Adaptable Pluggable Multimodal Input with Extensible Accessible Soft Dialogues for Games

Anthony Savidis[1,2], Yannis Lilis[1]
[1]Institute of Computer Science, FORTH
[2]Department of Computer Science, University of Crete
Crete, Greece
{as, lilis}@ics.forth.gr

ABSTRACT
We propose a game input system with *two-level multimodality*: an action may be performed via a number of virtual input devices, while the latter may be associated to a number of physical input devices that can be plugged or unplugged on-the-fly. Our input system includes a mode manager, capable to dynamically judge if the offered game commands are not possible via the available physical input devices. In such situations, the mode manager automatically activates interactive graphical panels providing all game commands, supporting pointing and switch-based input, the latter accessible by hand-motor impaired users. We refer to such panels as *soft input dialogues*, offering an automatic, extensible, and adaptable intermediate input layer among the game system and the input devices. Our input system supports pluggability, enabling locally or remotely connected devices to be utilized on-the-fly, a feature particular useful for pervasive games.

Keywords
Adaptable input; Multimodal input; Accessible input.

1. INTRODUCTION
Currently, entertainment in a pervasive computing perspective is increasingly receiving attention, introducing new challenges. In this context, the following requirements have been identified: (a) *multiplicity* - support alternative configurations for game input including: gamepads, joysticks, laser pointers, computer vision, digital gloves, binary switches (for hand-motor impaired), voice input, and touch panels; (b) *multiuser input* – provide concurrently input methods for each player (requiring multi-cursors and concurrent dialogues); (c) *diversity* - allow different players utilize different types of input methods; (d) *multimodality* - allow players utilize in parallel multiple input methods; (e) *dynamicity* – allow players abandon or engage new input methods on-the-fly; and (f) *adaptability* – support the automatic profile-based activation of best-fit input methods per individual player.

In our discussion we adopt the distinction among *physical input*, to denote events coming from a particular device carrying activation data, and *logical input* to imply high-level user actions triggering responses from the game logic. Such user actions are called *input commands*, while the mapping of physical input to input commands is called *input binding*. This distinction reflects the early proposition in [2]. Normally, at different runtime points, different sets of input commands become available to the player. Such command sets characterize the *interaction modes* of a game. If for an input command there is at least one binding supported by the available input devices, the command is said to be *plausible*. To consider disabled user needs, if the binding is supported by accessible input devices it is said to be *accessible*. Hereafter we revisit plausibility to *also imply* accessibility. Game accessibility gains increasing attention, reflected by the formation of an IGDA SIG for Game Accessibility [3], and by the inclusion in the top 50 greatest game innovations, according to Ernest Adams [1], of "Reconfigurable controls and other accessibility features". Following the previous definitions, a mode is plausible if all its commands are plausible.

Although none designs games with non-plausible modes, in case of (un)pluggable input, implausible modes may dynamically arise. In our input system we resolve this issue by supporting mode-specific animated soft dialogues providing all respective input commands in the form of graphical command panels merely via pointing or switch-based scanning interaction.

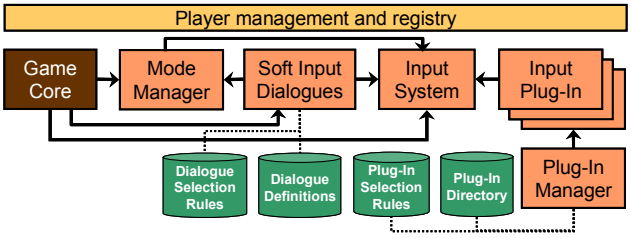

Figure 1: High-level overview of the input system architecture; arrows indicate package dependencies.

An overview of the overall software architecture is provided under Figure 1. As shown, the input system generally relies on two key layers: plug-in support handled by the plug-in manager and soft input dialogues. The activation of the most appropriate soft dialogues and input device plug-ins (from those available) is carried out following the selection rules defined in the Decision Making Specification Language – DMSL [5]. Examples of adaptation rules for plug-in selection are provided in Figure 3. Soft dialogue definitions, including appearance, command mapping, active areas and meta-information, may be freely extended. Examples of soft dialogues from our board game are provided under Figure 2.

