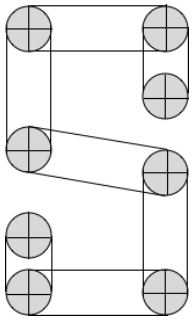

ShapeShifter

Programmatic geometric manipulations made simple

Stephanie Burgos, Ishan Guru, Rashida Kamal,
Eszter Offertaler, and Rajiv Thamburaj

Table of Contents

1. Introduction

ShapeShifter is a programmatic geometric modelling language that seeks to allow a user the ability to manipulate simple primitive shapes in order to create increasingly complex ones. Rather than require the user to manage meshes and vertex connectivity information, Shapeshifter abstracts these nitty-gritties away from the user, so that the user only has to call on familiar mathematical shape names with a set of initializing properties to summon it into existence. The user does not need to worry about the underlying mathematical or technical representation of the shape in order to enjoy a visually displayed representation. Shapeshifter relies on OpenGL for the rendering via the GLUT OpenGL toolkit[1]. For mesh manipulation, Shapeshifter uses the Cork Boolean Library[2].

As much of the technical manipulations are abstracted away from the user, this manual seeks to instruct the user about the nature of the Shapeshifter language, the specificities of the available lexical elements, operators, program syntax, etc., and provide instruction on how to build up increasingly complex visual representations based on the available primitives.

2. Lexical Elements

2.1. Identifiers

An identifier consists of a sequence of letters, digits and the underscore (_). The identifier must begin with a letter or underscore, but subsequent characters can be any combination of the previously mentioned characters. By convention, identifiers are are lowercase, and are most frequently used as variable names. Case does matter: abc and aBC are regarded as two unique identifiers.

Legal identifiers include: abc, _abc, abc123, _123abc, ab12c3

Illegal identifiers include: 123b, 67

2.2. Keywords

ShapeShifter has a set of keywords that cannot be used as identifiers. These include data types, booleans, control flow statements, file input/output statements and keywords for pre-defined shapes. Each of these can be seen below (more information regarding each can be seen in the tables below) :

Primitive Data Types: int, double, bool, string
Advanced Data Type: Array, void, Shape, Sphere, Cube, Tetra, Cone, Cylinder

[1] https://www.opengl.org/resources/libraries/glut/
[2] https://github.com/gilbo/cork

Booleans: true, false
Control Flow: if, elif, else, return, break, for, while
I/O: render, save, print
Special Value: NULL
Unit Shapes: SPHERE, CUBE, CYLINDER, TETRA, CONE
Operators: UU, NU (Note: Arithmetic and Logical operators are defined by symbols)
Built-in Functions: Scale, Rotate, Translate, Reflect, Intersect, Union, Difference, Copy, Render, Save

The table below shows each of these keywords along with descriptions and sample ShapeShifter syntax.

Primitive Data Types

Keyword	Syntax
int An integer value such as in C. *More information in the Integer Data type section.*	`int x = 7`
double An 8-byte precision value. *More information in the Double Data type section.*	`double x = 7.12345`
bool A representation of the true/false logical values.	`bool x = true`
string A collection of characters supported from the local character set.	`string x = "ShapeShifter"`

Advanced Data Types

Keyword	Syntax

void
A special data type used to represent return types of 'nothing'. *More information in the Data type section.*
	`void makeHat() {`
	`  ...`
	`}`

Shape An advanced data type that represents the underlying mesh representation. *More information in the Data type section.*	`Shape s1 = SPHERE` `Shape s2 = CUBE`
Sphere	`Sphere s1 = SPHERE`

A special type of Shape that represents the unit sphere.	
Cone A special type of Shape that represents the unit Cone.	`Cone c1 = CONE`
Cylinder A special type of Shape that represents the unit Cylinder.	`Cylinder c1 = CYLINDER`
Tetra A special type of Shape that represents the unit Tetrahedron.	`Tetra t1 = TETRA`
Cube A special type of Shape that represents the unit Cube.	`Cube c1 = CUBE`

