A New Programming Environment for Dynamics-based Animation

Leonardo L. Oliveira Paulo A. Pagliosa

Universidade Federal de Mato Grosso do Sul, Departamento de Computação e Estatística, Brasil

Abstract

This paper presents a new animation system initially designed for visualization of dynamics simulations in science and engineering applications. A hybrid language is used to describe the scene objects to be animated and the scripts and actions that modify the state of the objects over time. The system is made up of components responsible to compile and execute an animation, and render and exhibit the resulting frames. Currently, the animation system is being extended to support dynamic simulations of rigid and elastic bodies in interactive, real-time applications, including games. For rigid body simulations, AGEIA PhysX engine is used as a component. The paper presents the functionality of the main components of the original system architecture, introduces the main features of the animation language and describes how an animation is specified and then executed by the system.

Keywords: animation, dynamic simulation.

Contact with authors:
{llo,pagliosa}@dct.ufms.br

1. Introduction

Computational simulation of a physical phenomenon consists in the implementation of models that represent relevant aspects of the structure and behavior of the objects involved in the phenomenon. Simulation can be divided in three phases: modeling, analysis and visualization.

In the modeling phase, or pre-processing, are built the geometric, mathematical and analysis models used in a simulation. A geometric model describes the position, dimension, and shape of an object, among other attributes (textures, materials, etc.). A mathematical model is a set of differential equations governing the physical behavior of an object. An analysis model is a mesh of non-overlapping cells, or elements, resulting from a subdivision of the domain and/or the boundary of an object. The analysis model is usually required to numerically solve the mathematical model for the general case of geometry and boundary conditions. The analysis phase, or processing, takes as input an analysis model and produces as output a (generally large) set of numbers that represents the discrete solution of the mathematical model. The visualization phase, or post-processing, transforms the analysis output in graphics primitives that, once rendered, allow a more immediate comprehension of the effects of the phenomenon being simulated. In dynamic simulations, specifically, the analysis is performed over a time interval, which is divided in a number of time steps. The visualization of analysis results at each time step requires, in order to give to the user a view of what happens in the simulation over time, the use of animation techniques.

The paper introduces an animation system AS initially designed to be a visualization tool for dynamic simulation in science and engineering applications. In order to make the system as applicable as possible, the authors have developed an animation language AL to describe the objects and scripts of an animation [Oliveira 2006]. In addition, it is available an application programming interface (API) which implements classes that represent scenes and their components (actors, lights, cameras, etc.), and scripts, actions and events. From these base classes, new ones can be derived for specific applications.

The AL animation language is derived from a general-purpose, hybrid (i.e., supports both global functions and data structures as well as object-oriented features, like C++) language L [Oliveira 2006] in which were added productions to facilitate the creation of scene components and the specification of scripts, actions and events. The animation system has as components an AL compiler and an animation virtual machine (AVM) that executes the resulting compiled bytecode and controls the update cycle of an animation. Efficiency can be achieved with native methods, i.e., methods whose body is implemented in a language other than the animation language (usually C++). A number of methods in system API are native.

Although it was possible to extend an existent language (such as Java), it has decided to implement a proper animation language, since L and its virtual machine LVM had already been developed by the authors and were available to be used in the animation system. Because AL was derived from L and AVM from LVM, it was easier to integrate the virtual machine with the other components of the system; and in the future it will be easier to add new features in the language, such as behavior rules for actors.

In the animation system, scripts and actions can be used to create new objects and control over time any changes on the state of objects in a simulation (not only movement, but also appearance, acting forces and torques, etc.). In addition, one of the AS components is a physics engine responsible for computing the effects of constrained dynamics on the objects. The version of the system described in the paper uses the AGEIA PhysX engine [AGEIA 2006] for collision detection and dynamic simulation of rigid bodies only. Currently, AS is being extended to support interactive, real-time dynamic simulation of both rigid and elastic bodies.

The paper depicts the proposed animation system and is organized as follows. Section 2 presents related work. Section 3 describes the system architecture. Section 4 introduces how to use the animation language and the main classes of the system API to specify an animation. Section 5 gives an overview of the internal structure of the AVM and addresses how an animation is executed. Section 6 points out the concluding remarks.

2. Related Work

Script languages constitute a considerable resource to governing an animation. Features such as automatic memory management, garbage collection and support to construction of dynamic data structures, amongst others, make script languages a tool that can be used beyond the scope of computer animation. An example is Lua [Ierusalimschy 2006], which combines simple procedural syntax with data description constructs based on associative arrays and extensible semantics. Lua has been currently used in game development.

ASAS [Reinolds 1982] is one of the first animation language based on actors and scripts. Its goal is to offer to the animator the ability to control an animated sequence through a script. ASAS is based on LISP and introduces specific concepts such as geometric and photometric characteristics, transformations operators and the data structure of an actor.

Zeleznik [1991] proposed a system for object-oriented modeling and animation that provides facilities of integration with some paradigms of animations. Objects can be geometric (actors) and not geometric (cameras, lights, etc.) and exchange messages among them. The list of messages of an object determines its behavior and its variable parameters over time. This list can be modified by the animator or another object, featuring interactions actor-actor and actor-animator; a message is abstract since the object is accountable to define itself how it will be affected by a message.

