Interaction Acts for Device Independent Gaming

Stina Nylander & Annika Waern
Swedish Institute of Computer Science
Lägerhyddsvägen 18
SE-75237 UPPSALA, SWEDEN
{stina.nylander, annika.waern}@sics.se
ISSN 1100-3154
ISRN:SICS-T--2002/04-SE

Abstract
Porting computer games to new platforms can be a cumbersome work, since many parts of a game is platform specific. This is due to the fact that game developers try to maximise user’s experience when playing a computer game, and to make the most of every device the game runs on.
We use game and device independent interaction acts to describe user-game interaction for networked games that are accessed from many different devices. A device independent description of the game interactivity allows for remote access from different (semi)-generic interaction engines, as well as device optimisation for the same game application.

Introduction
It is a mistake to think that games of today are platform dependent. Developing a high-end game is a huge investment, and to make this investment pay off, advanced games are typically developed to be portable. However, whereas there is a strong tendency towards making ordinary applications more and more platform-independent (e.g. by a web interface), games are multi-platform applications rather than cross-platform applications. Games typically try to make the most out of every device they run on, and will consequently look very different on a PC and on a mobile phone. Sometimes the different platforms will not even support the full game, but only certain functionalities. There are also examples of game families, where there are entirely different games for different platforms, but where data can be transferred between the different games. This makes existing approaches to platform independence such as media browsers (e.g. web) less well suited to support games that run on multiple platforms. The effect is that porting a game to a new platform is a tedious task that must be done over and over again for each new game.

In this paper, we suggest a middle route to platform independent game development. The approach is based on a clear-cut distinction between the game application and the human interaction with the game. We propose a way of describing game interactivity that is independent of both device type and game type using interaction acts (Nylander and Bylund, 2002). Since the interaction language is abstract and generic, it is possible to envision a set of (semi)-generic interaction engines that are able to handle any game based on the set of interaction acts it offers. The interaction engine may, for particular platforms and particular games also be replaced in part or in full by a game-specific interaction engine. This way, game interaction can still be optimised for a particular platform.
Limitation of the approach

The reader should note that the game interface – presentation and user interaction – is not the only aspect of a game that is specialised for a particular platform. A typical game architecture will contain several components that are rewritten for each platform; examples are external communication links, and memory management. In some games, even some of the critical game engines (an AI or physics engine for example) might be optimised for each platform.

This being true, the game interface is still perhaps the most critical part to handle. But the major reason for our work is that many games need not run all functionality at the user’s own machine. A networked game can very well be built using an architecture where most of the functionality executes at a game server that runs with a game service, and only the interface is distributed to the user’s own machine. This is the type of games we primarily target, and our example scenario below illustrates one such game.

Related Work

Developing device independent applications is neither a new research issue, nor an issue restricted to game research. During the seventies and early eighties there were little standardisation of hardware, and model-based systems like ITS (Wiecha, et al., 1990) tried to provide device independent applications. More recent attempts concern the area of mobile and ubiquitous computing, which is exemplified by Hodes et al. that provides user interfaces for different devices in automated service discovery (Hodes, et al., 1997).

The concept of interaction acts has been developed in an earlier project where we explored possibilities to create device independent on-line services (see (Nylander and Bylund, 2002) for details). During that project a first set of interaction acts was established and a prototype calendar service with three different user interfaces was implemented, Java Swing, HTML and std I/O. The calendar prototype showed that interaction acts are a way of developing device independent services, and that it is possible to generate user interfaces with different structure and different interaction modes based on a general description of the user-service interaction. However, it also showed that applications with high requirements on interactivity and real-time communication would need a more elaborated set of interaction acts and a more advanced system architecture, which is presented in this paper.

Scenario

Miriam is playing one of her favourite games the Sims\(^1\) on-line from a server. She is gaming on-line because she wants to make her family very rich, and she has trained them to a state where the family members get on very well by themselves and require little supervision. When she logs off, the family goes on living their life without her intervention.

