MML: A LANGUAGE AND SYSTEM FOR PROCEDURAL MODELING AND MOTION

Mark Green
Hanqiu Sun

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

ABSTRACT

This paper describes a special purpose programming language, called MML, that has been developed for procedural models. This language is used to specify both the generation of the model, and the motion of the objects in the model. A motion specification technique that works well with natural phenomena is also described. This motion specification technique has the advantages of being independent of the detailed structure of the object, and lending itself to the construction of good user interfaces for animation systems. The MML language can be used to produce both interactive and batch animation programs. The user interface for the interactive animation programs is automatically constructed by the MML system. Several examples of the use of MML are presented.

KEYWORDS: Computer Animation, Natural Phenomena, Interactive Animation Systems

1. Introduction

A number of procedural modeling techniques have been developed for natural phenomena. These techniques are based on using formal systems (such as grammars), or procedures to generate the geometrical primitives that describe the shape of an object. Some examples of this approach to natural phenomena are L-systems, graftals [Smith 1984], fractals [Fournier et.al 1982], and particle systems [Reeves 1983]. Each of these techniques is capable of describing a range of phenomena, but none of them is capable of describing the entire range of natural phenomena that we would like to model. One exception to this is the work of Smith that has attempted to generalize the graftal approach to cover the other modeling techniques [Smith 1987]. One of the major research topics in natural phenomena is the development of general modeling techniques.

A major problem with using procedural models in computer animation is that the traditional methods of motion specification don't work very well with them. It is usually not possible to use key framing or scripting techniques with the types of models that these techniques produce, due to the large number of primitives in them and the use of pseudo random numbers. In order to conveniently specify the motion of these objects new motion specification techniques are required. This issue is explored further in section two of this paper, and forms another main research area in natural phenomena.

In order to explore both of these research areas we have been developing a programming language for procedural models and their motion. The main motivation for developing this language was our frustration with using regular programming languages for the development of procedural models. The amount of coding required to produce a model using these languages tended to discourage experimentation. We wanted a system that would allow us to quickly and easily explore different types of procedural models and techniques for specifying their motion. We also wanted to produce interfaces to these models that were easy to use. We found that most of our programs had basically the same structure and used the same computational techniques. This suggested that there should be a higher level way of describing these models. The language described in this paper is our first attempt to produce a higher level notation for procedural models and motion. This is an experimental system for exploring the issues in procedural modeling and motion, it is not intended to be the final solution to these problems.

The second section of this paper briefly describes the technique that we have developed for specifying the motion of procedurally generated objects. The MML language is described in section three, and its implementation is discussed in section four. A number of examples of the use of MML are presented in section five. The last section of this paper discusses some of the issues that we hope to explore through the use of this system.

2. Procedural Motion

One of the main problems with procedural modeling is motion specification. Most procedural models are capable of generating large numbers of primitives, and the generation process usually involves pseudo-random numbers. As a result, the animator cannot directly specify the motion of each primitive in the model. For example, in the case of a model of a tree there may be several hundred branches and thousands of leafs. An animator cannot use a key frame system to position each branch and leaf in the key frames of an animation. Similarly, it would be hard to use a scripting language to specify the transformations to be applied to each primitive. In order to animate this type of object the animator must have some means of specifying the motion at a level higher than the individual primitives.

The motion specification technique that we are using for natural phenomena is based on the use of motion verbs and motion processes. The animator specifies the motion of the object in terms of the motion verbs which are acting on the object. A motion verb is viewed as an atomic motion that is applied to an object over a fixed length of time. For trees there could be motion verbs for growing and blowing in the wind, and for people walk and reach could be motion verbs. Motion verbs can have parameters that are used to control the details of the motion. For example, in the case of the blow in the wind motion verb, the parameters are the wind direction and wind speed. From the animator's point of view, a motion verb should be independent of the detailed structure of the object, and its meaning should be intuitively obvious. A motion verb should be applicable to a range of objects. That is, all trees should respond to the same set of motion verbs (the motion produced will depend upon the type of tree).

Each motion verb is implemented by a collection of motion processes. A motion process is a (virtual) process that is attached to one of the primitives in the model. This process is responsible for the motion of that primitive while the corresponding motion verb is active. Conceptually the motion processes are generated at the same time as the primitives in the model, and they have the same connectivity. That is, the motion processes for a primitive communicates with the motion processes on the primitives they are connected to. In the case of a tree, the motion processes attached to a branch would communicate with processes on the parent branch and its child branches.

