A Component Based Architecture for Distributed, Pervasive Gaming Applications

Carsten Magerkurth
Fraunhofer IPSI
AMBIENTE
Dolivostraße 15
64293 Darmstadt, Germany
magerkurth@ipsi.fhg.de

Timo Engelke
Fraunhofer IPSI
AMBIENTE
Dolivostraße 15
64293 Darmstadt, Germany
engelke@ipsi.fhg.de

Dan Grollman
Brown University
115 Waterman Street
Providence, RI 02912
USA
dang@cs.brown.edu

ABSTRACT

In this paper, we describe a component based architecture for developing distributed, pervasive games that integrate tangible and graphical user interface components. We first discuss some of the interface components we have developed and then present a coordination infrastructure called Pegasus that allows flexibly coupling and reconfiguring components during runtime. On top of Pegasus we have created a language for describing pervasive games called DHG and briefly present a first sample application that is based on the component based architecture and defined with the game description language DHG.

Categories and Subject Descriptors

H.5.1 [**Information Interfaces and Presentations**]: Multimedia Information Systems

General Terms

Management, Design, Human Factors, Languages.

Keywords

tangible interfaces, TUI, pervasive games, hybrid environments, tabletop games, computer games, entertainment.

1. INTRODUCTION

There is a growing trend in the computer gaming research community to augment traditional video games with aspects from the real world, e.g. [1][2][7][13][15]. These hybrid or pervasive games combine the virtual nature of traditional video games with physical and social context, thus creating immersive gaming experiences that pervade the boundaries of virtual, physical and social domains [13].

We have been active in the field of pervasive tabletop games that provide tangible interfaces borrowing interaction techniques from traditional board games (see fig. 1). The central idea is to combine the advantageous elements of traditional board games and computer entertainment technologies [8]. Board games emphasize the direct interaction between human players. They sit together around the same table, facing each other at an intimate distance. Their close face-to-face interaction integrates discussion, laughing, and all kinds of non-verbal communication hints creating a rich social situation.

On the other hand, playing computer games introduces many highly interesting possibilities that enhance the playing experience. Game presentation is enriched with audio and visual support and game content is limited only by the imagination of the developers. The drawback of computer games, however, is the lack of social interaction in a face-to-face setting, since the players' notion of each other is mostly conveyed by screen and keyboard.

Figure 1: A Pervasive Tabletop Game

Consequently, we aim at combining the human centered group situation of tabletop games with the advantages of pervasive computing technology and thereby realize new and engaging gaming experiences. For the remainder of the paper, we focus on pervasive games that comprise the following properties:

1.) Tangible and physical interfaces such as game boards, playing pieces, wands, etc. are available in addition to graphical displays.

2.) Gameplay is mostly turn based, since fast-paced real-time gaming styles contradict with creating a strong social situation and focus attention to graphical interfaces instead of human players.

3.) Interaction with the virtual parts of the game is distributed among multiple devices such as PDAs, large public displays, physical controllers, etc., from which some are public and some

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACE 06, June 14-16, 2006, Hollywood, California, USA.
Copyright 2006 ACM 1-59593-380-8 /06/0006 ...$5.00.

are private. The exact setup of the players' computing and interaction devices might not be known to the developer beforehand.

1.1 Developing Pervasive Games

One of the most crucial points and challenge for the success of pervasive games is the provision of appropriate interfaces that stimulate the direct interaction between human players and with the information that is relevant in the context of the given task. The goal is to provide the players with a set of interaction devices that support pervasive gaming applications appropriately with unobtrusive and intuitive interfaces fostering direct communication. Obviously, the development of hybrid gaming applications involves the integration of physical, social and virtual aspects, thus introducing a new dimension of complexity to the entertainment applications of the pre-pervasive computing era. Hence, it is crucial to be able to quickly prototype and tweak game rules as well as the involved user interfaces and interaction devices.

2. USER INTERFACE COMPONENTS

As [8] note, the design of physical interfaces with high-fidelity functioning prototypes is prohibitively expensive, if not supported by a dedicated user interface toolkit that allows for flexibly interchanging interface components. In the next sections such interface components and the appropriate interaction designs are discussed. The entirety of these devices can be flexibly integrated to any pervasive gaming application based on the Pegasus coordination infrastructure that is discussed later. The interaction devices presented here hence form the building blocks from which pervasive game applications can be developed.

2.1 Tangible Game Boards

The interaction with tangible game boards is a prototypical example for the spatial approach that [15] identify as the primary TUI approach in which artifacts are directly interpreted and augmented by a virtual application, not involving any additional layer of indirection.