Waiting for her bus, she checks in to the game site to see how her family is faring. This time, she accesses the game from her mobile phone, which gives her very little options to interact with the family. The output from the game is limited to statistics about the family and character status, given in text and simple graphics. The activity

\(^1\) See http://www.thesims.com, developed by Maxis.
options that Miriam has are high-level interventions such as forcing a character to take a new job, or simply to pause the game until she is back at her home computer.

At work, she gets an email from the game telling her that one of her characters had an accident and that the game has paused, waiting for her reaction to the accident. She logs into the game using a game browser\textsuperscript{2} to check how to handle the accident. She monitors development for a while and then again logs out, leaving the characters on their own. All game functionalities are accessible through the browser interface, but the browser uses only simple pre-rendered icons for characters and objects. Furthermore, this interface operates in asynchronous mode: Miriam can monitor the development of the game, but must press an ‘interact’ button to actively intervene. If she does so, the game pauses and she is given access to a number of actions through a standardised toolbar for the browser, as well as pop-up menus associated to the objects in the game world.

This scenario only illustrates one reason that makes access from multiple devices desirable, i.e. when a game goes on over a long period of time. A chess game can be played on-line in much the same manner, involving two players instead of one. Similarly, an on-line role-playing game is usually controlled in a rather high-level manner that easily can be transferred to simpler devices, sacrificing only the graphics. Even strategy games like \textit{Black and White}\textsuperscript{4} that are both multi-player and real-time can still be played from multiple devices by relying on its built-in intelligence (the game is influenced rather than controlled by the user) and by restricting the

\textsuperscript{2}The web browser is not an ideal tool for game interaction, as will be discussed below.

\textsuperscript{4}See http://www.bwgame.com, developed by Lionhead Studios Ltd.
interaction on simple devices to only a few of the game functionalities. Simpler multi-player games can also benefit from multi-platform functionalities since it increases the availability of other gamers.

For high-paced action games, we can expect that users will choose to use only one device, since the player’s skills are closely coupled to the skill of handling the device controls. However, even for this category, it is interesting to allow gamers with different devices play against each other in ‘mixed’ tournaments and ladders.

**Characteristics of game applications**

The example scenario was constructed to illustrate both how an advanced game can be designed for networking, as well as how games differ from other networked applications. The central requirement of this game is that it needs to be accessible from many different devices. This is a requirement that it shares with other networked applications. Often, the added devices are more restricted than a standard PC. Small devices are (and will continue to be) restricted in screen size, means of interaction, and computational power. Another property that our example game shares with networked applications is that the game resides ‘in the network’. The game is not an application that is installed on the user’s computer and controlled only by the user’s activities. It is a process that is distributed and synchronised between the user’s devices and a backbone server.

The scenario illustrates two properties that are typical for games but less common in other applications. Firstly, games are multi-platform applications rather than cross-platform applications. A common vision for networked applications is that their full functionality should be accessible in the same way from any device. Since games are built to maximise the user experience, a developer wants to make the most of every platform, both in terms of graphics and the kind of user controls it enables. In some cases, this leads to game series where the games for different platforms are entirely different games, only using a common concept and sometimes the ability to exchange information. Examples of this are the Pokemon\(^5\) and Harry Potter\(^6\) games. In our scenario, there is only one game, but the interaction with the game is instead more limited at the more limited platforms. Secondly, an important aspect of games is that they have a life of their own – things happen in games that gamers do not directly control, but that they must be able to notice. Games need to be able to push information to users through all media.

Moreover, games often present a ‘game world’ to users and the user is sometimes seen as having a precise position in the world. This world may be more or less realistic (as in 3D games), or entirely iconographic (such as in a board game). But in both cases, the activities that users can do are determined by the world status. This is not a unique feature for games, but outside the game arena it is more common to organise and present domain content based on what actions the user should be offered.

