A Data-Driven Entity-Component Approach to Develop Universally Accessible Games

Franco Eusébio Garcia and Vânia Paula de Almeida Neris

Federal University of Sao Carlos (UFSCar), Sao Carlos, Brazil
{franco.garcia,vania}@dc.ufscar.br

Abstract. Design and implementing accessible games can be challenging, particularly when the designers wish to address different interaction capabilities. Universally-Accessible Games (UA-Games), for instance, follow the principles of the Design for All, aiming to enable the broadest audience as possible to play. Although there are papers regarding the design of UA-Games, the implementation can still be challenging. This paper presents a flexible and extensible approach to implement an UA-Game. The approach relies in a data-driven and component based architecture to allow game entities to be created, managed and customized during run-time. Doing so, it is possible to change the behavior and presentation of the game whilst it is running, allowing the game to adapt itself to better address the interaction needs of the user. Furthermore, being data-driven, it is possible to create and customize user profiles to address specific interaction requirements.

Keywords: Universal Design, Game Accessibility, Universally-Accessible Game, Game Design, Game Development.

1 Introduction

Digital games importance increases every day, with their usage ranging from leisure and entertainment to learning and even healthcare. Gaming is becoming more social – users are sharing experiences, playing together and even creating new content for their favorite games. However, despite the growing importance, many people are still unable to play – for instance, due to a disability.

Playing a game requires many different users' sensory, cognitive and motor abilities [1–4]. Most of the required abilities to play are common to average users – users which belong to the standard deviation of a normal distribution of users [5, 6]. If, however, a user lacks a required ability, playing the game becomes harder or even impossible. This is often the case for users with disabilities: if a user cannot receive the game's stimuli or is unable to determine or provide a response, his/her overall playing experience is hindered.

There are different approaches to improve game accessibility. Some approaches try to address specific interaction requirements for a particular user group or disability – such as audio games for visually impaired users [4]. Other approaches aim to be accessible to as many users as possible. Universally-Accessible Games (UA-Games)

C. Stephanidis and M. Antona (Eds.): UAHCI/HCII 2014, Part II, LNCS 8514, pp. 537–548, 2014.

[1, 7] are an example of the latter: UA-Games follow the principles of the Design for All [8], aiming to enable as many users as possible, regardless of their (dis)abilities, to play.

In order to support the range of requirements and users abilities, accessible games must be flexible from its design to its implementation. For UA-Games, some of the design strategies include abstract design, polymorphic physical specialization and user profiles [1]. These strategies allow the designers to tailor the game interactions in order to address the interaction abilities and capabilities of a group of users. This way, it is possible to enable – or, at least, to improve – the game experience for users encompassed by the available interaction profiles.

As the tailored interactions may range from audio subtitles to completely different game presentations, the game architecture should be flexible enough from the start. Otherwise, implementing a new profile might require changes to core modules, potentially increasing the required development efforts and costs. It is, therefore, necessary to decouple all the logic (such as the game's rules and mechanics) from the presentation of the game without compromising the ease of creating new tailored interactions.

This paper presents a flexible and extensible approach to implement accessible games – in special, focused in the tailoring of UA-Games. It relies in an entity-component and data-driven architecture to allow the customization of game entities during run-time. This allows the game to adapt itself to better address the interaction needs of the user and, being data-driven, eases the creation and improvements of user profiles.

This paper is organized as follows: Section 2 briefly discusses game accessibility strategies and design. Section 3 discusses limitations of traditional game entities definition in games. Section 4 presents a more flexible approach to defining games entities and its application to UA-Games. Section 5 discusses how to enhance the approach using a data-driven architecture and how to apply it to UA-Games profiles. Section 6 discusses some limitations of the approach. Finally, Section 7 presents the conclusions and possibilities of future work.

2 Game Accessibility Overview

In 2004, the International Game Developers Association (IDGA) published the white-paper "Accessibility in Games: Motivation and Approaches" [2]. More recently, Yuan *et al.* [4], the "AbleGames Foundation" [9] and Barrie *et al.* [10] further discussed the theme. These works debate the importance of accessibility, describes common types of disabilities and discusses how some games approached them. According to them, there are two main strategies to creating accessibility games: supporting one specific disability or supporting many disabilities. Most of the studied games do the former: they focus on a specific disability and try to create the most accessible, compelling and interesting gaming experience for users with that disability. [2] and [4] discusses strategies employed by various games to achieve the best results for their target users.