Figure 2: Physical (left) and Virtual (right) Game Boards

Accordingly, the game board is a tangible interface that seamlessly integrates representations and controls and is thus preferable to graphical user interfaces in which spatial relationships are controlled in a different way (via the mouse) than they are represented. Tangible game boards as the primary input devices for pervasive computing board games are already discussed in [13] along with possible enabling technologies (magnetic fields, RFID, computer vision, etc) and are not further elaborated on here. Figure 2 (left) shows a tangible game board, whereas figure 2 (right) shows a GUI realization of the game board. Depending on the availability of a physical device, the same application might utilize the physical or the virtual (GUI-) version of the respective interface.

2.2 The Gesture Based Interaction Device

Nowadays, mice and keyboards are the common interaction devices for most tasks involving computer systems, although several application areas such as simulations or computer games utilize specialized devices such as steering wheels, joysticks, or gamepads. The interaction devices dedicated to traditional computer entertainment applications normally do not link between the physical and the virtual world in a way that mastering the interface itself is a central and demanding component. Hence, real world skills gained by training the interface are not necessary to achieve an effect in the virtual domain of the game application. However, the tremendous success of the Eye-Toy [14], a camera addon for the Sony Playstation that integrates the filmed shape of the player as an action object in the respective video games, is a clear indication that dedicated physical interfaces involving real world skills are an important future trend for pervasive games.

Figure 3: The Gesture Based Interaction Device

The gesture based interaction device is an interaction device to allow for novel forms of human computer interaction. It follows the approach of linking the exertion of a physical skill to a respective effect in a virtual application. It consists of a stick augmented with an accelerometer in its head that can be swung in a similar way as a conductor's baton or a magic wand (ref. [5]). It picks up and digitally converts the radial movements of the stick to discrete acceleration measures. There are three possible operating modes of the device, each relating to different usage scenarios and applications.

2.2.1 Gesture Recognition

As illustrated in Figure 3 (left), the primary operating mode of the device is gesture recognition. In a typical pervasive gaming application, the device translates and maps the real world qualities of the user's gestures to a virtual representation that has a certain effect in the game. The more accurate a player is able to perform a set of given gestures, the more successful is his outcome in the virtual world. Applied to a fantasy role playing scenario, one could for instance imagine several different magic spells that each have unique effects and are more or less hard to perform. A simple spell might turn the light on or off and consist of two easy gestures (e.g. a circle or a straight line) to be performed consecutively by the spell caster using her magic wand. A more

powerful spell might eradicate all life in a 50 yards diameter and require ten more difficult gestures such as certain characters or words from the alphabet to be performed in a sequence (possibly even in a certain amount of time). Hence, the application of the device in the gesture recognition mode ("wand" mode) can match the real world skills in mastering gestures of varying difficulty to a respective virtual proficiency. A computer game wizard is no longer mighty due to some numbers stored in her character database, but because of physical skills acquired by real experiences.

2.2.2 *Intensity Measurement*

The gesture recognition mode works by matching the features of a set of stored gestures to the current incoming stream of data from the device. Another way of using the gesture based interaction device is by regarding the magnitude of the raw sensor data in order to measure the force of the swinging. By doing so, it is possible to create pervasive games that use the gesture based interaction device like a hammer or a sword instead of a wand or a conductor's baton.

2.2.3 *Pointing*

The final operating mode requires an RFID antenna built into the device's head. As illustrated in Figure 3 (center) one can equip arbitrary artifacts with RFID tags that can be read and unambiguously identified by the antenna's head as shown in Figure 3 (right). This allows for unobtrusive multimodal interaction styles that follow Bolt's paradigm [2] by naturally specifying source and/ or target artifact of an arbitrary action.

For pervasive tabletop games that are in the domain of role playing games (such as the Caves & Creatures application discussed later), especially the capability of mapping the real-world skill of operating the physical device with the virtual effects of casting magic spells is an interesting feature that showcases the interaction between virtual and physical domains in pervasive games.

2.3 The Smart Dicebox

In the physical world, dice are important components of a wide range of games. They are used for creating variations in the game flow. By rolling dice, an element of chance is introduced to an otherwise static and deterministic flow of game actions. The chance in the dice, however, is not equal to the generation of a random number. Rolling dice involves both a physical act and skill (some people and some dice roll better results than others) as well as a social mechanism to supervise and control the physical act, because cheating is a common phenomenon associated with this particular way of adding variability to games. Hence, rolling dice is a very interesting example of a gaming interface that spans virtual, physical, and social domains and is thus particularly suitable for a computer augmented realization.

