NPSL-2D Language Reference Manual

Glenn Barney (gb2174@columbia.edu)

October 18, 2007

NPSL-2D is a simulation modeling language for 2D Newtonian physics. It is built around the customization of forces that act on Shape (rectangle or circle) objects. A simulation is coded then run, translated to a visual display for the user.

1. Lexical Conventions

Tokens:
Tokens consist of Identifiers, Keywords, Constants, and Operators.

Whitespace : Spaces (or blanks), all tabs, ('\t') and line terminators (treated as '\r' '\n' for DOS or '\n' for UNIX machines) are considered white space. They exist solely as token separators.

Comments
Multi line comments are supported, and begin with "/*" and end with "*/". Comments can not be nested.

Identifiers
Identifier is a sequence of characters containing any letter or digit, but must start with a letter. Identifiers are case sensitive.

Constants
Constants are either integers (any series of digits), floats (syntax explained later), the boolean value *true* or *false*, or string constants (syntax also explained later).

Keywords (explained in the document, capital words are internal types or objects)

int	float	bool	string	false
true	Point	Line	Circle	Rect
Force	at_tick	interval	foreach_shape	foreach_touching_shape
shape	touching	function	World	Logger
void	if	else	return	for
while	break			

Expressions
Expressions evaluate to some value at the end of their execution. They are generally evaluated from operators but can be as simple as the value true. Here is a list of valid expressions:

lvalue
function-call
unary operator expression (see operator list)

binary operator expression (see operator list)
assignment expression (example : x = 5;)
type definition

Statements and Flow control

A statement controls logic flow and occur in order. A statement list is one or many statements, or nothing:

```
statement-list :
        statement; | statement-list | e;
```

General statements.
-Type declaration and definition :

```
type <identifier>;  //declaration
type <identifier> = rval;
        rval : constant
                | expression evaluating to type;
```

Braces {} are found in the follwing conditions :

-to mark a begin end block for conditional statements (if, while, etc)
-to mark a begin end block for a function
-to mark a begin end block for a foreach section

Braces are not supported for general insertion arbitrarily into code.

Conditional (can be nested) :

```
if (expression) {
        statement-list
}
else { statement-list}
```

While statement :

```
while ( expression ) {
        statement-list
}
```

For statement :

```
for ( expression1; expression2; expression3) {

        statement-list;
}
```

A for statement executes first executes expression1. Then it repeatedly executes the statement list followed by expression3 as long as expression2 is true.

while and *for* also support the *break* statement, which exits the while or for loop :

```
break;
```

The *return* statement returns from a function call :

return; or *return* (expression);

2. Memory Management, Scope, Constructors and Functions

Any constructor C can access any object if it was declared and defined before C's constructor starts. Objects created in NPSL-2D are all of global scope and are only destroyed at the end of the program. The language has no support for multiple threading and therefore objects in the main memory of the program are only accessible by the thread running the simulation. Functions can take objects of both basic and complex types as parameters; they will be able to directly access these objects through the passed reference. In this sense, function scope acts like a goto statement, with the same scope as the code that called the function.

All points and shapes are treated in units on a Cartesian coordinate system starting with point 0,0 in the bottom right. x direction increases to the right and y direction increases up[1].

Functions are defined as follows:

```
function <return type> <identifier>(<ParamType> <param identifier>, <ParamType>
                                        <param identifier>…. ) {

        statement 1;
        statement 2;
        ….
        statement n;
        return <type>;

}
```

More formally :
function return-type *identifier (* arg-list) body
arg-list :
    type *identifier*
    | type *identifier*, ag-list

body :
    statement-list

Return types can be one of any basic or complex built in type, or *void*. Parameter types can be one of any basic or complex built in type, including the type Shape. Functions may have side effects as parameters are passed by reference, but no function can pass another function as a parameter; there are no function closures in NPSL-2D.

[1] This is in contrast to the original proposal document. I will use a AffineTransform in Java in the translation layer to display pixels with the bottom left being the origin.

