Guana, V., Stroulia, E. PhyDSL: A Code-generation Environment for 2D Physics-based Games.
IEEE Games, Entertainment, and Media Conference (IEEE GEM'14). October 22 - 24, 2014, Toronto. ON. Canada.

PhyDSL: A Code-generation Environment for 2D Physics-based Games

Victor Guana
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
guana@ualberta.ca

Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
stroulia@ualberta.ca

***Abstract*—Video-game design and development involves a variety of professionals working together to create games with engaging content, and an efficient and flexible software architecture. However, more often than not, video-game development environments are designed for software developers, supporting programming tasks agnostic of the needs of the non-computer experts on the team. Code-generation environments offer an alternative methodology to building families of software systems that systematically differ from each other. They provide high-level domain-specific languages that express the domain concepts and features of interest, and isolate the low-level implementation concerns, so that even non-programming experts can prototype and efficiently create software systems. In this paper we describe *PhyDSL*, a code-generation environment for the creation of mobile physics-based games in 2D. We report how we have used *PhyDSL* for the rapid prototyping of customizable and cost-effective games based on physics.**

I. Introduction

As mobile devices become intrinsic to people's everyday lives, so does casual gaming. The challenges involved in video-game development, meeting short times-to-market, deploying in evolving platforms, and answering to a heterogeneous population of game consumers [1] can be traced to basic reasons: the diversity of the teams involved in game development, and the need to explore (and prototype) multiple alternative designs before deciding which one to implement.

Video games are designed by teams with diverse skills. These teams include storytellers, artists, graphic designers, and software engineers [1]. However, more often than not, video-game development environments are designed for software developers, supporting programming tasks agnostic of the needs of the non-computer experts on the team. Callele et al. [2] identify two main stages of video-game development: *pre-production* and *production*: in the former stage developers reduce design uncertainty by defining the game story, characters, and visual effects; the latter stage involves the formalization of the game's requirements, architecture, and implementation details. Furthermore, considering the roles of the two major groups participating in both these stages, *game designers* produce the documents that contain a game concept and gameplay structure, and the *software engineers* formalize the game requirements and implement the designers' vision. Callele et al. also point out that the transition process between the *pre-production* and *production* stages is challenging due to the absence of generally accepted practices that facilitate the interaction between the two teams; *game designers* do not necessarily understand the limitations of the implementation technologies and *software engineers* often limit the designers' creative vision. Acknowledging these challenges Tang and Hanneghan [3] highlight the need of game-authoring environments to facilitate the rapid prototyping of games by non-computer experts.

Video game development is a fundamentally iterative process. In the early stages of video game prototyping, variable versions of a game need to be implemented and analyzed in order to improve its overall design [4]. Among many others, perceived gameplay difficulty, visual aesthetics, actor mechanics, and goal cohesion, are some gameplay features that need to be experimented with in order to shape games to their target audiences [5]. This iterative process is expensive and full of technical uncertainties. Developers not only have to pay attention to changing gameplay features, but also deal with evolving pieces of code that, if not properly managed, may end up diverging from the game's original software architecture. Long prototyping stages can turn the development of a simple casual game into a budget and software architecture nightmare[1].

Model-driven code-generation environments offer an alternative process for building families of software systems [6]. They rely on *domain-specific languages* to capture the semantics of the problem-domain concepts, in terms of reusable code fragments. Developers specify software systems in terms of the language, and the specifications are input to a *generation engine* that generate compilable/executable code [7] [8]. In the context of video-game design, Tang et al. [9] have studied the benefits of using model-driven engineering techniques as a means to abstract the implementation details of video games, and allow non-programming experts to prototype, and efficiently create gameplay designs.

In our work, we focus on physics-based games that represent a large segment of the casual-games and rehabilitation-games markets, including *platform*, *shoot 'em up*, *puzzle* and *maze* games, and we propose a model-driven construction environment for this broad class of gameplay designs. We have identified a number of basic common features across physics-based video games and we built *PhyDSL*, a model-driven code-generation environment that consists of a textual domain-specific language for gameplay definition, and a generation engine capable of taking gameplay specifications and translating them into executable code for Android devices.

