output OpenGL canvas. An intermediate canvas could either be a framebuffer object (FBO) or a texture or memory area which cannot be displayed immediately. Only an OpenGL canvas specifying the output canvas is displayed immediately. The specification of all canvases needed is done through configuration of the *display configuration* component.

Specifying the canvases is done by first creating the output OpenGL canvases. Then all intermediate canvases are derived from these output canvases. Using the above mentioned case with an AR and VR layer, two intermediate canvases are required. These canvases are represented by two FBOs. Every FBO stores the color and depth buffer of the associated layer.

Every OpenGL canvas can be associated with one or more virtual cameras. These cameras are also attached to the derived intermediate canvases. In order to make the canvases and cameras accessible to other components they are organized in so called *render targets*. One render target stores one canvas and one camera and is accessible through a unique name.

The following example shows a configuration for the case we need to merge real and virtual world using the depth buffer. *AR-RT* and *VR-RT* are the names of the auxiliary targets storing the intermediate canvases used for rendering.

```
Canvas "GL" (StereoType("SideBySide") Pos(0 0) ... )
Camera "Cam" ( Type("Stereo") ...
               EyeSeparation(5.0)
               Attendee("User"))
CreateRenderTarget    "Out"     ( Canvas("GL") Camera("Cam") )
CreateAuxiliaryTarget "AR-RT" ( Canvas("FBO") RenderTarget("Out") )
CreateAuxiliaryTarget "VR-RT" ( Canvas("FBO") RenderTarget("Out") )
```

Another major difference to the previous display abstraction is that the new one is passive. It is not called through the main loop. Instead, all renderer components that need one or more render targets actively query these render targets from the display abstraction. Thus this component was transformed into a passive administrator of data buffers, here for display output. This allows us to post-process the rendering data in an arbitrary way. However, the last processing component, writing all data into the OpenGL framebuffer, must be aware of this and do the buffer swap.

These major changes resulted in a complete redesign of the display component of the system.

E. Data storage and Graphics-Rendering

Since the layer concept is a central part of the kernel, data and rendering plug-ins must be adapted to this concept. A scene data representation that encapsulates a third-party scene graph, has to store every layer in an individual scene graph. This enables the renderer to draw the different layers independently from each other.

Converting the display configuration into a passive component led to two new tasks every graphics renderer has to

accomplish: Distinguish between mono and stereoscopic rendering as well as activating the render target before rendering and finalizing the render target after rendering. In case of an OpenGL canvas this results in calling the makeCurrent and the swapBuffers function.

For the scene data plug-ins no additional configurations are needed. In case of the graphics renderer a render target has to be joined with the layers that have to be rendered into this target.

Using the basic configuration from above the renderer would render all scene elements of the AR layer into the *AR-RT* render target and the VR layer into the *VR-RT* render target. Both render targets are specified in the display configuration as auxiliary targets. An example of such a configuration is shown below wherein *AR-Layer* and *VR-Layer* are the layer names defined in the application.

RenderTarget "AR-RT" ("AR-Layer")
RenderTarget "VR-RT" ("VR-Layer")

The configuration for a VR setup may look like this:

RenderTarget "Out" ("AR-Layer" "VR-Layer")

In the VR case no intermediate canvases are needed. The rendering is directly done in the OpenGL canvas and both layers are rendered.

The OpenGL renderer and the associated data component used is not based on a third party scene graph. It supports generic GLSL vertex and fragment shaders as well as geometry shaders. This feature will be used by the example application in order to create the triangle geometry needed.

F. Combiner

The *Combiner* component's task is to merge the video stream with the virtual scene. Consequently, it has to be called after the graphics renderer. Using the basic AR requirements from above the combiner has to compare the AR and VR depth buffer pixel by pixel and copy the color component associated with the nearest depth component into the output buffer. However, if the requirements change a new combiner component has to be implemented.