Boolean Keywords

Keyword	Syntax

true
A representation of the logical value true. *More information in the Integer Data type section.*
	`bool x = true`
	`if (x == true){`
	`  ...`
	`}`

false
A representation of the logical value false. *More information in the Double Data type section.*
	`bool x = false`
	`if (x == false){`
	`  ...`
	`}`

Control Flow Keywords

Keyword	Syntax

if
Allows programs to control whether or not certain code needs to be run, provided conditions are met.
	`if(condition) {`
	`  //run this`
	`} else {`
	`  //run this`
	`}`

else
Allows programs to control whether or not certain code needs to be run, provided conditions are not met. This is not optional.
	`if(condition) {`
	`  //run this`
	`} else {`
	`  //run this`
	`}`

elif
Allows programs to control whether or not certain code needs to be run, provided conditions are met or not met.
	`if(condition) {`
	`  //run this`
	`} elif(condition) {`
	`  //run this`
	`} else {`
	`  //run this`
	`}`

return
A keyword used to define the return value for a function. Every statement that comes after a return will not be executed. Each branch of execution must include a return statement corresponding to the function return type.
	`Shape makeHat {`
	`  Shape hat = ...`
	`  ...`
	`  return hat;`
	`}`

break
A keyword used to break out of a loop regardless of whether or not the condition is satisfied. In the case of nested loops, it breaks out of the innermost loop.
	`while(true) {`
	`  ...`
	`  break;`
	`}`

for
A key word used to represent a loop that runs a certain number of times. Similar to the for loop in C. Variables must be predefined.
	`for (i = 0; i < 10; i++ ){`
	`  ...`
	`}`

while
A keyword used to represent a special type of loop that runs while certain conditions are met.
	`while(condition) {`
	`  /*run this section of code until condition is broken*/`
	`}`

Shape Keywords

Note: Each of the Shape keywords are predefined mesh values for the unit Shapes

Keyword	**Syntax**
SPHERE A special keyword used to as a constructor for the unit sphere.	`Sphere s1 = SPHERE`
CONE A special keyword used to as a constructor for the unit cone.	`Cone c1 = CONE`
CYLINDER A special keyword used to as a constructor for the unit cylinder.	`Cylinder c1 = CYLINDER`

CUBE A special keyword used to as a constructor for the unit cube.	`Cube c1 = CUBE`
TETRA A special keyword used to as a constructor for the unit tetrahedron.	`Tetra t1 = TETRA`

Operator Keywords (Excluding Symbols)

Keyword	Syntax

UU
A keyword that maps to the Union() function. It can only be applied to shapes.
	`Shape s1 = SPHERE`
	`Shape c1 = CUBE`
	`Shape t1 = TETRA`
	`s1 UU c1`
	`Shape sct = s1 UU s2 UU t1`

NU
A keyword used for the operator that maps to the Intersect() function. It can only be applied to shapes.
	`Shape s1 = SPHERE`
	`Shape c1 = CUBE`
	`Shape t1 = TETRA`
	`s1 NU c1`
	`Shape sct = s1 NU c1 NU t1`

DU
A keyword used for the operator that maps to the Difference() function. It can only be applied to shapes.
	`Shape s1 = SPHERE`
	`Shape c1 = CUBE`
	`s1 DU c1`

Built-in Functions

Note: More information on the functions can be found in the Built-In functions section of this reference manual

Keyword	Syntax
Scale(Shape x, double x, double y, double z) This is a predefined function that is used to scale shapes. The x,y, z components scale the Shape by the input values, which are positive-valued doubles.	`Shape s1 = SPHERE` `Scale(s1, 2.0, 2.0, 2.0)`
Rotate(Shape x, double x, double y, double z) This is a predefined function that is used to rotate shapes. It takes in a shape, three double values that correspond to counter-clockwise	`Shape s1 = SPHERE` `Rotate(s1, 90.0, 0.0, 0.0)`

rotation about the X, Y, and Z axes. The values correspond to degrees and the effective values are mod 360.0	
Translate(Shape x, double x, double y, double z) This is a predefined function that is used to translate shapes. It takes in the shape to translate and the input x,y,z values represent the offset in the corresponding directions.	`Shape s1 = SPHERE` `Translate(s1, 10.0, 0.0, 10.0)`
Reflect(Shape x, double a, double b, double c) This is a predefined function that is used to reflect shapes. It takes in a shape, as well as 3 doubles that define the plane of reflection: **a**x + **b**y + **c**z = 0.	`Shape s1 = SPHERE` `Reflect(s1, 0.0, 1.0, 0.0)`

