Modifying a game interface to take advantage of advanced I/O devices

A case study

Rubén J. Garcia[1], Milán Magdics[1,2], Antonio Rodríguez[1], Mateu Sbert[1]
[1]Universitat de Girona
Girona, Spain
[2]Budapesti Műszaki és Gazdaságtudományi Egyetem,
Budapest, Hungary
e-mail: rgarcia@imae.udg.edu

Abstract—**This paper describes the modifications required to use advanced, immersive I/O devices such as domes and Kinects in videogames (the case study includes four games). A standalone library of components has been created to allow easy adaptation of other games, with few modifications required in the game architecture. The changes include the creation of a cubemap to obtain the whole surroundings of the user in the game, and the use of post-processing GPU shaders to render this cubemap realistically in the curved surfaces. Additionally, the adaptation process of the GUI is described, and additional interaction modes using Kinects are presented.**

Keywords-Serious Games; Advanced I/O devices; Virtual Reality

I. INTRODUCTION

Large virtual reality visualization devices are becoming more common and affordable, and intuitive I/O control devices based on tracking the users gestures allow for an ever increasingly immersion in virtual worlds. However, seamless integration of these visualization devices in game and rendering engines is not yet available, so mostly custom-build software is available for the devices.

We present here the adaptation process of four serious games to take advantage of these devices. The software created during the adaptation process will be integrated in a generic library for commercialization purposes.

The following sections present a description of the different games studied and the technologies used for immersive interaction. Then section II describes the modifications in the games required to render to curved immersive devices while preserving mouse interaction. Section III describes the integration of Microsoft Kinect to control the game. Finally section IV concludes the paper.

A. Serious games

Four games have been tested using our libraries: Legends of Girona [1] is a historical serious game which describes the siege of Girona during 1285. The game uses first and third person perspective and is designed as a graphic adventure. Jaume I is a strategy game which takes place during the battle of Portopí (1229). LISSA [2] is a medical simulator teaching CPR. Musical World creates a virtual world where points are obtained by singing.

The games are implemented using Unity [3], which is a multiplatform game engine running on Microsoft Windows and Apple OSX. Unity also provides support for game consoles and mobile devices. Screenshots of the different games can be seen in Fig. 1, 2 and 4.

B. Dome and Immersapod

A dome is an immersive, hemicircular environment which can be used as a display. Domes are commonly used in planetariums. The device we used (Fig. 2 (h)) is composed of an inflatable hollow hemisphere measuring 1.5 m in radius [4], and a video projector with a lens system to project the planar image of the video projector into the hemispheric walls [5]. The projector takes as input a standard HDMI, DVI or VGA cable.

An immersapod [6] is a display device composed of a projector and a rigid, vertically positioned spherical cap (truncated on the lower side). Fig. 2 (i) shows a photograph of the device.

To create the video feed for these output devices, a fisheye camera using a 6mm to 8mm objective can be used. In the case of the dome, the camera points upwards, while for the immersapod the camera points forwards. Some post-processing of the video (masking and cutting) is required for correct visualization on the devices. Another possibility is the use of six cameras to create a cubemap, obtaining the final image from the cubemap.

Examples of games using these technologies are driving simulators, such as [7] (immersapod) and [8] (full dome), and flight simulators [9] (large curved screen). Some work has been performed previously on adding support for the immersapod to the Unity and Blender game engines [10], [11], although some rendering artifacts are visible in these methods. The Half-Life engine has also been ported to a

Figure 1. Level 1 of the Legends of Girona game

virtual reality theatre with two planar walls [12].

C. *Kinect*

Microsoft Kinect [13], originally intended for the Xbox 360 game, is a webcam-style add-on peripheral designed to support the most natural ways of communication with the computer: gesture recognition or spoken commands (often referred to as *natural user interface*). It is both equipped with a color camera and an infrared projector extended with a sensor providing depth information, turning the Kinect into a low-cost, real-time full body 3D motion capture device. According to the documentation, two skeletons, and up to 6 people within its field of view can be detected. For a single skeleton, 20 joints can be used in standing posture and 10 joints while sitting. By analyzing the gestures and poses of the user, Kinect can provide basic interactions similar to mouse, keyboard and touch interactions (i.e. selecting buttons, zooming and panning around a surface). Kinect control has been successfully applied to many different areas, such as computer games and entertainment, education or healthcare.