In order not to lose the physical and social aspects of rolling dice by simply creating random numbers in a computer application, we tried to preserve the multi-faceted nature of dice-rolling in our computer adaptation. Due to the size and feasibility problems associated with augmenting individual dice with respective sensor technology, we integrated multiple dice into one single smart artifact, the Smart Dicebox (see fig. 4).

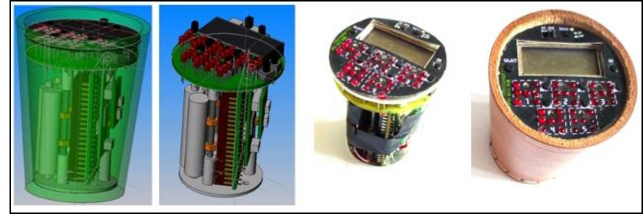

Figure 4: The Smart Dicebox

The augmentation of a dice box allows for utilizing a physical manipulation technique (shaking) that influences the virtual outcome; a dice box is also a well-known interaction device for games that players are used to. Of course, a natural drawback of the approach is that a dice box is not identical to rolling physical dice and some players might not be used to using a dice box, although with games requiring multiple dice to be rolled simultaneously (such as Yahtzee or several role-playing or tabletop conflict games) it is common to use such a device, in fact, most editions of Yahtzee are shipped with dice boxes.

2.3.1 *Interaction Design*

The interaction was designed to be as similar to a traditional dice box as possible. To generate random numbers, the device is lifted, shaken, put on a plain surface upside down, and then finally lifted again to see the results. However, in contrast to traditional dice, the sum of the spots is not counted from the physical dice after being tossed on the surface of the table. Instead, the spots are displayed via light emitting diodes (LEDs) on the surface of the dice box top.

Shaking the device also emits a sound mimicking the sound of shaking a traditional dice box, although the integrated sound hardware does hardly deliver sound of acceptable quality. Since the smart dice box is capable of communicating with the environment via radio transmission, it is more preferable to let another sound source outside the device perform the respective audio output.

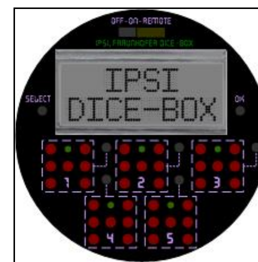

Figure 5: The Dicebox Interface

In addition to the basic interface of shaking, dropping and turning the device, there is also a conventional button interface with a graphical display integrated in the top surface of the dice box as shown in Figure 5. The button interface is used for advanced configuration of the device when no other, more sophisticated interface such as a respective GUI application (running on a nearby PC) is available or when single dice are to be "held" or "released", i.e. when they are to be included or excluded from tossing. Each of the five dice displayed on the surface of the dice box consists of seven red LEDs that represent the spots of the respective die. Whenever the device is shaken, the respective light

patterns change in accordance to the tossed result. A small green LED is used to indicate whether the respective die is held or released, i.e. if its face changes when the device is shaken. To toggle between holding and releasing a die, a small button is associated with each die that turns the respective green LED on or off with each press, thus ensuring an intuitive way of changing the individual held states by providing visual feedback and adhering to the Gestalt law of proximity regarding the button layout.

The device can be turned off and on and put into a remote controlled mode with a respective three-state-switch above the graphical display. The remote controlled mode allows the device to be controlled from the environment via radio transmission as discussed in the following section.

2.3.2 Operating Modes

The Smart Dice Box was designed to function both as an autonomous smart artifact that performs its functions independent of the environment and as an interaction device that offers its services to other instances (software applications, smart artifacts etc) in the vicinity. For the stand alone operating mode, the device's integrated memory and the built-in games offer added value compared to a traditional dice box. For instance, it replaces pen and paper when games like the aforementioned Yahtzee are played that require the counting of individual results. In fact, a similar game (the Counter Game) is currently implemented on the device, albeit by far simpler than its original. When the dice box is brought to a pervasive computing environment, it can be put into a remote controlled mode in which it is both able to receive commands from the environment and to convey back information about its own state.

2.3.3 Technical Realization

The core of the dice box consists of a Smart-Its particle computer [8] equipped with additional LC displays, LEDs and switches. Such a particle is "a platform for rapid prototyping of Ubiquitous and Pervasive Computing environments, for Ad-Hoc (Sensor) Networks, Wearable Computers, Home Automation and Ambient Intelligence Environments. The platform consists of ready-to-run hardware components, software applications and libraries for the hardware and a set of development tools for rapid prototyping" (http://particle.teco.edu/). The particle is built around two independent boards, a core board that consists mainly of processing and communication hardware and basic output components and a sensor board containing a separate processing unit, various sensors and actuators. In the case of the dice box, these were accelerometers, other sensors include light, humidity, temperature, or force/ pressure. The sensor board also runs the user application. The user applications are programmed in C using a respective application programming interface (API).

