PROGRAMMING LANGUAGES AND TRANSLATORS
Language Proposal

ShapeShifter

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

Domain: Programmatic Geometry Modelling
Ultimate goal: Be able to cleanly and programmatically define complicated shapes by performing a series of operations on simple primitive shapes.

TEAM

Manager:	Ishan Guru	ig2333
Language Gurus:	Eszter Offertaler, Rashida Kamal	eo2309, rsk2161
System Architect:	Rajiv Jude Thamburaj	rjt2132
Tester:	Stephanie Burgos	sb3539

LANGUAGE DESCRIPTION

ShapeShifter is a programming language designed to make geometric shape manipulations as sexy as it sounds. Instead of having to muck about with meshes and vertex connectivity information, people can use ShapeShifter to create and display three dimensional shapes like spheres and cubes simply by providing their names. ShapeShifter allows these primitives to be modified and combined through a series of transformations with all of the actual implementation and underlying mesh representation abstracted away. Functions and control flow is included to allow programmatic and recursive modelling.

Instead of worrying about inputting the correct functions or areas, displaying three dimensional shapes, such as spheres, cubes, cones, or any manipulation is as simple as providing the name of a shape and its origin coordinates. ShapeShifter then allows for these primitives to be modified and combined to through a series of transformations, with options to define loops and functions for programmatic modelling.

It is important to note that ShapeShifter does not support an interactive GUI for shape modifications, such as CAD, but rather relies on user programming the visual manipulations they intend to make. It does, however, support an interactive display that allows users to view their generated shapes. The display supports basic viewing actions like zoom, rotate, etc. but does not make any changes to the actual shape. While this does rather limit the usefulness of ShapeShifter for users who would like to make heavier use of the GUI, the intention is not to replace GUI-based CAD systems, but to provide a programmatic alternative. It is useful in cases where a scene may consist of many well-defined but tedious operations, e.g. stacking thousands of spheres.

The display for ShapeShifter is based on a minimalistic OpenGL output that places the elements of the scene onto a “grid” environment and allows for simple camera movement. The view can be zoomed, translated, and rotated. There is also an option for basic animation: the displayed scene can rotate on its own to allow for viewing from all angles without direct user manipulation.

The user is insulated from the underlying mesh-based representation of each geometric primitive. ShapeShifter will be wrapped around an appropriate Constructive Solid Geometry library like CGAL[1] or OpenCSG[2] that will handle the brunt of the mesh operations, like intersection or union.

PARTS OF THE LANGUAGE

Each program defines a “scene” that contain the “shapes” that are comprised of “primitives” that undergo a series of “operations”.

Basic Data types

- int: an integer data type
- double: a double-precision floating point data type
- vec3d: a vector of 3 doubles, with named, accessible fields x,y,z.
- vec3i: a vector of 3 integers, with named, accessible fields x,y,z
- String: C++ style string
- C-style arrays
- Point: this data structure represents an actual vertex in the underlying mesh representation of the shape, and contains position information. Used to capture a notion of geometric vertices.

Scene

Each program defines a scene. The syntax is:

```
Scene {
        ...
}
```

It acts similarly to a main() function in other languages. The expressions in the curly braces of Scene are sequentially performed when the program is run. The Scene itself can be displayed, but mostly it acts as an abstract environment - with a world coordinate system - for shapes to exist in.

[1] http://www.cgal.org/
[2] http://opencsg.org/

Shapes and Primitives

Shapes are the predominant data type. They include primitives, which comprise the most basic SHAPE types and are the only things that can be initialized directly, and the results of operations on shapes.

Users can query primitive-specific information on shapes that have not been modified by topology-altering operations. Generic shapes only provide position and orientation information. The position is measured at the center of the bounding box of an arbitrary shape. The orientation is initialized somewhat arbitrarily, but tracked over the course of rotation operations.

Common properties of arbitrary shapes

Property name	*type*	*description*
position	vec3d	(x,y,z) coordinates of the center
orientation	vec3d	(x,y,z) offset in degrees from world's coordinate system
SPHERE:		
radius	double	radius, only accessible under uniform scale
CUBE:		
width	double	width of the de facto rectangular prism
height	double	height
length	double	length
vertices	Point[8]	the vertices of the cube
CYLINDER:		
radius	double	radius of the face
height	double	height
CONE:		
radius	double	radius of the base
height	double	height
vertex	Point	the point of the cone
TETRAHEDRON:		
length	double	length of the sides
vertices	Point[4]	the vertices of the cube

These can be accessed by shapeName.propertyName, e.g.

SHAPE ball = SPHERE;

ball.radius -> returns a double equal to 1.0

These properties can not be changed by the user, only queried.

Operations

Operations broadly do one of two things: they perform transformations on a SHAPE, or cause topological changes to the underlying mesh, resulting in the creation of a new SHAPE object (the inputs are not modified).

Initialization:

SHAPEs are initialized by writing the name of a primitive shape. The following code creates a new SHAPE of type sphere unit size at the scene origin:

SHAPE shapeName = SPHERE; // Or CUBE or one of the other primitives

The following operations modify the state of the SHAPE parameter.