II. I/O Interface in Domes and immersapods

The use of traditional mouse and keyboard interaction techniques in the dome is difficultated by the lack of suitable resting surfaces (they impede mobility and produce large shadows obscuring the display). The use of a (preferably wireless) gamepad or Kinect controller increases game enjoyability significantly.

Concerning the camera transform used for these devices, modifications are required in the final render, in the interaction with the virtual world and in the management of the GUI. The following three sections describe the details and caveats, while section III details the Kinect controls.

A. *Visualization of game geometry*

The camera required for rendering on these devices has a field of view of up to 360×180 degrees. However, most game engines (and in particular the Unity 3.5 engine we used) are not prepared for such extreme fovs. See Fig. 2 (a) for an example rendering.

Therefore, another approach is needed. A cubemap centered at the main player position is updated at every frame, and a post-processing GPU shader uses the cubemap to generate the correct image. We have implemented two shaders: one handles dome devices and the other immersapods. The pseudocode can be seen in Fig. 10 and 11. Fig. 2 (b), (g) and (c), (f) show the result for the dome and the immersapod for the different games, respectively. Fig. 2 (d), (e), (i) shows photographs of the resulting visualizations (the photograph only spans a fourth of the display in the dome because of the camera fov).

The most important input of the shaders is a cubemap texture of the scene. To handle the positioning of the projector, a height parameter is used to change the apparent position of the camera. The resolution of the cubemap has an important impact in the performance of the game. We have found that a 1024^2 cubemap texture is sufficient for rendering at full HD, while smaller textures produce too much smoothing.

The dome master shader requires the vertical angle that the dome spans; while the immersapod master shader requires two angles: the angle at which the sphere cap is cut at the bottom (see Fig. 3), and the fov. Additionally, since minute changes in position and orientation of the projector can make a noticeable difference in position of the bottom border (projecting a strip of the image on the walls behind the immersapod), we have added a circular mask aligned to the bottom of the immersapod to avoid the issue.

