A Domain-Specific Language for Augmented Reality Games

Rubén Campos-López
Universidad Autónoma de Madrid
Madrid, Spain
ruben.campos@uam.es

Esther Guerra
Universidad Autónoma de Madrid
Madrid, Spain
esther.guerra@uam.es

Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain
juan.delara@uam.es

ABSTRACT

Augmented Reality (AR) applications have become popular over the last few years, with significant impact on video games. AR does not require advanced technology, but a mobile device with a camera is enough. However, building AR games is time-consuming and requires deep expertise in the tools, technologies and programming languages of the field, as well as on mathematical concepts related to the graphics and physics of the virtual objects. We attack this problem by means of a Domain-Specific Language (DSL) named ARGDSL, tailored to create AR games. It offers primitives to customise the domain and logic of the game, the physics of the virtual objects, and their graphical representation. We provide an Eclipse environment enabling the definition of AR games using the DSL, and an iOS client able to run the defined games.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages**; • **Human-centered computing** → *Ubiquitous and mobile devices*; **Mixed / augmented reality**;

KEYWORDS

Domain-Specific Languages, Augmented Reality, Games

ACM Reference Format:
Rubén Campos-López, Esther Guerra, and Juan de Lara. 2024. A Domain-Specific Language for Augmented Reality Games. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, Article 1, 3 pages. https://doi.org/10.1145/3605098.3636111

1 INTRODUCTION

Augmented Reality (AR) [1] technologies enable the visualisation of virtual objects as part of the real world. The increasing capabilities of mobile devices and the emergence of head-mounted widgets have enabled the use of AR for all sorts of applications, from industrial settings to education, health and video games [5].

Unlike Virtual Reality (VR), AR is quite accessible as it only requires widely used hardware (smartphones and tablets, with their camera and sensors) to run this type of applications. AR is becoming increasingly popular, achieving high commercial success in entertainment and video games [7]. Prominent examples of AR games include Pokemon GO and Pikmin Bloom (both by Niantic) and the large AR ecosystem by SnapChat.

However, creating AR games with current approaches is complex, as they require high development effort, deep technical expertise, and knowledge of computer graphics and physics. To mitigate this problem, we propose a Domain-Specific Language (DSL) to create AR games, called ARGDSL (Augmented Reality Games DSL). This is a declarative, textual DSL that allows customising all aspects of an AR game, including the domain elements, their graphical representation, their behaviour and the game logic. The goal of ARGDSL is to avoid the need of programming or the use of complex AR frameworks, lowering the entry barrier to AR game development. Currently, it is focused on physically realistic skill games (e.g., labyrinths, balance games, shooters). We have built an Eclipse editor that permits defining AR games with ARGDSL, and an iOS client able to run the defined games on iPhones and iPads.

2 THE ARG DOMAIN-SPECIFIC LANGUAGE

Fig. 1 shows a scheme of our approach, which is based on model-driven principles [2]. Specifically, we have created the DSL ARGDSL to allow AR game developers to create AR games by defining the elements used in the game, their graphical AR representation, their physics, and the game logic. ARGDSL specifications conform to a meta-model, and abstract away technical low-level details of the game.

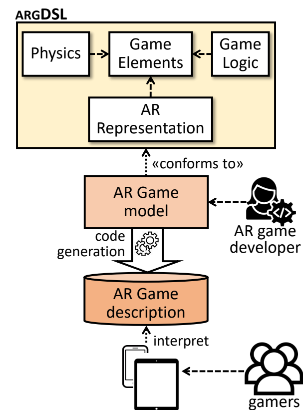

Figure 1: Our approach.

Given an AR game specified with ARGDSL, a code generator produces a low-level representation of the game, which gets uploaded into a server. This representation is then interpreted by an iOS client which runs on iPads and iPhones, so that gamers can play the game on their devices.

Next, we describe how to describe an AR game with ARGDSL using an AR football game as example. The player needs to throw penalty kicks to the goal without touching any obstacles, and with 60 seconds to score as many goals as possible.