3. Types

Basic types

```
<data type> :
              int | bool | float | string
```

Basic types all initialize to default values if declared but not defined. Integers, floats, initialize to 0. Booleans initialize to false, and strings initialize to the empty string “”;

Integers – declared with keyword *int*.
Consist of a series of Digits

Booleans – declared with keyword *bool*
Consist of either the keyword “false” or “true”.

Floats – declared with keyword *float*
Floats are floating point numbers. They are an optional integer number followed by either a fractional or exponential part, or both, and are computed base 10. Exponent part is “E” or “e” followed by an optionally signed integer. So these are valid floats (in order):

- Integer, decimal point, fraction, exponent
- Integer, decimal, fraction
- Integer, decimal, exponent
- Integer, decimal
- Integer, exponent
- Decimal, fractional
- Decimal, exponent.

Strings – declared with keyword *string*
The String type is used to store character strings, and is primarily used for logging purposes and for use as keys when defining shape instance properties (such as coefficients of friction). The default value of a string upon declaration is the empty string, "". A string can be initialized by setting it to a constant, and you can use the backslash character as an escape to include the newline character or quote character in your string.

```
String example; // example is the empty string
example = “test\n”;
```

Declarations and Definitions

Variables can be declared with the type and then the identifier :
<type> *identifier*;
And can be both declared and defined at the same time:
<type> *identifier* = value;

Value must be a literal of the same type, or an expression that evaluates to that type.

For Primitive data types:
int *identifier* = (<int> | expression evaluating to <int>) //int x = 4;
float *identifier* = (<float> | expression evaluating to <float>); //float y = 3.452e10
boolean *identifier* = (<bool> | expression evaluating to <bool>); bool isTrue = true;

Other legal statements are :

```
     int z = 3 + 2;
    float k = 3 + 3.34;
    bool test = true & false;
```

Built in Complex Types

<complex type> :
    Point | Line | Shape | Circle | Rect | Force

The general format for constructing a complex type is :

```
    <type> identifier {
        field1 = defined variable or constant;
        field2 = defined variable or constant;
        ….
        //for each type that is an optional or required for construction
    }
```

And the general form for accessing fields in complex objects is the dot notation:

```
    log(indentifier.field1);
```

Point

A Point is a pair of integers, declared with the keyword *Point*. A Point must be constructed with two arguments, x and y, both integers.

```
    Point <identifier> {
        x = int;
        y = int;
    }

    pont myPont {
        x = 1;
        y = 2;
    }
```

The default value for Points declared but not initialized is 0,0. To access x and y use myPoint.x and myPoint.y respectively.

Line

A *Line* is simply a set of two Points. To define a *Line*, create two *Point* objects and use these to create your *Line*. Both *Point*s are required. Be careful when using undefined *Point*s in your Line object as they will initialize to 0,0, creating a valid, but perhaps unexpected *Line*.

```
    Line <identifier> {
        point1 = Point;
        point2 = Point;
    }
    Point firstPoint { x =1; y =2; };
    Point secondPoint {x = 2; y =3; };

    Line myLine { point1=firstPoint; point2=secondPoint };
```

A *Line* can represent a *Point* if both point1 and poin2 have the same x and y values. *Line* also has the member functions:

float length() which returns a float value of the length of the *Line*.
float slope() which returns a float value : rise over run. Returns 0 if Line is a point.

Shape Types

There are two basic shape types supported, Circles and Rectangles. The names of these types are Circle and Rect. They both derive from abstract type Shape which cannot be defined or declared. A Circle or Shape can be treated as a Shape through functions or through the automatic iterator keywords *foreach_shape* and *foreach_touching_shape*, which are only accessible inside a force, inside a *at_tick* block.

```
    ex : function int comapreShapes(Shape first, Shape second);
```

Shape fields can also be accessed from the Shape object, as described in the “Shape fields” chart.