The second approach tries to address multiple disabilities. As interaction needs may vary, designing games for everyone can be challenging or, depending on the desired gameplay, even impossible. Thus, the goal of this approach is to enable a game to have the broadest audience as possible [9].

As indicated in Section 1, UA-Games are an example of the second approach. These games combine many of the strategies of accessible games to support as many users as possible. Some interesting UA-Games include Universal Tic-Tac-Toe, UA-Chess, Access Invaders and Terrestrial Invaders [1]. These games offer different polymorphic specializations, tailoring the game to suit the abilities of visually, hearing and motor impaired users.

2.1 Game Stimuli

In [4], Yuan *et al.* describe an interaction model for games. The model describes the required sequential steps a user performs to play a game: (1) receiving stimuli, (2) determining the response and (3) providing input to the game. The model is cyclic, going from (1) to (3) and restarting at (1). The authors state that the subsequent steps rely on each other, *i.e.*, if a user cannot receive the stimuli, he/she will be unable to perform the other step and, therefore, to play.

Yuan *et al.* categorize the stimuli in two groups: primary stimuli and secondary stimuli. Primary stimuli is essential to the game comprehension – the case of graphics in most games. Therefore, if a user is unable to perceive or understand a primary stimulus, he/she will be unable to play. Secondary stimuli are usually complementary to a primary stimulus, such as sound effects in most game, and, therefore, not essential to the comprehension. Thus, if a user is unable to perceive it, he/she might be able to play, albeit with a minor loss in the experience.

2.2 UA-Games Design Framework: The Unified Design

The UA-Games mentioned in Section 2 were design using the Unified Design [1, 7]. According to Grammenos *et al.* [1], the Unified Design is a framework which offers a "step-by-step, top-down approach, starting with a high-level abstract task definition process, leading eventually to the creation of a complex, but well-structured, design space, populated by numerous interweaved physical designs".

The framework describes how to design a game in an abstract level, eliminating references to physical-level interactions (such as interactions with input and output devices). The designers tailor the physical-level interactions for each disability they want to support in a posterior step – the polymorphic physical specialization. In this step, the designers choose and tailor the primary and secondary stimuli of the game to suit the interaction needs of a disability.

The tailored interaction can be grouped in a user profile. When a user plays the game, he/she chooses the best profile for his/her needs. The chosen profile determines how the game is presented to the user and how he/she interacts with the game.

Although guiding the design, the Unified Design leaves the implementation to the designer. Thus, hoping to contribute with the implementation of UA-Games, the next sections describe a flexible and extensible approach to the implementation[1].

3 Object-Oriented Entity Hierarchies

The Object-Oriented Programming (OOP) is one of the most used paradigm in software [11]. In the game industry, it is not different: C++, C# and Java are among the most used programming language [12] – and all of them favor OOP. As the implementation of games usually employs the OOP paradigm, game entity implementations are often class hierarchical. They define a base class containing the common data for all entities (such as the object's world position and orientation) and derived classes extending the base class functionality.

Human perception is mostly visual [13]. As such, the primary stimuli of most digital games are also visual – hence the word videogame. Thus, it is usual consider the visual representation as part of a game entity and, for convenience, near the root of the hierarchy, as most entities shall have one (*e.g.* Fig. 1).