As a consequence a generic combiner was implemented that allows to process arbitrary input data buffers and write the output to a specified output buffer. The input data can be an FBO or memory area. An output buffer can be either an FBO or OpenGL canvas. The configuration allows also to specify an arbitrary GLSL fragment shader as program and define input textures, an arbitrary parameter set, and the output buffer. Input textures can be all FBO textures or scene elements with a video stream attribute. Input textures can be bound to any GLSL uniform sampler2D variable. At the moment, the parameter set can be a collection of float and integer parameters.

Processing is done by an orthographic projection of a textured OpenGL quad in such a way that it covers the whole viewplane. Since the input textures are render targets or a video stream their size is equal to the viewplane and therefore every texel will be processed by the specified GLSL fragment shader.

The following configuration is an example for the setup used in the earlier paragraphs. *DepthMerge* is the GLSL shader that will be used. It compares both depth values texel by texel and sets the color value associated to the nearest depth texel.

```
DepthMerge ( Input( (FBO("AR-RT" Depth) "Texture1")
                    (SceneElement("User") "Texture2")
                    (FBO("VR-RT" Depth) "Texture3")
                    (FBO("VR-RT" Color) "Texture4") )
             Output(Canvas("Out")))
```

Inside the *Input* section all input textures are specified. *SceneElement("User")* adresses a scene element that has to store a Video attribute. In this case it is the attendee and the video data is copied to a texture every frame. *Output* specifies the render target the result has to be written into. Here, it is the OpenGL render target *Out*. However auxiliary render targets can be used also allowing to build a post processing pipeline.

V. Results: Example Application

The example application that already uses the developed VR/AR framework has been presented at ISMAR 2011 [11]. However, the focus of the poster was a non technical description of the application itself. Here, the focus lies on describing the technical details of the application and the integration into the developed VR/AR framework.

The aim of this application is to augment or display magnetic field lines either using an HMD in an AR setup or a mono/stereo display in a VR setup. Its motivation is to give pupils a better understanding of invisible physical forces leading to a better model-building, since there exists a large barely comprehensible gap between the experiment and its theoretical background for a lot of pupils.

Our objective is that augmented/virtual reality techniques can help to narrow this gap. Magnetism as physical concept has been chosen as the initial experiment to gain knowledge about the usefulness of our approach to augment real physical experiments or to display the experiment in an virtual environment while interacting with the physically existing experiment setup.

A. Hardware setup

In case of the magnetic field experiment two optically tracked magnets are used as 6-DOF input devices. Using real magnets provides us with natural haptic feedback allowing the student to move the magnets freely while feeling the magnetic field forces and simultaneously seeing the changing magnetic field lines as a model for the magnetic field. This gives a direct visual cue of the perceived haptic feedback and should help to understand the concept of magnetism.

1) AR setup: The first hardware setup as seen in figure 4 uses a stereo video see-through HMD with built-in optical marker tracking. The HMD consists of two OLED microdisplays with near eye optics and a camera with four pixel-synchronous image sensors (2 color, 2 b/w infrared sensitive).

The two color camera sensors are located near the center of the HMD and are used for the stereo video see-through. The two b/w sensors are located at the outer ends of the HMD and are used for tracking. To minimize latency and to avoid jitter,

Fig. 4. AR setup using a video see-through HMD with included tracking and real magnets as interaction device. The separate monitor displays the image generated for the user's left eye.

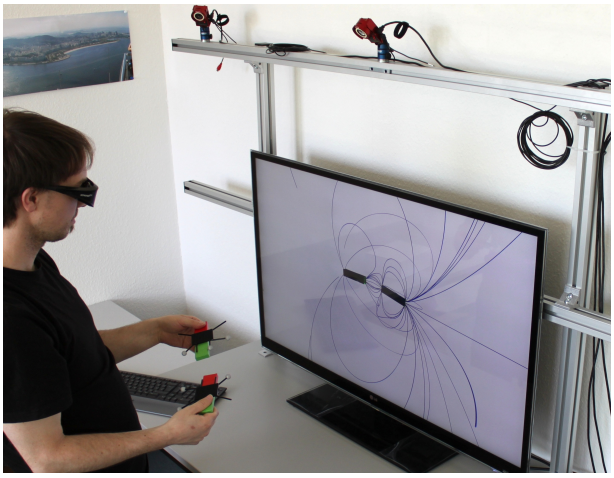

Fig. 5. VR setup using an active stereo LCD-TV, optical tracking and real magnets with tracking markers as interaction devices.

the image acquisition is synchronized with the refresh rate of the OLED displays.

