Mapwad

A 3D modeling language

Language Reference Manual

Ben Smith – bhs16@columbia.edu
AvrumTilman – amt77@columbia.edu
Josh Weinberg – jmw211@columbia.edu
Ron Weiss – ronw@cs.columbia.edu

1. Language Semantics

A Mapwad program is a standard ASCII text file that can be generated on any computing platform that has an ASCII editor. Mapwad is a case-sensitive language that matches seven types of tokens: white space and comments, identifiers, number, strings, operators, keywords, and other tokens.

1.1. White space and Comments

1.1.1. White space

Spaces, tabs, newlines, and carriage returns are all considered whitespace. All white space will be ignored

1.1.2. Comments

C and C++ style comments are supported. That is, "//" start a single-line comment, while all text, multi-line or single-line, between "/*" and "*/" will be considered a comment. All text that is considered a comment will be ignored.

1.2. Identifiers (Variable, Function, and Object names)

Identifiers are used for naming variables, functions, and objects. They can contain a mixture of alphanumeric characters as well as the underscore character ('_'). It should be noted, that identifiers must begin with either a letter or an underscore. This follows the C specification.

1.3. Numbers

Mapwad supports integer and floating point numbers. Integers consist of one or more digits. Floating point numbers can be defined in a couple different ways, with or without a decimal point, exponent, or the combination of the two. An exponent is defined as either the 'e' or 'E' character, followed by an optional sign character ('-' or '+'), followed by an integer. For example:

```
5         // integer or float (depending on type specified at declaration)
0.45      // float
5.2       // float
5e10      // float
5.3E-10 // float
```

1.4. Strings

Strings are character sequences enclosed within a set of double quotes. To place a double quote inside the string type two double quotes. You cannot place a newline character within the string.

For example, the following assignment,

```
string text = "He said, ""hi there""";
```

would set `text` equal to: He said, "hi there"

1.5. null

Mapwad supports a null value like Java.

1.6. Operators

=	**%**	**--**	**%=**	**<=**
+	||	+=	==	**.**
-	**&&**	**-=**	>	
*	**!**	*=	<	
/	++	/=	>=	

1.7. Keywords

The following identifiers are reserved as keywords:

Map	**Location**	**string**	**while**	**false**
Room	**int**	**if**	**break**	**return**
Wall	**float**	**else**	**continue**	**null**
Thing	**boolean**	**for**	**true**	

1.8. Other Tokens

The following tokens are used for building statements:

,	**(**	**{**	**[**
;	**)**	**}**	**]**

2. Program Layout

A Mapwad program is a combination of variable, and function definitions as well a special “Map” program entry function at the end of the program. Aside from the “Map” function, which must be located at the end of the program, all variables, and functions definitions can be mixed together in any order. It should be noted, that any variable or function must be declared before use.

3. Variables

Mapwad supports both explicit/user-defined and implicit variables. User defined variables must be declared before use and have a specific structure for declaration and assignment. The structure forces types on variables at declaration time making Mapwad a strongly-typed language. Implicit variables are Mapwad built-in variables that are associated with specific types.

3.1. Explicit (User-Defined) Variables

3.1.1. Definition

A user-defined variable is declared in the following manner:

```
<type> <identifier>
```

3.1.2. Assignment

As stated above, a variable cannot be assigned a value or accessed until it has been declared. However, Mapwad does allow for variable initialization, that is, a variable can be assigned a value at the time that it is declared.

```
// incorrect – invalid declaration
a;
a = 5;

// correct
int b;
b = 5;

// correct
int c = 5;
```

3.1.3. Scope

Variables in Mapwad can have either global or local scope depending on where they are declared. If a variable is declared in the program body (i.e outside function definitions), it is automatically assigned a global scope and can therefore be accessed by any construct that follows in the program (i.e. function definitions, conditionals, loops, etc). Variables that are declared within functions and constructs are local to that function or construct.