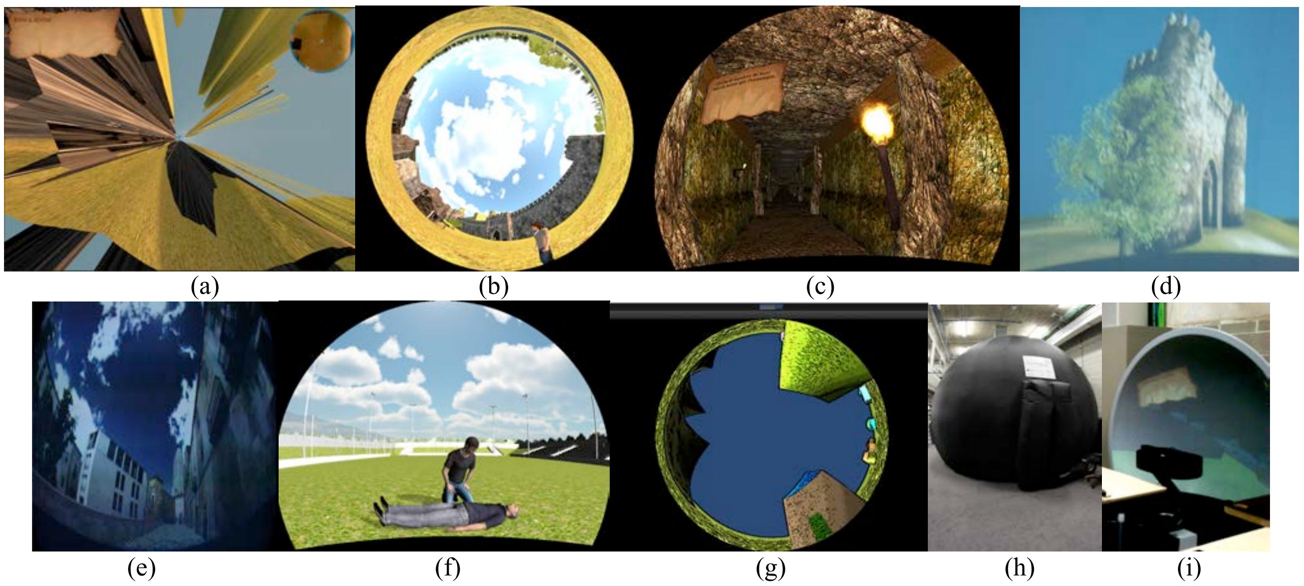

Figure 2. (a) Unity camera with a 350° field of view; Output of the shader for visualization on Legends of Girona (b) dome and (c) immersapod; Photograph inside the dome (d) Jaume I; (e) Legends; (f) LISSA; (g) Musical world; (h) Photograph outside the dome; (i) immersapod

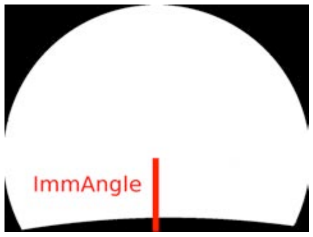

Figure 3. Immangle parameter of the immersapod shader

Another issue which needs to be taken into account is that the high area of the screens means that the projection will have a lower brightness and contrast than normal. An extra ambient light activated when in dome mode takes care of the issue (the intensity must be calibrated on each scene). Skymaps are not affected by ambient lights, so in some cases a specific skymap may be required.

When domes are used in planetariums the projector is positioned on the center of the sphere, so that a 180° fov simulates the night sky. However, we have found that lowering the projector to the floor (achieving an almost 360° fov) increases the immersive experience significantly, since the projection almost reaches the floor.

The skymaps, ambient and height parameters need a per-scene or per-camera tuning, while the rest of the parameters need only be calibrated once on the hardware.

Since the rendering of a cubemap texture is substantially slower than a normal rendering pass, the framerate of the games suffers a corresponding decrease. The use of these output devices thus requires the use of a better CPU and GPU combination. However, given the cost of the devices, the expense is not significative. For our tests, we used a 3.1 GHz Intel Core i5 Mac with 8 GB of RAM and an AMD Radeon HD 6970M with 1GB of RAM. Our games can be rendered at 200 fps in a flat monitor, while the Dome and immersapod version run at 66 fps, which is three times as slow. We consider that the reduction in performance is justified by the increase in realism. We couldn't find any significant differences in the conversion procedure of the different games. The cost of the procedure is proportional to the number of distinct cameras.

B. *Object Selection*

When using the mouse to click on objects, there are two ways to know which object was clicked:

- One approach is to use a buffer of identifiers. In this case, polygons are rendered to two different buffers: a color buffer which creates the image that the user views, and a second buffer in which each object has a unique color or identifier. To know which object was clicked, the second buffer can be queried with the mouse position. In this case, our transform should be applied to both buffers.
- Another approach is to create a ray originating at the camera position and passing through the mouse position. This ray can be used to intersect the scene tree. In order to take into account the transform of the output device, the direction vector should be transformed by the algorithms described earlier. This is the approach used by the Unity engine.

We have created a DomeMouseClick interface which exports three functions: MouseClick, MouseOver and Deselect. Clickable objects should use the interface. A script has been created to take into account the rendering mode and send the corresponding events. This script contains an implementation of the algorithms described in the previous section, along with the empty transform for planar visualization devices. Again, the cost is proportional to the number of distinct active objects.

C. *GUI*

The standard GUI is normally painted as a post-processing step, using 2D primitives. Therefore, it is not affected by our algorithms. The result is a clipped, curved and deformed GUI. If only simple, small, centered GUIs are used, the deformation remains small, and the main problem is that the GUI appears in the zenith of the scene in the Dome. If this is acceptable, no changes are required to use domes or immersapods. The possible GUI position can be seen in Fig. 4 (a).

