Universidad de Castilla-La Mancha
Escuela Superior De Informática

Memoria de Prácticas

# Blendify: Lenguaje para la definición de simulaciones físicas

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

*Fecha:*
8 de octubre de 2017

# Índice

# 1. Presentación del problema

Dentro de la enseñanza de Física, la visualización de los problemas es una herramienta muy importante para que los estudiantes comprendan mejor los conceptos teóricos existentes tras ellos. La teoría del aprendizaje de Ausubel[1] propone este enfoque, ya que permite al alumno esclarecer y dotar de utilidad aquello que está aprendiendo [1]. En este contexto, la simulación es una opción que lleva disponible durante años, pero que está siendo muy infrautilizada.

Esto tiene una fácil explicación: la mayoría de programas de generación de simulaciones físicas imponen interfaces gráficas que requieren una inversión de tiempo grande por parte del usuario: partiendo de un enunciado textual de un problema, el usuario, que puede ser tanto un profesor como un estudiante de física, se ve obligado a traducirlo manualmente a un formato gráfico, lidiando muchas veces con interfaces de usuario tediosas que hacen perder mucho tiempo, tanto para aprenderlas, como por las operaciones repetitivas que deben realizarse.

Lo ideal sería que, a partir de un enunciado de un problema, fuese posible la generación automática de su simulación. Dejando de lado las dificultades que el Procesado de Lenguaje Natural supone (aún dentro de un dominio tan acotado), una primera aproximación puede ser la creación de un lenguaje con un poder expresivo similar al de los enunciados de problemas, al que sea lo más sencillo posible traducir (para un humano) dichos enunciados escritos originalmente en lenguaje natural.

Como plataforma de simulaciones físicas, vamos a considerar Blender, ya que incorpora, entre muchas otras funcionalidades, un motor de físicas muy potente, y, tantoo más importante, una API para Python que expone prácticamente todas las funcionalidades del programa.

Para nuestra práctica, por tanto, consideramos la definición de un lenguaje que permita escribir simulaciones físicas, y la construcción de un procesador de lenguajes que lo traduzca a código Python que use la API de Blender, que posteriormente podrá ser interpretado por este programa para generar una simulación. Nuestro objetivo principal, pues, es simplificar al máximo posible la generación de simulaciones que de otra forma serían muy tediosas de realizar.

[1] psicólogo y pedagogo estadounidense de gran importancia para el constructivismo

# 2. Descripción del lenguaje

Como se ha dicho anteriormente, nuestro lenguaje, llamado **Blendify**, deberá tener el mismo poder expresivo los enunciados de problemas, y posiblemente, permitir definir ciertos hechos sobre el contexto que se dan por supuesto (tipo de problema, dimensionalidad, etc.). Si analizamos la estructura de los enunciados, nos encontramos con que, primero, se nos da información sobre los elementos que intervienen en el problema (cubo, esfera, polea, rampa, electrón, planeta...) y sus parámetros (masa, velocidad, fuerzas que actúan sobre este...); posteriormente, se suele dar una condición que podemos considerar como de parada de la simulación, y una pregunta (que podemos felizmente obviar) sobre algún parámetro de un elemento en determinado momento. Nuestro lenguaje, pues, deberá permitir declarar diferentes objetos y sus correspondientes atributos, así como ciertos parámetros globales que afecten a toda la simulación, alguna condición de parada, y posiblemente, información sobre qué quiere hacer el usuario con la simulación (ejecutarla, generar una animación, etc.).

## 2.1. Tabla de Tokens

En esta sección se muestra la tabla de *tokens* que componen le vocabulario de nuestro lenguaje — ver Tabla 1. Esta tabla nos ayudará a formar las producciones que generarán nuestro lenguaje final. En el siguiente apartado se podrá ver como se ha hecho uso de estos *tokens* en las diferentes producciones.

## 2.2. EBNF

Una vez tenemos definidos nuestros *tokens*, podemos comenzar a especificar nuestro lenguaje. Como producción de partida se ha pensado en una que contenga terminales simples para que el usuario (normalmente destinado a aquellos que no tiene mucha experiencia en el campo de la programación) no tenga dificultades a la hora de crear diferentes escenarios, ya que la herramienta está pensada para ayudar en el campo de la física. La producción sería la siguiente:

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

Se pretende que el usuario define el comienzo y el fin del programa para ayudarle a no cometer errores que puedan entorpecer el desarrollo del mismo.