It should be noted that Mapwad uses static scoping.

```
int a = 5; // exists within the following if statement
if(a > 3)
{
      int b = 6; // exists only within this if statement
}
```

3.2. Implicit Variables

Mapwad includes a list of implicit variables that are associated with some of the built-in types (see Types/Objects section for a list of implicit variables belonging to the different types and objects). Furthermore, an implicit variable may or may not be read-only (see specific type or object for information on its associated implicit variables).

4. Types/Objects

Mapwad has many built in types and objects. The basic types include booleans, integer numbers, floating point numbers, and string literals. The advanced types and objects include Walls, Rooms, Things, Locations, and arrays. It should be noted that Mapwad does not support type casting, but does have limited support for type promotion. If an integer is added to a float, the integer is promoted to a float, and the result is a float.

4.1. Basic Types

4.1.1. `boolean`

Boolean values that can take the values ‘true’ or ‘false’.

4.1.2. `int`

A 32-bit signed integer.

4.1.3. `float`

A 64-bit floating point number.

4.1.4. `string`

Basic string of characters enclosed in double quotes. As explained in the first section, if double quotes are needed in the string itself, use a repeated double quote as an escape sequence (see first section for an example). Mapwad's support of strings is very basic and extends only to string creation, and string equality comparison. Other string manipulation functions (i.e. regular expressions) are not supported.

4.2. Advanced Types/Objects

Mapwad has several built-in objects and advanced types. Mapwad's objects are struct-like constructs. Because of their complex nature, each of these objects requires a form of instantiation, and therefore has an associated constructor. Furthermore, as is done in Java, all advanced types and objects are passed by reference.

4.2.1. `Wall`

Walls are a struct-like construct that are used to make up Rooms. Walls are made up of implicit variables that can be set and changed depending on the desired situation. Each of the Wall's implicit variables has a default value.

Walls can technically be created outside the context of a Room, but their main purpose is to connect together to form a Room. Since each Room has a list of Walls (see Room definition), the connection between a Wall and its neighbors is built into the Wall object. That is, each Wall stores the angle between itself and the next Wall in the Room's Walls list.

Furthermore, in order to create a multi-Room environment, there must be a means of interconnecting Rooms. Mapwad provides this ability with built-in functions to merge Walls from different Rooms, and also to create doorways between Rooms (see description of the Attach() and AttachWithDoor() built-in functions later in the manual).

4.2.1.1. Implicit Variables

Unless otherwise specified, each of the Wall's implicit variables is read/write.

`string Name` – Name associated with the wall. Default value is ""
`float Length` – The length of the wall. Default value is 20
`float AngleNext` – The angle between this Wall and the next Wall in a Room's Walls list. Default value is 90
`Wall Next` – The next Wall in a Room's Walls list (read-only). Default value is itself

`Wall Prev` – The previous Wall in a Room's Walls list (read-only). Default value is itself

`string Texture` – The type of texture to be mapped onto the wall (inherited from the quake .WAD file – to be defined later). Default value is ""

`boolean IsEntry` – Whether or not this Wall is connected to another Wall (read-only). This variable can only be set by a built-in Attach-type function. Default value is false

`Wall ConnectedTo` – The Wall that this Wall is connected to (read-only). This variable can only be set by a built-in Attach-type function. Default value is null.

`Room FromRoom` – The Room that this Wall is a part of (read-only). Default value is null.

4.2.1.2. Wall Constructor

Wall(float Length=10, float AngleNext=90, string Name="", string Texture="")

To create a Wall, use the following syntax:

```
Wall <var name> = Wall(..);
```

4.2.2. `Room`

The Room object is a struct-like construct that is arguably the most important object in Mapwad. Simply stated, a Room is a circular connection of Walls that is singular (that is, Walls cannot cross). These Walls are stored in a special implicit Walls list that is associated with each Room object. The other defining attribute of a Room is its height, whose value is stored in a Room's implicit Height variable.