Shapes

Declare and define a shape of type Circle or Rect with the following syntax. Optional fields below are marked with default. Shapes must be declared and defined at the same time, there is no declaration only syntax for shapes. If you must declare a shape without any detail, initialize all the parameters to zero and set them later through access. Like any other float to int access, an implicit int to float cast will occur if you set any of these parameters to integers. xCoord and yCoord are the center of the Shape.

Shapes can be accessed in

```
Circle <identifier>{
    mass = float;
    free = bool;  --optional; default to true
    radius = int;
    xCoord = int;
    yCoord = int;
    color = color type; --optional; default to grey
    velocity = float; --optional; default to 0
    direction = float; --optional; default to 0
}

Rect <identifier>{
    mass = float;
    free = bool;  --optional; default to true
    width =  int;
    height =  int;
    xCoord = int;
    yCoord = int;
    xTopLeft = int; -- optional set to xCoord – ½ width
    yTopLeft = int; --optional set to yCoord – ½ height
    xBottomRight = int; -- optional set to xCoord + ½ width
    yBottomRight = int; --optional set to yCoord + ½ height
    color = color type; --optional;default to grey
    velocity = float; --optional; default to 0
    direction = float; --optional; default to 0
}
```

Shape fields
Common to all Shape objects

Name	Type	Purpose	Required / default
mass	float	Weight in "units" (eg kg) of the Shape	required
free	bool	True if not tied to background	def : true
xCoord	int	x coord of center point	required
yCoord	int	y coord of center point	required
color	Color	color of Shape when drawn	def : grey (so sad)
velocity	float	current speed of Shape	def: 0
direction	float	current direction of velocity	def: 0

Specific to Circle

Name	Type	Purpose	Required / default
radius	int	radius of circle	required

Specific to Rect

Name	Type	Purpose	Required / default
width	int	width of rectangle	required
height	int	height of rectangle	reqired
xTopLeft	int	x coord of top left corner	def: xCoord – ½ width
yTopLeft	int	y coord of top left corner	def: yCoord - ½ height
xBottomRight	int	x coord of bottom right corner	def: xCoord + ½ width
yBottomRight	int	y coord of bottom right corner	def: xCoord + ½ height

Accessors for Circle's and Rect's member fields are used with the dot function. For example, assume a Circle name myCircle was declared and defined. Then you can say :

```
 int width = myCircle.mass;
myCircle.height = 2.45;
```

Forces

Forces act on a shape at each tick. The *foreach* keyword is used here, and it is the only time it is used. There are two types of forces, global forces and object forces, but it's really up to the user to implement a force by having the force act at each tick. For example, we can create member variables unique to a Force before the at_tick statement. These variables can be referenced later in other forces. To define a force use the following syntax :

```
Force <identifier> {
        statement 1;  //variable declaration
        statement 2; //variable declaration

        at_tick <int when>? < interval>?[2]{
                statement 1;
                statement 2;
                ….
        }
}
```

There are two util functions for a "force" to break a force up into X and Y component. You can pass it a magnitude and a direction, and this beaks up the vector into a X and Y magnitude (positive if in the +x or +y direction, negative in the –x or –y direction) They return new forces and are :

```
        function int getXPart(float magnitude, float direction);
        function int getYPart(float magnitude, float direction);
```

Note these functions cast the result to an int, they are like a dx and dy on a point.

For example you can call

```
        Force myForce { //initialized} ;
        Force xComponent = myForce.getXpart();
```

The at_tick <int when> < interval> section is the main area where forces are applied, and they can act on two sets of Shapes. The first set of Shapes is the entire set in the world, this set is maintained by the compiler and accessed as *shape* under the *foreach_shape* block. The second set is the set of Shapes that the current shape is touching, this is accessed as *touching* under the *foreach_touching_shape* block. Any statement that is legal can be executed in the foreach block, and statements can access other forces.

```
foreach_shape {
  shape.y = shape.y - 9.8; (* apply gravity *)
 }
```

[2] ? here means one or zero times, the value when and interval is optional

Every shape has a list of Shapes it is touching at the current tick. This list can be accessed only through the foreach_touching keyword, which lets you iterate on the touching Shape list. Access to the individual Shape in the loop is accessed through the "touching" keyword. They are used as so

```
foreach_touching_shape {
        touching.mass = 4;
        float x;
        float y;
        if  (touching instanceof Circle) {
                print touching.radius;
        }
}
```

If an integer number is provided after the "at_tick" phrase, then a force will only act at that specific tick number. So for example at_tick 40 will have the force act once and only once at tick 40 (time = 40 seconds). If no integer is provided, then the constant force acts *on every tick*. Gravity is an example of such a constant force.