Conceptually, the motion processes are viewed as executing concurrently, but in practice this is usually not necessary. For most motions, a motion process has two distinct tasks it must perform. The first task is to determine the state of the primitive it is attached to, and the environment it is acting in. The environment of a motion process usually consists of the states of the primitives it is connected to, plus any factors external to the model. In the case of blow in the wind, the motion process must determine the forces exerted on it by its child branches, and the interaction between the wind and the branch. The second task is to compute the motion of the primitive. The main input to this computation is the state information that was gathered in the first task. Thus, for each time step in the motion we need two passes through the primitives in the model. The first pass starts at the youngest generation of primitives (leaves in the case of a tree) and works towards the oldest generation (the root of the tree). This pass computes the state of each primitive, which is stored with the primitive. The second pass starts with the oldest generation and works towards the youngest generation. The pass computes the motion of the primitives.

From the above description of the motion computations, it can be seen that motion specification is in terms of primitive types, and not the individual primitives. For each type of primitive, we need to specify a function, called f1, that computes the state of the primitive, and a function, called f2, that computes the motion of the primitive. When the motion verb is executed, each primitive's type is used to determine the functions to be executed in order to compute its motion. Note that the specification of the motion is independent of the detailed structure of the object, since the motion is specified in terms of the primitive types, and not the primitives themselves.

3. The MML Language

The main purpose of the MML language is to provide a high level notation for describing procedural models and their motion. The MML system can produce two types of animation programs, which have two distinct purposes. The first type of animation program has an interactive interface and is used in the exploration and evaluation of different types of procedural models and techniques for specifying their motion. For this purpose the two most important feature of MML are the ability to quickly implement a procedural model, and then be able to interactively vary its parameters and instantaneously observe the results. The MML system automatically provides the bookkeeping utilities that are normally associated with procedural modeling, and provides a framework that facilitates the development of this type of model. It also produces an interactive interface to for the animation program. This interface is based on the motion verbs in the model, and the parameters to the motion verbs and the generation process. Thus, the programmer doesn't need to provide a user interface to each of the models that he or she creates. This also ensures that there is some uniformity in the interfaces that are used by different modeling programs.

The second type of animation program is used for producing high quality animation based on the procedural model. These programs are script driven, where the commands in the script are the motion verbs in the model. These commands specify the frame in which the motion verb is active, and the parameter values for the motion verb. The script parser for the model is automatically produced by the MML system. The interactive animation programs can be used to produce scripts that are used as the input to the corresponding batch animation programs. In this way the animator can interactively develop a animation using a simple rendering of the objects, and then reproduce the motion with better rendering through a batch animation program.

The MML language is an extension of the C programming language. There are three reasons for basing MML on an existing programming language. First, most of the MML programs perform a fair amount of computation, thus most of the features of a programming language would need to be included in MML. We felt that it would be better to use an existing language for these features, rather than designing one of our own. Second, the use of an existing language allows us to use packages and subroutines that have already been written, saving us some programming effort. This also makes it easier to intergrate MML programs with our existing animation software. Third, basing MML on an existing programming language greatly simplified its implementation. Instead of writing a compiler for MML, all we needed to do was write a preprocessor that converts MML programs into C programs.

An MML program is divided into four sections; which are called Primitives, Generate, Motion, and Render. The Primitives section describes the geometrical primitives that appear in the model. The Generate section describes how the primitives in the model are generated. The motion verbs that operate on the model are described in the Motion section. The conversion of the geometrical primitives into display primitives is defined in the Render section. The following sections contain brief discussions of some of the features of MML, a complete description of the language can be found in its reference manual [Green 1987].

3.1. Primitives Section

In order to describe the generation of the model and its motion, we need descriptions of the geometrical or modeling primitives that are used in it. The Primitives section of an MML program, which starts with the keyword Primitives, contains the definitions of the primitives in the model. A primitive definition consists of the name of the primitive followed by a list of the its properties. Each property definition consists of a type followed by the property name. The property definitions have the same syntax as C variable declarations. Examples of primitive definitions can be found in the examples section. Each of the primitives is converted into a C structure which can be referenced in the other sections of the program. The name of this structure is the name of the primitive with the prefix "MML_".

3.2. Generate Section

The Generate section describes how the primitives in the model are generated. A language for procedural modeling must be based on some general model for the generation of primitives. At the present time, the rule or grammar based notations seem to be the most general [Smith 1987]. As a result, the model that we are using for the generation process is rule based. The generation process consists of a sequence of generations. Some of these generations (the number of generations is under animator control) occur before the first frame of the animation is generated. After that there is one generation in each frame. In a generation, each of the primitives that are in the model (as of the last generation) is considered by all of the rules in the model. If the primitive satisfies the rule, it is applied resulting in the generation of new primitives. Internally, the primitives in each generation are stored together, and it is easy to determine the generation that a primitive belongs to (this information can be used in the generation rules, and in the motion computations). The current generation number is available to the rules, making it possible to have rules that apply to only the primitives in the previous generation.