In the prototypical tracking system presented here, the tracking markers are integrated into the real magnets. They contain 3 IR-LEDs each. The LEDs are arranged in well distinguishable patterns. One advantage of this setup is that no external cameras are needed. Since the tracking cameras are in a fixed position with respect to the see-through cameras, tracking of the user's head is not necessary. The field of view of the video see-through sensors is always contained in the tracking volume.

The HMDs-API provides functions to query the color images, the view and projection matrices for left and right eye as well as the position and orientation of the tracking markers. All data is queried in one input device plug-in. However, no data processing is done in this plug-in. Its only task is to retrieve all data and pass it to the configured processing units allowing an application independent usage of the input device. The two OLED microdisplays are addressed as a single display by the operating system and they can be directly addressed through the display configuration.

2) VR setup: The second hardware setup is a VR setup. The display is an active stereo LCD-TV. For optical tracking a three camera setup using the Natural-Point OptiTrack system is used. As interaction devices magnets extended with tracking markers are used. The tracking system is directly mounted on the back of the TV-Screen using aluminium profiles. Figure 5 shows the setup in use.

All tracking data is sent to the application using VRPN [12]. Since the Natural-Point tracking software only runs under Microsoft Windows and the MagSim application under Linux a virtual machine with Microsoft Windows has been installed on the same computer to avoid a two computer setup. No problems occurred while running the tracking software inside the virtual machine. On the framework side a generic VRPN input device plug-in already existed and redirecting the received input data was done through configuration.

In both cases (VRPN and HMD input device) the tracking data is sent to a special action that sets both virtual magnets to the new tracking positions. In figure 6 this action is represented by *Interaction*.

B. Application MagSim

The MagSim application consists of one short application source, that creates the two used layers and the two magnets that represent the real magnets as shadow scene elements. They are used to generate a depth image for the real existing magnets and for the magnetic field simulation. Their depth image will be used by the Combiner to merge the real and virtual world to resolve occlusion by the real magnets and the magnetic field lines. The second part is a configuration file specifying and configuring the plug-ins the application has to load. Figure 6 shows the used plug-ins for the AR setup and the work flow between the different components. The dashed line shows the calling order of the components and the solid lines show the data flow between the different components. The calls (1) and (3) are triggered through the systems main-loop. Plug-ins components are drawn with a thick border. The only component that is more or less application specific is the *Magnetic Field Simulation* plug-in. It uses the two magnets specified in the AR layer to calculate the superimposed magnetic field of both magnets. The resulting field lines are stored as line strips. They are transformed into triangle meshes and are colored using a GLSL geometry shader associated as attribute to the *Magnetic Field* scene element.

1) Real-time Magnetic Field Simulation: The magnetic field of a steady electric current can be calculated using the law of Biot-Savart [13]:

$$\mathbf{B}(P) = \int \frac{\mu_0}{4\pi} \frac{Id\mathbf{l} \times \mathbf{r}}{|r|^3} \tag{1}$$