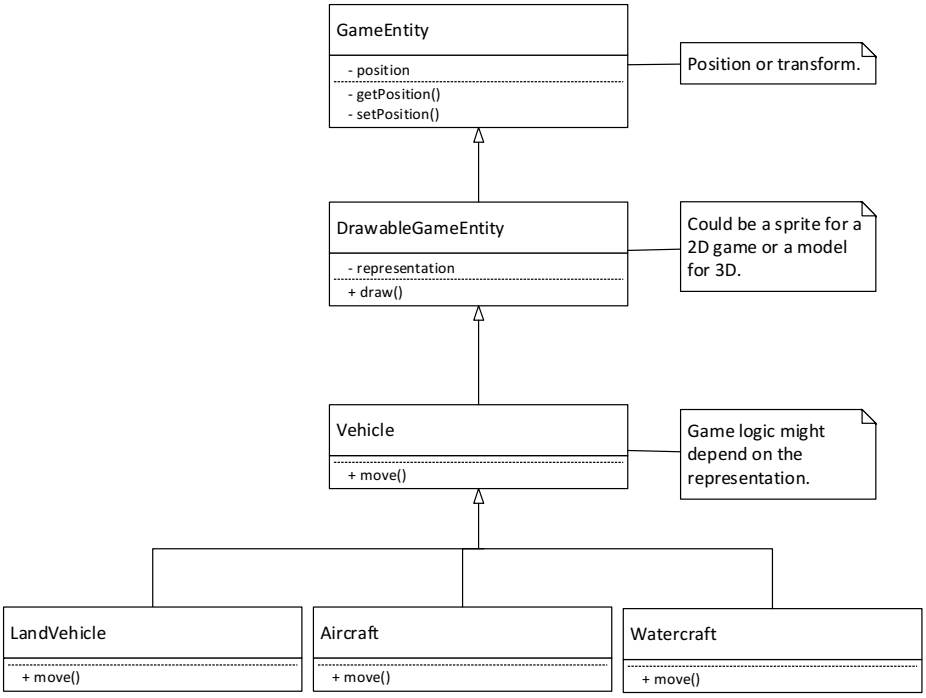

Fig. 1. A hierarchical approach for game entities. Creating a hybrid vehicle class (such as a land and air vehicle) would require multiple inheritance or code duplication.

[1] A reference implementation of the approach is available at <http://lifes.dc.ufscar.br/>.

For a UA-Game, however, the entity representation can greatly. For a certain disability, the best representation might be an image; for another, a sound or a haptic stimulus. The best form to convey the representation varies as well; therefore, it is difficult to create a general enough abstraction.

It could be argued to implement the representation at the bottom of the hierarchy. However, this would compromise reusing existing specializations when they are appropriate for multiple cases. Where would a multisensory representation fit? The new class either would inherit from multiple classes (causing the diamond problem – refer to Fig. 1 caption for an analogy) or would require duplicating existing code.

Whilst some languages have constructs to deal with the issue (*e.g.*, virtual inheritance in C++), there is a more flexible way of solving the problem: using an entity-component approach.

4 An Entity-Component Approach to UA-Games

An entity-component[2] approach aims to decouple the characteristics and functionality of a game entity into smaller, self-constricted components[3] [14–20]. It has been used in various games, promoting faster prototyping, iteration times and content creation [17, 19]. In addition, as discussed in this section, it can contribute to developing accessible games.

The approach can be seen as an extreme case of favoring "composition over class inheritance" [21] – has-a is preferred to is-a relationship. The entity can be very simple – as simple as an identifier in some implementations. This way, an entity definition is not rigid nor static: its data and behavior only depends on (and are defined by) the components attached to it.

It is possible to attach to or detach components from an entity during run-time – thus, it is even possible to customize an entity while the game running. In a UA-Game context, this means it is possible to switch from one polymorphic physical specialization to another by switching physical-level interaction components whenever it is appropriate.

Depending on the implementation, components can be data-only or have data and logic. Data-only components are processed by appropriate game subsystems. For instance, a Transformable Component can store the position, scale and orientation of an entity. A Physics or Movement system may process and manipulate these data.

Components with data and logic can also perform data processing themselves, *i.e.*, the component can update its own logic or state. Although this can appear flexible, it may cause problems when a component depends on other components. Besides, for the purposes of this paper, it may be more appropriate to have a very specialized subsystem than a component – keeping the component simple helps it being agnostic.

---

[2] The name and implementation of the approach may vary according to the author. Some variations include entity systems, component based systems, actors model and properties. Their goal, however, is similar.

[3] The components, therefore, should not be confused with Commercial of the Shelf components (COTS).

The data structure utilized in an entity-component implementation also varies – it depends on the chosen language and its available paradigms or on how often the game attach or detaches components from entities. Common choices include the use of collections or in-memory databases. The former, usually implemented in OOP languages, define a component interface. It acts as a handler to all the components – new components inherit and implement the interface, adding their data members[4]. The latter uses database facilities to store the data-only components.