The Generate section is divided into three sub-sections. The first sub-section contains the definitions of the parameters used in the generation of the model. Each parameter definition consists of a type, a parameter name, and an optional range of values. The syntax of the types and parameter names is the same as in the Primitives section.

The Initial sub-section is used to defined the initial state of the model, which is the set of primitives that belong to the first generation. Each of these primitives is listed in the Initial sub-section, along with values for some, or all, of their properties. Any of the primitives in this sub-section can be given a name so they can be referenced in other sections of the program.

```
parent: primitive_type(generation_number)

when:       initial
            motion
            always
            generation(expression,expression)
            frame(expression,expression)

number(expression,expression)
```

Fig. 1 MML condition clauses

The Rules sub-section consists of a set of rules that describe how the primitives in the model are generated. Each rule has a condition part and an action part. The action part of the rule is a sequence of C statements. The condition part of the rule is made up of a sequence of clauses separated by semicolons. A clause can be a C expression or one of the MML clauses shown in fig. 1. The parent clause must be included in each rule, and it states the type of primitive that the rule can be applied to. The rule is applied to each primitive of the indicated type that satisfies the other clauses in the condition. The primitive under consideration is called the parent of the rule, and it can be referenced in the clauses of the condition, and the body of the rule. The when clause specifies when the rule is to be applied. A generation rule can be applied either before the animation starts, or in each frame of the animation. In the first case the rule is called an initial rule, and in the second case it is called a frame rule. The initial and generation values for the when clause specify an initial rule, with the generation value indicating a range of generations that the rule can be applied in. The motion and frame values indicate a frame rule, with the frame value specifying a range of frames that the rule is to be applied in. The always values specifies that the rule is both an initial and a frame rule. The number clause specifies that the rule is to be applied a random number of times for each parent primitive. The two expressions in this clause give the range of the uniformly distributed random number used as the repeat count for the rule.

3.3. Motion Section

The Motion section contains the declarations of the motion verbs for the object. This section starts with the keyword Motion followed by a list of motion verb declarations. A motion verb declaration, as shown in fig. 2, is divided into two parts. The first part contains the declaration of the parameters to the motion verb. These declarations have the same format as the parameter declarations in the Generate section. The second part of the declaration consists of the f1 and f2 functions that are applied to the primitives in the model. The body of an f1 or f2 function is a sequence of C statements. If an f1 or f2 function is omitted, the primitives of that type are not processed in the corresponding pass of the motion computations. If one of the primitive types is not listed in the Functions part of a motion verb declaration, primitives of that type don't take part in the motion computations.

```
verb_name
            Parameters
                        type parameter_name (min,max);
                                    .
                                    .
                                    .
                        type parameter_name (min,max);
            Functions
                        primitive_name
                                    f1: {
                                                C statements
                                        }
                                    f2: {
                                                C statements
                                        }
                                    .
                                    .
                                    .
```

Fig. 2 Motion Verb Declaration

3.4. Render Section

The render section describes how the modeling primitives are converted into display primitives. This section consists of three sub-sections, as shown in fig. 3. The statements in this section are used to construct a set of procedures that are called in each frame of the animation. The statements in the Initialization section are called at the start of the frame, and their purpose is to initialize the rendering process. The statements in the Finalization section are called at the end of the frame, and they are responsible for any operations that must be performed to transfer the image to a display device or file. The remaining statements in this section specify how the modeling primitives are converted into display primitives. Each of the primitive types has a sequence of C statements associated with it, and these statements are responsible for performing this conversion. If one of the modeling primitives is not mentioned in this section, it will not be converted into display primitives.

4. Implementation

The implementation of MML is divided into two parts. The first part is a preprocessor that converts MML programs into C routines. The second part consists of two libraries that support the interactive and batch versions of the animation programs. The output from the preprocessor is compiled and then loaded with one of these libraries to produce either an interactive or batch animation program. This process is illustrated in fig. 4.

```
Initialization
            C statements
Finalization
            C statements
primitive_name
            C statements
primitive_name
            C statements
                    .
                    .
                    .
```