with

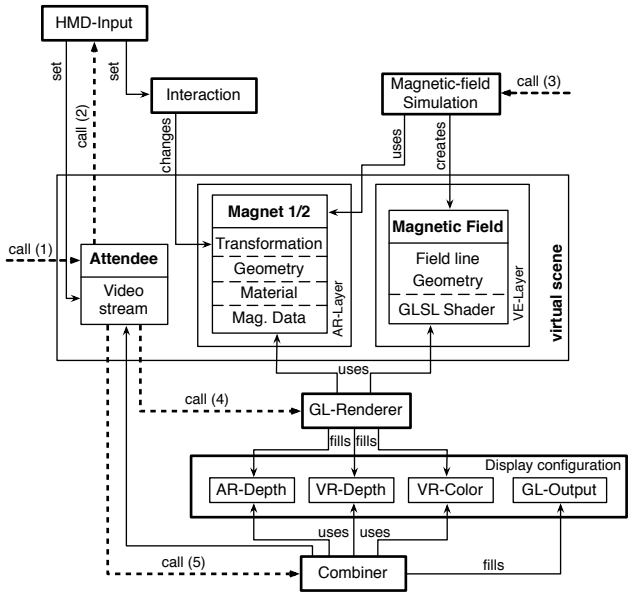

Fig. 6. Work-flow of the application using the AR setup. Dotted lines indicate system calls and the solid lines show the internal "calls" beetween the used plug-ins and the virtual scene. Plug-ins are drawn with thick border. All attributes (Video stream, Geometry, ...) are also part of one or more plug-ins.

$\mathbf{B}(P)$	the magnetic field at point P
P	a given point, at which the field is to be computed
I	the current
$d\mathbf{l}$	the differential element of the wire
$\mathbf{r}$	the vector from the wire element $d\mathbf{l}$ to the point P
μ_0	the magnetic constant

This law describes the direction as well as the strength of a magnetic field at a given position P for an electric circuit of an arbitrary, infinitely narrow wire. The resulting vector is integrated over the complete wire. The superposition-principle of magnetic fields does allow to integrate respectively accumulate the field vectors to get a single result at a certain point. By defining a wire of a coil, an electromagnet can be simulated. A geometrically equivalent permanent magnet possesses the same magnetic field outside the magnet as the solenoid, so we can use Biot-Savar's law to efficiently simulate the magnetic field. In our final AR-setup the interior of the magnet is not visible, therefore there is no need for a proper simulation of that part of the field.

In our case, cuboid-shaped permanent magnets are used for interaction. Their surface can be approximated by plenty straight wires (see fig. 7). The magnetic field of each of these wires is, without the need of integration, directly computable by (from [14]):

$$\mathbf{B}(P) = \mathbf{e_H}\frac{\mathbf{I}}{4\pi * a}(\cos\beta_1 - \cos\beta_2) \qquad (2)$$

with:

$\mathbf{e_H}$	a unit vector, defined by $\frac{\mathbf{PA}\times\mathbf{PB}}{\lvert\mathbf{PA}\rvert*\lvert\mathbf{PB}\rvert}$, which is perpendicular to the plane spanned by P and the two endpoints A, B of the wire.
$\mathbf{I}$	the current
a	the projection of P to the vector $\mathbf{BA}$ defining the wire.
β_1, β_2	angles between $\mathbf{AB}$ (the wire) and $\mathbf{PB}$

This formula is derived from equation 1 and describes the magnetic field of a straight wire of finite length. Only the start- and endpoint of the wire, the current and the Position P in space is needed. An electric current is only possible in a closed circuit, so simulating only one straight wire makes no sense. Instead, four straight wires are used, one for each side of the simulated magnet, forming a closed loop. A stack of these wires defines the complete magnet. The more wire-layers are used, the more accurate the resulting magnetic field will be (see fig. 7). The resulting magnetic field is easily computed by accumulating the field vectors of the calculation of the single wires.

Since we have an interactive environment, the magnetic fields of all involved magnets have to be computed many times per second. To achieve real time frame rates, the computation is done using modern graphics hardware (GPU). The problem fits perfectly to this architecture, because identical calculations ("Single Instruction") need to be done for many different points in space and many different wires defining the magnets ("Multiple Data"). By using this SIMD-Principle, many computations can be done in parallel, perfectly taking advantage of the many cores of modern graphics cards.