If both the when and interval numbers are present then the force acts every "when" ticks. So at_tick 4 interval would mean the force acts every 4 ticks.

Note on internal representation of force application: The language directly supports the mechanism for changing the location of a Shape by having the programmer move the x and y components of the object inside the force action. NPSL-2D runtime will apply all calculations effecting xCoord and yCoord on each tick for every Shape. It will then attempt to move every Shape in the world from its original xCoord and yCoord position to its new xCoord and yCoord position at the same time. If there is a collision then this will be calculated - the Shape can clearly end up in a different location then the pure forces intended after one tick has passed.

Shape Interaction

Every Shape can keep track of user defined numeric variables that relate one object to another under a specific force. We call these variables *values*, but they can be used for any reason. To set a *value* on a Shape, call the Shape setValue() member function. Pass in the force it relates to, the unique name for this value relative to this Shape, the second object that it is relative to, and finally the float value to set.

void setValue (*Force* f, *string* name, *Shape* s, *float* value);

To get a value on a Shape, call getValue(), passing in the same parameters as getValue() for retrieval.

float getValue(*Force* f, *string* name, *Shape* s);

If you set a unique value (that is a force, name, and shape pair tuple) to a new value, it will overwrite the old value.

We can also store data from one force into a specific coefficient (named so after the coefficient of friction) in a Shape, so that shape can use that force for other force calculations. This function is called as a Shape member function named setCoefficient(). Pass the unique name for this coefficient relative to this Shape, the second object that it is relative to, and finally the float value to set.

void setCoefficient (*string* name, *Shape* s, *float* value);

To get a value on a Shape, call getCoefficient(), passing in the same parameters as setCoefficient for retrieval.

float setCoefficient(*string* name, *Shape* s);

If you set a unique coefficient (that is a name, and shape pair tuple) to a new coefficient, it will overwrite the old coefficient.

Compile time checking ensures that every Shape be constructed with it's basic elements, and every object force that is referenced in a setCoefficient() call exists. However it does NOT ensure that a coefficient exists and if a getCoefficient() call can not find your value then the code will throw a runtime exception.

Collisions and the Built in Elastic Coefficient.

In order to support collisions, the runtime will automatically have objects in fully elastic collision by default. This means objects will lose no kinetic energy on collision. In order to have inelastic collisions, the user must make the call :

    shape.setElasticityCoef(Shape s, float val).

This call is commutative; that is

    shape1.getElasticityCoef(shape2) == shape2.getElasticityCoef(shape1).

When one call is made both elasticity coefficients are set. The runtime will use these coefficients to effect the dx and dy between ticks representing the loss of kinetic energy in a collision.

4.Operators

Mathematical numeric operators act on *floats* and *int*s. For binary operators, if a *float* and *int* are mixed as parameters, then an *int* is automatically cast into a *float*. If an *int* is stored to a *float*, it is also implicitly automatically cast to a *float*. However *float*s are not cast back into *int*s for loss of precision. For that reason the statement int x = 3 + 2.3 is illegal. The variable x must be declared as a *float*.