Central to the Room object is the list of Walls that make up the perimeter. The Walls list is a super data structure that has the dynamic growth functionality of a linked list and the ease of access provided by arrays and associative arrays. When a Room is first created, its Walls list is null.

Access to a Room's Wall objects is accomplished by either indexing into the Walls array (i.e. Walls[0], Walls[1], and Walls[2]), or grabbing the first Wall object's handle and iterating using a Wall object's list properties (i.e. Walls[0].Next refers to Walls[1], and so on, with the last Wall's Next variable pointing to Walls[0]). If a Wall in the Walls list has a name, it can be accessed associatively by that name (i.e. Walls["three"]). It should be noted, that if multiple Walls have the same name and associative access is desired, the first Wall in the list relative to Walls[0] that matches will be returned.

Adding and removing Wall objects from the Walls list is done using the built-in functions Add(), AddAfter(), and Remove() (see description of these built-in functions later in the manual).

The flexibility of the Walls list allows the programmer to iterate through a Room's Wall objects in a few different ways (assume x is the name of a Room object):

a) Iterate using normal array functionality

```
Wall currentWall;
int i;
for(i = 0; i < x.Walls.Size; i++)
{
      currentWall = x.Walls[i];
}
```

b) Iterate using list functionality

```
Wall iterator = x.Walls[0];
Wall currentWall = iterator;
iterator = iterator.Next;
for(; iterator != x.Walls[0]; iterator = iterator.Next)
{
      currentWall = iterator;
}
```

4.2.2.1. Implicit Variables

Each of the Room's implicit variables is read/write.

`float Height` – Height of the Room. Default value is 10.
`Wall[] Walls` – List of Walls associated with the Room object (explained above). Walls is empty (null) by default.

4.2.2.2. Room Constructors

Room(float Height=10)

To create a Room, use the following syntax:

```
Room <var name> = Room(..)
```

4.2.3. Thing

Things are simple entities supported by Quake that can be put within Rooms. Things include monsters, lava pits, guns, power-ups, etc. A Thing is placed into a Room using a Location (see below).

4.2.3.1. Implicit Variables

Each of the Thing's implicit variables is read/write.

`string Type` – Type of Thing. Default value is ""
`Location Position` – Thing's start location. Default value is null.

4.2.3.2. Thing Constructor

Thing(string Type="", Location Position=null)

To create a Thing, use the following syntax:

```
Thing <var name> = Thing(..)
```

4.2.4. Location

A Location is a three-tuple consisting of a Wall object, a string representation of a position relative to the Wall, and a distance perpendicularly outward into the Room from the Wall.

4.2.4.1. Implicit Variables

Each of the Location's implicit variables is read/write.

`Wall NearWall` – The Wall that this Location will be relative to. Default value is null.

`string WallPosition` – Where along the NearWall to place this location. This can take the values, "left", "right", or "center". Default value is "center."

`float WallDistance` – How far out into the Room from NearWall this Location will exist. Default value is 2.

4.2.4.2. Location Constructor

Location(Wall NearWall=null, string WallPosition="center", float WallDistance=2.0)

To create a Location, use the following syntax:

```
Location <var name> = Location(..)
```

4.3. Arrays

Mapwad arrays are very similar to C and Java arrays. Before an array can be used, it must be declared with the number of elements. For simplicity, Mapwad also allows the use of array constants, where an array is initialized with all the element values. Elements in an array are indexed by number starting with 0. The array type extends to every type and object that Mapwad supports. Furthermore, multidimensional arrays are allowed.

