

Highlights

Building Augmented Reality Games with `argDSL`

Rubén Campos-López, Esther Guerra, Juan de Lara

- We present the domain-specific language ARGDSL for augmented reality games.
- ARGDSL permits defining the domain, logic and virtual objects of the game.
- The games defined with ARGDSL are deployed on a server, and played on iOS clients.
- We showcase ARGDSL by creating four games, which gamers find simple and usable.

Building Augmented Reality Games with ARGDSL

Rubén Campos-López, Esther Guerra and Juan de Lara

Computer Science Department
Universidad Autónoma de Madrid, Madrid, Spain

ARTICLE INFO

Keywords:
Domain-Specific Languages
Model-Driven Engineering
Augmented Reality
Games

ABSTRACT


Augmented Reality (AR) has become popular. It does not require advanced technology, but only a mobile device with a camera to interact with virtual objects. However, developing AR applications – especially games – is time-consuming and requires in-depth knowledge of highly specialised technologies, and mathematical concepts related to the graphics and physics of the virtual objects.

To address this problem, we propose the domain-specific language ARGDSL for creating AR games. It allows customising the game logic, and the virtual objects' domain, physics and representation. We provide an Eclipse editor to define AR games using the language, and an iOS client to run the games. Our experiments show the versatility of our proposal and the usability of the games.

Metadata

Nr.	Code metadata description	
C1	Current code version	1.0
C2	Permanent link to code/repository used for this code version	<ul style="list-style-type: none">• ARGDSL editor: https://github.com/Superrub1997/ARGDSL• iOS client ALTER GAMING: https://github.com/Superrub1997/ALTER-gaming
C3	Permanent link to Reproducible Capsule	<ul style="list-style-type: none">• Videos: https://alter-ar.github.io/gaming.html• App Store: https://apps.apple.com/us/app/alter-gaming/id6453476416
C4	Legal Code License	EPL-1.0 License
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	Eclipse IDE for Java Developers 2023-12, EMF 2.35.0, Java 11, Xtext 2.33.0, Xtend 2.33.0, Acceleo 3.7.14, Swift 5, Xcode 15, ARKit 5
C7	Compilation requirements, operating environments and dependencies	<ul style="list-style-type: none">• ARGDSL editor: Microsoft Windows 10 64-bit or later• iOS client ALTER GAMING: iOS 14
C8	If available, link to developer documentation/-manual	https://alter-ar.github.io/argdsl.html
C9	Support email for questions	rubencampos.97@gmail.com

Table 1: Code metadata

 rubencampos.97@gmail.com (R. Campos-López); Esther.Guerra@uam.es (E. Guerra); Juan.deLara@uam.es (J.d. Lara)
ORCID(s):

1. Motivation and significance

Augmented Reality (AR) [2] 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 widgets¹ has enabled the use of AR for all sorts of applications, from industrial settings to education, health and video games [11].

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. Fig. 1 shows an illustrative example of an AR game being played on a mobile device. The camera of the device captures the real world and displays virtual objects (a green plane and a ball) on top. The virtual objects can be controlled via gestures in the device (e.g., the tilt of the plane is controlled via the accelerometer). This way, the virtual ball will move according to the plane inclination and gravity.

AR is becoming increasingly popular, achieving high commercial success in entertainment and video games [16]. Prominent examples of AR games include Pokemon GO and Pikmin Bloom (both by Niantic²) and the large AR ecosystem by SnapChat³.

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) [9]. 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, we focus 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. We have evaluated the versatility of the approach based on four case studies, and the usability of the created games by means of a user study, showing good results.

Overall, this paper contributes a novel DSL to define AR games declaratively. It does not require expertise in imperative programming or low-level frameworks, and allows for concise game specifications that focus on the essence of the games and avoid the need to write boilerplate, repetitive code across games. The defined games are uploaded to a server and can be played immediately without requiring compilation or deployment into the mobile device.

2. Software description

Fig. 2 shows a scheme of our approach, which is based on model-driven [6] and language engineering [20] principles. Specifically, we have created the DSL ARGDSL to allow AR game developers to define AR games, namely: the elements used in the game, their graphical AR representation, their physics, and the game logic. Following a model-driven approach, ARGDSL specifications conform to a meta-model, and abstract away technical low-level details of the game.

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.

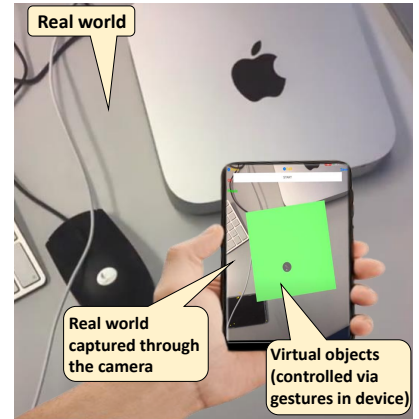


Figure 1: Example of AR game played on a mobile device.

