Universidad de Castilla-La Mancha
Escuela Superior De Informática

Language processors

Blendify: Language for the definition of physic simulations

Authors:
Alberto Aranda García
Cristian Gómez Portes
Daniel Pozo Romero
Eduardo Sánchez López

Date:
October 19, 2017

Contents

1 Problem presentation

In the teaching of physics, the visualization of problems is a really important tool for the correct comprehension of the theoretical concepts. The Ausubel's Learning Theory[1] talks about this approach, because it allows the students to see how useful is the theoretical concepts in reality[1].. In this context, the simulation is an option that's been available for many years, but it has not been used a lot.

This has an easy explanation, the majority of programs for the generation of physic simulations have graphic interfaces that require a huge amount of time to make something useful out of them: from the problem statement the user has to translate it into a graphic format, which needs time to learn how to use them correctly and for the many repetitive tasks that have to be done.

The ideal scenario would be that, from a problem statement, the generation of simulation will be automatic. But that's not possible using the natural language, even inside this very concrete domain. A first approximation could be the creation of a language with an expressive power similar to the problem statements, one that could be easy to translate (for a human) said statements written originally in natural language.

As a platform for the physic simulations, we are going to use Blender, since it incorporates a very potent physic engine and a Python API that allows to use every functionality of the software by code.

For our practice, we consider the definition of a language that allows the user to write physic simulations, and the construction of a language processor that translates it to Python code, that will be interpreted in Blender to generate said simulation. In summary, our main objective is simplify the generation of physic problems simulations.

2 Description of the language

As said previously, our language called **Blendify**, will have the same expressive power for the creation of program statements, allowing the definition of some facts like the type of problem or the dimensionality. If we analyze the statements structure, we will get to the conclusion that first there's some information about the elements that take part in the problem (Shapes like a cube, sphere, ramp, planet...)

[1]Psychologist and pedagogue of great importance for the constructivism

and their parameters (mass, speed, force and so on). Alongside this, there is a condition that we can use as a stop for our simulation, for example when an element gets to a certain position, and a question about the state of the world that the student has to solve. Our language then will allow to declare different objects and their respective attributes, alongside some global parameters that affect the entire simulation, a stop condition and will also show in screen information about what the user wants to do with the simulation (start the simulation, generate an animation...)

2.1 Tokens table and Reserved words

In this section we show the *tokens* table and some reserved words that make up the vocabulary of our language. — — see Tabla 1 and 2. This table will helps us work to form the productions that will generate our final language. In the next part we show how we used these *tokens* in the productions.

2.2 EBNF

Once we have defined our *tokens* and reserved words, we can start to specify our language. As the default production we tought in one that contains simple terminals so that the user (We have to remember this software is oriented to people that doesn't have experience in programming) won't have problems when creating different scenes. The production will be the following:

program ::= **begin** id_program body_program **end**.

The user has to define the start and the end of the program to help he or she to not make any mistakes. *body_program* ::= **declaration** body_declaration **scene** body_scene **action** body_action.

As mention previously, the objective of this production will be the same, since there is a part for the declaration of the variable, for the scenes and for the actions. *body_declaration* ::= '{' { **(static** attribute_declaration '=' value_static) |(**dynamic** attribute_declaration '=' value_dynamic)} '}'.

body_action ::= '{' **start_simulation** '}'.

body_scene ::= '{' {(attribute_case '=' goal)} '}'. extbfdynamic.

This three productions define the declaration, action and scene fields. Basically, what we will do in this part is to define a type, which will be associated to an

Tokens	**Lexeme**	**Pattern**
Assign	=	=
Comma	,	,
Open Bracket	{	{
Closed Bracket	}	}
condition	AND |OR	AND |OR
Open parenthesis	(	(
Closed parenthesis	)	)
Coordinates	(1.2, 1.2, 1.2)	'(' real ',' real ',' real ')'
Alphabetic	a ,..., z , A ,..., Z	lower_case |upper_case
lower_case	a ,..., z	[a-z]
upper_case	A ,..., Z	[A-Z]
Real	23.21	digit + ['.' digit +)]
Digit	0 ,..., 9	1 |2 |3 |4 |5 6 |7 |8 |9 |0

Table 1: Tokens table.

Key Words	**Lexeme**	**Pattern**
begin	begin	begin
end	end	end
declaration	declaration	declaration
scene	scene	scene
action	action	action
start_simulation	start_simulation	start_simulation
type figure	static, dynamic	static, dynamic
type value	position, rotation, scale weight, speed	position |rotation |scale |weight |speed
form figure	Cube, Sphere, Cone Cylinder, Force_field, Ramp	Cube |Sphere |Cone Cylinder |Force_field |Ramp

Table 2: Reserved words table.

identifier. For example in the case of *body_declaration* we will define if the variables are static (objects that will have position, rotation and scale) or dynamic (object that will have speed and weight, alongside the previously mentioned characteristics) and also we will have to define the type and an identifier. In the case of *body_scene* it will be the same, discarting the **static** and **dynamic** part.

attribute_declaration ::= type_figure id_attribute.

type_figure ::= **Cube** |**Sphere** |**Cone** |**Cylinder** |**Force_field** |**Ramp** |**Plane**.

This last productions are those that define the types of figures that we can specify: **Cube**, **Sphere**, **Cone**, **Cylinder**, **Force_field**, **Ramp**, **Plane**. In addition, the identifier can be written if it starts by a letter (In capital letters or not), followed by more letters or numbers, but with only one letter is enough.

attribute_case := **Case** id_case.

As mentioned before, *attribute_case* defines the type of scene that's going to be created. **Case** is a reserved word that mentions the type of scene. *id_case* gives a name to a scene in particular. The formation of *id_case* is the same as *id_attribute*.

goal::= gplane |gspeed |gcollision |goal2.

goal2 ::= '(' goal { condition goal } ')'.

goal and *goal2* would allow us to create conditions when creating scenes. With these productions we can define one o more different goals that will help the user to see the problem with different points of view.

value_static ::= [position] [rotation] [scale].

value_dynamic ::= [position] [rotation] [scale] [weight] [speed].

value_static and *value_dynamic* are those production that allows us to define if an object will be static, that means it cannot be moved, or dynamic with a defined speed and weight.

id_attribute ::= alphabetic {alphanumeric}.

id_program ::= alphabetic {alphanumeric}.

position ::= **position** coordinates.

rotation ::= **rotation** coordinates.

scale ::= **scale** coordinates.

weight ::= **weight** real.

speed ::= **speed** coordinates.

With these 5 productions we can try to define the position, rotation, scale, weight and speed of an object in a specific scene. All of these characteristics will be formed with coordinates (A set of 3 values defined by the axis 'x', 'y' and 'z' specified in *coordinates*) with the exception of the weight, that will be a real number.

coordinates ::= '(' real ',' real ',' real ')'.

condition ::= **AND** |**OR**.

condition production that contains the logic conditions **AND** and **OR**.

alphabetic ::= lower_case |upper_case.

lower_case ::= [a-z].

upper_case ::=[A-Z].

alphabetic is the production that generates lower cases with *lower_case* and capital letters with *upper_case*.

alphanumeric ::= alphabetic |digit.

alphanumeric will be the production that generates letters (lower cases or capital letters) and digits.

real ::= digit + ['.' digit +)].

Finally, *real* is that production that allows us to generate real numbers.

References

[1] David Ausubel et al. Teoría del aprendizaje significativo. *Fascículos de CEIF*, 1:1–10, 1983.