4.3.1. Usage

Syntax for array declaration follows C except that the brackets are glued to the type and not the variable name. For example,

```
int[] a; // correct
int b[]; // incorrect
```

There are two ways to assign values to an array. Firstly, before adding elements to an array the size must be stated explicitly. Secondly, Mapwad supports array constants where all values are enclosed in curly braces, and

separated by commas. Access to elements in the array is performed by using the variable name immediately followed by a left bracket, then the index, then a right bracket. For example, to create an array called numbers that will hold 4 integer values 1,2,3,4, the following two implementations would be correct.

```
int[] numbers = int[4];
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;
numbers[3] = 4;
```

or,

```
int[] numbers = {1,2,3,4};
```

For multidimensional arrays the following implementation would be correct.

```
int[][] numbers = int[2][4];

numbers[0][0] = 1;
numbers[0][1] = 2;
numbers[0][2] = 3;
numbers[0][3] = 4;
numbers[1][0] = 1;
numbers[1][1] = 2;
numbers[1][2] = 3;
numbers[1][3] = 4;
```

Multidimensional array constants are not supported.

4.3.2. Implicit Variables

Mapwad arrays have an implicit variable, Size, that stores the maximum number of elements that the array can contain. The Size variable is read-only.

`int Size` – Maximum number of elements that the array can store. For multidimensional arrays each dimension has a size. So given the above example numbers.Size=2 and numbers[0].Size=4.

5. Statements

Statements make up the heart of a block of code. Every code block is made up of multiple statements that define how the block operates. If a code block is thought of as a recipe, then the variables would be the ingredients and the statements would be the instructions for how to cook the recipe. Within Mapwad there are four basic types of statements. An Operator statement (also called an expression), an assignment statement, a control flow statement, and a function call statement.

5.1. Operator Statements

5.1.1. Arithmetic Operator

Arithmetic operators allow basic arithmetic to be performed on numerical values. Mapwad allows arithmetic operators to be performed on integer variables, float variables, or constant numerical values.

Mapwad supports multiple types of arithmetic operators. The first basic types are the +, -, *, /, and % operators. These operators take two variables or constants and return the value of the operation. They must be combined with another statement to be useful within the program.

The next type of operator is the assignment arithmetic operators. These are +=, -=, *=, /=, and %=. These operators act similar to the operators above except that the left hand operand must be a variable that can store the return value of the operation. So x=x+5 would be written as x+=5.

Finally there are two more arithmetic operators that are defined for programmer convenience. They are the increment (++) and decrement (--) operators. These operators act similarly to the assignment operator above except they only have a left hand operator. The right hand operator is assumed to be 1. So x+=1 would be written as x++.

Mapwad supports limited upcasting for arithmetic operations. If an arithmetic operation is given a mixture of floats and ints all the ints will be promoted to floats and a float will be returned. However, if the return type expects an integer then an error value will be thrown.

5.1.2. Boolean Operators

Mapwad supports two different types of Boolean operators. The first type compares two values and returns true or false. These operators are ==, !=, <, >, <=, >=. The == and != are valid for all types, while the other comparison operators are only valid for ints and floats. You can mix ints and floats and Mapwad will upcast an int into a float in order to perform the operation. The are also valid for strings and booleans. However, both operands must be a string and the other operators(<, >, <=, >=) are not supported for strings.

Beyond those operators Mapwad also allows Boolean algebra. Mapwad supports &&, ||, and !. && compares the Boolean value of the left hand operand to the Boolean value of the right hand operand. If both of them are true it returns true otherwise it returns false. || acts similarly to && except that it returns true if either the left hand or the right hand operand is true. Finally, ! returns the opposite of its right hand operand. ! does not take a left hand operand.

5.1.3. Dot Operator

The dot operator '.' accesses a member variable of the object. For example if x is a room object then x.Height will access the height variable associated with x.