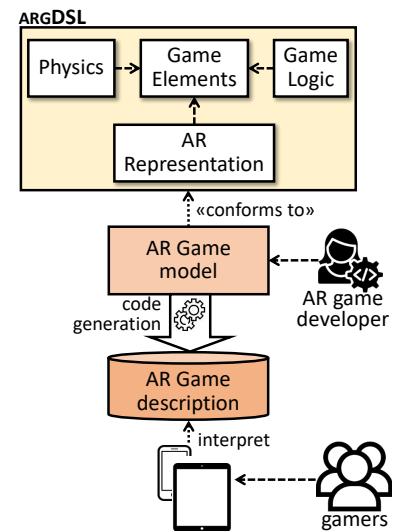


Figure 2: Our approach.

¹<https://www.zdnet.com/article/best-ar-glasses/>

²<https://nianticlabs.com/>

³<https://ar.snap.com/>

Next, we describe how to describe an AR game with ARGDSL: the game elements (Section 2.1), their AR representation (Section 2.2), their physics (Section 2.3), and the game logics (Section 2.4). As a running example, we will define an AR football game (introduced in [9]) to throw penalty kicks to the goal, without touching any obstacles. The player will have 60 seconds to score as many goals as possible.

2.1. Game elements

To create an AR game with ARGDSL, the game developer needs to declare the elements of the game together with their properties and relations (i.e., the domain of the game).

Fig. 3 shows the meta-model for this part of the DSL, which is inspired by the OMG's MetaObject-Facility (MOF) [13]. Each type of element of the game is described by a class, defined by its name, attributes, and references. A class may be abstract and have superclasses. In addition, the derived attribute `noGraphic` signals if class objects have no graphical representation, and hence they are kept invisible.

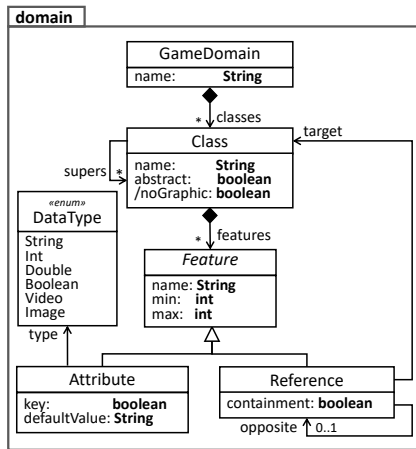


Figure 3: Domain elements meta-model.

```

1 Game football {
2   elements {
3     ball {} /* No attributes */
4     net {}
5     obstacle {}
6     floor {}
7   }
8 }
    
```

Listing 1: Definition of domain elements for the AR football game.

Class attributes are described by their name, type, minimum and maximum cardinality, and default value. In addition to typical data types (e.g., `String`, `Int`), we support images and videos, which get displayed on the object's representation. In turn, references have a name, cardinality, and point to a target class. References can be compositions, and be paired with an opposite reference to define a bidirectional association.

Listing 1 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, but Section 4 will show examples of this.

2.2. AR representation

ARGDSL allows assigning an AR representation to each class and reference in the game domain (cf. Fig. 4). 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, the following features of the AR objects can be configured (via class `NodeConstraint`):

- **Overlapping:** It controls if the AR object can overlap other objects.
- **Size:** It determines the initial, minimum and maximum size that the object can take during the game. If their value is 1.0, then the default size of the SCN object is applied.
- **Distance to original position:** Attributes `x-`, `y-`, and `zToOriginPos` are used to restrict the distance each object can be displaced from its original position (where -1.0 indicates no restriction).
- **Rotation:** It permits rotating the object horizontally.
- **Planes:** It allows configuring whether the AR object can be placed only on horizontal planes, only on vertical planes, on any kind of plane, or anywhere.

Listing 2 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.

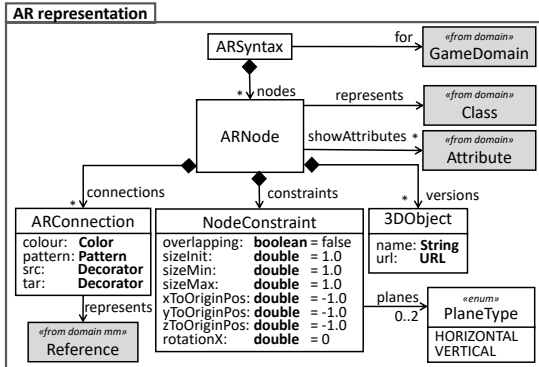


Figure 4: AR representation meta-model.

```

1 Graphics {
2   element ball {
3     versions {
4       v1 = "http://url.com/ball.scn" /* just 1 AR represent. */
5     }
6     constraints {
7       plane horizontal
8       overlaps
9     }
10  }
11  element net { ... }
12  ...
13 }

```

Listing 2: AR representation of elements for the football game (excerpt).

2.3. Physics

In ARGDSL, each element class may have physical information on how its objects should behave and move realistically during the game. Fig. 5 shows the meta-model for this part of the DSL, which is inspired by the specifications of Apple’s ARKit⁴.

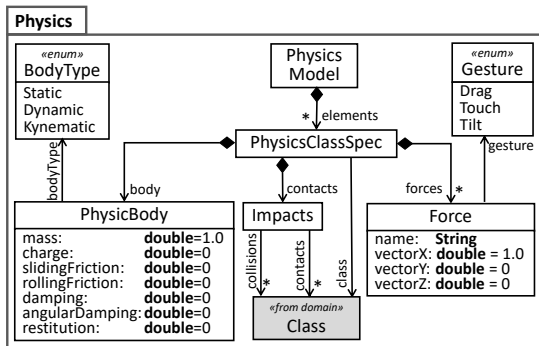


Figure 5: Physics meta-model.

```

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 ...
21 }

```

Listing 3: Physics of the football game (excerpt).