The rest of this paper is organized as follows. Section 2 discusses the related literature and some other strategies to build games based on high-level specifications. Section 3 presents *PhyDSL*, a domain-specific language, and explains it through a game-generation example. Section 4 describes the architecture of *PhyDSL*. Section 5 presents a brief experience report on using *PhyDSL*. Finally, Section 6 concludes and describes our future research agenda.

II. Related Work

Although numerous tools exist to support the flexible prototyping of video games, relatively few academic articles have been published describing their technical implementation and evaluating for their ease of use. Let us now review some of the tools that use graphical and textual languages for the specification and implementation of video games.

Furtado and Santos [10] present SharpLudus, a code-generation environment for stand-alone action-adventure games. SharpLudus includes two domain-specific languages for defining game layouts and behaviors. It also includes a generation engine for deriving C# code supported by Microsoft's DirectX API. Focused on the concept of "rooms", SharpLudus is specifically tailored to define game "worlds" with interconnected rooms viewed from above. It also allows the specification of characters that "live or die" depending on a life counter, and NPCs (non-playable characters) with primitive AI behaviors. Although it is not clear whether the SharpLudus generation engine is built using on model-driven engineering technologies, *software engineers* have access to the generated code for post-generation edition and refinement.

Reyno et al. [11] present a prototype game-authoring tool based on model-driven engineering techniques: the tool produces C++ code from UML models extended with stereotypes. It allows developers to define the structure and behavior of platform games for PCs. Although concrete examples of game specifications are provided, the underlying language is not described in detail.

In [12], Robenalt proposes a model-driven engineering framework to develop multiplatform 3D games. The proposal envisions the usage of the Eclipse Modeling Framework (EMF) together with model-transformation technologies, in order to describe platform-independent models that capture the geometry, textures, lighting, animation, and sound of games. These high-level specifications of gameplay designs provide the basis for automatically generating code, to be executed on 3D game engines.

Karamanos et al. [13] describe a 2D graphical authoring environment to create 3D action role-playing games. It allows users with limited technical background to specify 3D spaces using 2D projections. The environment allows the designer to specify the location and attributes of the gameplay elements, and using a programmatic Java translator, it creates a rendering client capable of materializing the high-level definitions using the OpenGL-ES API.

Finally, proprietary authoring environments such as GameSalad[1], GameMaker[2], Stencyl[3], and Construct2[4] provide multiplatform publishing mechanisms integrated with visual editors that enable the creation of video games through drag-and-drop operations, action events, and advanced scripting. Although very flexible in terms of semantics designed for non-computer experts, and powerful code generation capabilities, most the proprietary authoring environments are designed to meet the requirements of a complete video game production process. Consequently, their learning curve, and the limited availability of the code that they produce, make their purpose different from quick game prototyping and testing.

III. The PhyDSL Declarative Language

PhyDSL was designed for the purpose of supporting the fast prototyping of physics-based games, including *platform*, *shoot 'em up*, *puzzle* and *maze* games. This is a broad category that includes many of today's casual games.

PhyDSL has been designed as an Eclipse[5] plugin. As a front-end, the plugin provides a syntax-directed text editor (including syntax highlighting and text completion), supported by a domain-specific language that defines gameplay in high-level terms describing the gameplay static and dynamic elements and their behaviors. The *PhyDSL* plugin also provides a component, ChainTracker [14], for tracing and visualizing the dependencies between the language specification, the resulting model elements and the constructed code.

Considering the three major design components for gameplay specification presented by Hunicke et al. in [15], namely *Mechanics*, *Dynamics*, and *Aesthetics*, the *PhyDSL* language consists of four gameplay definition sections: (i) *mobile and static actor definition*, (ii) *environment and layout definition*, (iii) *activities definition*, and (iv) *scoring rules definition*.