Advances in Computer Entertainment Technology 2008, Yokohama, Japan.

```
stereotype HandMotorImpaired: hasattr(user.hand_impaired) and params.user.hand_impaired = true
stereotype CanUseMouse: hasattr(user.mouse_pointing) and params.user.mouse_pointing in {expert, good}
stereotype PrefersKeyboard : hasattr(user.prefer_keyboard)  and params.user.prefer_keyboard = true

component input [
        if HandMotorImpaired   then    activate "KeyboardSwitchpad bindings/switchpad.xml"
        if CanUseMouse         then    activate "MousePointing"
        if PrefersKeyboard     then    activate "KeyboardKeypad bindings/keyboard2.xml"
]
```

Figure 3: Excerpt of adaptation logic in DMSL for player stereotype definition and player-adapted plug-in activation logic.

2. RELATED WORK

Multimodality recently, it became the focal point of W3C through the MMI act [6]. Although most games support expensive and custom peripherals, little progress has been done in terms of dynamic interoperability and extensibility. There are games where the input has been augmented to support multimodal input via both voice and hand-gestures, as in [4]. However, in such cases the input system is rebuilt from scratch as a customized extension to the basic game, with no extra capabilities for extensibility of the multimodal features, such as adding new channels of input. *Adaptability* implies system capability for automatic User Interface configuration to individual user profiles. While in the general HCI field adaptation methods have been widely applied, input adaptability for games is not that popular. *Accessibility* is a field recently gaining more attention in the game development domain [4]. There are numerous games implemented with accessibility extensions for various user groups, however, most of them are customized, hardly resulting in generic methods.

3. MULTIMODALITY

Our support for multimodality relies on the architectural split among virtual input and physical input. Firstly, we identified a small set of virtual input devices supporting extensible input commands, while transferring the responsibility of binding physical to virtual commands directly at the physical device level (e.g., as part of a device configuration file). Then, we enabled multiple physical input devices map to a single virtual input device during runtime. The latter essentially allows multi-channel input, since distinct physical devices may imply varying input modalities. Additionally, physical devices may be comprehensive subsystems controlling multiple hardware devices, such as gesture recognizers, voice input, vision systems, etc. In other words a physical device is not bound to a single hardware device. Next we elaborate on implementation details for virtual and physical input management.

Virtual input management The notion of a virtual input device is reflected in a respective superclass, as shown under Figure 6 - middle left. Virtual devices encompass a list of their respective attached (installed) physical devices as pointers of physical device superclass Figure 6, top left). This way, the input system can always tell if a set of input commands is plausible by the physical input devices that a player is using. Consequently, the mode manager (Figure 6, top right) may test if the currently entered mode is plausible (Figure 6, bottom right, method *IsPlausible*). In a negative case, it will decide to launch an appropriate (to the mode) soft dialogue. In case polling is required, virtual devices offer a *Poll( )*actually polling all its associated physical devices, and *PollAll()*, invoking *Poll()* for each virtual device. Next we elaborate on the three virtual input device types we have defined in our input system: *keypad*, *pointing* and *scanning*.

Virtual keypad (Figure 4) It represents physical devices with a number of hardware keys, posting distinct codes on press and optionally on key release. Examples of physical device instances covered by the virtual keypad are keyboards, phone keys, gamepads, driving pedals, and gesture recognizers (including computer vision software); essentially, the string key codes will map to the recognized commands. The adoption of string representation for key codes allows embrace extra types of physical devices (e.g., steering wheels, pressure switches, etc.) delivering more structured information as device events, instead of a plain command name.