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 in the game, or by forces applied by the players to impulse the element. Specifically, the physical 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 the screen (*Touch*), sliding on the screen (*Drag*), or tilting the screen (*Tilt*). The magnitude of the force is mandatory when touching the screen, and optional

⁴<https://developer.apple.com/documentation/realitykit/physicsbodycomponent>

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 we will explain in the next section.

Listing 3 defines the ball physics using the DSL textual syntax. The ball is set to be a dynamic object with a mass of 0.5 kilograms and some friction forces. It also defines the force kick, which gets activated when dragging on the object. In addition, the ball defines collisions and contacts with floor, obstacle and net. Finally, the dynamic properties of the ball that are not explicitly provided take the default values specified in the meta-model (cf. Fig. 5), like 0 for *charge*.

2.4. Game logic

The meta-model in Fig. 6 permits defining the game logic. A game (class *GameLogic*) can show a different message to signal the start, win and lose situations. It also defines the starting and winning game conditions (class *ScoreSystem*): initial score and number of lives of the player, and final score needed to win the game.

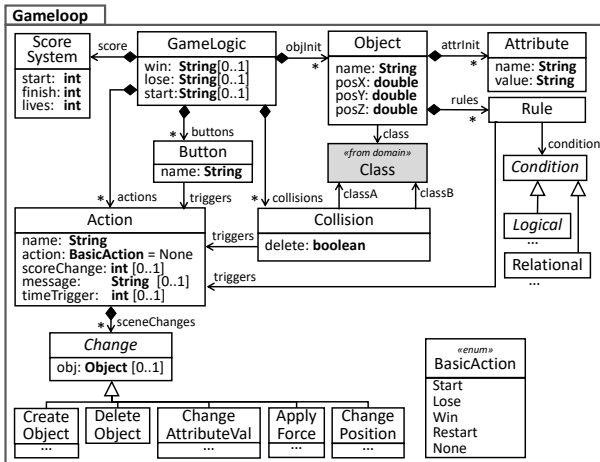


Figure 6: Game logic meta-model (excerpt).

```

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 { /* delete the ball and create new one */
20        do delete fb /* fb is a ball 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 is a ball 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    post1: obstacle [5.0, 0.0, 10.0]
41    post2: obstacle [-5.0, 0.0, 10.0]
42    limit: obstacle [0.0, 0.0, 12.0]
43    barrier: obstacle [0.0, 0.0, 5.0]
44  }
45 }
  
```

Listing 4: Definition of the football game logic.

The game logic is defined by means of actions (class *Action*). Each action performs one of four possible basic actions (start the game from scratch, lose, win, restart the game saving the progress), or none of them. In addition, it may display a message, change the score, self-trigger the action in a time interval of seconds, or change the game objects (class *Change*) by creating or deleting objects, modifying attribute values, applying forces, or changing object positions. The actions can be triggered when an object meets some conditions (specified by class *Rule*), there is a collision (class *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, and an operation to count the number of objects of a class.

Listing 4 shows the logic for the running example in the DSL syntax. The Display block (lines 2–10) corresponds to the GameLogic class. It defines messages for start and winning (as this game cannot be lost) and the score system. Since finish is set to * (internally encoded as -1), 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* object position. Note that creating objects requires providing their type (e.g., `ball`) and name (e.g., `fb`), and having several objects of the same type is possible. The last action, `miss`, presents a message, deletes the ball and creates another in front. Next, lines 32–35 declare two collisions. The first one triggers the `goal` action when the `ball` collides with the `net`. The second one triggers the `miss` action when the `ball` collides with an `obstacle`.

Finally, lines 36–44 declare 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.

Fig. 7 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 of the game is available at <https://youtu.be/LUV3uTLBg2o>.

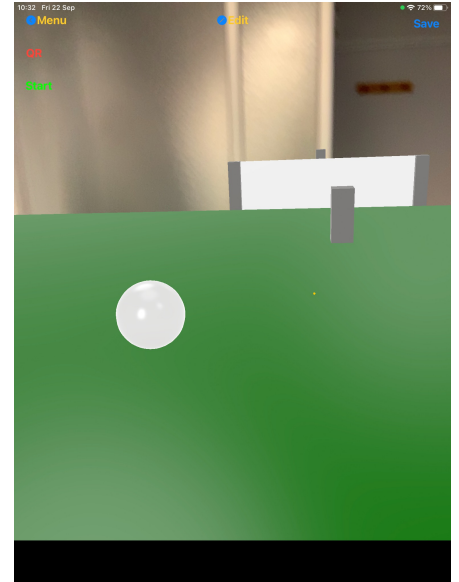


Figure 7: Resulting AR football game.

3. Software architecture

Next, we overview the architecture of our approach (Section 3.1) and describe the tooling (Sections 3.2 and 3.3). Its source code is available at <https://github.com/Superrub1997/ARGDSL>.