Union(Shape x, Shape y)
Union ([Shape x, Shape y, ...])
This is a predefined function that is used to union shapes. The operator UU maps to this function. It takes in a two shapes and returns the union as a new shape. Alternatively, Union can accept an array of length 2 or more shapes.
	`Shape s1 = SPHERE`
	`Shape c1 = CUBE`
	`Union(s1, c1)`

Intersect(Shape x, Shape y)
Intersect ([Shape x, Shape y, ...])
This is a predefined function that is used to intersect shapes. The operator NU maps to this function. It takes in a two shapes and returns the intersection as a new shape. Alternatively, Intersect can accept an array of length 2 or more shapes.
	`Shape s1 = SPHERE`
	`Shape c1 = CUBE`
	`Intersect(s1, c1)`

Difference(Shape x, Shape y) Difference ([Shape x, Shape y, ...])
This is a predefined function used to take the difference of shapes. The operator DU maps to this function. It returns the result of y subtracted from x as a new shape. Alternatively, Difference can accept an array of length 2 or more shapes.
	`Shape s1 = SPHERE`
	`Shape c1 = CUBE`
	`Difference(s1, c1)`

Copy(Shape x)
This is a predefined function that is used to copy shapes. It returns a copy of the input shape.
	`Shape s1 = SPHERE`
	`Shape c1 = NULL`
	`c1 = Copy(s1)`
	`Shape s2 = Copy(c1);`

Render()
This is a predefined function that is used to display a shape on the screen. The function takes in a Shape.
	`Scene SCENE {`
	`  …`
	`  Shape s1 = SPHERE`
	`  Render(s1)`
	`}`

Save()
This is a predefined function that is used to save the current scene into a file. The function takes in a Shape.
	`Scene SCENE {`
	`  …`
	`  Shape s1 = SPHERE`
	`  Save(s1)`
	`}`

Integer literals are sequences of one or more decimal digits. Negative integer literals are expressed as integer literals with a hyphen prefix.

Examples: -5, 12

Double literals consist of one or more digits, a decimal point, and one or more decimal digits. As with integer literals, negative float literals are expressed by prepending a hyphen.

Examples: 0.1, 123.456, 12.1
Invalid examples: .0, 50.

String literals are double-quoted sequences of zero or more ASCII characters. Special characters in a string can be represented with escape sequences.

Examples: "ShapeShifter is the best", "Line one\nLine two", ""

2.3. Separators

Separators in ShapeShifter are used to separate tokens. Separators include:

```
, ; { } ( )
```

2.4. Comments

Comments in ShapeShifter follow the same convention as C, where single line comments go from // to the end of the line while multiline comments are everything in between /* and */. Nested comments, however, are not supported by the language.

For example,

```
// This is a comment in ShapeShifter
```

```
/*
      This
      Is
      Also
      a
      Comment
      In
      ShapeShifter
*/
```

2.5. Operators

ShapeShifter uses several operators to compare and combine data. Consult Figure 1 for a complete overview of relative operator precedence.

2.5.1. Arithmetic Operators

The general binary arithmetic operators are supported for numeric types, and are listed below:

```
+     -     *     /
```

Each of these are left associative, however, * and / have a higher precedence level than + and -.

2.5.2. Relational and Logical Operators

The following relational operators are all supported by ShapeShifter on numeric types, are left associative and have the same precedence level:

```
>     <     >=    <=
```

Similarly, the equality operators are also supported by the language, however, have a precedence level slightly lower than the relational operators listed above. These operators are:

```
==    !=
```

Compared to the arithmetic operators defined above, relational operators have a lower precedence.