In this section, we explore *PhyDSL*'s syntax, semantics and expressive power through the specification of *"Meteorite Dodger"* a casual game with similar gameplay characteristics than Asteroids[6] (Atari, 1979). The objective of *"Meteorite Dodger"* is to continuously dodge meteorites that randomly appear on the screen and bounce during a given amount of time. The user, represented through a touchable alien avatar, has to avoid collisions with the asteroids, and, optionally, collect gems in order to obtain extra points. Once the game ends, the player's performance is scored taking into account the number of times the avatar collided with a meteorite (-30 points), the number of gems collected (+20 points/gem), and the length of the game (+5 points every 5 seconds).

A. Defining The Game Actors

Using *PhyDSL*'s *mobile and static actor definition* section, game designers specify the game actors and their physical properties. *PhyDSL* distinguishes between mobile and static actors; while mobile actors can be affected by collision and the game's environmental forces, such as gravity or acceleration

[1] http://gamesalad.com/
[2] https://www.yoyogames.com/studio
[3] http://www.stencyl.com/
[4] https://www.scirra.com/construct2
[5] http://www.eclipse.org/
[6] http://atari.com/arcade#!/arcade/asteroids/

vectors, static actors are not affected by any force. In our example (see Figure 1) mobile actors are suitable for defining the game's meteorites and the player's avatar. Static actors can be used to define layout elements, such as the wall sections that will enclose the "world" of the game, or static obstacles, or platforms for a more complex gameplay design.

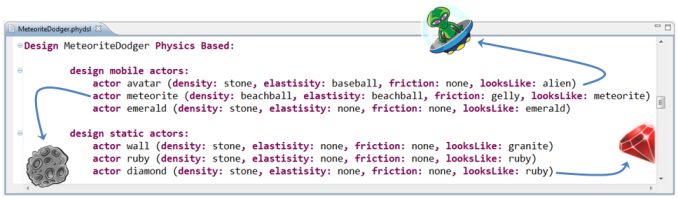

Fig. 1. PhyDSL Generated: Meteorite Dodger - Mobile and Static Actor Definition

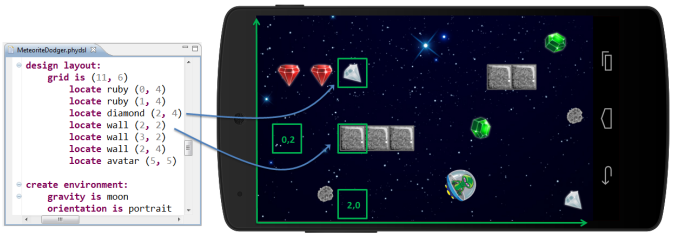

Fig. 2. PhyDSL Generated: Meteorite Dodger - Environment and Layout Definition

For both mobile and static actor definitions, physical properties such as *density*, *elasticity*, and *friction* have to be specified in order to model how the actors behave in the simulated physics environment of the game. *PhyDSL* provides a predefined library of data-types to specify physical property values. In our example, the specification of a meteorite's density is done using the constant *"stone"*, which is included in *PhyDSL* native data-type library. In this case, *PhyDSL* maps the conceptual meaning of *"stone"* to a numerical value that can be interpreted by the *to-be-generated* physics engine configuration. Such predefined data-types offer reusable implementations of domain-specific concepts with direct semantic relevance for the construction of a gameplay.

To provide more flexibility, especially for experienced programmers, *PhyDSL* allows game designers to extend data-type libraries and define their own types as restrictions of numeric, string, or boolean-based subtypes.

In our example, three types of gems are defined: rubies, emeralds and diamonds. Rubies and diamonds have predefined locations since the beginning of the game. Furthermore, they are not affected by environmental forces and serve as simple elements that the player has to capture. Consequently, both rubies and diamonds are modelled as static actors. Emeralds are defined as mobile actors that randomly appear on the screen. Section III-C showcases how, through the use of *PhyDSL*'s *activity* language concepts, actor appearance events can be modelled during gameplay design.