3.1. Architecture

Fig. 8 shows the architecture of our solution based on ARGDSL. Game developers can define AR games using the ARGDSL editor, built with Xtext [4]. The meta-models describing the abstract syntax of the DSL are defined with the Eclipse Modeling Framework (EMF) [8], the de-facto meta-modelling standard within Eclipse. Developers can use the editor to define the elements of the game textually, as illustrated in Listings 1– 4. The editor does not offer specific support to design 3D objects though, but they can be created with professional 3D editors (e.g., Blender, Maya, 3ds Max) and then exported to SCN format for their use within the games.

After defining an AR game, developers can invoke a code generator that synthesises four JSON documents with the game information. This generator was built using Acceleo⁵, a template-based language to emit text from EMF models. The generated JSON documents conform to a custom JSON Schema⁶, and contain a lower-level representation of the DSL models, used by the iOS client. 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 [10], is in charge of interpreting the JSON files, so that gamers can play the games.

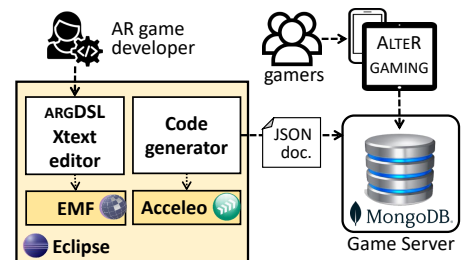


Figure 8: ARGDSL architecture.

⁵<https://eclipse.dev/acceleo/>

⁶<https://json-schema.org/>

Next, we provide details on the ARGDSL editor (Section 3.2) and the iOS client (cf. Section 3.3).

3.2. argDSL editor

Fig. 9 shows a screenshot of the ARGDSL editor integrated within Eclipse (label 1). To facilitate the game definition, the editor features content assistance (activated by `ctrl-space`, cf. label 2) and prompts error indications when detecting missing or incorrect elements in the game. The left of the figure shows the project explorer (label 3) containing some game files (with `.arg` extension). Right-clicking on an `.arg` file displays a menu option to generate the JSON documents from the game definition.

Listing 5 shows an excerpt of one JSON document, generated from the physics definition of the running example. While the mapping of the features of the physics body into JSON is direct (lines 4–7), the generator produces bit masks for the collisions (lines 9–13) as the iOS client requires. These masks are calculated by assigning each class a power of 2 (1, 2, 4, etc.), and adding the values of the classes involved in the collision.

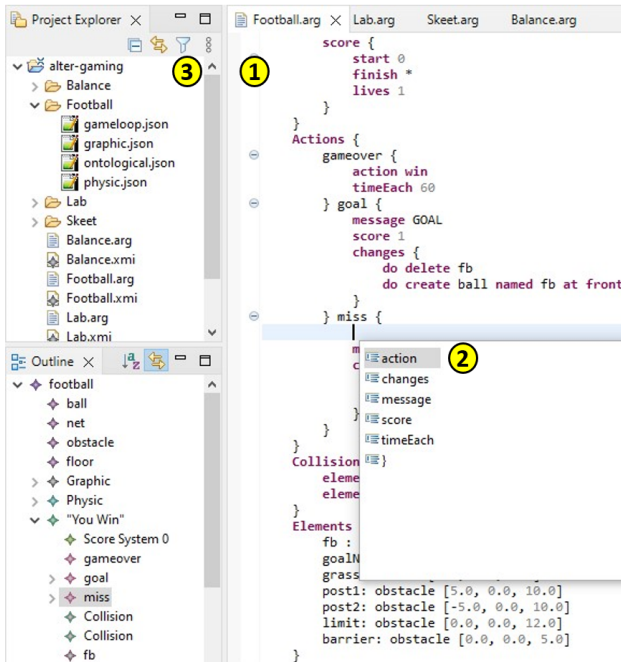


Figure 9: Screenshot of ARGDSL editor.

```

1 {
2   "name": "ball",
3   "physicBody": {
4     "bodyType": "dynamic",
5     "mass": 0.5,
6     "charge": 0,
7     ...
8   },
9   "bitMasks": {
10    "category": 1, /* power of 2 for this class */
11    "collision": 14, /* addition of all colliding classes: 2
12      ↵ + 4 + 8 = 14 */
13    "contactTest": 14
14  },
15  ...
16 }

```

Listing 5: Small excerpt of generated JSON document for physics.

3.3. Alter gaming

After generating the JSON documents, they can be uploaded to the game database. Then, our iOS client, called ALTER GAMING, is able to load the documents and run the defined games. This client is implemented using Swift and ARKit, Apple's library for AR development. When the client is started, it calls the game server to retrieve all available AR games. Upon selecting a game, the client loads all the game data and switches to the camera view, where it displays the game (cf. Figs. 1 and 7). Technically, the client was built by extending the ALTER tool [10], an iOS client to create AR editors for modelling languages. ALTER GAMING extends the client with dynamic objects (with physics), an interpreter for the game logic, and the textual DSL described in Section 2.

4. Illustrative examples

In addition to the running example, we have created three further AR games with different features, which are described next. The complete game definitions can be found at <https://github.com/Superrub1997/ARGDSL/tree/master/samples>.