2.4 Other Interface Components

Dice, pieces, and game boards are the most prominent physical building blocks, from which tabletop games are constructed. Pervasive tabletop games pick up these physical interaction devices and their respective interaction techniques to preserve the social group situations found in traditional board game systems. Many other interaction devices such as ordinary computing devices (PCs, PDAs etc.) or simple interfaces such as physical buttons or RFID-augmented playing cards (ref. [8]) might be used complimentarily in order to form a heterogeneous, distributed landscape of user interfaces. In the following section we discuss the software infrastructure developed to integrate and coordinate these multiple user interface components into pervasive gaming applications. The central requirement of such an infrastructure is to unburden the developer from anticipating any concrete setup of interface components available at the players' site. Ideally, the developer abstractly defines the game rules so they will work equally well with hardware as well as virtual substitutes (A physical board vs. a GUI or a Smart Dicebox vs. a random number generator). Without changing game definitions it is thus possible to change interface components and systematically study the effects their idiosyncrasies have on a concrete game.

3. COORDINATION INFRASTRUCTURE

The coordination infrastructure 'Pegasus' provides the functionality to let heterogeneous devices connected in an ad-hoc manner share information and synchronize distributed data objects. It is designed as a lightweight communication solution which is capable of integrating also resource-constrained devices such as PDAs or the Smart Dicebox powered by diverse operating systems and providing wired or wireless communication protocols. The main design considerations are briefly outlined before the actual system architecture is presented.

Distribution and Dynamic Device Integration

As pervasive gaming applications are comprised by a distributed set of heterogeneous interaction devices that exchange information among each other, the issue of centralized versus direct communication immediately becomes relevant. Pegasus does not rely on a central server component, but allows communicating entities to directly exchange information. This allows for the realization of distributed applications that are highly dynamic regarding the integration and disintegration of communicating entities. Physical interaction devices and their corresponding software proxies can be added and removed at any time with their state information being synchronized over an anonymous, decoupled communication bus. In the same way, the application logic can be distributed among communicating entities so that the disintegration of any single central server component does not necessarily have fatal consequences.

Decoupled Communication

For the design of a distributed pervasive computing coordination infrastructure, the choice of an appropriate communication model is most crucial, because of its implications for scalability and flexibility. The decoupling of communication between participants is the primary design goal in order to address the highly dynamic nature of pervasive computing. Pegasus consequently adheres to a publish/ subscribe interaction scheme in order to realize a loose coupling of participating components. As [5] point out, the advantage of the publish/ subscribe interaction style over the majority of similar schemes such as shared spaces, message queuing, or remote procedure calls lies in the full decoupling on all three relevant dimensions, namely space, time, and synchronization.

Subscription Scheme

In order to save communication resources in a publish/ subscribe system it is important to specify which particular events a communicating entity is interested in, so that subscribers do not

have to receive all events that are published. Due to the small-scale nature of typical pervasive gaming applications that involve only a limited number of communicating entities, the overhead of sophisticated content-based protocols is uncritical. Therefore, Pegasus follows a content-based approach that allows filtering events based on the evaluation of predicates that can be dynamically altered during runtime. Events are stored as distributed data trees / XML structures that are hierarchically ordered, so that the communication scheme of Pegasus can be seen as a combination of hierarchical and content-based systems. Accordingly, Pegasus introduces the notion of Functional Objects that get triggered on freely configurable changes within distributed data trees. A Functional Object subscribes to an arbitrary tree or sub-tree and is triggered on the evaluation of predicates that relate to changes in the subscribed tree. This is explained in more detail in the respective section of the system architecture.

3.1 System Architecture

The Pegasus system architecture comprises functionality on three layers of abstraction. As illustrated in figure 6 these include 1.) the Basic Tools Layer that provides low level functions for dealing with data trees, network transfer, and XML parsing, 2.) the Network Data Layer that abstracts access to shared information via Accessor objects and 3.) the Functional Object Layer that provides various Functional Objects that implement multiple event handlers triggering arbitrary actions within the respective Functional Objects.

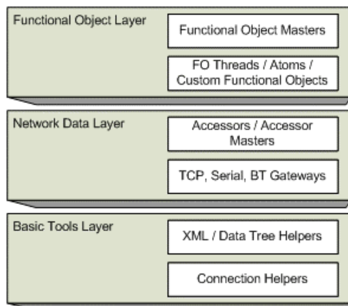

Figure 6: The Pegasus Coordination Infrastructure

The functionalities on each of the layers are now discussed in more detail.