Fig. 3 Render Section

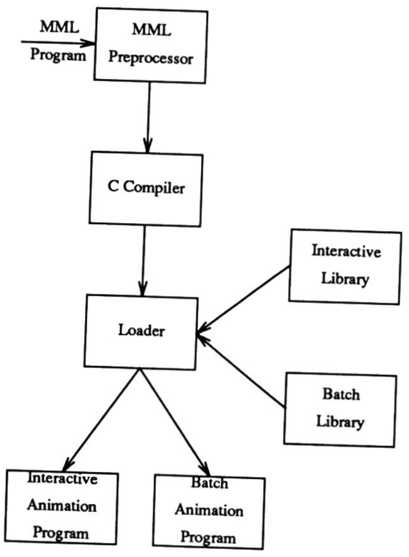

Fig. 4 Conversion process for MML programs

The routines produced by the preprocessor fall into two groups. The first group of routines corresponds to the C statements that appear in the MML program. These routines are responsible for generating the model and computing its motion. The second group of routines form the interface between the support library and the first group of routines. While the preprocessor is processing the MML program it gathers information about the generation process and the motion verbs. This information includes the names, types and ranges of the parameters for the generation process and the motion verbs, and the names of the motion verbs. This information is used to construct the user interface in the interactive animation system. The batch version of the animation system uses this information to parse the script provided by the animator.

The user interface that is provided by the MML system is divided into four sections, which are called view, generate, motion, and edit. The view part of the user interface is used to set the viewing parameters and specify whether the frames produced in the session should be saved for interactive playback. The view part of the user interface is also responsible for the playback of the recorded frames. All the view parameters can be modified through the use of graphical potentiometers.

The generation part of the user interface controls the generation of the primitives in the model. When this part of the user interface is entered, a bank of graphical potentiometers is displayed across the bottom of the screen. There is one potentiometer for each of the parameters in the Generation section of the MML program, and a potentiometer for specifying the number of generations to be produced before the first frame of animation. Once the parameter values have been selected the animator can select the start command from the menu to produce the model. There is an erase command that can be used to remove all the primitives from the model, allowing the animator to start over again with a new set of parameter values.

The motion part of the user interface is used for controlling the motion verbs that are applied to the model. This part of the user interface has a menu containing the names of all the motion verbs. When the animator selects a motion verb from this menu, a bank of potentiometers appears across the bottom of the screen. There is one potentiometer for each of the parameters to the motion verb, and one potentiometer for specifying the number of frames that the motion verb is active in. Once the animator has selected the parameter values that he or she wants, the okay command is selected from the menu. This command is used to add the motion verb to a table of currently active motion verbs. At this point the animator can select other motion verbs to be applied to the model, or select the start command to start the computation of the motion. Before the motion is computed, the table of currently active motion verbs is scanned to find the motion verb with the shortest duration. The motion is then computed up to the last frame in which this motion verb is active. The animator can then enter new motions, or restart the motion computations.

All the motion verbs that the animator enters in a session are recorded in a motion table. This table can be edited in the edit part of the user interface, and can be used to generate a script for a batch animation program. In the edit section of the user interface, the motion table is displayed on the screen, and the user can edit its contents by pointing at a field and entering a new value for it. If the motion table doesn't fit on the screen, there are commands that can be used to scroll forward and backwards through the table.

5. Examples

In this section three examples of the use of MML are presented. The purpose of these examples is to illustrate the range of models that can be represented in MML, and show how the features of this language are used.