4.1. Balance

The goal of this game is to maintain a ball balanced on a platform, avoiding its collision with an obstacle. The tilt of the platform is controlled with the accelerometer, and the goal is to keep the ball balanced for as long as possible so that it does not go off the platform. Players earn points for every second they balance, but colliding with the obstacle deducts points. After 10 seconds, the platform gets smaller, making it harder to balance. If the ball falls off, the game is over and the score is displayed. Fig. 10 shows a screenshot of the game, and an illustrative video is available at https://www.youtube.com/shorts/GJhkWZz_zIs.

Listing 6 shows the definition of the physics for the platform, which declares a force named `balance` activated by tilting (line 12), and has contacts and collisions with the ball.

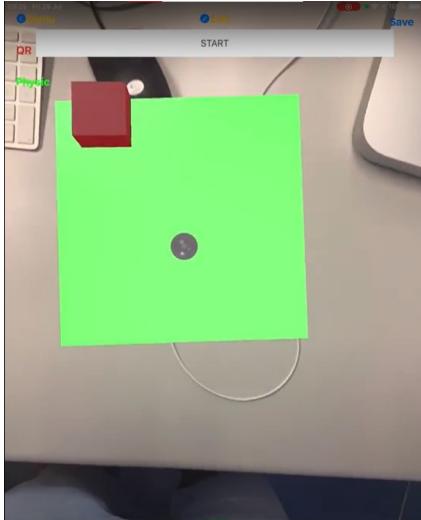


Figure 10: Balance game.

```

1 Physics {
2   element Floor
3   body {
4     mass 0.0
5     bodyType static
6     slidingFriction 0.5
7     restitution 0.5
8     damping 0.1
9     angularDamping 0.1
10  }
11  forces {
12    balance : gesture tilt
13  }
14  contacts {
15    collision Ball
16    contact Ball
17  }
18 }

```

Listing 6: Physics definition for the balance game (excerpt).

4.2. Skeet

The aim of this game is to shoot at targets that are in the air, using a limited amount of time and ammunition. Shooting is performed by clicking a button, and the player needs to orient and move the mobile device to hit the targets. The game is won if all targets are hit within the time, and lost if there are any targets left or the player runs out of ammunition. Fig. 11 shows a screenshot of the game, and a video is available at https://www.youtube.com/shorts/8K4S_yEpo0I.

Listing 7 shows the definition of the game's domain. The element class named `player` declares the attribute `ammo` (line 6) with initial value 6, which gets decreased on each firing. Moreover, the `player` is set to invisible, since it is not graphically represented in the game.

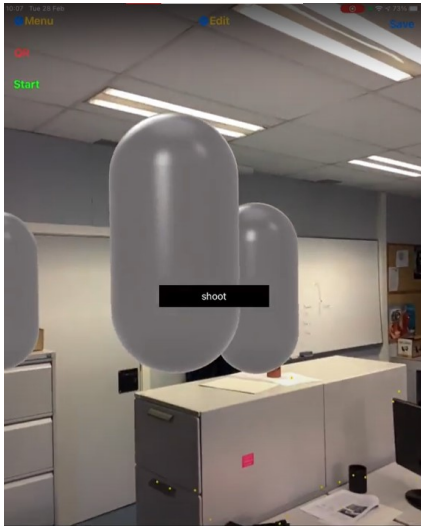


Figure 11: Skeet game.

```

1 Game skeet {
2   elements {
3     bullet {}
4     target {}
5     player {
6       ammo : Int [1] = 6
7     } invisible
8   }
9 }

```

Listing 7: Domain definition for the skeet game.

4.3. Labyrinth

This game shows an AR labyrinth, as in Fig. 12, and the goal is to guide a ball into the green pivot without hitting the red walls, using the direction buttons. If the ball hits a red wall, the game is lost. When the ball reaches the pivot, the maze returns to the initial state, but reducing the platform and increasing the size of the walls, making the game more difficult. The game is won when the pivot is reached three times. A video illustrating the game is available at https://www.youtube.com/shorts/e_j3qINJviI.

Listing 8 shows a small excerpt of the game logic. The action named `levelUp` restarts the game (line 3), displays a message (line 4), increases the score (line 5), scales down the platform (line 7), and enlarges the walls (lines 8–11).

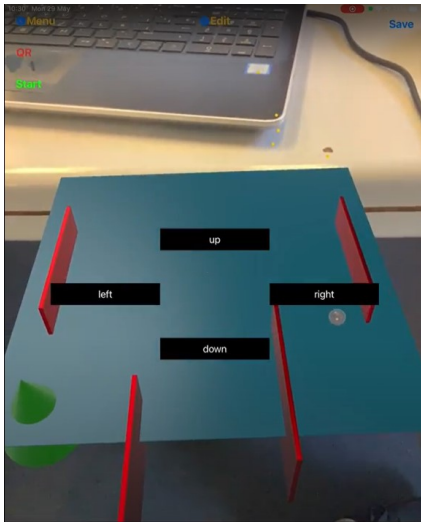


Figure 12: Labyrinth game.

```

1 Actions {
2   levelUp {
3     action restart
4     message LevelUP
5     score +1
6     changes {
7       do edit Plane named floor scale -0.1
8       do edit Wall named wall1 scale 0.3
9       do edit Wall named wall2 scale 0.3
10      do edit Wall named wall3 scale 0.3
11      do edit Wall named wall4 scale 0.3
12    }
13  }
14  ...
15 }

```

Listing 8: Game logic definition for the labyrinth (excerpt).