B. Defining The Game Environment and Layout

PhyDSL provides a 2D grid-layout system to locate actors on screen. Figure 2 portrays how, in order to create our example's gameplay, static elements such as a *"wall"* and a *"diamond"* are located on the (2,2) and (2,4) screen coordinates respectively. For each game, the dimensions of the canvas grid-layout, the screen orientation, and the gravitational force of the simulated environment need to be defined. The gravitational force can be configured in three modes. Specifying a gravity force value, in this case by using *PhyDSL*'s native type *"moon"*. Or by using the *"gyroscope"* constant, indicating that the environment will follow the device's orientation and perceived gravity for the simulation. Or using the constant *"none"*, to denote that the environment does not suffer from gravitational force influence.

In the case of *Meteorite Dodger* two platforms are defined using static elements (Figure 2). Moreover, the rubies and diamonds are located in strategical places for the player to capture. The player's avatar, defined using a mobile actor, is originally located in the coordinates (5,5). In the current version of *PhyDSL*, all mobile actors react to touch events on the screen. Thus, different strategies can be used in order to play our game example; the player can either push away the approaching meteorites, move the avatar around avoiding collisions, or combine both strategies to achieve better results.

C. Defining The Game Activities

Having described how actors are defined and placed on the screen, let us now introduce *PhyDSL*'s *activity* concept, which allows game developers to introduce interactive elements in their gameplay design. The *activity* concept is used to define the interactive gameplay elements dictated by time, such as the *appearance* and *movement* of mobile actors. They are modelled as Event - Condition - Action rules, associated with a given actor. Specifically, two types of rules can be modelled in the form of *"When <timer condition> actor <actor ID> moves <MoveProperties>"* and *"When <TimerCondition> actor <actor ID> appears <AppearanceProperties>"*.

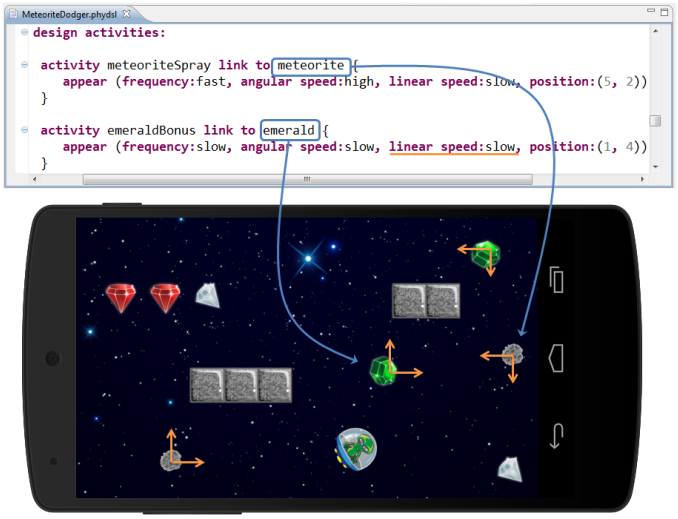

Fig. 3. PhyDSL Generated: Meteorite Dodger - Activity Definition

In our *Meteorite Dodger* example, two activities are specified (see Figure 3). The goal of the first activity, called *meteoriteSpray*, is to create the obstacles that the player has to avoid colliding with. Appearance rules have three

configuration parameters. *Frequency* determines how of often the appearance event will be triggered. The *linear* and *angular speed* specify the magnitude of the initial rotational and linear movement of the meteorites. As mentioned before, native data-type libraries can be used to define physical property values. In this case, the types *"slow"* and *"fast"*, are used to define 6- and 3-second intervals respectively. Furthermore, different game designers might opt for defining what *"slow"* or *"fast"* mean in their gameplay design. The value of these variables can be changed using *PhyDSL*'s text editor.

The goal of the second activity, called $emeraldBonus$, is to model the appearance of emeralds on the screen so the player can capture them, and obtain extra points. Unlike diamonds and rubies, modelled and placed on the screen as *static actors*, emeralds are defined as *mobile actors* and, once they appear during the execution of the game, they move around the game world.