The first example, shown in fig. 5, is a model of a water fountain. In this example a water is modeled as a collection of particles. There are two types of primitives in this model, the water particles and the source of the water fountain. The source of the water fountain rotates about the z axis, and the rate of this rotation is specified when the model is created.

```
Primitives small
 source
     double x, y, z ;                /* source position */
     double x_size, y_size, z_size ;/* size of source */
     double vx, vy, vz ;             /* mean velocity of particles */
     double theta;                   /* current angle */
 end ;

 particle
     double x, y, z ;                /* particle position */
     double vx, vy, vz ;             /* particle velocity */
     int life_time ;                 /* length of time left for particle */
 end ;

Generate
 Parameters
   int nparticle ( 20, 500 ) ;       /* mean number of particle produced */
   int life ( 10, 100 ) ;            /* average life of a particle */
   double dtheta ( 0, 5) ;           /* increment in source angle */

 Initial
   source fountain
     x = 0.0 ; y = 0.0 ; z = 0.0 ;
     x_size = 0.0 ; y_size = 0.0 ; z_size = 0.0 ;
     vx = 0.2 ; vy = 0.2 ; vz = 10.0 ;
     theta = 0;
   end ;

Rules

 /* rotate the source about the zaxis in each frame
    of the animation
 */

 parent: source ; when: motion
 => {
     struct MML_source *p;
         double cos(), sin();

         p = get_primitive(parent);
         p->vx = 0.4 * cos(p->theta);
         p->vy = 0.4 * sin(p->theta);
         p->theta = p->theta + dtheta;
     };

     /* generate a random number of new particles */

     parent: source ; when: motion ; number(nparticle-10,nparticle+10)
     => { double t ;
         struct MML_source *p ;
         struct MML_particle *current;
         double range();
```

```
          p = get_primitive ( parent ) ;
          current = generate ( particle ) ;
          current->x = range ( p->x - p->x_size, p->x + p->x_size ) ;
          current->y = range ( p->y - p->y_size, p->y + p->y_size ) ;
          current->z = range ( p->z - p->z_size, p->z + p->z_size ) ;
          current->vx = range ( p->vx - 0.1, p->vx + 0.1 ) ;
          current->vy = range ( p->vy - 0.1, p->vy + 0.1 ) ;
          current->vz = range ( p->vz - 1.0, p->vz + 1.0 ) ;
          current->life_time = irange ( life - 5, life + 5 ) ;
          t = range ( 0.0, 1.0 ) ;
          current->x += t * current->vx ;
          current->y += t * current->vy ;
          current->z += t * current->vz ;
          add_primitive(current);
        };

     /* delete particle at the end of their life time */

     parent: particle ; when: motion ;
       ( p->life_time < 0 )
       => remove_primitive ( parent ) ;

     /* delete particles that fall below the ground level */

     parent: particle ; when: motion
       => {
          struct MML_particle *c;

          c = p;
          if(c->z + c->vz < 0)
            remove_primitive (parent);
         };

  Motion

    Update

    Parameters
      double g ( 0, 25 ) ;          /* gravitational constant */

    Functions
      particle

        /* compute the motion of a water particle */

        f2: {
            struct MML_particle *c;

            c = current;
            c->x += c->vx ;
            c->y += c->vy ;
            c->z += c->vz ;
            c->vz -= g ;
            c->life_time -= 1 ;
           };
  end;

Render

 Initialization
    color(BLACK) ;
    clear() ;

 Finalization
    swapbuffers() ;

 particle
    color(WHITE) ;
    pnt(current->x, current->y, current->z) ;
```

Fig. 5 MML Code for water fountain example

The generate section of the model has four rules. The first rule is responsible for rotating the source of the water fountain. The second rule is responsible for the generation of the particles in the water fountain. In this rule the generate function is used to allocate memory for the primitives, and the add_primitive procedure is used to add the primitive to the model. In an MML program the primitives are referenced by an integer primitive number. This allows the run-time system to select the most efficient scheme for storing the primitives (for example, in the case of large models some of the primitives can be stored on disk) without changing the MML program. The get_primitive function is used to convert a primitive number into a pointer to its C structure. The last two rules delete particles from the system when they reach the end of the lifetime, or when they fall below ground level.

The Update motion verb updates the particle properties on each frame of the animation. The Render section of the program converts the particles into point primitives that are displayed on an IRIS workstation.

The MML program shown in fig. 5 can be converted into an interactive animation program by following the steps outlined in section 4. When this program is executed the display shown in fig. 6 appears on the screen. This part of the interface coordinates the execution of the different parts of the user interface. At this point in the interaction the user would enter the generate system and create the initial primitive in the model. The user would then enter the motion part of the user interface. When this happens the display shown in fig. 7 appears on the screen. In this display the user has selected the Update motion verb, and the potentiometer for its parameter has appeared at the bottom of the screen. The user can now select the okay and start commands from the menu. Once these actions have been performed, the program will calculate the frames in the animation and show them to the user as they are computed.

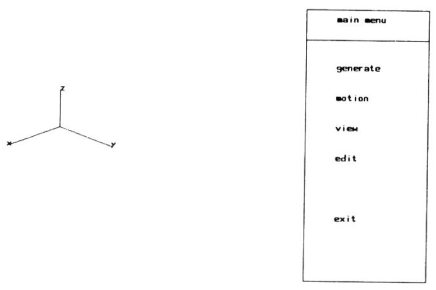

Fig. 6 Initial screen display

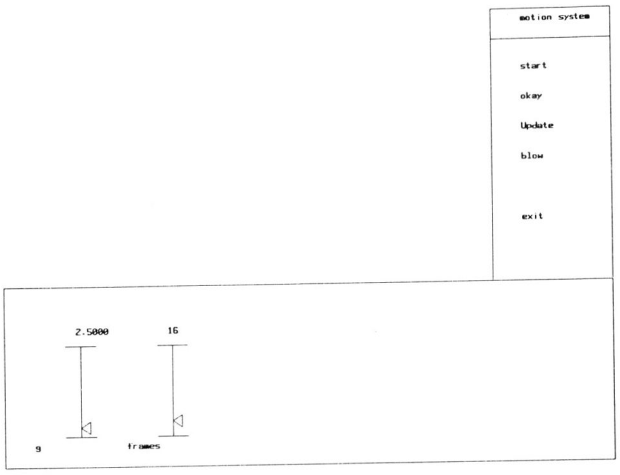

Fig. 7 Motion system display

Once the motion has been computed, the view part of the user interface can be used to view the frames at a controlled rate. Fig. 8 shows the one frame of the water fountain animation viewed through the view system. The speed potentiometer controls the rate of display, in terms of the delay between frames (in units of 1/60 second).

