LUMIERE

CS22BTECH11051 | SHAIK ARMAAN*
CS22BTECH11022 | SHIVENDRARAJE GADEKAR*
CS22BTECH11038 | MEDOJU SRIMANNARAYANA*
CS24BTKMU11001 | BIKRAJ SHRESTHA*
CS24BTKMU11005 | NIGAM NIRAULA*
CS24BTKMU11003 | SHASHWOT PAUDEL*
CS22BTECH11015 | GAURAV CHOUDEKAR*
CS22BTECH11019 | DUDEKULA DHEERAJ*

CONTENTS

1 LANGUAGE BACKGROUND

With advancement in field of science and technology computer simulations have been a major part of new innovations. However, domain expert often struggle with lack of simple interface, knowledge regarding programming and lack of hardware optimization for the simulation. Even if some language do provide these readily available interfaces they often restrict the user from having flexibility in their simulation. The study of classical physics has long been a cornerstone of engineering. Fundamental principles such as Newton's Laws of Motion, conservation of momentum, and the mechanics of rigid bodies form the basis of understanding how physical systems behave under various forces and torques.Traditional programming languages often lack the necessary abstractions to effectively model mechanical systems, resulting in inefficient code. To deal with this problem effectively, **Lumiere** is proposed as a DSL(Domain Specific Langugage) specifically for the simulation of classical physics, focusing on rigid body dynamics and the associated mathematical principles.

Lumiere leverages object-oriented programming (OOP) principles to create a structured environment where users can easily define physical entities and their interactions. By encapsulating physical properties and behaviors within classes and objects, Lumiere streamlines the development of simulations, enabling users to concentrate on the physics rather than the intricacies of programming syntax. The ability to visualize simulations enhances the understanding of complex concepts in mechanics. Lumiere uses visual components that allow users to observe the behavior of rigid bodies in real time, facilitating an interactive learning experience.

Lumiere consists of different physics simulations that will help users understand about the different perspective of the bodies in newtonian physics. Lumiere will simplify the modeling, simulation, and analysis of the mechanical systems involved in this classically based physics. It will be concerned with rigid body dynamics via imposed forces and torques, resting its physics on Newtonian mechanics. Typical equations are Newton's Laws, but also stress-strain relationships and laws of elasticity, conservation of momentum and energy. The applications of the DSL consists the physical aspects of fundamental principles of force and mechanics.

* *Department of Biology, University of Examples, London, United Kingdom*
[1] *Department of Chemistry, University of Examples, London, United Kingdom*

2 AIM OF THE DSL

The proposed DSL aims to simplify the simulation processes for the end users.

- Complexity Abstraction: Creating a high-level interface for the specification of Physio-mechanical systems and interactions, abstracting the usually low-level numerical methods needed for the simulation process.
- Simplify Integration: Enable the user to describe solid body dynamics by means of familiar mathematical notations and physical laws, easing their task of defining boundary conditions, forces, and material properties.
- Optimization: DSL will understand the semantics of the provided system and use the properties of mechanics to produce an optimized simulation while also incorporating the hardware optimization.
- Real-Time Visualization: A significant feature of Lumiere is its real-time visual simulation capability, allowing users to observe and interact with the behavior of rigid bodies dynamically.

Overall, Lumiere will provide for ease in modeling complex mechanical systems and reduce manual coding burdens from general-purpose languages, enabling high accuracy, efficiency.

3 PROGRAM FLOW/STRUCTURE

```
~Sample DSL Program: Simulating a Moving Body with Forces
__USE__ lib.mech
__HARDWARE__ ~can be __CPU__ __GPU__ or __MAC__

~static type
__FIX__ MAX_BODIES  10;
__FIX__ GRAVITY  9.81;

~Declare variables
int body_count = 0;
float simulation_time = 0.0;
boolean is_simulation_running = true;
vec body=<10, 20deg>;
string yes= "car";


~Main simulation function
int sim() {
    Body bodies[MAX_BODIES];  ~object of class bodies
    int i;

    ~Initialize bodies
    check_until (i = 0; i < MAX_BODIES; i++) {
        bodies[i] = Body(x=10, y=10, vel=(12,10), force=(10,10));
    }

    ~ Run simulation loop
    check_until (is_simulation_running) {
        ~ Update all bodies
        check_until (i = 0; i < MAX_BODIES; i++) {
            bodies[i].force = bodies[i].calculate_force();
            bodies[i].update_position(0.01);  ~ Time step = 0.01
        }

        ~ Increment simulation time
        simulation_time += 0.01;

        ~ Stop simulation if time exceeds limit
        if (simulation_time > 10.0) {
            is_simulation_running = false;
        }
        animate():
    }
}
```