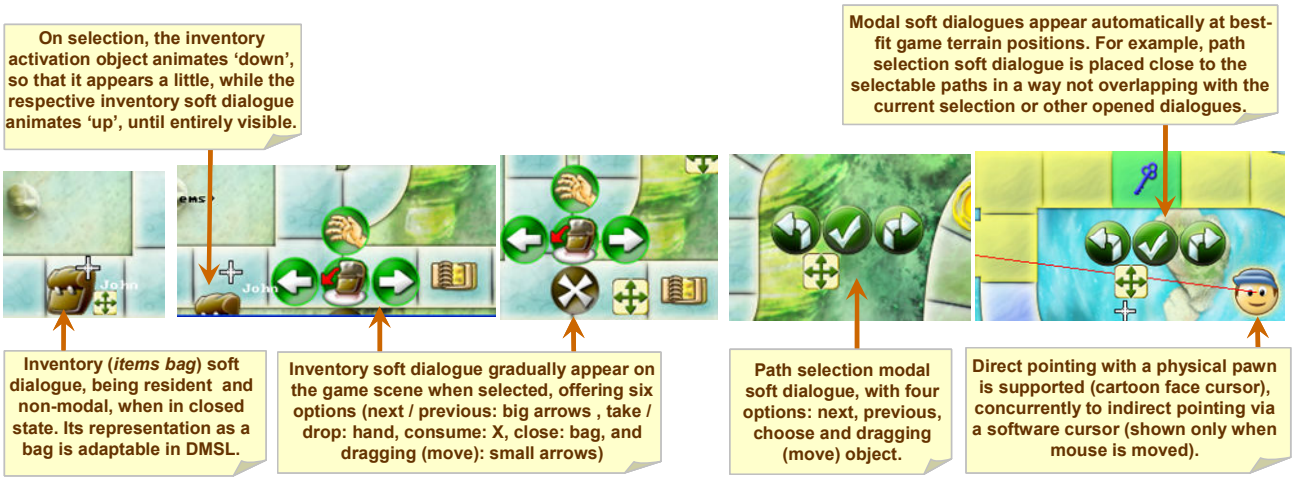

Figure 2: Examples of soft dialogues from the 'Four Elements' board game for inventory and path selection.

```
class VirtualKeypad : public VirtualInputDevice {
   typedef std::string Event;
   typedef void (*Handler)(const Event& event, void* closure);
   typedef unsigned PlayerId;
   typedef map<PlayerId, pair<Handler, void*> > HandlerMap;
   HandlerMap handlers;
   void PostEvent (const Event& event, PlayerId player) {
         { HandlerMap::iterator i = handlers.find(player);
           if (i != handlers.end) (*i.first)(event, i.second); }
   void SetHandler (Handler handler, void* closure, PlayerId player)
         { handlers[player] = pair<Handler, void*>(handler, closure); }
   VirtualKeypad (PlayerId id): VirtualInputDevice(id){}
};
```

Figure 4: The *VirtualKeypad* class.

Virtual pointing (Figure 5) It abstracts relative (e.g. mouse) and direct / absolute pointing (e.g. touch screen), supporting multi-cursors (one per player) via player identification included in the event data type, and multi-touch (i.e. a player touches more than a single spot) if the physical device provides information on touch points. The *PostEvent* and *SetHandler* methods have the same semantics as in *VirtualKeypad* class, with similar implementation.

```
class VirtualPointing : public VirtualInputDevice {
   enum Type {
        Touch, DoubleTouch, Untouch, // surfaces, composite systems
        MoveTo, MoveRel, Click, DoubleClick, Release, // mouse style
        MoveObjectAt  // physical objects like pawns, dices, etc.
   };
   struct Event {
       Type          type;
       string        data; // Extra data, mostly device specific event info
       unsigned      x, y;  // Absolute (position) or relative (offset) data
   };
   void PostEvent (const Event& event, PlayerId player);
   void SetHandler (Handler handler, void* closure, PlayerId player);
   VirtualPointing (PlayerId id): VirtualInputDevice(id){}
};
```

Figure 5: The *VirtualPointing* class.

Virtual scanning (Figure 7) Hierarchical user interface scanning via switch-based accessible input (mostly binary switches) is a well-known technique enabling access for users with impairments on the upper limbs. Technically, scanning is provided as an extra input method of soft dialogues, playing the role of an input redundancy method. We introduced a virtual scanning device through which all scanning-related physical input devices may be connected.

```
class VirtualScanning : public VirtualInputDevice {
   enum Event {
        Next, Select, // basic events, 2 switches or 1 switch time scanning
        Previous, ReverseDir, ExitContainer, // Extra switches
        AtFirst, AtLast, EnterNextContainer // Extra switches
   };
   // Same methods and similar constructor as the rest of virtual devices
};
```

Figure 7: The *VirtualScanning* class.