\(^5\) [http://www.pokemon.com](http://www.pokemon.com) by Nintendo.

Requirements posed by the scenario

The scenario described above introduces two important requirements on networked game applications: games must be able to handle different devices (sometimes with great differences in interaction and presentation capabilities), and they need to be able to push information to clients. In our example, we introduced the concept of a generic game browser to allow access from many different devices. This game browser does not use a game-specific engine for rendering or user interaction, and will necessarily impose some limitations on what the game can do and how the user can control the game. To handle the need of pushing data, our example uses email and the generic game browser instead of web interaction. Although the game browser is very limited in its functionality, it is still able to push information to the user about what is happening in the game.

To create a device-independent way to describe interactions in computer games, we need to find a level of abstraction where we can describe user-game interaction in a general way. This means that the description must be independent of device, game type and interaction mode. In the scenario above we saw two of these parameters changing, the device (from PC, to mobile phone and back to a PC with a game browser), and the interaction mode (from direct manipulation, to a text based UI on the mobile phone, to limited game browser UI), while the game was the same. If we could find a general description of the user-game interaction that covers all these modes, the game application will be able to serve many different devices without changes in the core application.

Interaction acts

We claim that by fixing our interface description language at the level of user-application interaction, we have found a level of abstraction that is independent of device, application, and interface model. The user-application interaction can be described in terms of a small set of basic interaction acts, which in different combinations allow the user to accomplish different tasks. A basic interaction act is an abstract unit of user-application interaction, which is stable over different types of user interfaces as well as different types of applications. For example, the act of presenting data to the user is the same independently of systems or modalities, while the means of presenting the data may change, e.g. pictures, synthesised speech or text output. We do not include any information that is dependent on media or modality in an interaction act, be it for input or output. Instead, we allow each interaction act to be mapped at run-time to any kind of rendering template, to create the actual user interface that might be both device and game specific.

In an earlier project (Nylander and Bylund, 2002), we have created and implemented a simple architecture model and a first preliminary set of interaction acts. In the case of very simple applications, the background application as a whole can be seen as a single object that users interact with, using a restricted set of interaction acts: output, input, select and modify. The four types correspond to the following four interaction activities.

− Output: The object can show users information.
− Input: Users can be offered to input data.
− Select: The object can offer a set of alternatives that the user will choose from. A button can be seen as a single option menu.
− Modify: The application offers a piece of data that the user can make changes to.

This particular set of basic interaction acts was selected based on analysis of user interaction in existing simple applications and tested through the design of a simple calendar service. We also allowed interaction acts to be grouped into hierarchical groups, and both groups and individual interaction acts could be given symbolic names.

In our first implementation of an interaction engine, we allowed a service to configure the interaction engine by a customisation form that mapped the basic types of interaction acts, as well as the application-specific names for interaction acts, to different interaction behaviours and presentation functions. At runtime, the service dynamically generated the user interface by sending a hierarchically grouped set of interaction acts to the interaction engine, which in turn generated the user interface. All user activities were encoded as responses to the presented interaction acts, and sent back to the application.

This first approximation of the role of interaction acts was a bit too simplified to fit more complex applications such as games. In the previous section, we discussed the fact that games often are modelled as an object space, containing many individual objects that may offer interaction acts to the user. In a game application, objects can be objects that reside in the game ‘world’, such as avatars for other gamers or artefacts. It is useful to introduce an additional level of abstraction that apply to an object space, and that offers the user acts that manipulate the objects and control their lifecycle: create, destroy, select, and move.

− Create: Offer the user a possibility to generate a new object in the space.
− Destroy: Enable the user to destroy an object.
− Select: Enable the user to select a set of objects in the space. A selection act can refer both to spatial properties of the objects (e.g. objects within a selected area) or any other properties that the objects have in common (all red objects).
− Move: Move an object (or a selected set of objects) to a new position in the space.