4 LEXICAL FORMAT

4.1 Comment

DSL will support single and multiline comment. The single line comment would be specified by ~ and multiline comments would be specified by using \ ~ (start) and ~ \ (end). A simple example is given below:

```
~This is a single line comment.
/~This is amultiline comment with
different specificaiton ~/
```

4.2 Identifiers

The identifiers can have letters, numbers or underscore. However, they must start with a letter or underscore.

4.3 Reserved Word

The reserved words in the DSL are:

int	float	char	boolean
string	vec	Cluster	void
Body	Basic	force	if
else	checkuntil	break	Continue
__FIX__	__USE __	__ HARDWARE__	__GPU__
__CPU__	OS	fix	return
try	Catch	typeof	Class
Extends	Inherit	hidden	

Table 1: Reserved Words in the DSL

5 DATA TYPES

Data Type	Description	Memory Size	Range	Example
`int`	Integer type for whole numbers without decimals.	4 bytes	-2,147,483,648 to 2,147,483,647	`int num = 25;`
`float`	Floating-point type for numbers with decimals.	4 bytes	1.2E-38 to 3.4E+38	`float num = 1.75;`
`char`	Character type for storing single characters.	1 byte	ASCII values (0 to 255)	`char text = 'A';`
`boolean`	Boolean type for true/false values.	1 byte	`true` or `false`	`boolean value = true;`
vector	Vector type for vectors with magnitude and direction	8 bytes	-2,147,483,648 to 2,147,483,647	vec force = <10,12>

Table 3: Basic Data Types in the DSL

6 OPERATORS AND EXPRESSIONS

The major operators used in the DSL are:

Operator	Associativity	Description
+	left-to-right	Used for addition.
-	left-to-right	Used for subtraction.
*	left-to-right	Used for multiplication.
\	left-to-right	Used for division.
%	left-to-right	Used for modulo operations(remainder calculation).
| |	left-to-right	Used as absolute value operator for data types likes int, float but for vectors computes the magnitude.
+=	right-to-left	Compound assignment operator.
-=	right-to-left	Compound assignment operator.
*=	right-to-left	Compound assignment operator.
\=	right-to-left	Compound assignment operator.
<	left-to-right	Less than operator.
<=	left-to-right	Less than equal to operator.
>	left-to-right	Greater than operator.
=>	left-to-right	Greater than equal to operator.
ˆ	right-to-left	Power operator.

7 STATEMENTS

The specified DSL supports following statements:

- **Conditionals**: These statements control the flow of execution based on conditions.

```
    if (condition) {
        // Code to execute if the condition is true
    } else {
        // Code to execute if the condition is false
    }
```

  Eg.

```
if (simulation_time > 10.0) {
    is_simulation_running = true;
    time_step=0.02;
    }
else{
        time_step=0.01;
    }
```

- **Loop**: Used for repeating a block of code based on a condition or a range.

```
    check_until (initialization; condition; increment) {
        // Code to repeat
    }

    check_until(condition){
        //code to repeat
    }
```

  Eg.

```
        check_until (is_simulation_running) {
            check_until (i = 0; i < MAX_BODIES; i++) {
            bodies[i].force = bodies[i].calculate_force();
            bodies[i].update_position(0.01);
            Time step = 0.01
        }
    }
```

- **Use**: A specialized statement used to activate features or components.

  ```
  __use__PACKAGE_NAME;
  ```

- **Hardware Specification**: This allows the specification of hardware components such as memory, CPU, or peripherals.

  ```
  __HARDWARE__CPU__ , __GPU__ or __MAC__
  ```

8 DECLARATIONS

The DSL allows for the declaration of variables, constants, and functions. These declarations define the data types, initial values, and functions that can be used throughout the code.