The following logical operators are also supported:

```
&&    ||
```

These refer to boolean AND/OR, which return either true or false based on whether or not both or none of the comparators are true. These operators are both left associative and have a precedence lower than both equality and binary operators.

Comparisons and logical operators between Shape types are not supported.

2.5.3. Shape Operators

ShapeShifter also supports the following Shape operators:

```
UU    NU    DU
```

These represent the UNION, INTERSECTION, and DIFFERENCE of the shapes they are acting upon, and are both left associative. UU, NU, and DU have a lower precedence level than arithmetic operators, but a higher precedence level than relational operators.

2.5.4. Unary Operators

ShapeShifter also supports the following Shape operators for numerics and booleans:

```
!      –
```

Each of these operators are the NOT and NEGATION operators, with the simple functions of negating a boolean, or negating an integer value (multiplying by negative one). The precedence level of unary operators can be seen in relation to all other operators in Fig. 1.0; however, unlike all previously mentioned operators, NOT and NEGATION are right associative.

2.5.5. Other Operators

The following uncategorized operators are also supported by ShapeShifter:

```
( expression )      =
```

These operators are the PARENTHESES and ASSIGN operators, each of which artificially influences precedence levels by evaluating what is inside the parentheses first, and assigning the values to expressions respectively. Their associativity can be seen in Fig 1.

Figure 1: Table of Operators and Associativity
Note: Precedence increases as you go down the table

Operator Symbol	**Name**	**Associativity**
=	Assign	Right
|| &&	Or And	Left Left
== !=	Equals Not Equals	Left Left
< > <= >=	Less than Greater than Less than or equals Greater than or equals	Left Left Left Left
UU NU DU	Union Intersect Difference	Left Left Left
+ -	Addition Subtraction	Left Left
* /	Multiplication Division	Left Left
! -	Not Negation	Right Right
( )	Parentheses	Left

3. Data Types

3.1. Primitive Data Types

The standard data types, int, double, string and bool, are all supported by ShapeShifter, along with the generic Shape type and specific types for a number of basic geometric solids. Any of these types can be NULL. Details on each of these types are given below, while the syntax can be seen in the code samples below.

3.1.1. Integer

Integers in ShapeShifter are represented by 32-bit memory chunks in 2's complement. All the standard operators can be applied to integers.

The sample below shows the use of integers in ShapeShifter:

```
int x = 7;
int y = 14;
int z = x + y; // z == 21 and z is an integer value
```

3.1.2. Double

Double values are also supported by ShapeShifter in order to represent more precise values. Each double is required to have a decimal point and numbers both prior to and following the decimal point. They follow the same 8-byte standard as C.

The sample below illustrates the use of double values in ShapeShifter.

```
double x = 7.12345;
double y = 14.12345;
double z = x + y; // z == 21.2469 and z is a double value
```

3.1.3. Boolean

Boolean data types in ShapeShifter are data types with two values, true or false, that represent the logical truth values.

The sample below illustrates the use of boolean values in ShapeShifter.

```
bool x = true;
bool y = false;
bool z = x || y; // z == true
```

3.1.4. String

The string data type in ShapeShifter is a sequence of characters that are declared within double quotes. Strings are converted and represented as an array of characters, and hence, can take the same letters and symbols as characters from the same local character set.

String usage in ShapeShifter can be seen in the sample code below:

```
String s = "ShapeShifter";
print(s); // Prints out 'ShapeShifter' to the console
```

3.2. Non-Primitive Data Types

3.2.1. Shape

The Shape data type is the defining factor of ShapeShifter, since this is where shapes can be declared and modified. Shapes are objects with an underlying triangle mesh representation that

the user can manipulate, render to the screen, and save to a file. Hence, they can be modified with either predefined functions or operators, or instantiated from one of the unit shapes that are built into the language, which are spheres, cubes, cones, cylinders, and tetrahedrons.