It is important to mention here that rules that define how actors appear need a position where the appearance action will take place. Since appearance events are typically linked to mobile actors, the position determines the initial location of the actor using the grid-layout system. Gameplay designs that include elements such as cannons, particle sprays, and other type of shooters, are suitable to be modelled with this type of events. Simple static elements can be also used in scenarios such as for doors that open and close. In fact, the usefulness of activities to model gameplay elements is limited only by the designer's imagination.

D. Defining The Game Scoring Rules

A big component of any gameplay design is the ability to define scoring rules that provide feedback, and motivate the player [16]. Similarly to *activities*, *PhyDSL scoring rules* use an Event - Condition - Action structure (see Figures 4 and 5). *Scoring rules* can be triggered by *time events*, *collision events*, or *touch-screen events*. Furthermore, each *scoring rule* can result in one of three possible actions: (i) the point count of the game changes positively or negatively; (ii) the game comes to an end; and (iii) the player final state changes to either *"losing"* or *"winning"*.

In *PhyDSL*, resulting actions are not mutually exclusive and can be used simultaneously. Game examples that can be defined using *PhyDSL*'s scoring structure include titles such as Flappy Bird[7] (Gears Studios, 2013). In Flappy Bird, a clumsy bird has to avoid obstacles while flying horizontally in a 2D physics world. The player gets points every time an obstacle is avoided. In the case of a collision event between the bird and an obstacle, three actions are executed, i) the game comes to an end, ii) the final state of the game changes to *"player loses"*, and iii) the total points accumulated are reported without a change. Similarly, in Castelvania[8] (Konami, 1986), the player uses platforms to travel between two points in the "world" of the game. While travelling through each level, enemies have to be avoided or killed. In the case of a collision between the player's avatar and an enemy, i) the game does not end, ii) the player does not win or lose, but iii) his/her live points are reduced. In *PhyDSL*, having the flexibility of defining simultaneous actions as a result of gameplay events, provides the ability to model scoring rules that impact the game in different scopes.

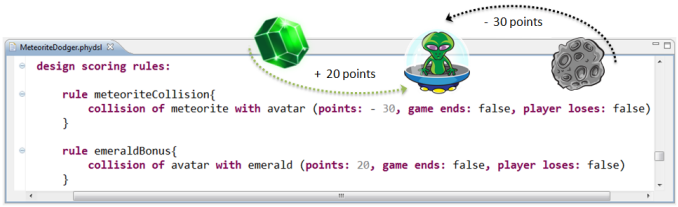

Fig. 4. PhyDSL - Scoring Rule Definition (Collision Rules)

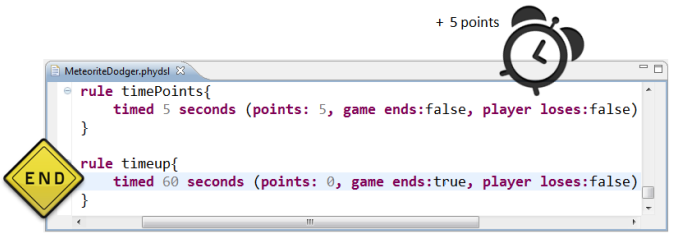

Fig. 5. PhyDSL - Scoring Rule Definition (Time Rules)

Going back to our example, in *Meteorite Dodger* collision and timed scoring rules are used to define a) how the player obtains points, b) how (s)he can be penalized after a collision with a meteorite, and c) when the game ends. Figure 4 displays the definition of two rules, namely $MeteoriteCollision$ and $EmeraldBonus$. The first rule defines that, in the event of a collision between a *"meteorite"* and the player's *"avatar"*, the point count of the game is negatively affected by a factor of 30. The second rule specifies that, in the event of the collision between the *"avatar"* and an *"emerald"*, the point count is positively affected by 20. The scoring events defined for *"diamonds"* and *"rubies"* are modelled in the same way.