- **Variable Declaration**: Variables can be declared with specified types and optionally initialized with values.

  ```
  type variable_name = value;
  ```

  Example:

  ```
  int counter = 0;
  string message = "Hello";
  ```

- **Constant Declaration**: Constants are immutable values that can be used throughout the code but cannot be modified.

  ```
  const type constant_name = value;
  ```

  Example:

  ```
  const int MAX_SIZE = 100;
  ```

- **Function Declaration**: Functions can be declared to perform operations and can return values. They accept parameters and may include type definitions for the return type.

  ```
  type function_name(parameter1_type param1, ...) {
      // Function body
      return value;
  }
  ```

  Example:

  ```
  int add(int a, int b) {
      return a + b;
  }
  ```

9 OOPS

The specified DSL also supports Object Oriented Programming. The paradigm supported are inheritance and overloading. For this, two base class have been provided.

1. Basic class: This class represents any basic object that has mass, velocity and certain position in space. Thus, the atrributes of the basic class are *mass*, v_x,v_y, *x*, *y* and a base *id*.
2. Body: The body class inherits the basic class and adds some of its own attributes to it like force, equation of forces, etc.

The `Basic` class represents any basic object with the following attributes:

- `mass`: The mass of the object.
- `vx`, `vy`: Velocity components in the x and y directions.
- `x`, `y`: Position components in the x and y directions.
- `base_id`: A unique identifier for the object.

Example of a `Basic` class in the DSL:

```
class Basic {
    float mass;
    float vx, vy;
    float x, y;
    int base_id;
}
```

Body Class

The `Body` class inherits the `Basic` class and adds the following attributes:

- `force`: The net force acting on the object.
- `equation_of_forces`: An equation or function that describes the forces acting on the body.

Here is an example of the `Body` class in the DSL:

```
class Body extends Basic {
    float force;
    string equation_of_forces;
}
```

CLASS INHERITANCE EXAMPLE

Class	**Attributes**	**Inheritance Example**
Basic	`mass`, `vx`, `vy`, `x`, `y`, `base_id`	`class Basic {...}`
Body	Inherits `Basic` attributes + `force`, `equation_of_forces`	`class Body extends Basic {...}`

Table 5: Class Inheritance in the DSL

10 UNDERLYING PHYSICS

10.1 Force

Force is something that acts on a 'Body' defined in our DSL to produce motion or alter the body's state of rest. It is the primary cause of change in velocity and is a vector quantity. In our DSL, we are concerned with two types of forces.

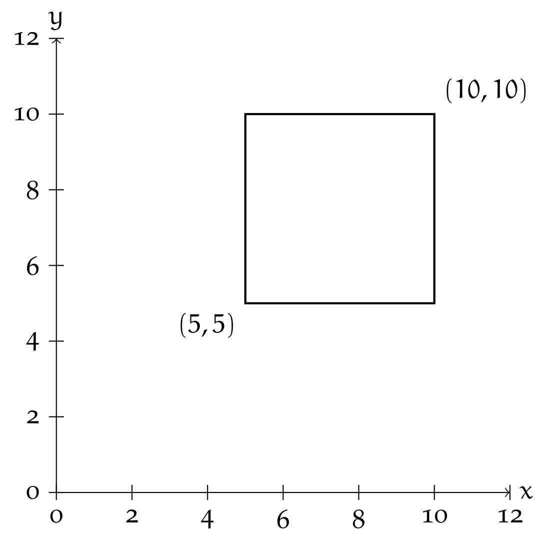
1. **Force of Motion**
   The most basic force that acts on a body of mass 'm' to produce and acceleration 'a'. Mathematically , it is represented as :-
   $F = m \cdot a$

2. **Springs Force**
   The force exerted by a spring when it is compressed or stretched and given by Hooke's Law. Mathematically , it is represented as :-
   $F = -k \cdot x$
   where,
   k is the spring constant and x the displacement from spring's natural length.