Initialization is done by assigning a new Shape variable either to the result of a function with Shape as a return type, or by assigning it to one of the built-in primitive types, specified by the name of that type in uppercase. If an assignment is performed on a variable that already had a definition, that Shape will be completely overwritten. If there are no references to the original Shape, it will be deleted and the underlying mesh freed (as happens when Shapes go out of scope). For instance, the following code sample initializes `ball` to be a unit sphere, and then creates a new Shape clone by invoking the Copy function on it.

```
Sphere ball = SPHERE;
Shape clone = Copy(ball);
```

Assignment of Shape types, when not done using a built-in initializer or the results of a function, is done by reference, not value. For instance, in the following example, both ballA and ballB refer to the same underlying data structure, and the scale affects both.

```
Sphere ballA = SPHERE;
Sphere ballB = ballA;
Scale(ballB, 1.0, 2.0, 1.0);
```

Shape is the parent class of Sphere, Cone, Cylinder, Cone, and Tetra. The subclasses automatically become a Shape once a topology-changing operation is applied. Rigid transformations, on the other hand, preserve the original geometric type. A subclass can become a Shape, but the reverse is not allowed. Shapes are the more general form.

The sample below illustrates the use of the Shape data type in ShapeShifter:

```
Shape makeHat(double y) {
      Shape brim = CYLINDER;
      Shape top = CYLINDER;
      Scale(top, 1.0, y, 1.0);
      return Union(top, brim);
}
```

3.2.2. Sphere

The Sphere data type is a subclass of Shape and represents a solid that is topologically equivalent to a geometric sphere. The initializer SPHERE creates a Sphere with radius 1 centered at the origin.

3.2.3. Cone

The Cone data type is a subclass of Shape and represents a solid that is topologically equivalent to a geometric cone. The initializer CONE creates a Cone with base radius 1 and height 1 centered at the origin.

3.2.4. Cylinder

The Cylinder data type is a subclass of Shape and represents a solid that is topologically equivalent to a geometric cylinder. The initializer CYLINDER creates a Cylinder with radius 1 and height 1 centered at the origin.

3.2.5. Cube

The Cube data type is a subclass of Shape and represents a solid that is topologically equivalent to a geometric cube. The initializer CUBE creates a Cube with height 1, width 1, and depth 1, centered at the origin.

3.2.6. Tetra

The Tetra data type is a subclass of Shape and represents a solid that is topologically equivalent to a geometric tetrahedron. The initializer TETRA creates a Tetrahedron with all edges equal to one centered at the origin.

3.2.7. Arrays

ShapeShifter also supports the Array data type, which can be a series of any of the other six previously mentioned data types.

An array is a data type that can be looked at as a series of other data types within the same container of a predefined size. It can be an array of any of the previously mentioned data types, including shapes.

Sample code for arrays in ShapeShifter is provided below:

```
int[] a1 = [1,2,3]; // array of type int[]
Shape[] a2 = [s1, s2, s3]; // where s1..3, are all spheres
```

3.2.8. Void

The void data type represents a non-existent value that can be returned by a function. In other words, if a function does not return a result, it returns void. Note that void and NULL are not the same thing.

The sample code below illustrates the void data type in action in ShapeShifter:

```
void addShapeProperties() {
      ...
}
```

4. Expressions

Expressions in ShapeShifter, similar to C, include at least one operand and zero or more operators, and can be grouped using parentheses.

Possible expressions include:

```
42
SPHERE UU CYLINDER
(4 + 2) * 42
```

4.1. Assignment

The assignment operator = stores values in variables. The left operand, or the lvalue, must be an identifier. Shapeshifter requires a type declaration prior to the identifier. The rvalue must be an accepted data type.

Sample variable assignment includes:

```
Shape box = CUBE; // box is the variable name
int i = 0;
Shape terr1ble_1cecream_mess = SPHERE UU CONE;
```

5. Statements

Statements are single atoms of code execution, delimited by semicolon characters.

5.1. Expression Statements

An expression statement consists of an expression (defined above) followed by a semicolon.