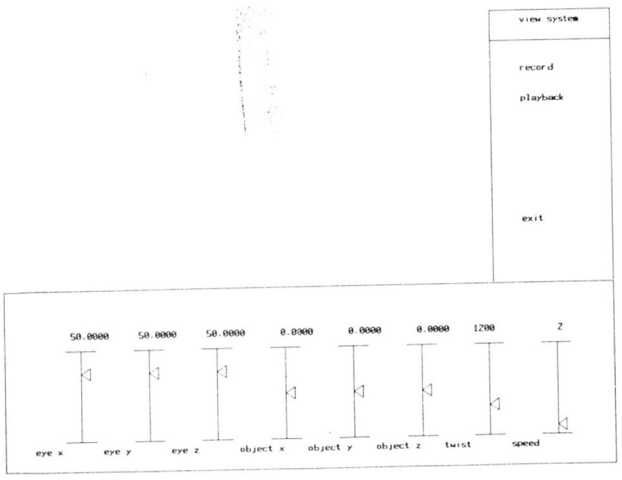

Fig. 8 View system and water fountain

The MML program shown in fig. 9 implements the blow in the wind motion verb for trees. The tree model developed by Aono and Kunii [Aono and Kunii 1984] is used to generate the model of the tree. The single rule in the Generate section of the program generates two new child branches for each branch produced in the previous generation.

```
Primitives small
 branch
  double  position[3];          /* distal end of branch */
  double  B[3];                 /* vector along the length of the branch */
  double  f1;                   /* f1 value for the branch */
  double  mass;                 /* mass of the branch */
  double  length;               /* length of the branch */
  double  radius;               /* radius of the branch */

  int     child1, child2;       /* the branch's children */
  int     parent;               /* the branch's parent */
 end;

Generate
 Parameters

 /* parameters for tree model */

 Initial

 /* generation of initial branch */

 Rules

 parent: branch(generation-1) ; when: initial
 => {

   /* generate child branches */

   };

 Motion

 blow

  Parameters
   double wx ( 0, 30 );          /* wind direction */
   double wy ( 0, 30 );
   double wz ( 0, 30 );
   double strength ( 0, 20 );    /* wind speed */

  Functions
   branch

    /* compute the forces acting on a branch */

    f1: {
      struct MML_branch *b, *p;
      double l, sqrt();
      double x,y,z;
      double t;
      double range();

      b = current;

      /* compute direct interaction between wind
         and the branch
      */

      l = sqrt(wx*wx + wy*wy + wz*wz);
      t = wx*b->B[0] + wy*b->B[1] + wz*b->B[2];
      t = t/(b->length * l);
      b->f1 = range(0.1,0.3) * strength * b->radius * (1-t);

      /* add in the forces from the children */

      if(b->child1 != 0) {
       p = get_primitive(b->child1);
       b->f1 += 0.1 * p->f1;
       p = get_primitive(b->child2);
       b->f1 += 0.1 * p->f1;
      }
      };
```

```
/* compute the motion of the branch */

f2: {
   struct MML_branch *b, *p;
   double l, sqrt();
   double x,y,z;
   double t;
   double w;

   b = current;

   /* compute w, the relative displacement of
      the branch
   */

   l = sqrt(wx*wx + wy*wy + wz*wz);
   t = wx * b->B[0] + wy*b->B[1] + wz*b->B[2];
   t = t / (b->length*l);
   w = 0.2 * b->f1 * (1-t) / b->mass;

   /* compute the vector from the proximal to
      the distal end point of the branch
   */

   x = (w*wx + (1-w)*b->B[0]/b->length) * b->length;
   y = (w*wy + (1-w)*b->B[1]/b->length) * b->length;
   z = (w*wz + (1-w)*b->B[2]/b->length) * b->length;

   /* update the distal position of the branch */

   if(b->parent == 0) {
     b->position[0] = x;
     b->position[1] = y;
     b->position[2] = z;
   } else {
     p = get_primitive(b->parent);
     b->position[0] = p->position[0] + x;
     b->position[1] = p->position[1] + y;
     b->position[2] = p->position[2] + z;
   }
   };
end;

Render
 /* draw the tree */
```

Fig. 9 MML code for tree example

The Motion section of the program contains the declaration of the blow motion verb. This motion verb uses an f1 and an f2 function in the computation of the tree's motion. The f1 function computes the force acting on the branch and stores the result in the f1 property of the branch. This force is divided into two components. The first component is due to the direct interaction between the wind and the branch. This component is proportional to the strength of the wind and the dot product between the wind direction and the branch direction. The second component of the force is generated by the child branches. In this example a fraction of the child's force is added to the force on the parent.