5.1.4. Precedence

The following chart defines the precedence order for Mapwad. Note the first row has the highest precedence:

f(r,r,..)	(expression)	ID	boolean constant	string literal	int constant	float constant
a[i]						
s.m						
l++	l--					
+a	-a	!a				
a*b	a/b	A%b				
a+b	a-b					
a>b	a<b	a>=b	a<=b			
a==b	a!=b					
x&&y						
x||y						
x=a	x+=b	x-=c	x*=d	x/=e	x%=f	

5.2. Assignment

An assignment statement takes a value and assigns it to a specified variable. Within Mapwad only variables can have a value assigned to them. The format of an assignment statement is <variable name> = <value>. The variable name can be any variable that was previously defined and matches the type of the value. Value can be any constant, variable, or statement with a return value. Assuming that X and Y are integers and addfunc() is a function that returns an integer then the following would all be valid assignment statements.

```
X=5;
X=5+1;
X=y-2;
X=addfunc(2,3);
```

5.3. Control Flow

A control flow statement describes what statements will be executed next. There are two basic types of control flow constructs, conditional statements and loops. Within all of the control flow statements a section of code is designated with { }. The code within that section has its own scope and any variable that is declared within that section will not exist outside of the section.

5.3.1. Conditional Statements

Conditional statements allow for statements to be executed only in specific cases. The format for a conditional statement is as follows:

```
if (<Boolean expression>)
{
      <statement>*
```

```
}

 or

if (<Boolean expression>)
{
      <statement>*
}
else if (<Boolean expression>)
{
      <statement>*
}
else
{
      <statement>*
}
```

The statement within the {} for if will be executed if the Boolean expression evaluates to true. If the statement evaluates to false then the next else if statement will be checked and the program will continue to check the else if statements until a Boolean expression evaluates to true. If the program reaches an else without an if those statements are evaluated. Once one Boolean expression returns true and the corresponding block is executed the rest of the conditions are skipped and the program continues after the else or the final else if.

5.3.2. Loop Statement

A loop statement allows the same code to be executed multiple times before continuing with the rest of the program flow. There are two types of loops supported in Mapwad, for loops and while loops.

5.3.2.1. For Loop

A for loop in Mapwad follows the same basic syntax as a C/C++/Java for loop.

```
for (<variable initialization>;<Boolean
expression>;<iteration action>)
{
   <statement>*
}
```

The variable initialization can be any valid expression; however it is intended to initialize some loop variable for the loop. The one exception to this is that a variable can not be declared in a for loop statement.

The Boolean expression is evaluated each time the loop is iterated. If the value returns true the loop statements are executed one more time.

The iteration action can be any valid expression; however it is intended to increment the loop variable.

5.3.2.2. While Loop

The while loop is a more general purpose loop than the for loop. The syntax for the while loop is as follows:

```
while (<Boolean expression>)
{
   <statement>*
}
```

Any valid Boolean expression can be used and any number of statements can then be put within the body of the loop. The Boolean expression is evaluated before executing the code block on each iteration. If the Boolean expression becomes false the loop terminates.

6. Functions

A function is basically an encapsulation of a section of code. This function can then be called at a later time and the code within the function will then be executed. A function in Mapwad performs similarly to how a function in C/C++/Java performs.

6.1. Structure

A function is made up of four parts, a return type; an identifier; a parameter list; and the function body. A basic function declaration follows the following format

```
<return type>? <identifier>(<parameter list>)
{
     <body>
}
```

The return type can be any valid type such as, a basic type, an advanced type, or an object. However, if the function does not return a value then a return type is not given at all.

The identifier is any valid Mapwad identifier as defined earlier in this manual.

The parameter list is a list of 0 or more variable declarations (type and name) separated by a comma. Any variable can have a default value specified. If a default value is specified then the variable is optional when the function is invoked. A parameter list would look like the following:

```
(int x, int y=1, int z=2, string a="default", string b)
```

In the above example, variables y, z, and a are all optional variables that do not have to be specified when the function is called.

The body of the function is made up of 0 or more statements.