Two timed rules are required to specify how the player obtains 5 points every 5 seconds as a continuous reward for surviving the falling meteorites, and to define the total duration of the game as 60 seconds (see Figure 5 rules $TimePoints$ and $TimeUp$ respectively). As we mentioned before, scoring rules triggered by touch-screen events can also be used. These are particularly useful when the gameplay design includes slicing, or cancellation gestures that trigger different actions to change the score or state of the game. Fruit Ninja[9] (Halfbrick Studios, 2010), is a good example of a game that requires touch-based scoring events. Its gameplay design is based on the user's ability to find-and-touch diverse fruits, that appear randomly on the screen, to "slice them up" and obtain points.

IV. PhyDSL Generation Architecture

The software architecture of *PhyDSL* consists of two main components: the *PhyDSL* domain-specific language editor and its generation engine. Using model-driven engineering techniques, the generation engine takes as input a game specification, and through a composition of model-transformations, refines it to produce executable code. The role of the model

[7] http://flappybird.wikia.com/
[8] http://castlevania.wikia.com/
[9] http://fruitninja.wikia.com/

transformations is to inject the execution semantics missing in the high-level definitions of the domain-specific language. More specifically, as shown in Figure 6, ATL [17] and Acceleo 3.0 [18] model-transformation languages are used to inject execution semantics in three different dimensions of the game architecture: (i) the game dynamics (i.e. environment and activities), (ii) the game scoring rules, and (iii) the game layout. Currently, *PhyDSL* allows the derivation of games for Android devices implemented using Java, and Box2D[10] physics engine as underlying technologies.

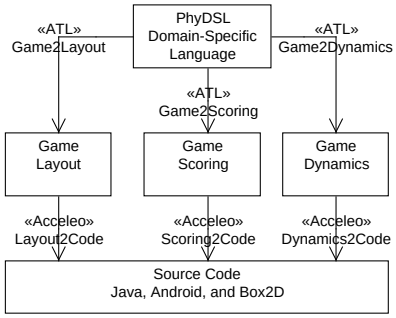

Fig. 6. PhyDSL Generation Architecture

Although the current implementation of *PhyDSL* is limited to producing Java code for the Android platform, model-driven engineering techniques allow to extend the generation environment in order to derive code for different target platforms. Section V describes our research agenda towards extending *PhyDSL* generation capabilities.

V. Experience Report

PhyDSL provides a comprehensive domain-specific language tailored to the rapid prototyping of 2D physics-based games and enables developers to define complex game designs, within this fairly broad domain, with only few high-level concepts. We have observed the flexibility and limitations of *PhyDSL* in rapid game prototyping for rehabilitation, and casual game scenarios.

We are now using *PhyDSL* to create tablet-based games in support of cognitive rehabilitation. We work closely with the University of Alberta's Department of Speech Pathology and Audiology, and the Department of Occupational Therapy, to develop a family of games as interventions that can help patients to improve from cognitive impairments suffered after brain injuries. Through the gamification of rehabilitation therapies, health professionals have been able to provide tailored experiences to patients depending on, for example, their age, gender, and medical condition. In this context, so much like therapies, games need to be quickly adapted to the needs of each patient. Such adaptations include, but are not limited to, increasing the difficulty level of the game with new gameplay actors, introducing new therapy challenges through new scoring rules, and even adapting visual or physical themes.

Regarding the two main challenges that we portrayed at the beginning of this paper, we have adopted different strategies in *PhyDSL* to tackle them.

[10]http://box2d.org/

1. Video game development is a fundamentally iterative process: Our proposal includes a generation engine that takes high-level game specifications, and automatically generates code that implements gameplay definitions. Automatic code generation has the advantage of guaranteeing software architecture consistency. This is, no matter the gameplay definition, the generation engine will always use the same software architecture, and well formed snippets of code in order to realize the designer's vision. Consequently, technical uncertainty is considerably reduced by abstracting the game implementation details into a high-level language. By doing so, our proposal allows fast game prototyping, testing, and gameplay refinement. Indeed, trade-offs have to be assumed to achieve this goal. *PhyDSL* is specifically tailored for 2D physics-based games and can't be used to, for example, prototype complex adventure games. However, using the same development technologies embedded in *PhyDSL*, other code-generation environments can be created for different game genres.