4.4. Summary

Table 2 displays the size of the AR game specifications: number of relevant elements to define the game domain, graphics, physics and logic, and number of lines of code (LoC) of the ARGDSL specifications. The LoC are directly proportional to the number of classes in the domain, graphics and physics, and to the number of game actions. Static and invisible objects require less LoC, as they lack physics and graphics

specifications. Overall, all the games were specified in less than 250 LoC, containing tens of objects. This suggests a moderate effort, as games were defined in minutes. In comparison, coding a game (like the football one) directly in Swift/ARKit would result in around 1 000 LoC (around 650 LoC for the game logic, 100 for the physics, and 250 for the elements' graphical representation). This code, in addition to being one order of magnitude longer than the corresponding ARGDSL specifications, involves low-level programming in Swift/ARKit. Still, in future work, we plan to conduct a user study with developers to understand the real effort, gain and acceptance of the DSL.

Game	Domain	Graphics	Physics	Logic	LoC
Football	4 objs, 1 attr	4 objs	4 objs	3 actions, 6 elements	165
Balance	3 objs	3 objs	3 objs	3 actions, 3 elements	141
Skeet	3 objs, 1 attr	2 objs	2 objs	3 actions, 7 elements	113
Labyrinth	4 objs	4 objs	4 objs	8 actions, 7 elements	220

Table 2: Size of use cases, counting objects and LoC.

Regarding game functionality, note that the AR objects modelled with ARGDSL are not just background images, but they can occlude and be occluded by real-world objects, and be placed on real and virtual horizontal and vertical planes. Moreover, they are connected with reality, since the physical movement of the device can affect the game, and some games may require the player to physically move in the real world to get closer or away from AR objects. For example, the Skeet game requires the player to move around the physical space to locate and shoot the balloons (Section 4.2), while the Balance game determines the tilt angle of the horizontal plane using the accelerometer to detect changes in the speed or direction of the device (Section 4.1).

5. Impact and evaluation

We have conducted a user study to understand the usability of the generated games. In this study, participants first played the labyrinth game (Section 4.3) and then filled some questionnaires.

5.1. Experiment design

The experiment started with a 10-minute explanation of the tool. After that, participants freely played the labyrinth game until passing the game or having three failed attempts. Then, they were asked to fill in three questionnaires: one collecting demographic information (age and usage level of smartphones and AR games); the System Usability Scale (SUS) questionnaire [7], which has become a standard for measuring usability; and six questions designed by us on specific aspects of the game.

The SUS questionnaire comprises ten questions with five response options each, from *strongly disagree* to *strongly agree* (a 5-point Likert scale). Five of these questions have a positive tone and their best score is 5, and the other five have a negative tone and their best score is 1. The participants' scores to the questions are converted into a number between 0 and 100, which gives a measure of the system usability. The questionnaire ends with open questions where participants can provide up to three positive and three negative aspects of the tool, as well as suggestions for improvement.

Our game-specific questions also used a 5-point Likert scale, three with positive and three with negative tone. The questions were: *Q11: I think the game is easy to play*, *Q12: I had difficulties playing the game*, *Q13: I think the buttons used in the game are adequate*, *Q14: I had problems using the buttons or understanding how they work*, *Q15: The virtual objects used in the game are suitable*, *Q16: I missed some virtual objects that could be in the game*.

5.2. Experiment execution and results

We recruited ten participants and provided them with an iPad mini to play the game. Their average age was 33, their average usage level of smartphones was 4.5 out of 5, and of AR games 1.6 out of 5.

The responses to the SUS questionnaire yield 87.25. According to [3], this qualifies the overall usability of the studied game as excellent (>85.5). As for the game-specific questions, Table 3 shows the results. The median of questions with positive tone (*Q11*, *Q13*, *Q15*) is 5, and the median of the negative ones is 1. This indicates that the participants are generally satisfied with the game generated by ARGDSL. Questions *Q13* and

Q16 had the lowest scores, having both the AR game interface as a common point. Therefore, participants found the game simple, adequate and usable, but the interface could be improved. These results are in line with the answers to the open questions, where participants considered that the game was easy to use, fun and intuitive, but they also suggested improving the graphical appearance of the game buttons and the virtual objects. Overall, the SUS results yield an excellent usability for the game, and the specific questions suggest that participants were satisfied with the game, though there is some room for improvement regarding the user interface.

Question	Average	Median	Std Dev.
Q11	4.9	5	0.32
Q12	1.2	1	0.42
Q13	4.4	5	0.84
Q14	1.2	1	0.42
Q15	4.5	5	0.71
Q16	1.9	1	1.10

Table 3: Results on game-specific questions.