6.2. Invocation

A function is invoked in a statement by its name followed by the necessary parameters enclosed in parentheses. The following example would call a function called myFunc() with the parameter list defined in the previous section.

```
myFunc(3, ,5 , ,"the end")
```

Recall, that variables x and b were not optional and therefore required that values be included in the parameter list. Variables y, z, and a were optional and could be left blank. However, z was included in the parameter list so that the new value of z will be used within the function instead of the default value of 2. Notice that any value that was not included in the function invocation still included the comma separator to indicate that the default should be used. However, if the parameter list includes optional parameters at the end of the list then the function can be called without including the extra commas. So, given a function myFunc2(int x=1, string y="name", int z=3) all of the following would be valid function calls:

```
myFunc2(5,"hi",9) //x=5, y="hi", z=9
myFunc2(6)        //x=6, y="name", z=3
myFunc2()         //x=1, y="name", z=3
```

6.3. Returning

If a function has a return type it must have a return statement somewhere in the body of the function. If the program flow reaches a return statement, the value given in the return statement is returned to whatever called the function, and the function terminates.

The function invocation can be a statement by itself or it can be part of a larger statement. If the function invocation is a statement by itself then any return value that the function might give is ignored. However if the function invocation is used as part of a larger statement then it must return a value. Once that value is returned, the rest of the statement executes as if that value had been supplied instead of the function call.

6.4. Map Function

Mapwad has one special function, the Map function. This is the equivalent to the main function in C. The Map function is run first and is used to describe the layout of the entire map. The Map function must be the last declaration in the file since anything declared after Map can not be used by Map and therefore will never be used.

In addition, there is one implicit variable, MapStart, that must be assigned in the Map function. MapStart defines where the player starts when the final map is run in the game engine. MapStart is of type Location.

6.5. Built-in Mapwad Functions

Mapwad includes some predefined functions. These are Add, AddAfter, Remove, Attach, and AttachWithDoor. They are defined as follows:

```
boolean Add(Room r, Wall newWall)
```

This function adds newWall to the end of the Walls linked list associated with Room r. Add() will return true if the operation succeeds. If you attempt to add a Wall that already exists in a Room other than r, then Add() will terminate and return false. When called on a Room with an empty Walls list (i.e. a Room that was just initialized), Add() will initialize the list, setting r.Walls[0] to newWall.

```
boolean AddAfter(Wall locat, Wall newWall)
```

This function adds newWall to the linked list of Walls that contains locat. newWall is added after locat in the linked list. AddAfter() will return true if the operation succeeds. If you attempt to add a Wall that already exists in a Room other than r, then AddAfter() will terminate and return false.

```
Remove(Wall rem)
```

Remove will remove the Wall rem from the Walls linked list that contains rem.

```
boolean Attach(Wall a,Wall b, string wallconnect)
```

This function is used to connect two rooms at the point specified by Wall a and b within each room.

Walls a and b are walls in separate rooms that are going to be connected with an entry. Attach() says to connect a to b and remove their intersection. wallconnect indicates where the two walls will be connected relative to each other, and can take the values "left", "right", or "center". That is, "left" connects the walls at their leftmost point, "right" at their rightmost point, and "center" at their respective center points. Note, Wall b's "left" point is defined as follows: When standing in the Room containing Wall a, facing Wall a, the "left" side of Wall b will be the side that is perceived to be on the left. Attach() will return true if it succeeds and false if it fails.

```
boolean AttachWithDoor(Wall a, Wall b, string wallconnect, string
doorpos, float height, float width)
```

This function works the same as Attach() except that instead of the entry being the entire size of the intersection, the entry is now the specified width and height. doorpos specifies where along the intersection the enrty will be placed, and can take the values “right,” “left,” or “center.” AttachWithDoor() will return true if it succeeds and false if it fails.

Mapwad will have a built-in limited library of math functions (i.e. Sqrt(), Sin(), Cos(), Tan(), RandInt(), etc.).