The f2 function computes the new position of the distal end of the branch. It is assumed that the new position will lie on the plane defined by the wind direction and the initial direction of the branch. A weighting factor, which is proportional to the force on the branch, the dot product between the wind direction and branch direction, and inversely proportional to the mass of the branch, is used to determine the branch position within this plane.

Fig. 10 shows a tree produce by the MML program in fig. 9.

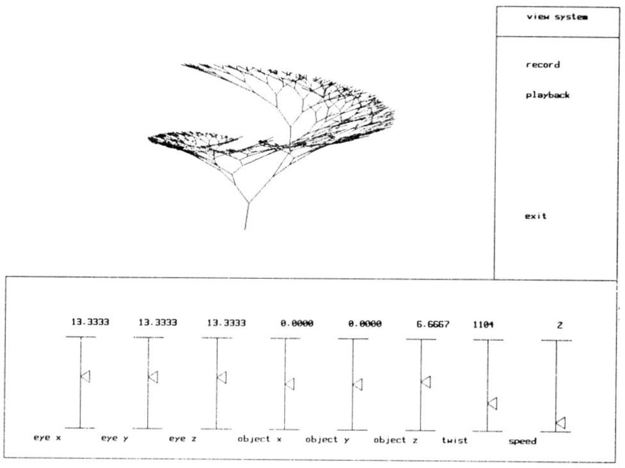

Fig. 10 Tree model

The MML program in fig. 11 computes a fractal based on a set of three seed triangles. The technique used to generate the fractal is essentially the same as the one described in [Fournier et.al. 1982]. The procedure middle, used in the Generate section of the program computes the midpoint of a stochastic line. The motion verb, Update, rotates the eye about the z axis to give different views of the fractal. One view of the fractal is shown in fig. 12.

```
Primitives small
 triangle
  double v1[3], v2[3], v3[3];     /* vertices of the triangle */
  double n1, n2, n3;
  int  color;
 end;

 eye
  double x, y, z;                 /* eye position */
  double radius;                  /* distance between eye and origin */
  double theta;                   /* angle of eye about the z axis */
 end;

Generate
 Parameters
  int levels ( 10, 50 );
  double rough_ratio ( 0.1, 1.0 );

 Initial
  triangle
   v1[0] = -2.5; v1[1] = -2.5; v1[2] = 0.0;
   v2[0] = 0.5; v2[1] = 2.5; v2[2] = 0.0;
   v3[0] = 2.5; v3[1] = -2.5; v3[2] = 0.0;
  end;

  eye Eye
   x = 10; y = 10; z = 10;
   radius = 10;
   theta = 0;
  end;
```

```
Rules

parent: triangle ; when: initial
=> { struct MML_triangle *p,*p1,*p2,*p3,*p4;
    double m1[3],m2[3],m3[3];
    int i;

    p = get_primitive( parent);
    p1 = generate(triangle);
    p2 = generate(triangle);
    p3 = generate(triangle);
    p4 = generate(triangle);
    middle(p->v1,p->v2,m1,rough_ratio);
    middle(p->v2,p->v3,m2,rough_ratio);
    middle(p->v3,p->v1,m3,rough_ratio);

    /* store the vertices of the new triangles */

    add_primitive(p1);
    add_primitive(p2);
    add_primitive(p3);
    add_primitive(p4);
    remove_primitive(parent);
    };

Motion

 Update

  Parameters
   double dtheta (-1, 1);
  Functions
   eye
    f2: {

        /* update the eye position */

        };
  end;

Render

 Initialization
    struct MML_eye *e;
    color(BLACK);
    clear();
    perspective(450,1.0,0.0,900.);
    e = get_primitive(Eye);
    lookat(e->x,e->y,e->z,0.0,0.0,0.0,1200);

 Finalization
    swapbuffers();

 triangle
    color(WHITE);
    move(current->v1[0],current->v1[1],current->v1[2]);
    draw(current->v2[0],current->v2[1],current->v2[2]);
    draw(current->v3[0],current->v3[1],current->v3[2]);
    draw(current->v1[0],current->v1[1],current->v1[2]);
```