2.1 Game elements

To create an AR game with ARGDSL, the developer needs to declare the elements of the game together with their properties and relations (i.e., the domain of the game). For this purpose, we built a meta-model inspired by the OMG's MetaObject-Facility [6].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SAC '24, April 8–12, 2024, Avila, Spain
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3636111

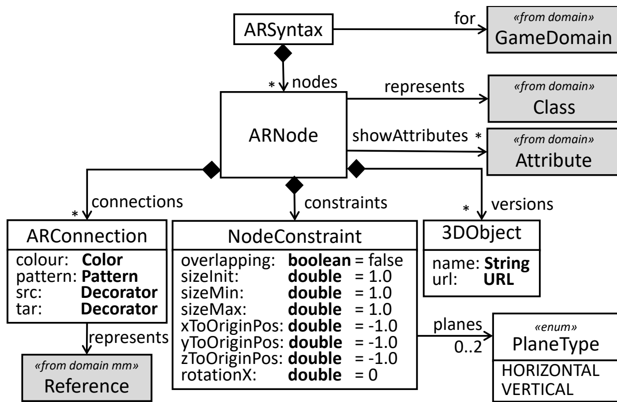

Figure 2: AR representation meta-model.

Each type of element of the game is described by a class, which may have attributes and references. In addition to typical data types (e.g., String, Int), we support images and videos, which get displayed on the object's virtual representation. The listing to the right shows the definition of the elements of the AR football game, using the textual syntax of ARGDSL. There are four types of elements: ball, net, obstacle and floor. No element defines attributes.

```
1 Game football {
2   elements {
3     ball {}
4     net {}
5     obstacle {}
6     floor {}
7   }
8 }
```

2.2 AR representation

The DSL permits assigning an AR representation to each class and reference in the game domain (cf. Fig. 2). Each class can be represented by one or several 3D objects, which can be swapped during the game. The 3D objects are described by a name and the URL of a file containing the AR image in Apple SceneKit (SCN) format. In addition, it is possible to configure the features of the AR objects (via class NodeConstraint), like their overlapping, size, the distance they can be displaced from its original position, their rotation, and the planes where they can be placed.

The listing to the right declares the AR visualisation of the ball in our AR game, using the DSL textual syntax. The ball is represented by one 3D object, can be placed on horizontal planes (i.e., atop the floor of the game) and can overlap with the other elements.

```
1  Graphics {
2    element ball {
3      versions {
4        v1 = "http://url.com/ball.scn"
5      }
6      constraints {
7        plane horizontal
8        overlaps
9      }
10   }
11   element net { ... }
12 ... }
```

2.3 Physics

Each element class may have physical information on how its objects should behave and move realistically during the game. Fig. 3 shows the meta-model for this part, inspired by Apple's ARkit.

Each element class has a physical body (PhysicBody) specifying its mass (in kilograms), electric charge (in coulombs), sliding and rolling frictions, object resistance to air (damping), rotational friction (angularDamping), and bouncing behaviour (restitution). In addition, it is possible to specify whether the movement of an element class will be affected by *collisions* with other objects, or by *forces* applied by the players to impulse the element. Specifically, the physical

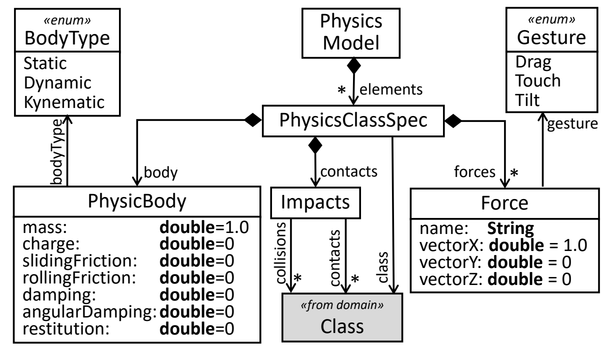

Figure 3: Physics meta-model.

body of an element class can be either *Static* (not affected by collisions or forces except gravity), *Dynamic* (affected by both forces and collisions), or *Kinematic* (affected by collisions but not by forces). Dynamic classes need to define the forces affecting them, by providing the name of the force, its magnitude in a 3-dimensional vector, and the gesture in the user interface that triggers the force. The latter can be either touching (*Touch*), sliding (*Drag*), or tilting (*Tilt*) the screen. The magnitude of the force is mandatory when touching the screen, and optional when dragging and tilting. Finally, each class can specify the game elements with which it can collide or have contact (class Impacts). Collisions affect the physic body of the element, but contacts do not. This information can be used in the game logic, as explained in next section.