3.1.1 Basic Tools Layer

The Basic Tools Layer comprises various lightweight XML-related library functions in which a document class represents the data tree. There are several methods for accessing the data structures via paths. Furthermore, various network related functions for establishing connections between multiple Pegasus software components and transferring data between them are provided at this layer. With the Basic Tools Layer available, we can load a data tree out of an XML file, establish a connection and transmit parts of the data tree to another Pegasus instance via different communication methods such as a socket, a serial line, or a Bluetooth connection.

3.1.2 Network Data Layer

The second layer is the Network Data Layer. It is responsible for the abstraction of access to shared information across Pegasus instances. It defines several classes of objects that closely work together. The most important one is the Accessor object that is capable of holding a tree structure either loaded from a file or referenced from a part of a tree of another Accessor. On a change of the referenced data, e.g. from another Accessor instance, the Accessor receives a notification. Similarly, the Accessor can inform other objects of its own change. Accessors hence provide the capability of representing and synchronizing arbitrary data trees among distributed software components based on Pegasus.

Another important class at the Network Data Layer is the Gateway Accessor. It passes information from published Accessors to a corresponding Gateway inside a different Pegasus instance from which other Accessors can access the corresponding data. Using this mechanism, two or more instances can refer to the same data, even when they reside on different devices with different operating systems. As the Gateway wraps the entire functionality of cross process data transfer, adapters for various communication protocols can be added by deriving from the Gateway class. At the current point in time, connections via TCP/IP sockets, represented by a TCP Server- and a TCP Client Gateway are implemented as well as Gateways for Bluetooth and serial communications. Other connection types such as IrDA or HTTP are trivial to integrate by deriving from the Gateway class. By using Gateway Accessors, the idiosyncrasies of specific transport protocols are mitigated, allowing for instance integrating wireless components such as the Dicebox next to USB/ serial devices like the magic wand.

3.1.3 Functional Object Layer

The third layer of the architecture is the actual Functional Object Layer that implements the aforementioned Functional Objects. Functional Objects build upon the communication infrastructure of the Accessors requiring only code for the additional functionalities they provide. A Functional Object augments the methods for data access and synchronization that an Accessor provides. It can be informed by other Functional Objects or Accessors of data changes and can evaluate certain conditions in respect to these changes. This can result in calling a virtual Do()-method of the Functional Object class (which is the actual interface between custom code and the distributed application data) to which the respective data is passed as parameters. The object itself can change data, which in return can result in "calling" actions on other objects, simply by changing the data regarded by them. Obviously, this Functional Object concept relates closely to the object oriented programming approach. Every object has its own data structures and certain methods which can be called on different context changes. The class definition is a hierarchic structure that can be derived from. With this concept, we can create networks of Functional Objects that react on data changes appropriately for their defined situations.

3.2 Representation of User Interfaces

Each user interface component is represented by a software proxy that uses a Functional Object to synchronize with its respective data tree. The connection and graceful disconnection of components is handled by each component’s Gateway Accessor. Since multiple Functional Objects can access a shared branch of data and anonymously inform each other of data changes, it is trivial to e.g. have a tangible game board with an associated data tree and then start up a graphical representation of the board that references the same data tree. Data changes in one of the

components would inform the other component of the respective changes and trigger appropriate actions. In this specific example, the Functional Object of the tangible game board would, by default, reverse any incoming data changes to enforce consistency with its own physical representation, whereas the graphical game board would accept any exterior data changes and update its graphical representation. Due to the decoupled communication scheme of Pegasus, an arbitrary amount of similar user interface components can be ad-hoc connected to a game application and automatically keep synchronized without any central coordination instance.

With the user interface components mostly aligning themselves, the developer can focus on describing the actual game rules that are independent of the actual user interfaces. The game description language used to define gaming applications is discussed in the next section.

4. GAME DESCRIPTION LANGUAGE

The game description language DHG is implemented on top of Pegasus which enables us to store all the rules of gameplay in xml files that are loaded at runtime. This allows us to tweak and change the game without having to recompile any code.

DHG is an xml-based language syntactically similar to many scripting languages with the unique benefits of being based on Pegasus, allowing for close integration with the pervasive gaming environment. Functions, defined in a function xml file, consist of a function name, argument names, and a sequence of parenthetically or bracketed expressions describing what actions to perform. A return type is specified by the xml attribute name. The game engine calls certain pre-defined functions upon events such as the introduction of a new playing piece on a game board, a piece moving, or a card being played. For instance, the xml code:

```
<MOVE action= "(SOUND PUBLIC foobar.wav)"/>
```

defines a function named MOVE. Here MOVE is one of the aforementioned pre-defined function names that is called whenever a piece is moved on the board. SOUND is a built in function that causes a music file to be played over an audio channel. This functions's return type is the default of 'action', meaning that side effects occur. Other possibilities include 'integer', 'bool', or 'string'.