To facilitate the above forces, we define two distinct properties for the Force.

a) **Range**: The range of the force defined the spatial extent or boundary within which the force can act upon a body. If any body lies on this field, it will experience the force. Beyond this range , the force will have no effect on that body. The range can be defined by simple:

   - **Two Coordinates:**
     When just two coordinates is passed, it is interpreted as the bottom left and the top right coordinate of a rectangular boundary.
     For example:
     If (5,5) and (10,10) is passed it is interpreted by the compiler as :-



   - **More than two coordinates:**
     If more than two coordinates are passed it is interpreted as the vertices of a polygon and acts as the range of the force.

11 ERRORS

Errors are used to define how we handle many exceptions cases that can occur during program.

11.1 Types of Errors

There are various types of errors defined in our DSL.

11.2 Error Handling

The DSL supports structured error handling using `try` and `catch` blocks. These blocks allow for catching specific runtime errors and taking appropriate actions. The system can handle the following errors:

- **String Length Error**: This error occurs when a string exceeds the allowed length.
- **No Collision**: This error is raised when an expected collision between objects fails to occur.
- **Out of Canvas (Spring Branch Error)**: This error occurs when an object moves outside the canvas boundaries.
- **Time Limit Exceeded**: This error is thrown when a process takes longer than the allowed time limit.

Each `catch` block is designed to handle a specific error, ensuring the system can gracefully recover or notify the user of the issue.

12 BUILT-IN LIBRARIES

As built in library, a *Mech* libraries is served with the DSL to provide with the generic body features to aid in simulation and computation. Mech library includes two types of built-in bodies namely, spring and string.

1. Spring
2. String

13 BUILT-IN FUNCTIONS

Avg(vectors) -> this should give an average vector based on all the vectors like an average position vector
Center of Mass: Takes all the balls and gives the center of mass of all. This has a lot more scope to do things based on the cluster of balls or position boundaries. If we do this, then we will score marks for sure. Similar considerations apply to the center of gravity.
Solve Trajectories: This should take in the canvas and be able to deduce all the equations of motion, or at the very least, plot all the points.

- **Spring**: This body simulates the behavior of a spring in mechanical systems, allowing for elastic deformation and force computation based on Hooke's law.

  ```
  spring {
      length: value;
      stiffness: value;
      damping: value;
  }
  ```

  Example usage:

```
spring {
    length: 10;
    stiffness: 50;
    damping: 0.1;
}
```

- **String**: This body represents a flexible string-like object used to simulate tension, elasticity, and other physical properties.

  ```
  string {
      length: value;
      elasticity: value;
      mass: value;
  }
  ```

  Example usage:

  ```
  string {
      length: 15;
      elasticity: 100;
      mass: 0.5;
  }
  ```

The `Mech` library provides fundamental components for simulating physical systems, simplifying the development process for mechanical computations.

14 POSSIBLE OPTIMIZATIONS

1. Based on the specs, set up an OpenGL context that optimally utilizes available GPU features such as Shader Model 5.0 or higher for physics computations.
2. Lumiere automatically detects whether the system is Cartesian or radial by analyzing the syntax used, allowing it to adapt computations and optimizations accordingly for more efficient handling of physics equations.
3. Split physics computations into independent tasks, such as separate calculations for each axis in Cartesian systems or components in radial systems.

15 OPERATORS DEFINITION

15.1 Addition and Subtraction

For primitive data types these operator evaluates as expected but for the data types defined specifically for this language will behave depending on the type of operands.

1. Both operands are vectors:
   If both operands provided are vectors the evaluated result will simply be their sum or difference.

   ```
   // Say F1 and F2 are two forces.
   F3 = F1 + F2
   F4 = F1 - F2
   ```

If F1 is a vector of (1,2) and F2 is a vector of (5,7) the results of the operations will be :

- F3 : (6,9)
- F4: (-4,-5)

15.2 Multiplication

1. Both are vectors:
   In case of both operands being vectors , then their dot product is calculated.

   ```
   // Say F1 and F2 are two forces.
   res = F1 * F2
   ```

   If F1 is a vector of (1,2) and F2 is a vector of (5,7) the results of the operations in **res** will be 19.

2. One operand is a scalar or other vector:
   In this scenario the evaluated result is the vector being scaled up or down.

   ```
   // Say F1 and k is either an int of a float.
   F2 = k * F1
   ```

   If F1 is a vector of (1,2) and k = 10, the resulting F2 will be (10,20)

REFERENCES