The listing below defines the ball physics using the DSL. The ball is set to be a dynamic object with a mass of 0.5 kilograms and some friction forces (lines 3–10). It also declares the force kick, which gets activated when dragging on the object (lines 11–13). Finally, the ball can collide and contact with floor, obstacle and net (lines 14–17).

```
1  Physics {
2    element ball {
3      body dynamic {
4        mass 0.5
5        slidingFriction 0.5
6        rollingFriction 0.5
7        restitution 0.5
8        damping 0.1
9        angularDamping 0.1
10     }
11     forces {
12       kick : gesture drag
13     }
14     contacts {
15       collision floor obstacle net
16       contact floor obstacle net
17     }
18   }
19   element net { ... }
20 ... }
```

2.4 Game logic

A game can define the starting and winning game conditions: initial score and number of lives of the player, and final score needed to win the game. Moreover, it can display different text messages to signal the start, win and lose situations.

Games define its logic by means of actions, which comprise one of four possible basic actions (start the game from scratch, lose, win, restart the game saving the progress), or none of them. Each action may display a message, change the score, self-trigger the action in a time interval of seconds, or make changes to the game objects, namely, creating or deleting objects, modifying attribute values, applying forces, or changing object positions. The actions can be triggered when an object meets some conditions, there is a collision, or a button is pressed (class Button).

Finally, it is necessary to specify the object set-up when the game starts, indicating the objects' name, class, position, and attribute values. Objects may have associated rules, to be triggered when some attribute condition is met. Conditions can be arithmetic or logical, and their operands can be constants, attribute values, or an operation to count the number of objects of a class.

```
1  Gamelogic {
2    Display {
3      start "Game start"
4      win "You won!"
5      score {
6        start 0
7        finish * /* no limit */
8        lives 1
9      }
10   }
11   Actions {
12     gameover { /* Gameover after 60 secs */
13       action win
14       timeEach 60 /* in seconds */
15     }
16     goal {
17       score 1 /* increase score by 1 */
18       message GOAL /* present a message */
19       changes {
20         do delete fb /* fb defined in line 37 */
21         do create ball named fb at front
22       }
23     }
24     miss {
25       message MISS
26       changes {
27         do delete fb /* fb defined in line 37 */
28         do create ball named fb at front
29       }
30     }
31   }
32   Collisions {
33     element ball to net -> goal
34     element ball to obstacle -> miss
35   }
36   Elements {
37     fb : ball [0.0, 1.0, 1.0]
38     goalNet : net [0.0, 0.0, 10.0]
39     grass : floor [0.0, 0.0, 0.0]
40     ...
41 }}
```

The listing to the right shows the game logic for the running example. The Display block (lines 2–10) defines different messages for start and winning (as this game cannot be lost) and the score system. Since finish is set to *, the game has no upper score limit.

The game declares three actions: gameover (lines 12–15), goal (lines 16–23) and miss (lines 24–30). Action gameover specifies that the game is won after 60 seconds, which is controlled by a time trigger (timeEach). Action goal increases the score by 1, presents a message, and makes two changes on the game: it deletes the ball (named fb) and creates another one in the *front* position of the screen. The DSL also permits positioning objects at the *back* of the screen, or in the *default* position of the object. Creating objects requires providing their type (e.g., ball) and name (e.g., fb), and having several objects of the same type is possible. Action miss presents a message, deletes the ball and creates another in front. Next, lines 32–35 declare two collisions. In the first one, when the ball collides with the net, the goal action is triggered. In the second one, a collision of the ball with an obstacle triggers action miss. Finally, lines 36–41 show an excerpt of the definition of the initial object set-up. Each object has a name (e.g., fb, goalNet), a domain class (e.g., ball, net), an initial position in the 3D space, and optionally, attribute values.