In addition to basic math, string manipulation, Boolean logic, and function calls, DHG provides functions for accessing and manipulation of any Functional Objects used by the game. These objects store all the information related to the game, and it is via these objects that the various user interface components that make up the pervasive gaming environment interact.

Objects, identified by a globally unique ID (GID) have associated attributes which may be strings or integers. These attributes can be read, utilized, manipulated, and set by commands in a DHG function. Returning to our MOVE function, we can also update a field in the moving object by adding another command after the first:

```
<MOVE action= "(SOUND PUBLIC foobar.wav)
               (UPDATE+ TURN moves 1)"/>
```

Here TURN is a special GID that always accesses the piece whose turn it currently is, and this last line will cause the 'moves' attribute to be incremented by 1. Since DHG commands are stored as just strings, object string attributes can themselves be lines of code. There is a primitive (DO) that interprets a string attribute as code and executes it. Extending the MOVE command further, we can now call piece specific move code thusly:

```
<MOVE action= "(SOUND PUBLIC foobar.wav)
               (UPDATE+ TURN moved 1)
               (DO (TURN moving))"/>
```

The last line obtains the string-valued 'moving' attribute of the current piece and executes it as code. In addition, as DHG provides methods for manipulating strings, code stored in attributes can easily be rewritten dynamically during game play. It is even possible for code to modify and delete itself. As an example, consider an item that can only be used once. With DHG, it is a simple matter to set it up such that after the first use, it can never be used again, because the code that would need to be executed no longer exists:

```
<gid628 use='(actual effects here)
             (UPDATE STRING 628 use "")'/>
```

This code defines an object with global ID 628. The first time it is used in the game, the 'use' attribute is read as code and executed. The first parenthetical command contains the actual effects of the entity in the game, such as playing a sound, or updating an player's character's stats. The second parenthetical command causes the 'use' attribute of the entity with ID=628 to be set to the null string, or cleared. This results in the entity now to be set to <gid268 use="/>. The next time 628 is used in the game, nothing will occur, as there is no code in the attribute 'use' to execute.

In addition to acting as code, strings are also used to make object access simple and stackable. Since an object is just defined via its GID, and the GID is just a string, GIDs can be stored as attributes in an object. This means that an object can have sub-objects, and the DHG allows for them to be accessed and manipulated. There are also several pervasive-game specific concepts that are built into the DHG, making it easy for developers to write games without knowing the interior mechanics of the system. For instance, the concepts of boards, public/private audio and text output, and random number generation are all exposed in simple function calls. This allows a game designer to specify what they wish to happen, without concern for which devices are actually present. It is then the responsibility of the pervasive game environment to make it occur using whatever devices are actually present. To illustrate the typical game flow in a pervasive game, we examine a sequence of events, where objects modify each other's attributes in response to actions by the players. We shall need a player, defined as

```
<gid100 name="Player1" toy="" />
```

a toy he can play with:

```
<gid201 name="Rubber Ball"
             use="(UPDATES TURN toy 201)"/>
```

and the built in MOVE function can be defined as:

```
<MOVE action="(TEXT PUBLIC
    [CAT (TURN name) 'is playing with the '
                       (TURN toy name)])"/>
```

This causes a string containing the players name and it's toy's name to be displayed. On the players first turn, we will see:

```
Player1 is playing with the
```

since the 'toy' field is initially blank. Supposing then that the player activates object 201. This can be accomplished by selecting an icon on a screen, playing a card into an RFID reader, or pushing a button. The exact mechanics of object activation are left to the pervasive gaming environment and the Functional Objects of the user interface components. The game control code itself does not care. Once object 201 is activated, it will update the 'toy' attribute of the turn-taker, who in this case is object 100, to now read:

```
<gid100 name="Player1" toy="201">
```

On the players next turn, we will now see:

```
Player1 is playing with the Rubber Ball
```

This small example demonstrates functionality of objects (in this case 201) being able to effect gameplay without any modification to the game code. By more extensively modifying an object's attributes, and with appropriate hooks (like the DO() command described above), simple objects such as 201 can cause significant changes in the game.

5. A SAMPLE APPLICATION

To demonstrate and explore the benefits of flexibly integrating various interaction devices to a rather complex pervasive gaming application, we have developed a "Dungeons & Dragons" style tabletop roleplaying game called "Caves & Creatures" that is entirely defined using the DHG game description language. "Caves & Creatures" uses the Pegasus infrastructure to add many different tangible user interface components to the game as well as virtual GUI based components as substitutes.