We have chosen the actor's model defined by McShaffry and Graham's for this paper [17]. Their model uses a (mostly) data-only component implemented with a collection. **Fig. 2** illustrates a contextualized adaptation of their model, using data-only components and changing the focus to game accessibility.

In Fig. 2, all components implement the Component interface directly. If the designers create non-physical-level components to handle the game logic by themselves, then the game logic shall not depend on output data. For instance, a Transformable, a Collidable and a Kinematic component can hold the data for the game physics. The Physics subsystem process and manipulate these components and updates the game logic accordingly.

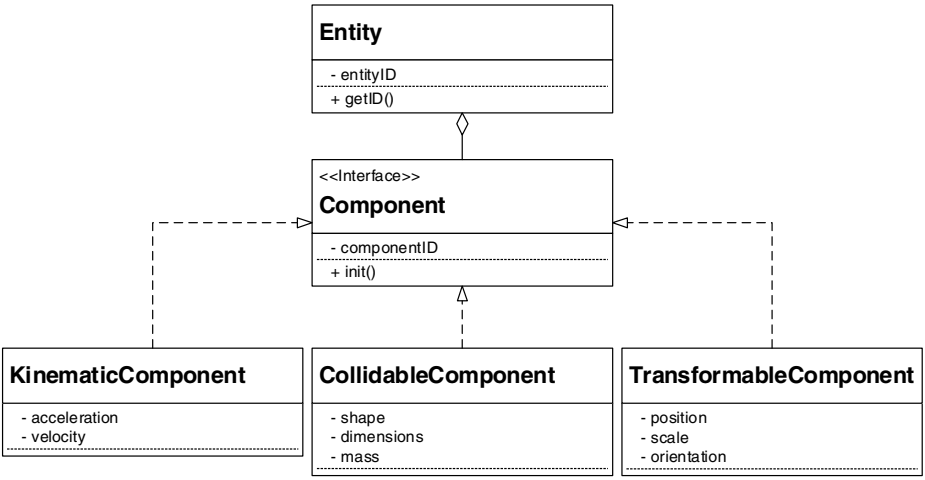

Fig. 2. An entity-component approach using data-only components

With this approach, implementing a new physical-level specialization consists of creating and specializing physical-level components, such as stimuli components. Thus, if the designers want graphical, auditory and haptic output for different polymorphic physical specialization, they create a Graphics, Audio and Haptic subsystem and their respective components. It is also possible to create components or subsystems considering the primary and secondary stimuli for each polymorphic physical specialization – as illustrated in Fig. 3.

[4] This is necessary in strong-typed languages to offer a common handle to the components.

When the game is running, the modal presentation will depend on the components attached to an entity. The game subsystems will read the data from the components and convey the information to the user as defined by the designers – resulting in an accessible gaming experience. If the input is also abstracted (for instance, into game commands), then it is possible to create a fully non-physical-level game logic using only components.