5.2. Declarations

ShapeShifter is a statically typed language. Variable declarations consist of a type, an identifier, and an rvalue such as:

```
int counter = 3;
Cylinder c = CYLINDER;
```

5.3. Control Flow

We allow for branches with if-else statements. These statements consist of an if block, followed by zero or more elif blocks, followed by an else block (note that the else block is required). The if and elif blocks are preceded by expressions in parentheses that, if true, cause the corresponding block to execute. Here is an example:

```
if (expr) {
      stmt;
} elif (expr) {
      stmt;
} else {
      stmt;
}
```

5.4. Loops

ShapeShifter allows for two loop constructs: while loops and for loops. Of the two options, while loops are more general (the body of the loop executes until the expression in parentheses evaluates to false):

```
while (expr) {
      stmt;
}
```

In addition to while loops, ShapeShifter supports for loops. The parentheses before the block contain three expressions. The first is used for initialization. The body of the loop is called until the second expression evaluates to false. Variables must be pre-defined. The third expression is called every time the body of the loop is executed:

```
for (expr; expr; expr) {
      Stmt;
}
```

6. Functions

6.1. Built-in Functions

ShapeShifter has a variety of built-in functions that can be used to manipulate shapes that have already been defined. These are built in order to simplify common functions that users may want to use while developing scenes and making geometric manipulations. Built-in functions include: `Scale()`, `Translate()`, `Rotate()`, `Reflect()`, `Union()`, `Intersect()`,

`Render()`, `Copy()`. Some functions, such as `Union()` and `Intersect()` have two representations in the language -- one as a functions, another as operators (represented as `UU` and `NU`, respectively). The purpose of this duplication is to allow users to have different approaches to organizing their shape-building process, as more interesting programs will rely on the creativity of the programmer to make complex, reusable shapes.

Each of these functions and their descriptions can be found below, along with sample code to illustrate their use in ShapeShifter.

The Scale Function: `Scale(Shape x, double x)`

The Scale function takes in four arguments, a shape to operate on, as well as three double values that the determine the scaling for the shape. It multiplies each of the mesh coordinates of the given shape by the scale factor in the corresponding X, Y, or Z directions. A snippet of code using the Scale function in ShapeShifter can be seen below.

```
Shape s1 = SPHERE;
Shape s2 = Scale(s1, 3.0, 3.0, 3.0); // uniform scaling
render(s2); //renders sphere with radius 3 on display
```

The Rotate Function: `Rotate(Shape s, double x, double y, double z)`

The Rotate function takes in two arguments, a shape to operate on, as well as a double value that the shape needs to be rotated by for each axis. Function takes the double value mod 360.0 and rotates the shape that many degrees counter-clockwise in the same axis. A snippet of code using the Rotate function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Shape c2 = Rotate(c1, 180.0, 0, 0);
Render(c2); /*renders the unit cone rotated 180 degrees CCW
around the X-axis. By default the unit cone is rendered with the
point pointing out of the screen*/
```

The Translate Function: `Translate(Shape x, double x, double y, double z)`

The Translate function takes in four arguments, a shape to operate on, as well as three double values that the shape needs to be translated by. By default, the second argument is translation in the x-axis, the third argument is translation in the y-axis, and the fourth argument is the translation in the z-axis. A snippet of code using the Rotate function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Shape c2 = Translate(c1, 9.0, 8.0, 7.0);
Render(c2); /*renders the unit cone translated (9.0, 8.0, 7.0)
units respectively*/
```

The Reflect Function: `Reflect(Shape x, double a, double b, double c)`

The Reflect function takes in four arguments, a shape to operate on, as well as three doubles that determine the plane of reflection: a*x + b*y + c*z = 0.
A snippet of code using the Rotate function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Shape c2 = Reflect(c1, 0, 1.0, 0.0);
Render(c2);  /*renders  the  unit  cone  reflected  over  the  xz
plane.*/
```

The Union Function: `Union(Shape x, Shape y)`

The Union function takes in two arguments, two shapes to operate on. The function takes the union of these shapes and returns the new shape. The input shapes are not modified in any way. By default, an operator mapping to this function is also available to the programmer in order to simply use. Alternatively, the Union function will also accept one argument, provided that the passed value is an array containing two or more shapes. The function takes the union of the first pair of shapes, then takes the union of the returned shape with the next shape in the array, and so forth. The underlying behavior when the function takes in an array essentially applies the logic of the two-argument Union function recursively. A snippet of code using the Union function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Shape s1 = SPHERE;
Shape sc1 = Union(s1, c1);
Render(sc1); /*renders the union of these two shapes. Think ice
cream cone with the ball of ice cream sticking through the cone*/
```