Unary Mathematical Numeric operators		
Operator	**Function**	**Associativity**
-	Negation	N/A
(	Open Parenthesis – Grouping operator	Left to right relative to

		close parenthesis
)	Close Parenthesis –grouping operator	Left to right relative to open parenthesis
Binary Mathematical NumericOoperators (operates on float and int)		
+	Addition	Left to right
-	Minus	Left to right
*****	Multiplication	Left to right
/	Division	Left to right
Binary Mathematical Relational Operators		
<	Less than	Left to right
>	Greater than	Left to right
=	Assign	Right to left
==, !=	Equal to, not equal to (int only)	Left to right
Binary Boolean Logical Operator		
&	Logical And	Left to right
|	Logical OR	Left to right

Binary Point		
==, !=	Equal to, if both x and y are equal, not equal if either one or both are different	Left to right
Binary Line		
==, !=	Equal to, if both point1 and point2 are equal, not equal if one or both are different	Left to right

Unary String operators		
Operator	**Function**	**Associativity**
[x]	Character substring – returns a substring with the character at index x	N/A
[x, y]	Character substring – returns a substring with the characters starting at index x and ending at index y	N/A
String Comparison (operates on magnitude of force component only)		
<	Alphabetically Less than	Left to right
>	Alphabetically Greater than	Left to right
=	Assign	Right to left
==, !=	Equal to (compares on string contents), not equal on string constants	Left to right
String Access Operators		
+	Concatenation – returns a new string	Left to right
Object (Shape) operators		
instanceof	Run time type checking – returns true of left-hand	Left to right

	side same type as right-hand side, otherwise false	

5.Constants

Built in constants:

Name	Type	Value
PI	float	3.14159265
LEFT	float	0
UP	float	PI / 2
RIGHT	float	PI
DOWN	float	3* PI / 2

Type Color, built in constant type.
Color.black - The color black.
Color.blue - The color blue.
Color.*cyan* - The color cyan.
Color.*darkGray* - The color dark gray.
Color.gray - The color gray.
Color.green - The color green.
Color.lightGray - The color light gray.
Color.magenta- The color magenta.
Color.orange - The color orange.
Color.pink - The color pink.
Color.red - The color red.
Color.white - The color white.
Color.yellow - The color yellow.

Built in functions
Math basic built in functions exist :
function float cos (float radian) ; //returns cosine of radian
function float sin (float radian) ; //returns sin of radian
function float tan Funciton (float radian); //returns tangent of radian

function float arcsin (float radian); //retunrs arcsin of radian
function float arccos (float radian); //retunrs arccos of radian
function float arctan (float radian); //retunrs arctan of radian

Special static world objects:
World – Sets the view of the world for the user and manipulates the speed of the simulations. Funcions available on it :

setBounds(int x, int y) : This function must be called sometime before the end of the program, and before the simulate function(). It set’s the viewable screen size in units. The x and y size is dependent on the size of the objects you define. A good way to think of the parameters to this function is each unit is a meter, and objects under gravity accelerate at 9.8 meters per second. However this is just one use case, it is up to user to define his objects in units relative to the bounds.

simulate() : This function is optional, as it must be called at the last line of the program. If you don't write it the compiler will insert it for you.

speedUp(float speed) : Speeds up the simulation by the float factor speed. As a default, one tick happens every second. For example, passing 4 to speedUp causes four ticks to pass every second.

slowDown(float speed) : Slows the simulation by the float factor speed. As a default, one tick happens every second. For example, passing 4 to slowDown causes one tick to pass every four seconds.

Logger – has the Logger.log(*string*) function, which logs a string to a console. All basic types, float, int, and bool are cast to strings for logging purposes only so that the function call below works:

```
int test = 4;
Logger.log(test); //prints 4 to log
```

Logger has one other function, setOutput. You can set the name of the log file here if you do not wish to log to console (the default).

```
Logger.setOutput("myFile.txt");
```