*body_program* ::= **declaration** body_declaration **scene** body_scene **action** body_action.

| **Tokens** | **Lexeme** | **Pattern** |
|---|---|---|
| Assign | = | = |
| Comma | , | , |
| Open Bracket | { | { |
| Closed Bracket | } | } |
| Case | Case | Case |
| scene | scene | scene |
| start_simulation | start_simulation | start_simulation |
| condition | AND \|OR | AND \|OR |
| Open parentheses | ( | ( |
| Closed parentheses | ) | ) |
| Type Figure | static, dynamic | static, dynamic |
| Type Value | position, rotation, scale weight, speed | position \|rotation \|scale \|weight \|speed |
| From Figure | Cube, Sphere, Cone Cylinder, Force_field, Ramp | Cube \|Sphere \|Cone Cylinder \|Force_field \|Ramp |
| 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 |

Tabla 1: Tabla de tokens.

Como se ha mencionado anteriormente, el objetivo de esta producción sería el mismo, ya que hay una parte de declaración de variables, otra de escenarios y acciones.

*body_declaration* ::= '{' { **(static** attribute_declaration '=' value_static) |(**dynamic** attribute_declaration '=' value_dynamic)} '}'.

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

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

Estas tres producciones defininen los campos de declaraciones, acciones y escenearios. Básicamente, lo que haremos en estos apartados será definir un tipo, el cual irá aosciado a un identificador. Por ejemplo, en el caso de *body_declaration* definiremos si las variables son estáticas (objetos que tendrán posición, rotación y escala) o dinámicas (objetos que tendrán velocidad y peso, además de las mencionadas anteriormente), además de especificar su tipo e identificador. En el caso de *body_scene* sería lo mismo obviando **static** y **dynamic**.

*attribute_declaration* ::= type_figure id_attribute.

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

Estas tres ultimas producciones son aquellas que definen los tipos de figuras que podremos especificar: **Cube**, **Sphere**, **Cone**, **Cylinder**, **Force_field**, **Ramp**, **Plane**. Además, el identificador que será posible concretar serán aquellos que comiencen por letra, sin distinción entre minúsculas y mayúsculas, seguido de letras o números.

*attribute_case* := **Case** id_case.

Aunque se ha mencionado anteriormente, *attribute_case* define el tipo del escenario que va a ser creado. **Case** es una palabra reservada que hace mención al tipo de variable escenario. *id_case* concreta el nombre que tendrá un escenario en concreto. La formación de *id_case* será el mismo que *id_attribute*.

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

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

*goal* y *goal2* nos permitirán crear condiciones a la hora de crear escenearios. Con estas producciones podremos definir una o diferentes metas que ayudarán al usuario a tener diferentes punto de vista de su problema.

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

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

*value_static* y *value_dynamic* serán aquellas producciones que permitan definir si un objeto será estático, es decir, no podrá moverse, o dinámico, tendrá una velocidad y peso asociados.

*id_attribute* ::= alphabetic {alphanumeric}.

*id_program* ::= alphabetic {alphanumeric}.

*position* ::= **position** coordinates.

*rotation* ::= **rotation** coordinates.

*scale* ::= **scale** coordinates.

*weight* ::= **weight** real.

*speed* ::= **speed** coordinates.

Con estas 5 producciones se intenta definir la posición, rotación, escala, peso y velocidad de un objeto en un escenario específico. Todas ellas se formarán mediante coordenadas (conjunto de 3 valores definidos por los ejes 'x', 'y', 'z' especificado en *coordinates*) menos el peso, que será un número real.

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

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

*condition* producción que contiene las condiciones lógicas **AND** y **OR**.

*alphabetic* ::= lower_case |upper_case.

*lower_case* ::= [a-z].

*upper_case* ::=[A-Z].

*alphabetic* es aquella producción que genera letras minúsculas debido a *lower_case* y letras mayúsculas debido a *upper_case*.

*alphanumeric* ::= alphabetic |digit.

*alphanumeric* será la producción que genere tanto letras, minúsculas o mayúsculas, como dígitos.

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

Finalmente, *real* será aquella producción que permita generar números reales.

# Referencias

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