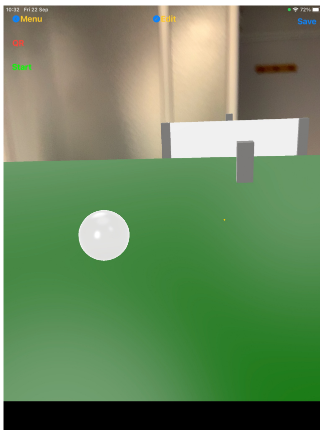

Figure 4: Resulting AR game.

Fig. 4 shows a screenshot of the defined game. The floor is green and horizontal, the net at the back is white, and there is a grey rectangular obstacle. A video is available at *https://youtu.be/lUV3uTLBg2o*.

3 ARCHITECTURE AND TOOL SUPPORT

Fig. 5 shows the architecture of our solution based on ARGDSL. Developers can define AR games using the ARGDSL editor, which is built with Xtext. The meta-models describing the abstract syntax of the DSL are defined with the Eclipse Modeling Framework (EMF) [3], the de-facto standard for meta-modelling within Eclipse.

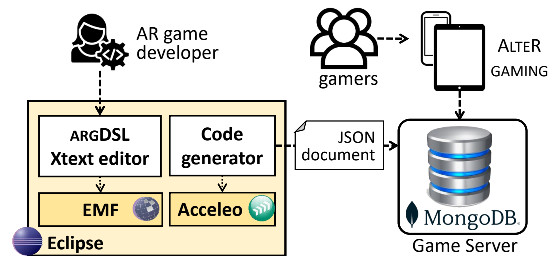

Figure 5: ARGDSL architecture.

After defining an AR game, developers can invoke a code generator that synthesizes a set of four JSON documents with the game information. This generator was built using Acceleo, a template-based language to emit text from EMF models. The IDE offers an option to upload these JSON documents into our game server, and store them in a MongoDB database. Then, an iOS client, built atop the ALTER tool [4], is in charge of interpreting the JSON files, so that gamers can play the games.

More information on the tool can be found at *https://alter-ar.github.io/gaming.html*.

4 CONCLUSIONS AND FUTURE WORK

Building AR games is time-consuming and requires deep expertise in AR technologies and programming models. To attack this problem, we have proposed a DSL to define AR games. Our solution allows describing the most relevant aspects of the game (domain, graphics, physics, logic), contributing to democratise AR game development.

As future work, we plan to extend our DSL to enable the use of the GPS, and the definition of multi-user and multi-level games.

ACKNOWLEDGMENTS

Work funded by the Spanish MICINN (PID2021-122270OB-I00).

REFERENCES

[1] R. T. Azuma. 1997. A survey of augmented reality. *Presence Teleoperators Virtual Environ.* 6, 4 (1997), 355–385.
[2] M. Brambilla, J. Cabot, and M. Wimmer. 2017. Model-Driven Software Engineering in practice. *Synthesis Lectures on Software Engineering* (2017).
[3] F. Budinsky et al. 2011. *EMF: Eclipse Modeling Framework.* Addison-Wesley.
[4] R. Campos-López, E. Guerra, J. de Lara, A. Colantoni, and A. Garmendia. 2023. Model-driven engineering for augmented reality. *J. Obj. Tech.* 22, 2 (2023), 1–15.
[5] H. Ling. 2017. Augmented reality in reality. *IEEE Multim.* 24, 3 (2017), 10–15.
[6] MOF. 2016. http://www.omg.org/MOF.
[7] A. Nikolaidis. 2022. What is significant in modern augmented reality: A systematic analysis of existing reviews. *J. Imaging* 8, 5 (2022), 145.