Fig. 11 MML code for fractal

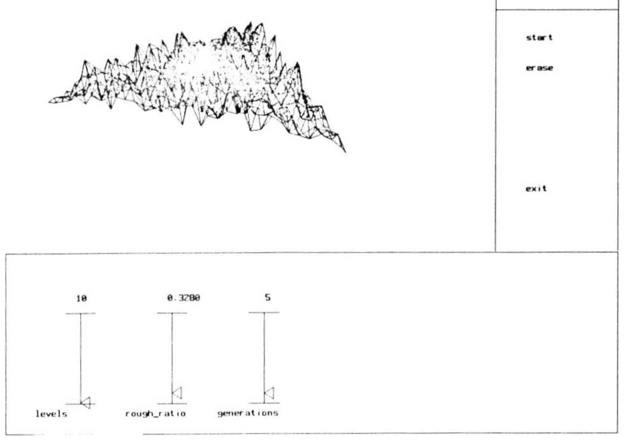

Fig. 12 Fractal display

6. Issues

The main reason for producing the MML system was to investigate different issues in the generation and motion of models of natural phenomena. These issues can be divided into two groups. The first group of issues address the basic research problems, while the second group is related to the production of animation software. In this section some of these issues are briefly discussed.

Model generation in MML is based on a collection of rules applied to the primitives in the model. As the examples show, this technique is general enough to handle fractals, particle systems, and trees, but is it general enough to handle other types of natural phenomena? Objects that are based on meshes of primitives, such as bodies of water, don't seem to fit into this framework very well. We have thought of rule schemes that produce meshes, and since MML is based on C, we could always use a for loop to generate the primitives. These solutions don't seem to be the most elegant ones, but they do work.

Motion computation in MML is based on two passes through the primitives in the model, for each time step of the animation. This scheme has handled all the types of motion that we have tried so far, but we may be biased by what we know is possible. When we start studying mesh based models we may find that this scheme doesn't work as well.

Both the interactive and batch animation programs allow multiple motion verbs to be acting on an object at the same time. There is the possibility that these motion verbs could interfere with each other. At the present time the MML system doesn't detect this interference, the motion verbs are executed in the order they were entered in the motion table without consideration for other motions in the object. We have taken this approach because we don't have a better solution to the problem. We could attempt to detect interfering motion verbs and apply some conflict resolution strategy, but it is not clear how expensive this would be and whether the results would be worthwhile. We are currently investigating techniques for handling conflicting motions.

A related problem is the interaction between objects in the same environment. This interaction could vary from objects avoiding obstacles and following terrain, up to complex synchronized motions between multiple objects (such as a group of dancers). This type of behavior requires some form of communications between the objects in the model and their motion processes.

The motion verb technique currently has no mechanism for combining several motions into one compound motion. Such a mechanism would allow the programmer or animator to develop complex motions out of simpler motions, and thus extend the animation system. Some of the mechanisms that we have considered are the parallel execution of several motion verbs (which brings us back to the previous issues), and the sequential execution of a set of motion verbs. Probably several composition mechanisms will be required.

There are several aspects of the MML system that we would like to improve. In the Generate section, the initial primitives in the model must be explicitly listed with their property values. In some cases this is not the most convenient way of specifying them. We would like to have interaction techniques for entering the primitives, a database system where different sets of initial primitives could be stored, and a means of computing the initial primitives.

MML is an extension of the C programming language, and as a result there is a tendency to use C code whenever MML doesn't provide the appropriate language construct. We would like to have a notation that is at a higher level and at the same time gives us the same flexibility that we have with MML.

Acknowledgements

We would like to thank our colleagues in the department for their comments on our work, and Christina Lau for implementing the first version of the MML preprocessor.

References

[Aono and Kunii 1984] Aono M., T.L Kunii, Botanical Tree Image Generation, IEEE Computer Graphics and Applications, vol.4, no.5, p.10-34, 1984.

[Fournier et.al 1983] Fournier A., Fussel D., Carpenter L., Computer Rendering of Stochastic Models, Comm. ACM, Vol. 25, p. 371-384, 1982.

[Green 1987] Green M., MML: A Modeling and Motion Language, Department of Computing Science, University of Alberta, 1987.

[Reeves 1983] Reeves W., Particle Systems - A Technique for Modeling a Class of Fuzzy Objects, ACM Transactions on Graphics, Vol. 2, p. 91-108, 1983.

[Smith 1984] Smith A.R., Plants, Fractals, and Formal Languages, Siggraph'84 Proceedings, p. 1-10, 1984.

[Smith 1987] Smith A.R., Formal Geometric Languages for Natural Phenomena, The Modeling of Natural Phenomena, Siggraph'87 Course Notes, 1987.