- *Rotate(SHAPE s, double x, double y, double z)*
  Rotates the SHAPE about its center (not the world center), inputs are in degrees. x corresponds to the amount of CCW rotation around the x axis, y around the y axis, etc.
- *Translate(SHAPE s, double x, double y, double z)*
  Translates the SHAPE by x along the world's x axis, by y along the y axis. etc.
- *Scale(SHAPE s, double x, double y, double z)*
  Scales the SHAPE by x along its own x axis, by y along its own y axis, etc.
- *Reflect(SHAPE s, double a, double b, double c)*
  This reflects the SHAPE across the plane defined by ax + by + cz = 0, in world coordinates.

The following operations make topological changes and result in a new SHAPE.

- *Union(SHAPE s1, SHAPE s2)*
  Returns a new SHAPE that is the geometric union of the two shapes
- *Intersect(SHAPE s1, SHAPE s2)*
  Returns a new SHAPE that is the intersection of the two shapes
- *Difference(SHAPE s1, SHAPE s2)*
  Returns a new SHAPE that is the result of subtracting s2 from s1
- *Copy(SHAPE s1)*
  This returns an exact copy of s1, makes no change to the input SHAPE

And there are a few general operations

- *Write(SHAPE s)*
  Dumps s to a general format file (probably .PLY)
- *Print(String msg)*
  Dumps msg to a log file.

Flow control

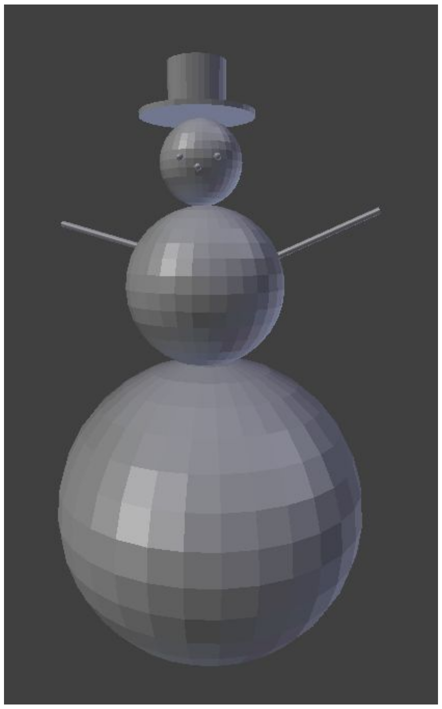
while loops, if/elif/else statements are supported in the language. All expressions must be surrounded by {}. !=, ==, <=, >=, <, > are supported in the if conditional.

Functions

The *function* keyword can be used to define a function, and *rec* is used to indicate that it is meant to be used recursively. Parameters are not passed by reference, so use them wisely. They can be used to avoid writing a lot of repetitive code, or to segment the creation of a complicated shape into less complicated subcomponents. E.g. when creating a model of a chessboard, you can have a function createPawn() return a pawn SHAPE and call it multiply times. Recursive functions are allowed, and are encouraged for the creation of fractals. It may be necessary to add more information queryable for primitives to fully take advantage of fractal behavior.

SAMPLE CODE

Sample program 1: Create a snowman from simple shapes and display it.
Resulting image (not an exact representation of the output)



```
/* This function returns a SHAPE that comprises the snowman's body */
SHAPE function makeBody()
{
        SHAPE base = SPHERE; // Initialize a unit sphere rotate at the origin
        Scale(base, 4, 4, 4); // Scale base by 4 in all coordinates
        Translate(base, 0, 2, 0);  // Translate base so that it sits on top of the xz plane
        SHAPE middle = SPHERE;
        Scale(middle, 2, 2, 2);
        Translate(middle, 0, 5, 0); // Translate middle so that it sits on top of the base
        SHAPE head = SPHERE;
        Translate(head, 0, 6.5, 0); // Place head on top of the middle segment
        SHAPE body = Union(base, middle);
        body = Union(body, head);
        return body;
}

SHAPE function makeHat(double y)
{
        SHAPE brim = CYLINDER;
        Scale(brim, 1.5. .01, 1.5);
        Translate(brim, 0.0, y, 0.0);
        SHAPE top = CYLINDER;
        Scale(top, .8, 1, .8);
        Translate(top, 0.0, y+.01, 0.0);
        return Union(top, brim);
}

Scene {
        SHAPE body = makeBody();

        // Add the arms
        SHAPE larm = CYLINDER; // Unit cylinder oriented along y axis
        Scale(larm, .05, 2, .01);
        Rotate(larm, 0, 0, -60); // Rotate 60 degrees CW around z axis
        Translate(larm, 1, 5, 0);
        SHAPE rarm = Copy(larm);
        Reflect(rarm, 1, 0, 0); // Reflect across the yz plane
        body = Union(body, larm);
        body = Union(body, rarm);

        // Add the face
        SHAPE reye = SPHERE;
        Scale(reye, .05, .05, .05);
        SHAPE nose = Copy(reye);
        Translate(reye, .3, 6.5, .45);
        SHAPE leye = Copy(reye);
        Reflect(leye, 1, 0, 0);
        Translate(nose, 0, 6.3, .48);
        body = Union(body, nose);
        body = Union(body, reye);
        body = Union(body, leye);

        SHAPE hat = makeHat(6.9);
        body = Union(body, hat);

        Display(hat);

}
```