The simulation is implemented in CUDA [15]. In CUDA, so called "blocks" are processed, which contain a number of threads. 32 of these threads, a so called "warp", always share a program counter, meaning they are processed in parallel in one computational unit. To achieve the best hardware utilisation, a block should therefore contain a multiple of 32 threads.

The magnets we used so far all have a rectangular base area. Thus a stack of eight wire-circuits and four wires for the four side does result in 32 identical computations with individual data, perfectly suitable for 32 threads. In the simplest configuration, computing a grid of vectors of the magnetic field, one block is created for computing each a single point of the grid, containing 32 threads which iterate over all the given magnets respectively their eight-fold-stacks of four-wire-loops. By defining a grid of N*N*N, the same amount of blocks is created and processed by the GPU.

One thread of each warp collects all the results of the other 31 threads using shared memory, accumulates the vectors to a single field direction and strength, and writes the result to main memory.

Calculating Field Lines: Since the the magnetic field's calculation result is a vector, a vector field can be derived. The field vectors can be interpreted as tangents of the field lines. By concatenating field vectors, field lines can be calculated. Since the tangent of a function is derived from the gradient, this function can be expressed as a first-order ordinary differential equation (ODE). The problem of calculating the field line is an initial value problem with a (random) starting point $\mathbf{P}$ in space as initial value and the following positions as the result

of the integration of this function. Common solving methods or approximations can be applied, using for example the Euler method or the more sophisticated Runge-Kutta integration schemes. Fig. 7 show the different results using euler and rk4, with different numbers of circuit stacks.

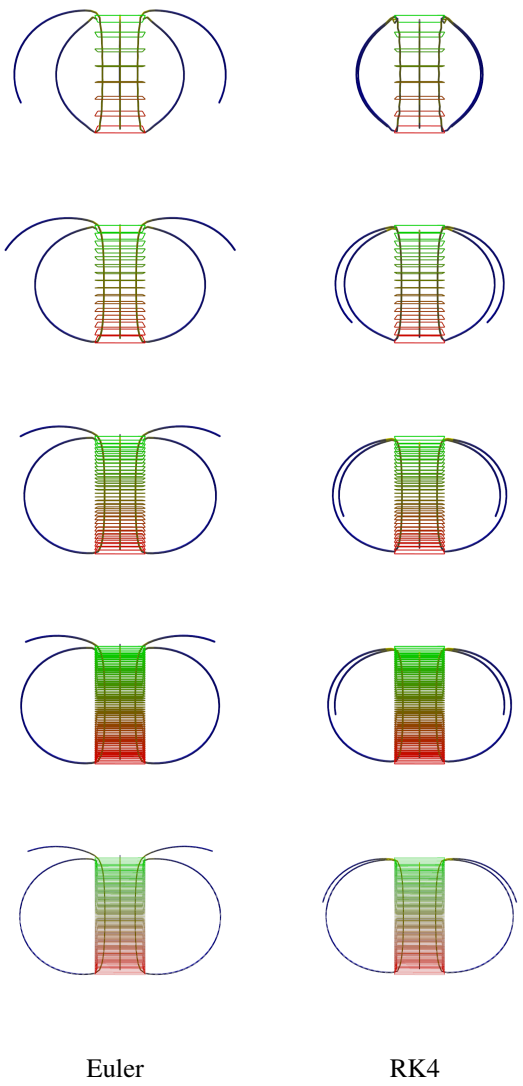

Fig. 7. Comparison of Euler and Runge-Kutta integration methods. The magnets are created of 8, 16, 32, 64 and 128 wire-circuits (f.t.t.b.), to show the different calculation accuracies.