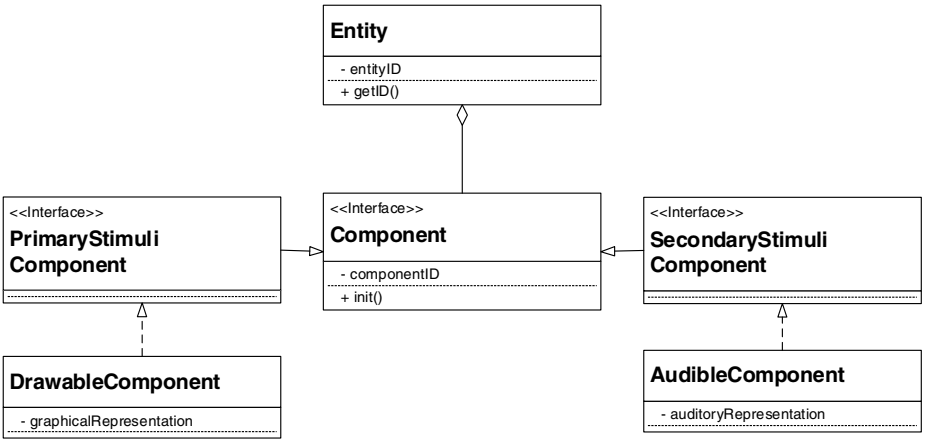

Fig. 3. Defining a stimuli component

To illustrate the approach, one can design an entity-component game - for simplicity, let us consider a Ping-Pong game. The ball of a Ping-Pong must move, collide with a table and paddles and be displayed to the user. Using the components of Fig. 2, the movement data could be stored in a Transformable component; the material, shape, inertia and mass of the ball in a Collidable component; and the velocity and acceleration of the ball in a Kinematic component. The Movement and Physics subsystem manipulates these components and update the ball's position and velocity.

Up to this point, there is no mention of a physical-level interaction: it can define to suit the user's abilities. For instance, for sighted users the primary stimuli component could be a Drawable component (such as the one from Fig. 3). The Graphics system would use its data along the Transformable component position, orientation and scale to draw the ball into the screen. For a visually impaired user, instead, the primary component could be an Audible one. This time, the Audio system would gather the required data from the components and play the sounds to the user. As one can see, the differences in the presentation are only due the chosen components – nothing else changes. Fig. 4 illustrates the example.

5 Improving the Approach with a Data-Driven Architecture

Section 4 discussed how entities and components contribute to make a game more flexible. However, using an entity-component approach alone either requires the developers

to hardcode the profiles or the user to choose the desired components when the game is running. It would be more interesting to allow the creation and customization of user profile from an outer data source – using a configuration file, for instance.

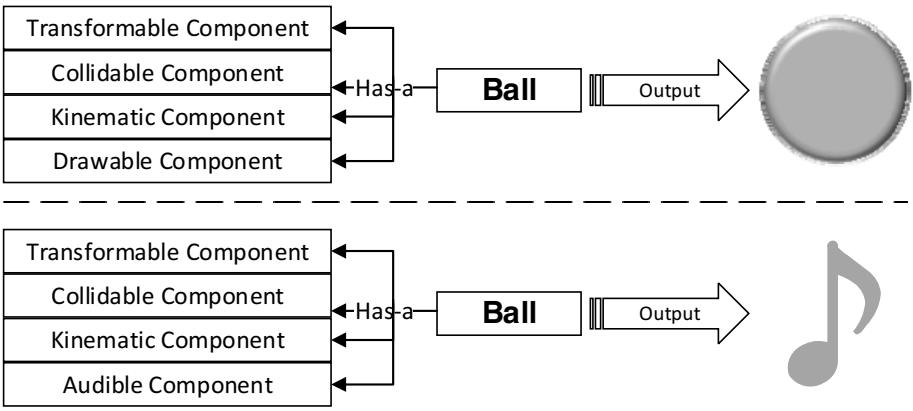

Fig. 4. If the game logic uses only non-physical-level interaction, it is possible to apply polymorphic physical specialization by changing physical-level interaction components

The use of data-driven architectures provides a way to separate game-specific code from application or engine code. Instead of hardcoding all possibilities into the game, the game-specific settings are defined in an outer data source. This contributes to increasing the flexibility and decreasing time for adjusting and tweaking settings for the entity – especially in case of a compiled programming language. Data-driven architectures share many of the advantages of entity-component approaches; they are also useful alongside game tools, such as editors, and for user generated content creation.

Combining an entity-component approach with a Factory pattern [21], it possible to load and create the components according to their definition from a data file [17, 19, 22]. For this paper, we once again choose McShaffry and Graham's implementation [17]. For each game entity, the designers define an Extensible Markup Language (XML) file containing all the entity's components and their initial data. **Listing 1** shows an example of the ball from the Ping-Pong game of Section 4. The game implementation parses the file and create the desired entity, with its respective behaviors and presentation data defined by the components, during run-time.

In the given example, the XML file contain both physical-level and non-physical level interaction. From a reuse perspective, it would be more interesting to split the file in two: one file for the physical-level components and another for the non-physical. From an accessibility perspective, however, the authors found it useful to replicate the data for some components. This way, it is possible to override the default values, allowing allows the designers to tweak the game's parameters to suit different interaction requirements.