Another important aspect of interaction acts that we did not explore in our previous work is that interaction acts need to have a pre-defined life cycle. The life cycle determines how long the interaction act stays accessible in the rendered user interface. Since we want to avoid all abstractions that are dependent on a particular presentation style or modality, we limit the life-cycle information to deal with which party, user, interaction engine, or application, that is responsible for terminating the interaction act. An interaction act can be rendered as a temporary, confirmed or persistent element of the final user interface.

− A temporary interaction act is sent for presentation exactly once and will not be available after it has been presented. Sounds are typically presented in a temporary manner. Note that any interaction act including buttons and menus can be temporary (like a button ‘click her in 10 seconds or the download will start’).

\[9\] A button can be seen as a single-option menu.
- A **confirmed** interaction act stays accessible in the user interface until the user acts upon it. A typical example is a login screen or a popup menu.

- A **persistent** interaction act is available in the user interface from the moment it is rendered until the application actively terminates it, or the application itself is terminated. An example of the former is the presentation of an avatar representing another user present in the game world. An example of the latter is a window menu bar.

Note that the fact that an interaction act is accessible need not necessarily mean that it is visible in the user interface. The only thing it means is that the interaction browser provides a way for the user to “get hold of” the interaction act. A voice menu system could for example contain a way for the user to get access to a persistent menu by speaking a command.

### The interaction engine

To realise user interaction, the general description of the user-game interaction need to be complemented with a device-specific **interaction engine** that knows how to render the different parts of the interaction description on a given device. Different devices have different capabilities of presentation (screen size, keyboard/keypad, sound capabilities etc.), and each game has its own look and feel.

We introduce three ways to deal with this. Firstly, we allow each game to supply its own interaction engine that is game and device specific. Secondly, we assume that a game can provide a customisation form to a device specific **interaction engine**, which will generate a game and device specific user interface for the device, e.g. a mobile phone or a desktop computer. In this case the customisation form will be specific to the game and the device. If no customisation form is provided, the interaction engine will generate a user interface from default settings. Thirdly, a game can send its interaction acts to a generic **game browser**, which is game-independent but device-specific, in that it targets the specific capabilities of the device. The game can control how the game browser should present the interaction acts through configuring the browser with a customisation form. This customisation form needs to be game-specific and browser-specific, but it is not necessarily device-specific since the same browser capabilities can be realised on many devices. Furthermore, a game browser can have a default behaviour for the basic set of interaction acts, making it capable of producing a very simplistic presentation of a game even without a customisation form.

In the scenario, we exemplify this idea in three ways. The PC client is both game- and device-specific, and allows the game interaction to be as sophisticated as it is possible on the PC. On the mobile phone, the game is accessed through an extremely limited interaction engine that generates text and simple graphics output, and mainly menus for input. At work, the game is accessed through a generic but very simple game browser that runs on a large range of devices, much similar to the web browsers of today.

As we saw in the scenario, different devices offer different degrees of interaction for the same game. The game functionality that is offered on a particular device can be restricted, depending on the capability and usage context for the device. However, it is important to note that we strive for a solution where the same functionality always is
described by the same interaction act or set of interaction acts, independent of which device it is sent to.

Figure 2 shows a basic architecture of how an interaction engine can be built to interact with a game-like application. The left-hand part of the figure depicts the game application and its game-specific object space. The right-hand side depicts the game-independent interaction engine.

In this figure we distinguish between three subsets of interaction acts, based on what level of the application they interact with. At application level we have interaction acts that are used to control the application. The set of available options at this level is up to the application. Most applications will contain start and stop, but the application may offer many more options, e.g. change of viewpoint. At the object space level we have the acts that can be used to manipulate objects in the object space; create, delete, select and move. Finally, acts that manage and manipulate individual application objects are based on the four types of interaction acts that we defined previously: input, output, select and modify. In the figure, the two first types are depicted as operations available on the application itself, whereas the latter are shown as interaction acts offered by individual world object.