In the case of more complex GUIs changes must be made to have the GUI follow the correct transform.

A new camera is positioned in an unreachable zone of the scene, and child object planes are created to represent the different GUIs. These objects are enabled and disabled when needed. Text is generated using 3D text primitives. Due to the reduced contrast in dome mode, care must be taken to assure that messages remain readable. An increase in size and using higher contrast between text and background are possible solutions. The GUI can be positioned on any visible surface of the Immersapod, but in the case of the Dome it should be located reasonably in front of the user (see Fig. 4 for an example; the reduced pixel count available for the GUI is apparent).

A new layer is used to distinguish between GUI and geometry objects, so that each camera only sees the corresponding objects. The GUI camera should clear the depth buffer, but not the color buffer, and it should render after the main camera, so that the GUI is visible in normal rendering mode. Clickable objects in the GUI should implement the DomeMouseClick interface, and the camera should use the script mentioned in the previous section.

In dome or immersapod modes, when the scene cubemap is generated, a second cubemap rendering using the GUI camera is used to transform the GUI.

Although the use of only one camera to render both geometry and GUI is theoretically possible, we have found that the GUI is occluded by the geometry objects if the camera is near these objects. The use of a special shader for the GUI which renders after all geometry and does not take into account the z-buffer alleviates the problem; however Unity's occlusion culling sometimes disables the GUI objects if they are farther than geometry objects without taking into account the fact that they should be rendered. The cost of the update is proportional to the number and complexity of the menus in the GUI.

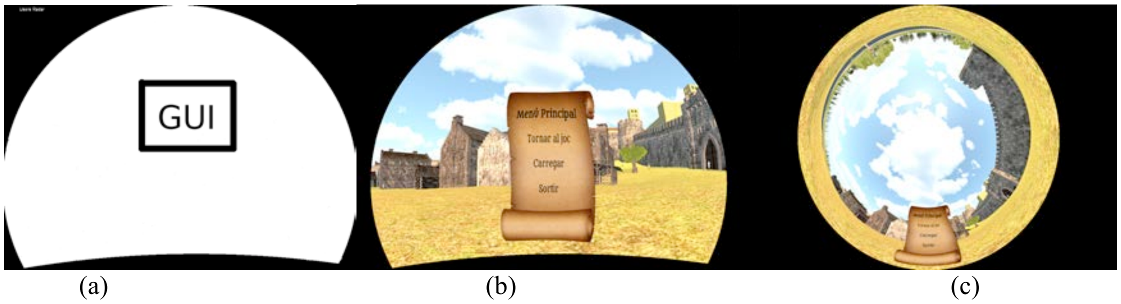

Figure 4. (a) Possible position of the GUI if no transform is used, (b) transformed GUI for Immersapod, (c) transformed GUI for Dome

III. Interaction with the Games

The Kinect device provides the raw color and depth images at real-time rates (at least 30Hz). On top of these data, computer vision algorithms were developed and packed into skeleton tracking middlewares to return the 3D position of user joints in each frame. We used the official OpenNI wrapper for Unity, along with NITE by PrimeSense. Kinect skeleton control is given by analyzing the position, velocity or acceleration of human joints.

A. Control definition with Unity and OpenNI

Game control in Unity is implemented by defining so called axes, that can respond to different input events such as a key press, mouse movement or in our case a human gesture; their state is represented by a floating point value. For example, the state of the axis corresponding to jumping in the game may be set to one when a corresponding input (e.g. key press or a jump gesture) was used and zero otherwise. As another example, the "camera rotation" axis may have both negative and positive values determining the angular speed and direction of the rotation. The OpenNI wrapper for Unity was designed to delegate the call of each event handler assigned to the axes to the built-in methods of Unity input with the same axis, after processing the Kinect related controls. This allows the development of games with both traditional keyboard–mouse and Kinect control enabled. However, this design assumes that the game always has Kinect plugged in, therefore, to enable seamless inclusion of Kinect control into an existing game we introduced an additional adapter layer that can select between Kinect and keyboard-mouse control depending on whether Kinect was successfully connected to the computer. The cost of the updating procedure is proportional to the number of gestures desired.