7. Appendix
   7.1. Example 1

```
/*
 *  FourWall()
 *  Creates a 4 wall regular room
```

```
 */
Room FourWall()
{
      Room newRoom = Room();

      int i;

      for(i=0;i < 4;i++)
      {
            // Adds a Wall of length 10 with an angle
            // of 90 to the Walls list.
            Add(newRoom,Wall(10,90));
      }

      return newRoom;
}

/*************************************************************
 * Generates a simple map consisting of 20 corridors
 * connected by a long corridor
 *
 *  something like this:
 *
 *  ====================||
 *  ====================||
 *  ====================||
 *  ====================||
 *
 *  This is done by first generating a 20x20 grid of
 *  simple 4 wall rooms.  We then connect all the rooms
 *  in each row, and connect all the rooms in the last column
 *************************************************************/
Map()
{
      //a 20x20 2D array
      Room[][] rooms = Room[20][20];

      int row;
      int col;

      //Instantiate all the room objects
      for(row=0;row < 20;row++)
      {
            for(col=0;col < 20;col++)
            {
                  rooms[row][col] = FourWall();
            }
      }

      for(row=0;row < 19;row++)
      {
            for(col=0;col < 19;col++)
            {
                  //For even columns we attach wall 0 to wall 2,
                  //for odd columns wall 2 to wall 0
                  if(col % 2 == 0)
                  {
                              Attach(rooms[row][col].Walls[0],
                                    rooms[row][col+1].Walls[2],
                                    "center");
                  }
                  else
                  {
                        Attach(rooms[row][col].Walls[2],
```

```
                              rooms[row][col+1].Walls[0],
                              "center");
                  }
            }

            //Build a corridor down the 20th corridor
            Attach(rooms[rows][19].Walls[1],
                  rooms[rows+1][19].Walls[1]);
      }

      //Set the location where the player enters the map
      MapStart = Location(rooms[0][0].Walls[0],"center",1);
}
```

7.2. Example 2

```
/*
 * AnyRoom(int walls, float length=0, float[] lengths={0})
 *
 * Returns a room with 'walls' number of walls.  There are two
 * ways to specify wall length.  If 'length' is used, then all
 * walls have their Length variable set to 'length'.  When
 * 'lengths' is used, then the length of each wall is passed in
 * to the function.  'length' and 'lengths' cannot both be used
 * in a single call (obviously).
 * i.e. AnyRoom(5,10) builds a pentagon shaped room with each
 * wall 10 units long
 *
 */
Room AnyRoom(int walls, float length=0,float[] lengths=null)
{
      Room newRoom = Room();
      int i;
      float angle = 360/walls;

      if(lengths == null)
      {
            lengths=float[walls];

            for(i=0;i < walls;i++)
            {
                  lengths[i]=length;
            }
      }

      for(i=0;i < walls;i++)
      {
            Wall wall= Wall();
            wall.Length=lengths[i];
            wall.AngleToNext=angle;

            Add(newRoom,wall);
      }

      return newRoom;
}

 /*
  * attachSpoke(Wall wall, Thing thing)
```

```
 *
 * Attaches a "spoke" to 'wall'.  A spoke is a corridor with a
 * pentagon shaped room attached to its end.
 *
 */
attachSpoke(Wall wall,Thing thing)
{
     float[] lengths = {10,20,10,20};
     Room corridor = AnyRoom(4,,lengths);
     Room vestibule = AnyRoom(5,10);

     Thing.Position = Location(vestibule.Walls[2],"center",2);

     Attach(corridor.Walls[2],vestibule.Walls[0],"center");
     Attach(wall,corridor.Walls[0],"center");
}

/*
 * This mapwad generates a map consisting of a large octagon
 * shaped room with four corridors leading out from it.  Each
 * corridor is terminated with a pentagon shaped room.
 *
 */
Map()
{
     Room central = AnyRoom(8,20);

     string[] thingTypes =
            {"Monster","","PowerUp","","RailGun","","Monster"};
     int i;

     for(i=0;i < 8;i += 2)
     {
          attachSpoke(central.Walls[i],thingTypes[i]);
     }
}
```