5.3. Threats to validity and limitations

Regarding *external validity* (generalisability of the results), our user study involves a small number of participants. Hence, to strengthen the generalisability of our findings, larger user studies should be conducted. In addition, the user study considers one AR game, and so, the results might not be extrapolated to other games. To mitigate this threat, we evaluated the AR game that we considered the most difficult to use among the case studies (i.e., the labyrinth).

As Section 4 showed, ARGDSL permits generating a variety of games, however it also poses some limitation to AR game development. The most evident is that games are mono-user (not collaborative), and there is no native support for multi-level games (even if levels can be emulated, as we did in the labyrinth game by periodically reducing the platform size). In this regard, if some feature is outside the reach of ARGDSL and no workaround allows its emulation, the only option would be to extend ARGDSL, since we currently have no facilities to export games (not even partially) to other platforms like Unity. We will address these limitations in future work.

6. Related work

Many AR tools are used within the Unity⁷ environment to create video games. In addition, some can be used in other development environments, like Xcode⁸ or Android Studio⁹. Next, we analyse the advantages, limitations, and supported functionalities of some prominent ones [19].

Some approaches rely on libraries for general-purpose programming languages. ARCore¹⁰ provides AR libraries for iOS and Android, and can be used in Unity, Android Studio and Unreal Engine. It is free and supports 3D figures and motion recognition. However, despite the compatibility, it targets Google (Pixel) and Samsung devices. ARKit¹¹ is Apple's AR library for iOS, and it can be used in Xcode and Unity. It has fewer features than ARCode (e.g., it lacks 3D recognition), but it can recognise 2D objects and the illumination level of the environment, optimising plane detection. Overall, both require low-level programming to create games, and expertise in the language (e.g., Swift).

ARFoundation¹² is the free AR tool for Unity, compatible with iOS, Android and HoloLens. It is focused on AR game development, and leverages on Unity's development style by combining drag&drop, forms and programming. The tool can then generate code for ARCode or ARKit. However, configuring and using ARFoundation is complex and requires advanced knowledge of Unity.

Other frameworks are also integrated in Unity. Vuforia¹³ is a free tool more advanced than ARCore and ARKit for environment recognition and 3D shapes. It is compatible with iOS, Android and Windows UWP and supports the creation of figures. Lightship ARDK¹⁴ is a tool used by the company Niantic, integrated into Unity and compatible with iOS and Android. It has similar features to Vuforia, but with some more advanced ones such as virtual positioning and multiplayer sessions. It is free only in trial version. Finally,

⁷<https://unity.com/>

⁸<https://developer.apple.com/xcode/>

⁹<https://developer.android.com/studio>

¹⁰<https://developers.google.com/ar>

¹¹<https://developer.apple.com/augmented-reality/arkit/>

¹²<https://unity.com/unity/features/arfoundation>

¹³<https://developer.vuforia.com/>

¹⁴<https://lightship.dev/products/ardk/>

Wikitude¹⁵ is a non-free tool for Google Glass, but also compatible with iOS and Android. It can be used in Unity, but also has its own programming environment, and supports GPS location. Overall, the Unity development cycle for AR can be tedious, since producing code for iOS requires another compilation on Xcode. Instead, our approach is lightweight, and since our client interprets the games, it does not require heavy compilation steps and the development cycle is more agile.

Many model-driven approaches and DSLs have been proposed to create specific types of games, like tower-defense games, 2D platformers, card games [5], maze games, educational games, serious games [17, 18] or role-playing games (cf. [22] for a survey). Technically, approaches either generate code or use an interpreter that executes the game model. In the former case, the code can be for a low-level framework or a game engine, and can cover part or the whole game. For example, RAIL [21] is a DSL for describing Non-Player Character (NPC) behaviours, and generates code for the Torque 2D engine. Our approach uses an interpreted approach, describes the whole game, targets AR, and focuses on realistic skill games (e.g., labyrinths, balance games, shooters).

Specific to AR, in academy, ZeusAR [12] is a development process to create AR serious games based on three phases: analysis, configuration and generation. While the process is programming language-independent, it is focused on educational games, leaving the generation of AR games in general as future work. Muff and Fill [14, 15] propose a visual modelling language for designing AR scenarios and AR workflows graphically. The system targets generic AR applications (e.g., to display the assembly process of furniture using visual markers on the physical elements), and is not specific to games. These would be challenging to create, as defining the physics of the AR elements and the game logic likely requires a more specialised DSL. Overall, we did not find any DSL specific to building dynamic AR games, as ARGDSL does.

7. Conclusions and future work

The demand for AR games has increased over time, raising the proposal of tools for their creation. However, 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. We have proposed tool support based on Eclipse and iOS, and demonstrated its effectiveness and usability through four cases studies and a user study.

As future work, we plan to extend our DSL to enable the use of the GPS, and the creation of multi-user and multi-level games. In the latter case, the game may change its elements and their behaviour at each level to provide more dynamism. We would like to provide a wizard to generate templates for different types of games (e.g., labyrinth, shooting, etc.). We are also considering migrating the editor to the web (e.g., using the web deployment option of Xtext) to provide a low-code web development environment for AR games. We plan to improve the client's graphical interface to make it more suitable for AR games (e.g., supporting other screen widgets besides buttons) taking into account guidelines for AR user interfaces [1]. Finally, to enhance compatibility, we will consider export facilities of the defined games into other platforms, like Unity.