B. Navigation and interaction in the Legends of Girona game

Since Legends of Girona is a first/third person game, the controls obviously need to support navigation in the virtual world (e.g. moving back and forward, rotate, jump, etc.). Additionally, to interact with the virtual environment, the "use" command has to be implemented. Additional controls such as open the map of the scene or open the quest log may be defined to increase gaming experience. Fig. 5 shows the controls.

1) Standard controls

A common way to implement navigation by gestures is to use one of the hands as the "tracking device" (like a mouse). A reference position is defined w.r.t. a fixed joint, such as the torso, and relative offset of the hand (or any other tracked joint) w.r.t. the reference position can define the value of the axis. As an example, movement of the hand left or right from the reference position can be assigned to rotation of the camera along the vertical axis left or right, respectively. We used this approach in our first control set, extended with looking up and down (hand movement upwards and downwards, respectively) and moving forward and backward (moving the hand forward or backwards). Since the distance from the reference point is a continuous amount, floating point values can be returned resulting in a smooth and gradual control, e.g. pointing completely forward may trigger sprint. OpenNI directly supports this type of control. However, it is sensitive to even the smallest movement of the detection joint, which might as well arise from the error of the skeleton tracking, changing from frame to frame resulting in unwanted oscillations. Thus, we re-implemented the hand control by defining a small dead zone around the origin where control is disabled and additionally, we require the hand to be raised to turn on hand-guided navigation. The rest of the actions are controlled with the other hand. Jump is triggered by pointing upwards, while interaction with a game object requires the user to point towards the Kinect.

2) Exergame

Another, more interesting but also more tiring approach

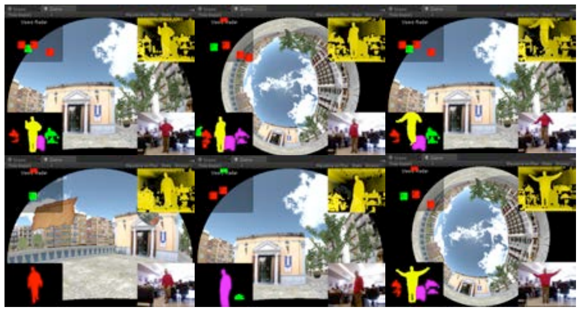

Figure 5. Kinect controls (hand, use, jump, step, turn, map)

to navigation is to require the user to mimic the action that he wants his virtual avatar to perform. More specifically, moving forward is triggered by stepping in place, jump requires the user to jump, sprint is controlled by stepping faster, rotation is given by the rotation of the user and opening the map requires the arms to be spread (similarly to holding a very big map in our hands).

Stepping is tracked by setting a minimum speed constraint on the user's legs, similarly to sprint, which has a higher speed threshold. Jump requires the torso to move upwards. The amount of rotation is calculated by checking the vector between the shoulders against the horizontal direction of the Kinect.

C. CPR evaluation with motion capture in LISSA

One of the features of the LISSA serious game is to teach Cardiopulmonary resuscitation (CPR) to users. Proper CPR is described by five criteria [14], among which two can be easily implemented using human motion capture, namely: correct arm pose and correct rhythm. CPR must be performed with straight arms, which is described by basic geometric constraints defined on the position of shoulders, knees and hands of the trainee. The rhythm of CPR is determined by analyzing the direction of the motion of the hands: a new compression is triggered when the vertical component changes from "down" to "up". Both criteria can be implemented as axes in Unity, which allows seamless integration of Kinect control into the game.

IV. CONCLUSIONS AND FUTURE WORK

We have described the modifications required to use full-dome and partial-dome immersive devices in four serious games, using a Kinect to control the game. A library of components to ease the porting of other games to these devices has been created. To the best of the authors' knowledge, this is the first serious game to use domes and immersapods.