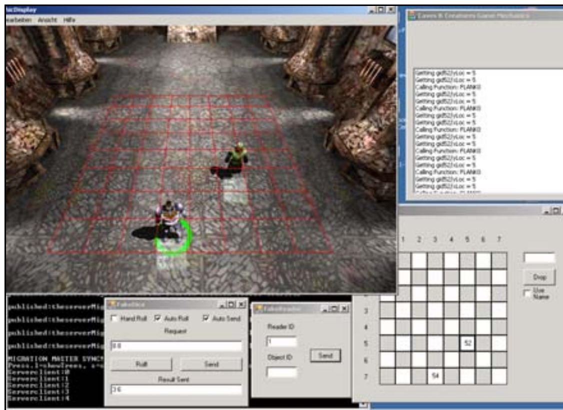

Figure 7: GUI Components of "Caves & Creatures"

The game currently integrates RFID augmented playing cards that represent items, weapons, armor, spells to be found, worn, used, cast, and traded between players. A physical game board is used for positioning and moving game characters. The Smart Dice Box implements the rolling of dice and gestures performed with the magic wand determine the success of magic spells. Some of these physical components are shown in figure 1, whereas figure 7 shows virtual counterparts of the physical components.

The beauty of the game's architecture lies in the complete decoupling of UI components from the actual gaming applications. It is irrelevant, if a physical game board as in figure 1, a 3D rendered display of the board (figure 7, top left area), or a simple 2D GUI control (figure 7, bottom right area) is used to control the movements of the pieces. The latter two components can even be executed multiple times and keep synchronized automatically due to the anonymous Pegasus communication bus. Likewise, physical cards or the dice box can be replaced by their virtual counterparts.

The entire game is defined in the DHG language, so that any aspect of the game including game data, flow, and rules can be changed while a game is running. Due to the implementation of DHG on top of Pegasus any components accessing game data or definitions are automatically synchronized. While a complete discussion of the game definition is beyond the scope of this paper, for illustration please regard the following XML sniplet that is triggered when a player wants to arm a weapon for his character, either by placing the weapon's physical card on an RFID reader or by using a corresponding GUI component.

```
<ARMWEAPON arg0="WEAPONID"
   action="[IF [EQ (WEAPONID used) 1]
    (TEXT PUBLIC 'Weapon already in use')
    { (DO (TURN weapon unuse))
      (UPDATE TURN weapon WEAPONID)
      (UPDATE WEAPONID used 1)
      (DO (WEAPONID my_use))
     }]" ></ARMWEAPON>
```

As can be inferred from this fragment, arming a weapon involves checking if that weapon is already armed elsewhere, unarming the previous weapon and then set it to being armed. The DO() hooks allow each object to have personalized code, e.g.:.

```
<gid100 name="Sword of Strength"
        my_use="(UPDATE+ TURN strength 2)"
        unuse="(UPDATE- TURN strength 2)"\>
```

Using this mechanism any object can define specialized behavior similar to derivation in other object oriented programming languages. The advantage of DHG over traditional OOP languages such as C++ or Java, however, is that this polymorphic behavior can be altered during runtime by the user or even by self-modifying code.

The actual gameplay of "Caves & Creatures" closely resembles that of the original "Dungeons & Dragons" tabletop miniatures game including also the more sophisticated rules for flanking or commander effects. Instead of having to browse through tables and lists of values and rules, the pervasive gaming adaptation unburdens the players from these mundane tasks and allows for concentrating on the actual gameplay. As a side effect, the game is also accelerated, because of these "user interface" improvements.

6. CONCLUSIONS

The component based architecture for pervasive gaming applications we presented allows for developing pervasive games without anticipating exact configurations of physical interaction devices. This reduces the complexity of game development and opens up the chance of experimenting with different configurations.

For instance, the sample game "Caves & Creatures" can be played without any physical interface components as well as with the specialized devices we have developed such as the magic wand or the Smart Dicebox. This allows for systematic evaluations of appropriate device compositions that we will conduct in the future.

7. RELATED WORK

There are several projects in the computer gaming research community that address the integration of the real and the virtual world. For instance, the academic research project False Prophets [11] is a pervasive board game, in which players jointly explore a landscape on a physical game board. A custom crafted infrared sensor interface helps identifying the playing pieces, while the game board is projected on the table. The realization of False Prophets is similar to parts of our platform, even though it is currently limited to a single exploration game. In the same spirit, but technically more constrained due to its very early realization in the mid-nineties is also the Digital Playing Desk from Rauterberg et al. [15].