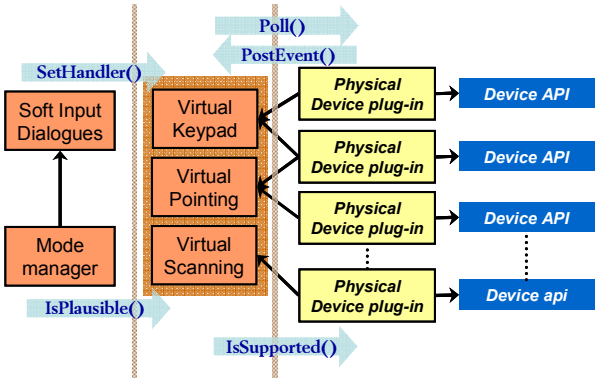

Figure 8: Architecture for physical input device management.

Physical devices should be wrapped-up as classes derived from the *PhysicalInputDevice*. The overall architecture, denoting method invocation among components, is shown under Figure 8.

```
class VirtualInputDevice { public:
   typedef list<VirtualInputDevice*> DeviceList;
   virtual const string  GetTypeId (void) const = 0; // Derived class type
   virtual bool          IsPlausible (const string& binding) const = 0;
   static DeviceList&    GetAll (int playerId);
   void                  Install (PhysicalInputDevice* dev);
   void                  Uninstall (PhysicalInputDevice* dev);
   void                  Poll (void) { poll every physical device }
   static void           PollAll (void) { poll every virtual device }
   VirtualInputDevice (int playerId);
};

class ModeManager { public:
  typedef string Id;
   void   AddMode (const Id& mode);
   void   AddCommand (
             const Id&     mode,     const string& cmmd,
             const string& vdevice,  const list<string>& bindings
          );
   void   EnterMode (const Id& mode);
   void   ExitMode (const Id& mode);
   bool   IsPlausible (const Id& mode) const;
   int    GetPlayerId (void)const;
};
```

```
class PhysicalInputDevice { public:
   virtual bool Poll (void)=0; // thows DisconnectionException;
   virtual bool IsSupported (string binding) const = 0; // String encoding
};

bool ModeManager::IsPlausible (const Id& mode) const {
   list<string>& cmmds = get all input commands of mode
   DeviceList& devs = VirtualInputDevice::GetAll(GetPlayerId());
   foreach cmmd in cmmds do
      foreach dev in devs do
         foreach binding in bindings do
          if (!dev->IsPlausible(binding))
              return false;   // All bindings should be plausible
      }
    return true; // All bindings found to be plausible
}

ModeManager& modes = GetModes(GetCurrPlayer());
modes.AddMode("NameEntry");  // Mode for text entry of player name

list<string> bindings; // Example of bindings
AddAllPrintableASCIICharacters(bindings); // All printable chars
modes.AddCommand("NameEntry", "Insert", "Keypad", bindings);
```

Figure 6: Virtual and physical device super-classes and mode management.

If an event is detected at the physical device it is forward via *PostEvent()* to its respective virtual input device. As shown, a single physical device may propagate events to more than a single virtual device type. Upon mode entry, the mode manager tests plausibility of the corresponding commands. The latter is performed by testing if the command bindings are plausible via *IsPlausible()*for the virtual device, forwarded to all attached physical devices as *IsSuppored()* calls.

4. PLUGGABILITY AND EXTENSIBILITY

Dynamic deployment of remote input devices is also supported by the software architecture of Figure 9, *RemotePhysicalDeviceProxy* inheriting from *PhysicalInputDevice*. During runtime, remote devices can connect to *RemoteDeviceConnector*; once connection is established a new *RemotePhysicalDeviceProxy* instance is created, initialized with the socket of the accepted input device.

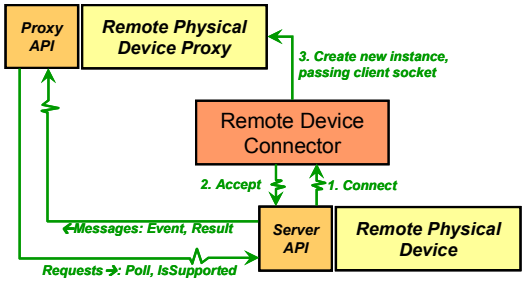

Figure 9: Plugging dynamically remote physical devices.