Euler's method method is the simplest calculation scheme. It uses, in this case, the vector $\mathbf{V}$ derived from the magnetic field calculation, scales it to a predefined length h and ads this vector $\mathbf{V}$ to the starting point $\mathbf{P_1}$ to define the starting point $\mathbf{P_2}$ for the next iteration. This process is repeated until a certain point or length of the resulting polyline is reached. Having a completely linear function, the result would be absolutely correct. In most cases however this simple method leads to a large error, since curves cannot be followed exactly because a vector is always straight. The accuracy thus depends on the amount of h, the scaling factor or "step size", which has to be very small to get a reasonable accurate result.

Around 1900 C. Runge and M.W. Kutta developed a better method to approximate an ODE, based on the idea of using not only one sample per iteration but sampling also the neighbourhood of the starting Point $\mathbf{P}$ by using derivatives resp. vectors derived from different weighted Euler-Steps starting all from Point $\mathbf{P}$. Then a weighted sum of these samples is used as the resulting vector $\mathbf{V}$. Many different types of Runge-Kutta-methods do exist, a very common one is the RK4-method, using four samples per step. The error per step is on the order of $O(h^5)$ and thus much better than Euler's method, which is on the order of $O(h^2)$.

The "Dormand-Prince" (Dopr853) method, taken from [16], also is a Runge-Kutta method which uses 12 function evaluations (here the field vectors), instead of four in the RK4-method. The error of this method is on the order of $O(h^8)$, making it even more accurate. Additionally it is a so called "embedded method", meaning that, by combining the 12 vectors with different constants, two results of a different order can be calculated. This is used in conjunction with an adaptive step size control. By giving a predefined tolerance, the size of the next step is increased/decreased to keep the field line's error within this tolerance. The version from [16], which is given as source code and which we used as a basis for our CUDA implementation, is written for "usual" functions $f(x, y)$. The mentioned tolerance does consist of an absolute and a relative part, scaled by the value of y:

$$tolerance = absTolerance + |y| * relTolerance \qquad (3)$$

This is done to prevent the integration from trying to keep the error very small, even if the value of y is very large. In our case, y is the position in space, thus not being a valid scale factor for the tolerance. Using other values as a scale factor were not successful, we tried the normalized field strength and a measure of the curvature of the line.

Fig. 8 shows a comparison of both the RK4 and the Dopr853 integration scheme. The error of the Dopr853 implementation is larger than the RK4 implementation, as is especially visible in the interior of the magnets. The lack of a valid scale factor possibly causes this problem, but this has to be evaluated further.

2) Configuration of the MagSim Application: The configuration of the plug-ins used for the MagSim Application is identical to the example configurations introduced above. All relevant parts of the AR case can be seen in figure 9 and in figure 10 for the VR case.

For both hardware setups the MagSim application source code is identical. The only difference is the configuration. Each configuration is stored in a text file and is parsed at application start. The configuration describes what plug-ins have to be loaded at run-time and the configuration parameters of every plug-in. Every plug-in specification and configuration is encapsulated by *Extension* and *!Extension* tags and the name behind Extension is the plug-ins name.

The differences between both configurations is, that in the VR setup no Combiner plug-in is needed and that the HMD input is replaced using a generic VRPN input device plug-in for 6 DOF input. Next to the different plug-ins loaded the display description and the renderer configuration is different,