As a concrete example, in the context of rehabilitation games, we reported how *PhyDSL* has been used to translate pen-and-paper tests for cognitive conditions, into game prototypes that can be adapted to the needs of particular patients [19]. More specifically, our task was to develop a game, inspired of the *cancellation test* for visual neglect [20]; the gameplay involved a set of mobile and static actors, that have to be cancelled by the player, while ignoring several visual distractors.

In this particular case, using *PhyDSL*, we reduced in 96% the total time required to produce game prototypes. Using manual coding techniques, our development team, consisting of three occupational therapists and one software engineer, took one month to design and create the first game prototype. On the other hand, using *PhyDSL*, the total development time was of 3 days. Furthermore, during initial clinical trials, and while answering to several requirements from patients and physicians, the team created several refined prototypes after gameplay testing. In this stage, the team, using manual software development, required 16 days to add gameplay features such as new reward mechanisms like incremental point count by time, new types of actors that serve as visual distractors on the screen, and modified visual effects for screen cancellation events, among many others. Using *PhyDSL*, this modifications were performed in less than a day by simply changing the high-level specification of the game.

2. Video games are designed by teams with diverse skills: The design of *PhyDSL*'s domain-specific language targets both *game designers* and *software engineers*. Although it uses stylistic features of general-purpose programming languages, such as Java and Phyton, that appeal to experienced *software engineers*, its lexical style is defined based on the hierarchical structure of physics-based game concepts, and language semantics close to *game designers*. As we mentioned before, *PhyDSL* is a language free of execution semantics, and declarative in nature. This means that developers do not need to define control statements, which minimizes language side effects, and allows developers to describe *what* the game should accomplish, rather than *how* to accomplish it.

Using *PhyDSL*, we have observed how occupational therapists, and other individuals without programming experience, have been able to prototype and test game designs with little,

and in some cases without, intervention of software engineers. This has reduced the cost and time of producing games that need to be rapidly tested in dynamic environments such as rehabilitation therapy. In our experience developing games for cognitive conditions, we observed that after two weeks of training, occupational therapists were capable of building and adapting game prototypes on their own. However, in some cases, software engineers were required to refine features of the game that were not captured in *PhyDSL*. In one particular occasion, an experienced developer was involved to modify the way the touchscreen events were detected for elderly patients. Nonetheless, in most of the cases, prototype refinements were handled by the designers, occupational therapists and physicians. Furthermore, in many cases such adaptations were performed in the same clinical environment.

Other games, like *"Meteorite Dodger"*, have been designed by high school students with little or no programming experience. Although it is hard to quantify the completeness of *PhyDSL* due to the variety of game prototypes and requirements of physics-based games, *PhyDSL* has been able to generate between 80% to 100% of the total lines of code needed for the implementation of our proposed gameplay designs. In the future we plan to conduct formal empirical studies to understand the developer's perceived difficulty when developing games with code-generation environments such as *PhyDSL*.

VI. Conclusions

Our research agenda aims at demonstrating how domain-specific languages can make the development of video games a more systematic experience for people of different backgrounds, and more importantly, significantly more cost-effective in terms of shorter time-to-market, fewer coding errors, and agile gameplay testing. Code-generation environments promise to close the interaction gap between *game designers* and *software engineers*. They allow the former to create quick prototypes and realize their vision using high-level semantics. At the same time, they enable *software engineers* to become active participants of a game's *pre-production* stage, reducing their programming tasks to code refinements targeted to complete the designers' vision, with features not captured in the domain-specific languages, or unsupported by the generation engines.