Soft dialogues are defined in XML, providing the bitmap ('film' field, being the bitmap's URI), commands with their hot (clickable) areas and optionally provided frames (to visualize click state). The path selection dialogue is provided below.

```
<Dialog name="paht_selection" film="pathSelection" defaultFrame="0">
<Commands>
   <Command name="previous" frame="1" box.x="0" box.y="0" box.w="44" box.h="44" />
   <Command name="select" frame="2" box.x="43" box.y="0" box.w="44" box.h="44" />
   <Command name="next" frame="3" box.x="87" box.y="0" box.w="44" box.h="44" />
   <Command name="move" frame="4" box.x="27" box.y="42" box.w="31" box.h="31" />
</Commands>
</Dialog>
```

5. ACCESSIBILITY AND ADAPTABILITY

In scanning, a special "marker" (green rectangle) indicates the focus interaction item, moving the marker to the *next* or *previous* interaction item using software or hardware switches. To navigate across soft dialogues we introducing two states: (i) in *exit* state (red rectangle) *select* moves to the next dialogue, while *next* changes state to *enter*; and (ii) in *enter* state (green rectangle), *select* moves focus to the first command of the dialogue, while *next* action changes state to *exit*.

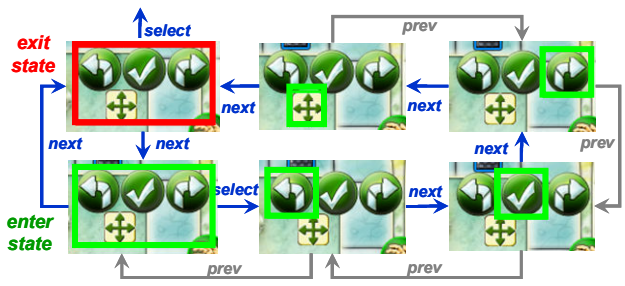

Figure 10: Navigation in a soft dialogue via *next* and optional *prev* action; select activates the area-associated command.

To support automatic player-adapted input configurations we adopted the DMSL language [9], embedding its rule compiler and evaluator as part of our input system. The following rules concern the adaptation of the scanning marker for colorblind players.

```
component ScanningMarker [
 if hasattr(user.colorblind) then [
       activate "Marker.OnEnter.Color:Blue"
       activate "Marker.OnExit.Color:Blue"
       activate "Marker.LineStyle:Dashed"
       activate "Marker.LineWidth:3"
       activate "Marker.EnterShapeStyle:Rect"
       activate "Marker.ExitShapeStyle:RectWithAnX"
 ]
 else activate "MarkerDefaults"
]
```

6. SUMMARY AND CONCLUSIONS

Our work emphasizes the critical role of the game input system with a focus on multimodality, pluggability, adaptability, and accessibility. We introduced soft dialogues as a dynamic intermediary among physical input and game commands, capable of supporting plausible interaction through pointing or switches, irrespective of the type of game input required during runtime. Additionally, through the virtual input layer, abstracting over a wide range of physical input devices, not only game input management is simplified, but pluggability is facilitated since device disconnection gets manageable at the low-end, introducing no failure to the game core itself. The latter may also help for game input recovery in games migrating on-the-fly from over different devices. Finally, we supported automatic configuration of all input features to individual player profiles by deploying a decision language and its respective adaptation engine.

7. REFERENCES

[1] ADAMS, E. (2007). 50 greatest game innovations. In *BusinessWeek* (November Issue). One line from: http://www.businessweek.com/print/innovate/content/nov2007/id2007115_528484.htm

[2] FOLEY, J. D., WALLACE, V. L., CHAN, P. (1984). The human factors of computer graphics interaction techniques. IEEE Computer Graphics & Applications, 4(11) (November 1984), pp 13-48.

[3] IGDA GAME ACCESSIBILITY SIG (2004). White Paper: Accessibility in Games: Motivations and Approaches. http://www.igda.org/accessibility/IGDA_Accessibility_WhitePaper.pdf

[4] LEE, K. B., KIM, J. H., HONG, K. C. (2007). An Implementation of Multi-Modal Game Interface Based on PDAs. Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications, IEEE Press, pp 759-768.

[5] SAVIDIS, A., ANTONA, M., STEPHANIDIS, C. (2005). A Decision-Making Specification Language for Verifiable User-Interface Adaptation Logic. *International Journal of Software Engineering and Knowledge Engineering*. Vol. 15, No. 6 (December 2005), World Scientific Publishing, pp 1063-1095.

[6] W3C MMI (2008). Multimodal Interaction Activity. Word Wide Web Consortium. http://www.w3.org/2002/mmi/