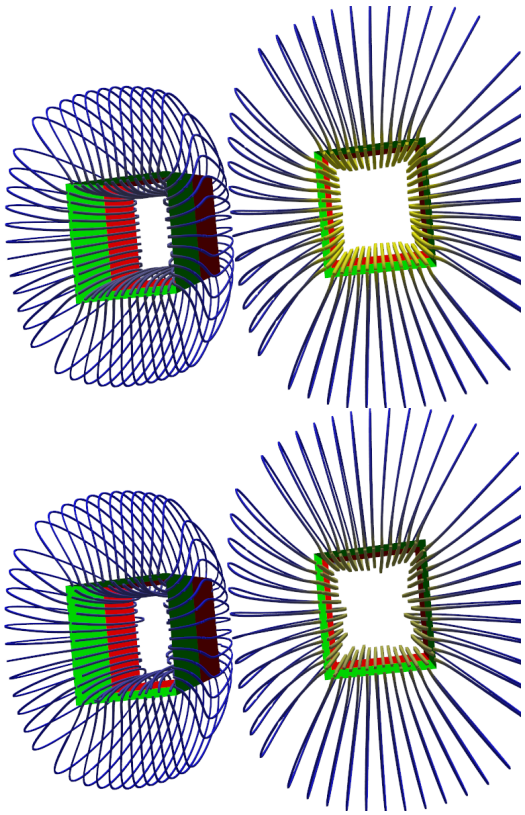

Fig. 8. A direct comparison of the RK4 method (top) and the Dopr853 method (bottom). Especially the field lines inside the magnets are more accurate for RK4, where Dopr853 should be much better.

```
Extension DisplayConfiguration
  Canvas "GL" (StereoType("SideBySide") Pos(0 0) Size(800 600 )
  Camera "Cam" ( Type("Stereo") ...
                 EyeSeparation(5.0)
                 Attendee("User"))
  CreateRenderTarget    "Out"       ( Canvas("GL")
                                      Camera("Cam") )
  CreateAuxiliaryTarget "AR-RT" ( Canvas("FBO")
                                  RenderTarget("Out") )
  CreateAuxiliaryTarget "VR-RT" ( Canvas("FBO")
                                  RenderTarget("Out") )
!Extension

Extension OpenGLRenderer
  RenderTarget "AR-RT" ( "AR-Layer" )
  RenderTarget "VR-RT" ( "VR-Layer")
!Extension

Extension Combiner
  DepthMerge ( Input( (FBO("AR-RT" Depth) "Texture1")
                      (SceneElement("User") "Texture2")
                      (FBO("VR-RT" Depth) "Texture3")
                      (FBO("VR-RT" Color) "Texture4") )
               Output(Canvas("Out")))
!Extension

Extension SceneData
!Extension

Extension HMDInput
!Extension

Extension MagneticFieldSimulation
!Extension
  ...
```

Fig. 9. Part of the AR configuration file used to load and configure all needed plug-ins.

```
Extension DisplayConfiguration
  Canvas "GL" (StereoType("SideBySide") Pos(0 0) Size(960 1050 )
  Camera "Cam" ( Type("Stereo") ...
                 EyeSeparation(5.0)
                 Attendee("User"))
  CreateRenderTarget    "Out" ( Canvas("GL") Camera("Cam") )
!Extension

Extension OpenGLRenderer
  RenderTarget "Out" ( "VR-Layer" "AR-Layer" )
!Extension

Extension SceneData
!Extension

Extension VRPNInput
!Extension

Extension MagneticFieldSimulation
!Extension
  ...
```

Fig. 10. Part of the VR configuration file used to load and configure all needed plug-ins.

since in the VR setup the magnets and the magnetic field visualisation has to be rendered onto one screen as seen in figure 5. As seen in the configuration file the OpenGLRenderer renders the VR-Layer and the AR-Layer directly into the output OpenGL canvas.

C. User workflow

Since experimenting with the magnetic field is a single user task, no collaboration is needed. In order to let others participate in case of the AR setup, the left eyes image is copied to an additional GL canvas as shown in figure 4. All other pupils can now observe the visual impressions of the user who wears the HMD.

However, in both setups the user interacts with the system by moving the magnets and observing the change in the magnetic field while receiving direct haptic response from the magnets. The visualized magnetic field gives the pupils a direct visual cue of the perceived haptic feedback and should help to understand the concept of magnetism.

For the MagSim application the user directly interacts with the virtual world using the magnets as tangible input devices without the need to select some object to interact with. This is desirable since there are no interaction metaphors to learn.

VI. Future Work

We would like to undertake more projects with our newly established VR/AR software to be able to evaluate it in more detail. It was noted, that the process of registering real and virtual objects involves many steps, like camera and HMD calibration and modelling 3D objects. As is, it is an iterative, trial-and-error process. This should be handled in a structured and integrated way by adding tools that support the developer.