For instance, as sound is perceived by time, it could make sense to raise the speed of the paddle for a non-visual profile in a Ping-Pong game. Likewise, in a low-vision profile, it could be useful to scale and increase the size of graphical components.

Listing 1. The components which will be used by the entity

```
<?xml version="1.0" encoding="UTF-8"?>
<Actor type="Ball" resource="entity/ball.xml">

  <TransformableComponent>
    <Position x="-85.0f" y="0.0f" z="0.0f"/>
    <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>
    <Scale x="1.0f" y ="1.0f" z="1.0"/>
  </TransformableComponent>

  <KinematicComponent>
    <Velocity vx="1.0f" vy="-1.0f" vz="0.0f"/>
    <Acceleration ax="0.0f" ay="0.0f" az="0.0f"/>
  </KinematicComponent>

  <CollidableComponent shape="sphere"/>
    <Density d="glass"/>
    <Material m="normal"/>
    <Mass m="1.0f"/>
    <Radius r="1.0f"/>
  </CollidableComponent>

  <PrimaryStimuliComponent>
    <DrawableComponent>
      <MeshFileName file="graphics/sphere.mesh"/>
      <MaterialFileName file="graphics/ball.material"/>
    </DrawableComponent>
  </PrimaryStimuliComponent>

  <SecondaryStimuliComponent>
    <AudibleComponent>
      <SoundFileName m="sounds/ball.wav"/>
      <SoundSettings volume="1.0f" loop="true"/>
    </AudibleComponent>
  </SecondaryStimuliComponent>

</Actor>
```

Using a data-driven architecture allows the creation of new profiles by mixing and matching the available physical-level interaction components and tweaking the

attributes values (Listing 2). This way, it is easier to create presets or default configurations for different polymorphic physical specialization.

Listing 2. As components can be attached to entities during run-time, each player can create his/her own custom profile

```
<?xml version="1.0" encoding="UTF-8"?>
<PlayerProfiles type="Profiles" re-
source="config/player_profile.xml">

  <PlayerProfile name="Average User: Default">
    <PrimaryStimuliComponent type="graphical"/>
    <SecondaryStimuliComponent type="aural"/>
    <Settings re-
source="config/average_user_default.xml"/>
  </PlayerProfile>

  <PlayerProfile name="Visually Impaired: Blind">
    <PrimaryStimuliComponent type="aural"/>
    <SecondaryStimuliComponent type="none"/>
    <Settings resource="config/blind_default.xml"/>
  </PlayerProfile>

  <PlayerProfile name="Visually Impaired: Low Vision">
    <PrimaryStimuliComponent type="aural"/>
    <SecondaryStimuliComponent type="graphical"/>
    <Settings resource="config/low_vision_default.xml"/>
  </PlayerProfile>

  <PlayerProfile name="Mary">
    <PrimaryStimuliComponent type="aural"/>
    <SecondaryStimuliComponent type="graphical"/>
    <Settings resource="save/mary_profile.xml"/>
  </PlayerProfile>

</PlayerProfiles>
```

6 Limitations of the Approach

The XML listings of Section 5 use only one resource for the stimuli component. The use of more resources is encouraged – and, many times, even required.

As the perception of stimulus varies depending on the human sense, only one representation may not be enough to fully convey the game information to the user. For instance, vision and hearing are very different senses. One model might be enough to represent graphically a game character; a single sound, however, is often not enough.

A possible solution to this problem is using the approach with an event based architecture – for instance, as described by the authors in [23]. This way, it is possible to use stimuli components alongside with events to convey more information to the user – which is even more interesting with scriptable events.

7 Conclusions and Future Work

While creating an accessible game, especially when considering very different interaction requirements for a UA-Game, both the design and implementation must be flexible. This paper presented the combination of a data-driven architecture with an entity-component to decouple game logic from game presentation.

With the approach, it is possible to create new presentations for the game without changing the game logic. Due to the nature of components, it is possible to choose and modify the presentation of the game during run-time. In addition, with a data driven-architecture, it is easier to create new profiles to suit different interaction abilities – which is essential for UA-Games.