We have developed the *PhyDSL* code-generation environment for generating source code for physics-based games on Android devices. *PhyDSL* provides a language through which developers can specify games in terms of their layouts, their static and mobile actors, and the behaviors of these actors driven by user interactions, timed events and game events (such as proximity and collision). Through our experience with the visual-neglect intervention game, we have shown that such high-level specifications are intuitive to develop (as the language terms are close to the game concepts) and can be used by non-software developers. Furthermore, we found that the language is indeed expressive enough to generate a family of game variants. We are currently in the process of extending *PhyDSL* with new features that include the definition of on-screen controls, and more flexible behaviors, such as *actor disappearance*, that can be triggered by *activities* and *scoring events*. Future work will allow *PhyDSL* to generate code for different platforms, such as iOS, using the same domain-specific language. Nonetheless, we believe that the advantages of rapid prototyping without technical uncertainty for a single platform, overcome the current limitations of our environment.

References

[1] J. Blow, "Game development: Harder than you think," *Queue*, vol. 1, no. 10, p. 28, 2004.

[2] D. Callele, E. Neufeld, and K. Schneider, "Requirements engineering and the creative process in the video game industry," in *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*. IEEE, 2005, pp. 240–250.

[3] S. Tang and M. Hanneghan, "State-of-the-art model driven game development: A survey of technological solutions for game-based learning," *Journal of Interactive Learning Research*, vol. 22, no. 4, December 2011.

[4] H. Desurvire, M. Caplan, and J. A. Toth, "Using heuristics to evaluate the playability of games," in *CHI'04 extended abstracts on Human factors in computing systems*. ACM, 2004, pp. 1509–1512.

[5] D. Pinelle, N. Wong, and T. Stach, "Heuristic evaluation for games: usability principles for video game design," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1453–1462.

[6] S. Mellor, T. Clark, and T. Futagami, "Model-driven development: guest editors' introduction." *IEEE software*, vol. 20, no. 5, pp. 14–18, 2003.

[7] K. Czarnecki, "Overview of generative software development," in *Unconventional Programming Paradigms*. Springer, 2005.

[8] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering-Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[9] S. Tang and M. Hanneghan, "State-of-the-art model driven game development: a survey of technological solutions for game-based learning," *Journal of Interactive Learning Research*, vol. 22, no. 4, 2011.

[10] A. W. Furtado and A. L. Santos, "Using domain-specific modeling towards computer games development industrialization," in *The 6th OOPSLA Workshop on Domain-Specific Modeling*. Citeseer, 2006.

[11] E. M. Reyno and J. Á. C. Cubel, "Model driven game development: 2d platform game prototyping." in *GAMEON*, 2008, pp. 5–7.

[12] S. A. Robenalt, "Mdsd for games with eclipse modeling technologies," in *Entertainment Computing-ICEC 2012*. Springer, 2012, pp. 511–517.

[13] C. Karamanos and N. M. Sgouros, "Automating the implementation of games based on model-driven authoring environments," in *Entertainment Computing-ICEC 2012*. Springer, 2012, pp. 524–529.

[14] V. Guana and E. Stroulia, "Chaintracker, a model-transformation trace analysis tool for code-generation environments," in *Theory and Practice of Model Transformations*. Springer, 2014, pp. 146–153.

[15] R. Hunicke, M. LeBlanc, and R. Zubek, "Mda: A formal approach to game design and game research," in *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2004, pp. 04–04.

[16] S. Rogers, *Level Up!: The Guide to Great Video Game Design*. John Wiley & Sons, 2010.

[17] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006, pp. 128–138.

[18] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire, "Acceleo user guide," 2006.

[19] T. Cheung, V. Guana, M. Labas, A. Lock, L. Liu, and E. Stroulia, "Computerized tablet-based cancellation assessment for spatial inattention," *Proceedings of the Canadian Association of Occupational Therapists annual conference (CAOT)*, 2013.

[20] S. Stone, B. Wilson, A. Wroot, P. Halligan, L. Lange, J. Marshall, and R. Greenwood, "The assessment of visuo-spatial neglect after acute stroke." *Journal of Neurology, Neurosurgery & Psychiatry*, vol. 54, no. 4, pp. 345–350, 1991.