Since smart phones have an ever increasing potential as AR devices, we would like to port our software to powerful cell phone platforms.

ACKNOWLEDGMENT

This work was partially funded by the German Federal Ministry of Economics and Technology (BMWi), Zentrales Innovationsprogramm Mittelstand (ZIM) (grant number KF2644102) and the Ministry for Innovation, Science, Research and Technology of the the state of North Rhine-Westphalia.

REFERENCES

[1] Dassault Systems, *3DVIA*, 2012 (accessed March 16, 2012). [Online]. Available: http://www.3dvia.com

[2] Metaio, *MobileSDK*, 2012 (accessed March 16, 2012). [Online]. Available: http://www.metaio.com/software/mobile-sdk/

[3] N. Aggarwal and A. Garg, "Article:using re-usable, secure software engineering principles for designing user focused mixed reality systems," *International Journal of Computer Applications*, vol. 32, no. 8, pp. 55–60, October 2011, published by Foundation of Computer Science, New York, USA.

[4] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner, "Design of a component-based augmented reality framework," in *Augmented Reality, 2001. Proceedings. IEEE and ACM International Symposium on*, 2001, pp. 45 –54.

[5] P. Grimm, M. Haller, V. Paelke, S. Reinhold, C. Reimann, and R. Zauner, "Amire - authoring mixed reality," in *Augmented Reality Toolkit, The First IEEE International Workshop*, 2002, p. 2 pp.

[6] M. Ponder, G. Papagiannakis, T. Molet, N. Magnenat-Thalmann, and D. Thalmann, "Vhd++ development framework: towards extendible, component based vr/ar simulation engine featuring advanced virtual character technologies," in *Computer Graphics International, 2003. Proceedings*, july 2003, pp. 96 – 104.

[7] J. Ohlenburg, I. Herbst, I. Lindt, T. Fröhlich, and W. Broll, "The morgan framework: enabling dynamic multi-user ar and vr projects," in *Proceedings of the ACM symposium on Virtual reality software and technology*, ser. VRST '04. New York, NY, USA: ACM, 2004, pp. 166–169. [Online]. Available: http://doi.acm.org/10.1145/1077534.1077568

[8] S. Irawati, S. Ahn, J. Kim, and H. Ko, "Varu framework: Enabling rapid prototyping of vr, ar and ubiquitous applications," in *Virtual Reality Conference, 2008. VR '08. IEEE*, march 2008, pp. 201 –208.

[9] R. T. Azuma, "A Survey of Augmented Reality," *Presence*, vol. 6, pp. 355–385, 1997.

[10] H. Kaufmann and D. Schmalstieg, "Mathematics and geometry education with collaborative augmented reality," in *ACM SIGGRAPH 2002 conference abstracts and applications*, ser. SIGGRAPH '02. New York, NY, USA: ACM, 2002, pp. 37–41.

[11] F. Mannuß, J. Rübel, C. Wagner, F. Bingel, and A. Hinkenjann, "Augmenting Magnetic Field Lines for School Experiments," in *International Symposium on Mixed and Augmented Reality (ISMAR 11)*, 2011.

[12] R. M. Taylor, II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser, "VRPN: a device-independent, network-transparent VR peripheral system," in *Proceedings of the ACM symposium on Virtual reality software and technology*, ser. VRST '01. New York, NY, USA: ACM, 2001, pp. 55–61.

[13] W. Demtröder, *Experimentalphysik*, ser. Springer-Lehrbuch. Springer, 2006.

[14] M. Marinescu, *Elektrische und magnetische Felder: Eine praxisorientierte Einführung*, ser. Springer-Lehrbuch. Springer, 2009.

[15] (2011) nVidia Cuda. http://www.nvidia.com/object/cuda_home_new.html.

[16] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. New York, NY, USA: Cambridge University Press, 2007.