The Intersect Function: `Intersect(Shape x, Shape y)`

The Intersect function takes in two arguments, two shapes to operate on. The function takes the intersection of these shapes and returns the new shape. The input shapes are not modified in any way. By default, an operator mapping to this function is also available to the programmer in order to simply use. Alternatively, the Intersect function will also accept one argument, provided that the passed value is an array containing two or more shapes. The function takes the intersection of the first pair of shapes, then takes the intersection of the returned shape with the next shape in the array, and so forth. The underlying behavior when the function takes in an array essentially applies the logic of the two-argument Intersect function recursively. A snippet of code using the Intersect function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Shape s1 = SPHERE;
Shape sc1 = Intersect(s1, c1);
```

```
Render(sc1); /*renders the intersection of these two shapes*/
```

The Difference Function: `Difference(Shape x, Shape y)`

The Difference function takes in two arguments, two shapes to operate on. The function takes the difference of these shapes and returns the new shape. The second shape is 'subtracted' from the first to form the new shape. The inputs are not modified. By default, an operator mapping to this function is also available to the programmer in order to simply use. Alternatively, the Difference function will also accept one argument, provided that the passed value is an array containing two or more shapes. The function takes the difference of the first pair of shapes, then takes the difference of the returned shape with the next shape in the array, and so forth. The underlying behavior when the function takes in an array essentially applies the logic of the two-argument Difference function recursively. A snippet of code using the Difference function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Shape s1 = SPHERE;
Shape sc1 = Difference(s1, c1);
Render(sc1); /*renders the difference of these two shapes*/
```

The Copy Function: `Copy(Shape x)`

The Copy function takes in one argument, a shape to be copied. It makes an exact copy of the first shape, with the same underlying mesh, and returns the new shape. The input shape is not modified. A snippet of code using the Copy function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Shape s1 = NULL;
Shape s1 = Copy(c1);
Render(s1); /*renders the unit cone*/
```

The Render Function: `Render(Shape x)`

The Render function takes in one argument, the shape to render to the display. The function takes the given shape value and renders the shape to the display using the defining shape properties. A snippet of code using the Render function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Shape s1 = NULL;
Shape s1 = Copy(c1);
Render(s1); /*renders the unit cone to the display based on the
defining values it contains*/
```

The Save Function: `Save(Shape x, string filename)`

The Save function takes in two arguments, the shape to save to file and the file to save it to. The function saves the underlying properties of the shape to a standard .off file that the user can then import in order to retrieve the shape. By default, the file is saved to the same directory that the user is working in. A snippet of code using the Save function in ShapeShifter can be seen below.

```
Shape c1 = CONE;
Save(s1, "cone.off"); /*saves the unit cone and all underlying
properties to a txt file that can later be imported by the user*/
```

6.2. User Functions

6.2.1. Declarations

Functions are declared in the following format:

```
return_type function_name(arg_type arg_name, arg_type arg_name, ...)
```

Sample code for a function that creates a shape might look like:

```
Shape makeHat(double y)
```

Shapeshifter does not require users to declare functions in a separate statement from actual function definitions.

6.2.2. Definitions

Function definitions are included in `{ }` after the function declaration. Each statement within the body of a function definition is punctuated with `;` as is the case with all statements through a Shapeshifter program. The use of whitespace is encouraged to increase the legibility of a program and make scope more apparent. A sample function definition may look like:

```
Shape makeHat(double y) {
      Shape brim = CYLINDER;
      Shape top = CYLINDER;
      Scale(top, 1.0, y, 1.0);
      return Union(top, brim);
}
```

Function names can not be repeated; attempting to define a function with the same name will result in a compiler error.