Acknowledgements

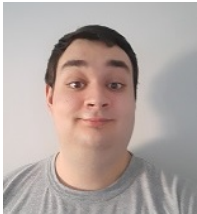
Work funded by the Spanish MICINN with projects TED2021-129381B-C21 and PID2021-122270OB-I00.

References

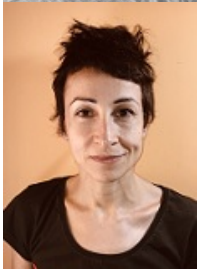
- [1] Aultman, A., Dowie, S., Hamid, N.A., 2018. Design heuristics for mobile augmented reality game user interfaces, in: Extended Abstracts of the CHI Conference on Human Factors in Computing Systems, ACM. pp. 1–5.
- [2] Azuma, R.T., 1997. A survey of augmented reality. Presence Teleoperators Virtual Environ. 6, 355–385.
- [3] Bangor, A., 2009. Determining what individual SUS scores mean: Adding an adjective rating scale. Journal of User Experience 4, 114–123.
- [4] Bettini, L., 2016. Implementing domain-specific languages with Xtext and Xtend. Packt Publishing.
- [5] Borrór, K., Rapos, E.J., 2021. MOLEGA: Modeling language for educational card games, in: DSM, ACM. pp. 1–10.
- [6] Brambilla, M., Cabot, J., Wimmer, M., 2017. Model-driven software engineering in practice. Synthesis Lectures on Software Engineering .

¹⁵<https://www.wikitude.com/>

- [7] Brooke, J., 2013. SUS: A retrospective. *J. of Usability Studies* 8, 29–40.
- [8] Budinsky, F., Merks, E., Paternostro, M., Steinberg, D., 2011. EMF: Eclipse Modeling Framework. Addison-Wesley.
- [9] Campos-López, R., Guerra, E., de Lara, J., 2024. A domain-specific language for augmented reality games, in: *ACM SAC (posters)*, pp. 1–3.
- [10] Campos-López, R., Guerra, E., de Lara, J., Colantoni, A., Garmendia, A., 2023. Model-driven engineering for augmented reality. *J. Object Technol.* 22, 1–15.
- [11] Ling, H., 2017. Augmented reality in reality. *IEEE Multim.* 24, 10–15.
- [12] Marín-Vega, H., Alor-Hernández, G., Colombo-Mendoza, L.O., Bustos-López, M., Zatarain-Cabada, R., 2022. ZeusAR: A process and an architecture to automate the development of augmented reality serious games. *Multim. Tools Appl.* 81, 2901–2935.
- [13] MOF, 2016. <http://www.omg.org/MOF>.
- [14] Muff, F., Fill, H., 2023. A domain-specific visual modeling language for augmented reality applications using WebXR, in: *ER, Springer*. pp. 334–353.
- [15] Muff, F., Fill, H., 2024. M2AR: A web-based modeling environment for the augmented reality workflow modeling language, in: *MODELS Companion, ACM*.
- [16] Nikolaidis, A., 2022. What is significant in modern augmented reality: A systematic analysis of existing reviews. *J. Imaging* 8, 145.
- [17] Thillainathan, N., Hoffmann, H., Leimeister, J.M., 2013. Shack City - A serious game for apprentices in the field of sanitation, heating and cooling (SHaC), in: *GI-Jahrestagung, GI*. pp. 2402–2413.
- [18] Thillainathan, N., Leimeister, J.M., 2014. Educators as game developers – model-driven visual programming of serious games, in: *KICSS, Springer*. pp. 335–349.
- [19] Trivedi, N.K., Anand, A., Sagar, P., Batra, N., Noonina, A., Kumar, A., 2022. A systematic review of tools available in the field of augmented reality. *J. Cases Inf. Technol.* 24, 1–9.
- [20] Wąsowski, A., Berger, T., 2023. Domain-specific languages effective modeling, automation, and reuse. Springer International Publishing.
- [21] Zhu, M., Wang, A.I., 2017. RAIL: A domain-specific language for generating NPC behaviors in action/adventure game, in: *ACE, Springer*. pp. 868–881.
- [22] Zhu, M., Wang, A.I., 2020. Model-driven game development: A literature review. *ACM Comput. Surv.* 52, 123:1–123:32.



Rubén Campos-López is a researcher at the modelling and software engineering research lab of the Universidad Autónoma de Madrid. His research interests include augmented reality, mobile development and model-driven engineering. Contact him at rubencampos.97@gmail.com.



Esther Guerra is Full Professor at the Computer Science department of the Universidad Autónoma de Madrid. Together with J. de Lara, she leads the modelling and software engineering research group (<http://miso.es>). She is interested in model-driven engineering, flexible modelling, meta-modelling, domain-specific languages and model transformation. Contact her at esther.guerra@uam.es, or visit <http://www.ii.uam.es/~eguerra>.



Juan de Lara is Full Professor at the Computer Science department of the Universidad Autónoma de Madrid. Together with E. Guerra, he leads the modelling and software engineering research group. His research interests are in model-driven engineering and automated software development. Contact him at juan.delara@uam.es, or visit <http://www.ii.uam.es/~jlara/>.