6.Example

Here's an example of how a dependent force hierarchy would be used, including coefficients. Note how much of the interaction is dependent on the user and how they define the force.

```
//assume user defines function to compute the line intersection of two Shapes,
//returning a Line. (Can be a line object with the same end and begin point (a //point) if
Shape is touching at just a point
function Line intersect(Shape shape1, Shape shape2) (){…};

//also assume user defines a function that returns an x component of a force
//given a magnitude and direction

World.setBounds(800,600);

Circle  ball{
        mass = 5;
        free = true;
        radius = 2;
        xCoord = 400;
        yCoord = 300;
```

```
}


Rect ground {
mass = 100;
free = false;
xCoord= 400;
YCoord = 50;
width = 200;
height = 100;
}

ball.setCoefficient("friction", ground, .35);

Force gravity {
        float direction = DOWN;
        float acceleration = 9.8;

        at_tick {
                foreach_shape {
                //from v = ½ at^2 + vi*t)
                shape.yCoord =  shape.yCoord 9.8; // apply gravity
                }
        }
}

Force normal {

        at_tick {
                //going to calculate the force normal, friction, and force down
                //the plane assuming that the touching shape  is fixed
                foreach_shape {
                    foreach_touching_shape {

                        //the normal force is gravity * cos (theta) where theta
                        //is the angle between the resting surface and the x  axis

                        float angle;  //angle from x axis is arctan slope
                        float forceNormalSize;
                        float forceNormalDirection;

                        float slope = intersect(shape, touching).slope;
                        angle = arctan(slope);
                        forceNormalSize = gravity.acceleration * cos(angle);
                        forceNormalDirection = gravity.direction +PI+ angle;

                        //store this value
```

```
                              shape.setValue (normal, “normSize”,
                              touching, forceNormalSize);
                              shape.setValue (normal, “normDir”,
                              touching, forceNormalDirection);

                              //apply the force;
                              shape.xCoord = shape.xCoord + getXPart(forceNormalSize,
                                                                      forceNormalDirection);

                              shape.yCoord = shape.yCoord + getYPart(forceNormalSize,
                                                                      forceNormalDirection);
                              }
                    }
          }
}

Force down_slope {

          at_tick {
                    //calculate the force force down assuming touching shape  is fixed
                    foreach_shape {
                        foreach_touching_shape {

                              //the down force is gravity * sin (theta) where theta
                              //is the angle between the resting surface and the x  axis

                              float angle;  //angle from x axis is arctan slope
                              float forceDownSize;
                              float forceDownDirection;

                              float slope = intersect(shape, touching).slope;
                              angle = arctan(slope);
                              forceDownSize = gravity.acceleration * sin(angle);
                                forceDownDirection = gravity.direction +3*PI/2+ angle;

                              //store this value
                              shape.setValue (down_slope, “downSize”,
                                        touching, forceDownSize);
                              shape.setValue (down_slope, “downDir”,
                                        touching, forceDownDirection);

                              //apply the force;
                              shape.xCoord = shape.xCoord + getXPart(forceDownSize,
                                                                      forceDownDirection);

                              shape.yCoord = shape.yCoord + getYPart(forceDownSize,
                                                                      forceDownDirection);
                              }
```

```
                }
        }
}

Force friction {

        at_tick {
                //calculate the force friction assuming the touching shape  is fixed
                foreach_shape {
                    foreach_touching_shape {

                        //the friction force is mu * force normal
                        //is the angle between the resting surface and the x  axis

                        float angle;  //angle from x axis is arctan slope
                        float forceFrictionSize;
                        float forceFrictionDirection;

                        float slope = intersect(shape, touching).slope;
                        angle = arctan(slope);
                        forceFrictionSize = gravity.accelration * cos(angle) *
                        shape.getCoefficient("friction", touching);
                        forceFrictionDirection = gravity.direction +PI/2+ angle;

                        //store this value
                        shape.setValue (friction, "frictionSize",
                                touching, forceFrictionSize);
                        shape.setValue (friction, "frictionDir",
                                touching, forceFrictionDirection);

                        //apply the force;
                        shape.xCoord = shape.xCoord + getXPart(forceFrictionSize,
                                                        forceFrictionDirection);

                        shape.yCoord = shape.yCoord + getYPart(forceFrictionSize,
                                                        forceFrictionDirection);
                        }
                }
        }
}

World.simulate();
```

Note with this example above, there is no kinetic energy used. The system could check the velocity of the shape object in the foreach_shape block and then apply a kinetic coefficient of friction vs a static one, it is all up to the language user.