As suggested in Section 5, by using a data-driven architecture with components, users can customize the game and create their own profile based on existing specializations provided by the designers (*c.f.* Listing 2). Potentially, this could allow designers to create accessible game editors, allowing users themselves to modify or create new content to the game – thus, turning people originally excluded from playing game into aspiring game designers.

Towards this goal and hoping to enable more people to enjoy digital games, the authors are currently working on an open-source game engine to aid and ease the development of UA-Games. More information is available at <http://lifes.dc.ufscar.br/>.

Acknowledgment. We acknowledge the financial aid from FAPESP (2012/22539-6) for the realization of this work.

References

1. Grammenos, D., Savidis, A., Stephanidis, C.: Designing universally accessible games. Mag. Comput. Entertain. CIE - Spec. ISSUE Media Arts Games 7, 29 (2009)
2. International Game Developers Association: Accessibility in Games: Motivations and Approaches (2004)
3. McCrindle, R.J., Symons, D.: Audio space invaders. In: International Conference on Disability, Virtual Reality and Associated Technologies, Alghero, pp. 59–65 (2000)
4. Yuan, B., Folmer, E., Harris, F.: Game accessibility: a survey. Univers. Access Inf. Soc. 10, 81–100 (2011)
5. Fischer, G.: Meta-design: Expanding boundaries and redistributing control in design. In: Baranauskas, C., Abascal, J., Barbosa, S.D.J. (eds.) INTERACT 2007. LNCS, vol. 4662, pp. 193–206. Springer, Heidelberg (2007)
6. de Almeida Neris, V.P., Baranauskas, M.C.C.: Interfaces for All: A Tailoring-Based Approach. In: Filipe, J., Cordeiro, J. (eds.) ICEIS 2009. LNBIP, vol. 24, pp. 928–939. Springer, Heidelberg (2009)

7. Grammenos, D., Savidis, A., Stephanidis, C.: Unified Design of Universally Accessible Games. In: Stephanidis, C. (ed.) HCI 2007. LNCS, vol. 4556, pp. 607–616. Springer, Heidelberg (2007)
8. Story, M.F., Mueller, J.L., Mace, R.L.: The Universal Design File: Designing for People of All Ages and Abilities. Revised Edition (1998)
9. AbleGamers Foundation: Includification - Actionable Game Accessibility, http://includification.com/
10. Ellis, B., Ford-Williams, G., Graham, L., Grammenos, D., Hamilton, I., Lee, E., Manion, J., Westin, T.: Game Accessibility Guidelines: A straightforward reference for inclusive game design, http://www.gameaccessibilityguidelines.com/
11. TIOBE Software: Tiobe Index, http://www.tiobe.com/index.php/tiobe_index
12. Sweeney, T.: The Next Mainstream Programming Language: A Game Developer's Perspective. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 269–269. ACM, New York (2006)
13. Parker, J.R., Heerema, J.: Audio Interaction in Computer Mediated Games. Int. J. Comput. Games Technol. 2008, 1–8 (2008)
14. Gregory, J.: Game Engine Architecture. A K Peters (2009)
15. Gamadu.com: Artemis Entity System Framework, http://gamadu.com/artemis/
16. Fox, M.: Game Engines 101: The Entity/Component Model, http://www.gamasutra.com/blogs/MeganFox/20101208/6590/Game_Engines_101_The_EntityComponent_Model.php
17. McShaffry, M., Graham, D.: Game Coding Complete, 4th edn. Course Technology PTR (2012)
18. Nystrom, R.: Component, http://gameprogrammingpatterns.com/component.html
19. Bilas, S.: A Data-Driven Game Object System. In: Game Developers Conference (2002)
20. Pallister, K.: Game Programming Gems 5. Charles River Media, Hingham (2005)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)
22. Laramee, F.D.: A Game Entity Factory. In: DeLoura, M.A. (ed.) Game Programming Gems 2, pp. 51–61. Cengage Learning, Hingham (2001)
23. Garcia, F.E., de Almeida Neris, V.P.: Design de Jogos Universais: Apoiando a Prototipação de Alta Fidelidade com Classes Abstrata. In: Anais do XII Simpósio Brasileiro sobre Fatores Humanos em Sistemas Computacionais, Manaus (2013)