A value of the type corresponding to the function's return type must be returned using the return keyword. If there is conditional branching, each branch must have a return statement.
If the function has a void return type, the return statement is not followed by anything, as such:

```
return;
```

6.2.3. Calls

Function calls behave like other statements within a Shapeshifter program. A sample function call may look like:

```
Shape hat = makeHat(6.9);
```

In this case, the returned Shape is automatically stored in a variable called `hat`.

7. Program Structure

There are some particularities within the language. Not unlike C's `int main()` function, the body of each Shapeshifter program must call `void scene()`. Scene refers to the abstract coordinate system environment that all following shapes exist in. Function definitions must occur prior to the `scene()`. Unlike C, Shapeshifter does not house function declarations separately -- users are meant to be encouraged to construct custom shapes by creating a function and then use the space of `scene()` to produce intermediary or a final visual representation. This workflow would be most relevant if the user seeks to create custom shapes that they mean to reuse throughout their program.

Each `Shape` is initialized as a specific subtype: a `Sphere`, `Cone`, `Cube`, `Cylinder`, or `Tetra` (tetrahedron). Successive transformations on these shapes which can then in turn result in more complex shapes whose properties are not limited to those of the basic subtypes.

Note that shapes can be initialized within the body of a function definition as well as the body of `scene()`. The scope of each object or variable is contained within the block (denoted by `{ }`) that it exists.

Statements, expressions, and variable definitions must be contained inside a user-defined function or Scene{}. There is no support for global variables.

8. Sample Program

```
/* This function makes a square pyramid of the given number of levels,
with the height of each level corresponding to stepHeight, a base side
length of baseSize, and the side length of the topmost level
corresponding to topSize. */
```

```
Shape makePyramid(int levels, double baseSize,
                  double stepHeight, double topSize)
{
      Shape base = CUBE; // Create a unit cube centered at origin
      Scale (base, baseSize, stepHeight, baseSize);
      // Move base to rest on the XZ plane
      Translate(base, 0.0, stepHeight / 2.0, 0.0);
      double sizeDiff = (baseSize - topSize) / levels;
      Shape prev = base; // Copy by reference
      double currSize = baseSize;
      int i = 0; // Variables must be pre-declared
      for (i = 0; i < levels; i = i + 1) {
          Shape next = Copy(prev); // Create an exact copy of prev
          Scale(next, (currSize - sizeDiff)/currSize, 1,
                      (currSizeSize - sizeDiff)/currSize);
          currSize = currSize - sizeDiff;
          if (currSize <= 0.0) {
              break; // Break out of closest loop
          }
          else {} // Do nothing
          Translate(next, 0.0, stepHeight, 0.0);
          // Build the pyramid by Unioning the levels together
          base = Union(base, next);
          prev = next; // Assignment by reference
      }

    return base;
}

/*  This program creates several pyramids of varying size and
orientations, applies rigid transformations, and merges them. The end
result is finally displayed and the underlying mesh representation
saved to a file.

Scene {
    Shape pyr5 = makePyramid(5, 5.0, 1.0, 1.0);
    Shape pyr3 = makePyramid(3, 4.0, 1.0, 2.0);
    Shape pyr5Inv = makePyramid(5, 1.0, 1.0, 5.0);
    Translate(pyr5i, 0.0, 5.0, 0.0);
    // Stack the inverted pyramid on its normal counterpart
    Shape pyr5Stack = Union(pyr5, pyr5Inv);
    Shape pyr3i = Reflect(pyr3, 0.0, 1.0, 0.0); // Reflect across y=0
```

```
    Translate(pyr3i, 0.0, 3.0, 0.0);
    Shape pyr3Stack = Union(pyr3, pyr3i);
    Rotate(pyr3Stack, 0.0, 0.0, 90.0); // Rotate around Z
    Shape merged = Union(pyr3Stack, pyr5Stack);
    Render(merged); // Display the result
    Save(merged, "pyramids.off"); // Save mesh representation to file
}
```