ACKNOWLEDGMENT

This work has been supported by the research projects coded TIN2010-21089-C03-01, IPT-2011-1516-370000 and IPT-2011-0885-430000 (Spanish Commission for Science and Technology), and by grant 2009SGR643 (Catalan Government). We would also like to thank the support of the BLOOM Centre 3D i Tecnologies Emergents, Girona.

[1] R. García, J. Gumbau, L. Szirmay and M. Sbert, Updated GameTools: Libraries for Easier Advanced Graphics in Serious Gaming, Asian-European Workshop on Serious Game and Simulation, 25th Annual Conference on Computer Animation and Social Agents, 2012

[2] V. Wattanasoontorn, I. Boada, C. Blavi and M. Sbert, The framework of a LIfe Support Simulation Application. 4th International Conference on Games and Virtual Worlds for Serious Applications. 2012

[3] Unity - Game Engine. "http://www.unity3d.com/" (2013), accessed 18 February 2013

[4] Inflatable Domes. "http://www.immersiveadventure.net/pdf/en_quim.pdf" (2013), accessed 18 February 2013

[5] Immersive Adventure. "http://www.immersiveadventure.net/" (2013), accessed 18 February 2013

[6] Immersapod. "http://www.immersivedisplay.co.uk/immersapod.html" (2013), accessed 18 February 2013

[7] V1 Racing simulator. "http://www.immersivedisplay.co.uk/v1racing.php" (2013), accessed 19 February 2013

[8] Full Dome Game Experience Example. "http://vimeo.com/15849286" (2013), accessed 19 February 2013

[9] Flight simulator. "http://www.youtube.com/watch/?v=QPzsMbDBEoU" (2013), accessed 19 February 2013

[10] P.D. Bourke, idome: Immersive gaming with the unity game engine. In: Proceedings of the Computer Games & Allied Technology 09 (CGAT09). pp. 265–272 (2009)

[11] P.D. Bourke, S. Hwy, and D. Q. Felinto, Blender and immersive gaming in a hemispherical dome. GSTF International Journal on Computing (JoC) 1(1) (2010)

[12] T. Schou and H. J. Gardner: A wii remote, a game engine, five sensor bars and a virtual reality theatre. In: Proceedings of the 19th Australasian conference on Computer-Human Interaction: Entertaining User Interfaces. pp. 231–234. OZCHI '07, ACM, New York, NY, USA (2007)

[13] Microsoft, Introducing Kinect for Xbox 360. "http://www.xbox.com/en-US/KINECT" (2012), accessed 26 December 2012

[14] European Resuscitation Council Guidelines 2010. "http://www.cprguidelines.eu/2010/" (2010), accessed 13 March 2013

```
algorithm DomeMaster (float fov, float verticalRot) {
  /* Coordinates range from (0,0) to (1,1) */
  Translate coordinates so (0,0) is screen center.
  Scale coordinates to [-1,1].
  Scale coordinates by aspect ratio of the image.

  if (coordinates outside of maximum circle)
              return black.

  vector=Apply azimutal equidistant projection with
        angle fov.
                    (over vertical direction)
  Apply verticalRot (vector).

  return Cubemap (vector).
}
```

Figure 10. Pseudocode for Dome master

```
algorithm ImmMaster (float fov, float Immangle,
        float verticalRot, float forwardRot,
        float[2] maskCenter, float maskradius) {
  /* Coordinates range from (0,0) to (1,1) */
  Center=(1/2, 1- pi/2 / (pi/2 + Immangle))
  ScalingFactor=1 + Immangle / pi/2
  Translate coordinates so Center is screen center.
  Scale coordinates using ScalingFactor.
  Scale coordinates by aspect ratio of the image.

  if (coords outside circle at Center with unit radius
    or coords inside circle at maskCenter with
    maskradius)
                return black.
  vector=Apply azimutal equidistant projection with
        angle fov.           (over forward direction)

  Apply verticalRot (vector).
        Apply forwardRotation (vector).
        return Cubemap (vector).
}
```

Figure 11. Pseudocode for Immersapod master