**Interaction engine functionality**

The interaction engine knows two things about the game that it is currently presenting. It knows about the set of interaction acts that the game currently offers, and for acts offered by individual world objects, it knows which object that offers which act.

**Figure 2.** A schematic picture of an interaction browser interacting with a game application. The browser is device and interaction model dependent, while independent of game application. The object space and active (darker) application objects present their interaction acts to the viewer.
The interaction engine uses this information to present the object space to the user, and detect when the user’s actions are responses to particular interaction acts. Such responses are sent back to the application or the specific object, depending on where the interaction act originated.

To be able to handle game interactivity, an interaction engine must have a set of properties that distinguishes it from an ordinary media browser.

− The fundamental modus of operandi of the interaction engine is *push*. The interaction acts are pushed to the interaction engine by the application, and the user’s activities are interpreted as responses to the pushed information. The application may at any point push a new object to the engine, independent of user activity, or request to delete an existing object.

− The interaction engine may very well filter away some of the pushed information based on local knowledge. For example, a 3D renderer may very well receive information about domain objects that never are rendered because they are outside of the user’s current view (or behind another object).

− Different interaction acts may have different life cycles. Some interaction acts stay accessible until the game application actively removes them. Other acts are offered only once, and accessible only until the user has acted upon them.

**Interaction engine architecture**

The basic building blocks of the interaction engine are the viewer and its cache, and the user interface engine.

− The **viewer** is the central component of the interaction browser. It is responsible for keeping track of the interaction acts that the application provides. The viewer keeps track of the interaction acts that currently should be available to the user. It also maintains the customisation form.

− The viewer has access to a local **cache**, which can be used to store interaction acts and media resources locally with the interaction engine.

− The **user interface engine** is more or less device-specific and responsible both for rendering the application presentations and capturing user activity with the input devices. The user interface engine is very often tied to a particular device, but more importantly it is always limited in the interaction paradigm it supports. For example, in our previous work we used the Java Swing package as a user interface engine. It is very simple to write a viewer for this kind of interface engine, but it is only suitable for rendering a very strict windows-based interface. There is no support for rendering a 3D environment or for real-time interaction. For a game, it is more suitable to use a game engine, or the user interface control functions from a game engine, as an interface engine.

**Game configuration**

Before the game can be rendered, the viewer might obtain a customisation form for the game, which contains information on how the user interface should be presented. This customisation form might be a simple table mapping each type of interaction act that the application can generate, to a presentation format that can be handled by the user interface engine. For extremely simple interface engines the viewer may need to be more advanced. In this case, the user interface engine may also call on viewer functionality to do things like
- filter away some interaction acts from immediate presentation,
- generate some interaction acts on its own that are used to help the user navigate the interaction acts provided by the application,
- request more interaction acts from the application, or
- request media resources from the application or a separate storage.

**Game-specific interaction engines**

As we discussed above, the entire interaction engine can also be game-specific. In this case, it makes no sense to distinguish between the viewer and the user interface engine. Still, it is useful to allow the game to interact with the interaction engine using interaction acts. In this case, interaction acts make it possible to allow the same game interactions to be rendered on a device that does not provide a game-specific interaction engine, but a semi-generic one. The only thing that is required to port the game to such an interaction engine is then to develop the customisation form.

**Revisiting the Scenario**

In the Sims example above, we can analyse the different interactions in the following way. To shorten the discussion, we limit it to actions that are available while the game characters are ‘in play’\(^{10}\), and not the wide range of actions available to set up the characters and their environment.

At the PC, the game uses a game-specific interaction engine much like the Sims game of today. To enable us to compare this engine to our other two examples, we summarise the functionalities of this engine below (limited to the ‘live’ gaming situation).