Bjork et al. [3] presented a hybrid game system called Pirates! that adds the world around us to gaming applications with players moving in the physical domain and experiencing location dependent mini-games on mobile computers. Thereby, Pirates! follows a very interesting approach to integrate virtual and physical components in game applications. Unfortunately, the mini-games on the PDAs do not involve multiple players, so that social aspects are not very relevant for Pirates!

From the domain of augmented reality are the works of Cheok et al. [4]. These augmented reality approaches involve the use of cameras and specialized AR glasses that project digital information over the standard camera images. The results create visually stunning hybrid words that involve tangible and artifacts and digital augmentations. The drawback, however, is that AR glasses need to be worn that might hamper social interaction, because players lose direct eye contact. [7] and [1] present a large scale game played on the real streets that also integrates players connected via the internet hence also pervading multiple realities.

A more detailed discussion of the related work can also be gained from this overview article [12].

8. ACKNOWLEDGMENTS

We thank all our colleagues and friends at Fraunhofer IPSI for revising earlier versions of this paper.

9. REFERENCES

[1] Benford, S., Magerkurth, C., Ljungstrand, P. (2005) Bridging the Physical and Digital in Pervasive Gaming. In Communications of the ACM, March 2005, vol. 48. No. 3. Pages 54-57.

[2] Bolt, R. A. (1980). "Put-that-there": Voice and gesture at the graphics interface. In Proceedings of SIGGRAPH '80. ACM Press, New York, NY, 262-270.

[3] Bjork, S., Falk, J., Hansson, R., and Ljungstrand, P. Pirates! using the physical world as a game board. In Proceedings of Interact 2001, 2001.

[4] Cheok, A. D., Yang, X., Ying, Z. Z., Billinghurst, M., Kato, H. (2002) Touch-Space: Mixed Reality Game Space Based on Ubiquitous, Tangible, and Social Computing. In Personal and Ubiquitous Computing (2002), 6: 430-442.

[5] Ciger, J., Gutierrez, M., Vexo, F., Thalmann, D. (2003). The magic wand. 19th Spring Conference on Computer Graphics (Budmerice, Slovakia, April 24 - 26, 2003). L. Szirmay-Kalos, Ed. SCCG '03. ACM Press, New York, NY, 119-124.

[6] Eugster, P. T., Felber, P. A., Guerraoui, R., Kermarrec, A. (2003). The many faces of publish/subscribe. ACM Computing Surveys. 35, 2 (Jun. 2003), 114-131.

[7] Flintham, M., Anastasi, R., Benford, S., Drozd, A., Mathrick, J., Rowland, D., Oldroyd, A., Sutton, J., Tandavanitj, N., Adams, M., Row-Farr, J. (2003) Uncle Roy All Around You. Level Up conference, Utrecht, Nederlands, 2003.

[8] Floerkemeier, C., Mattern, F. (2006). Smart Playing Cards – Enhancing the Gaming Experience with RFID, to appear at www.pergames.de

[9] Gellersen, H.-W., Kortuem, G., Beigl., M. ,Schmidt, A. (2004). "Physical prototyping with Smart-Its". IEEE Pervasive Computing Magazine, Vol. 04, 1536-1268, 10-18.

[10] Lee, J. C., Avrahami, D., Hudson, S. E., Forlizzi, J., Dietz, P. H., Leigh, D. (2004). The calder toolkit: wired and wireless components for rapidly prototyping interactive devices. In Proc' DIS '04. ACM Press, New York, NY, 167-175.

[11] Mandryk, R., Inkpen, K. False prophets: Exploring hybrid board/video games. In Extended Proceedings of CHI 2002, pages 640–641. ACM Press, 2002.

[12] Magerkurth, C., Cheok, A.D., Mandryk, R.L., Nilsen, T. (2005): Pervasive Games. In: ACM Computers in Entertainment, Vol. 3, No. 3, July 2005.

[13] Magerkurth, C., Engelke, T., Memisoglu, M. (2004) Augmenting the Virtual Domain with Physical and Social Elements. In ACM Computers in Entertainment, Vol. 2, No. 4, October 2004.

[14] Minkley, J. (2003). Eyetoy shifts a million. Online article at Computer and Video Games Market. http://www.computerandvideogames.com/news/news_story.php(que)id=97876

[15] Rauterberg, M., Mauch, T., Stebler, R. (1996). The Digital Playing Desk: a Case Study for Augmented Reality. 5th IEEE Workshop on Robot and Human Communication, Tsukuba, Japan, 410-415.

[16] Ullmer, B., Ishii, H. (2000). Emerging frameworks for tangible user interfaces. IBM Systems Journal, 39(3):915–931.