- The house itself and all objects in it are accessible to the user in that they might be shown to the user. Some objects may not be currently visible, that depends on the user’s current perspective. There are also several views available for each object – the object can be shown from different angles depending on the user’s current perspective. These presentation acts are best modelled as persistent – they stay available until the application kills them. The different views are pre-rendered media resources, which the interaction engine will select based on local knowledge about the user’s perspective. Characters are modelled much in the same way, only they move through the house and have idle behaviour.
- Objects and characters will sometimes offer possible activities to the user. These can be represented as persistent choice acts. Note that these are much less persistent than the presentation acts discussed above - the Sims game will often change the set of activities available to the user.
- These activities are not direct manipulation acts, but actually orders to one of the Sims characters that the user controls. The user can for example order a character to read a book or make dinner. The application contains a main menu that enables him or her to select the character that she wants to control.
- There are also a set of main controls for the game, that include selection between different information tabs that describe the status of a particular character, the ability to pause the game, zooming in and out in the level of

\(^{10}\) The game also contains functions that allow the user to select characters for their family, build the house, and buy furniture and household equipment for the family. We omit these from the discussion to keep it short.
detail, and selecting the point of view. Most of these are represented as menu
buttons, but the view is partially controlled by the keyboard arrows or mouse
movements. The menus and interaction options that control the user’s
perspective can be seen as entirely internal to the interaction engine. The menu
alternatives that access statistics for the Sims characters are provided by the
application, and rendered as buttons that provide the statistics on demand.
Finally, the game contains major controls such as pause and resume the ‘live’
action.

In the envisioned mobile phone interface, almost all of these interactions options are
missing. The interface is limited to presenting status information about the characters
and the family. These interaction acts are the same as the set of interaction acts for
accessing statistics in the PC version, but they are rendered with menus rather than
buttons. The ‘pause’ and ‘resume’ options are also provided by this interface.

In the generic game browser that Miriam uses at work, all of the interaction acts that
the application generates are accessible to Miriam, but the presentation is much
simpler and the navigation options are also simplified due to the standard
functionality of the browser. For the sake of the example, we assume that the browser
is limited to presenting a 2D space with smooth movements, and asynchronous
interaction.

- The house itself, all objects in the house, and the characters, are rendered by
  static pictures that either were sent with the customisation form, or can be
downloaded by the game browser when needed. The browser is able to show
  the characters moving through the space.
- The activities that are offered by characters and objects are not presented
  while the game is running. Since the browser is asynchronous, the user must
  first press ‘interact’ to pause execution, and then access the activities which
  then are available as pop-up menus associated to objects in the 2D space.
- All control menus, including game controls and the selection of which
  character to control, are offered from a standard toolbar menu.

This game browser can present all interaction acts that the application provides to the
game-specific interaction engine, but the game controls are considerably simplified.
To use this simple game browser, all the game developer has to do is to supply a
customisation form and supply the media resources (in this case, fixed images) that
the game browser should use to render the game. In this case, as often, the media
resources used can be a strict subset of those used in the ‘real’ interaction engine.

**Conclusions and future work**

We have presented a device and game type independent way to describe game
interactivity that is based on interaction acts. A generic description of the user-game
interaction allows for access to networked games from a wide range of devices with
the help of semi-generic interaction engines. An interaction engine can present a user
interface for the game either by using default renderings for each interaction act, or by
using game specific settings. This makes it possible to have one instance of a
networked game serving many different access devices, avoiding a complicated
porting procedure every time a new device will be used for access.

An early version of the interaction acts approach has been implemented in an earlier
project where a calendar served as test application. We are currently in the process of
designing and implementing an interaction engine that is capable of presenting
interaction for a range of games and applications, based on the architecture described in this paper.

Acknowledgements
This work has been funded by the Swedish Agency for Innovation Systems (www.vinnova.se), Gamefederation AB (www.gamefederation.com), and the Swedish Institute of Computer Science (www.sics.se). Thanks to the members of the HUMLE laboratory at SICS, in particular Markus Bylund, for thoughtful comments and inspiration.

References