Inprov [Perlin and Goldberg 1996] is an authoring system for scripting interactive actors in virtual worlds. It consists of two subsystems. The first one is an animation engine that uses procedural techniques to enable animators to create layered, continuous, non-repetitive motions and smooth transitions between them. The second subsystem is a behavior engine that enables animators to create sophisticated rules governing how actors communicate, change, and make decisions. The combined system provides an integrated set of tools for assign the “minds” and “bodies” of interactive actors. An Inprov actor can be doing many things at once, and these simultaneous activities can interact in different ways. The animator can place actions in different groups, and these groups are organized into a “back-to-front” order. Actions in the same group compete with each other and each action possesses some weight (global actions are located in the rear groups and local ones in the front groups). Different scripts can run in parallel and can be ordered on the same temporal referential by using instructions like `wait n seconds`.

Formella [1996] present a complete description of a simple animation language named AniLan. Parameters of objects to be animated can be changed over time with the help of actions, events and cues. AniLan rely your own animation model. The kernel of the model is an animation graph [Braun 1995], which represents the interactions of the objects and uses the dimensions library (parameters of an animation that have a type, which superposition and casting are two basic features) and the function library. An evaluation kernel that interprets the graph is used by an interface to produce the required values for the parameters at a certain instant of time or over a certain period of time. The graph holds all functions and parameters to be animated. A property of the model is that it is not possible to perform simulations, collision detection [Snyder et all 1993] or iteration directly with the graph.

More recently, a scenario language to orchestrate virtual world evolution was proposed by Devillers and Donikian [2003], which allows the description of scenarios in a hierarchical manner and their scheduling at simulation time. The language contains instructions such as `if`, `switch`, `repeat until`, `random choice`, `wait`, which are executed inside a time-step. It also contains more specific instructions (`waitfor`, `each-time`, `aslongas`, `every`) that spend more than one time-step to be finished and can run in parallel during the execution of the scenario they belong to. All those instructions can be composed in a hierarchical manner. To manage actors, the language offers also specific instructions to specify the interface of an actor, and to reserve and free actors.

3. System Architecture

The architecture of the proposed animation system is defined by the following components: animation language compiler, AVM, animation file linker, and animation file viewer.

The animation language compiler (ALC) derives from L compiler. It takes as input files containing the AL specifications of one or more scenes to be animated (`scn` files), and produces as output the corresponding animation object files (`oaf` files). These ones contain bytecode streams that are loaded and interpreted by the AVM. An animation code typically creates scenes and starts scripts and actions that modify the state of a scene over time. Scenes are animated sequentially by the system in order they are started; the scene being animated at a given time is called current scene.

During a simulation the AVM renders frames of the currents scene that can be packed by the animation file linker in order to produce movies in a number of formats (`avi`, `mpeg`, `flic`, among others), which can be displayed by the animation file viewer. For (preview of) animations directly exhibited in graphical windows these components are not used and not discussed here.

The AVM is the most important component of the animation system. It is made up of the subcomponents illustrated in the UML component diagram in Figure 1.

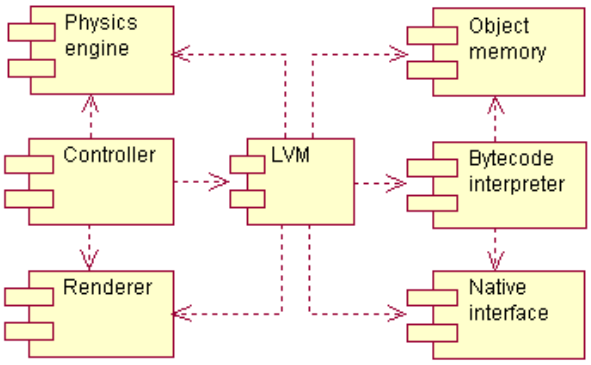

Figure 1: Architecture of the AVM.

The kernel of the AVM is the L virtual machine. This one is made up of the bytecode interpreter, the native interface and the object memory.

The interpreter "executes" object code, i.e., the "instructions" represented by the bytecodes generated by the ALC. There are instructions to create new objects, invoke methods, handle exceptions, etc. Native methods are executed with support of the native interface (NI). When a native method is invoked, the LVM pushes onto native stack the arguments passed to the method and also a reference to the NI object, then calls the native function that implements the method, and pushes the return value, if any, onto the LVM stack. A native code can use the NI object to access many of the functionalities of the LVM, such as to create objects, invoke methods, etc. The object memory is the place where live all objects created by an application. When an object cannot be reached by the LVM, a mark and sweep garbage collector automatically reclaims the memory used by the object.

The controller is the component responsible to orchestrate the execution of the scripts and actions of an animation. The total duration time of a simulation is divided in a discrete number of time steps named ticks. At each one tick the controller determines which pieces of (scripts and actions) code have to be executed by the LVM in order to update the state of the current scene. At the end of the update cycle, the controller invokes the physics engine to perform the physics simulation.

For a positive number of ticks called render resolution, the renderer takes the current scene and renders a scene frame. The current implementation of the system uses a simple, OpenGL based renderer.

The animation system is entirely implemented in C++. The current version runs on Windows.

4. Specifying an Animation

An animation is specified in the animation language which is an extension of the L language. The syntax of L is similar to C++ in some aspects and to Java in others. In common, the language supports selection (**`if-else`**, **`switch`**), iteration (**`for`**, **`while`**, **`do`**), control transfer (**`break`**, **`continue`**, **`return`**), exception handling (**`throw`**, **`try-catch`**), and expression statements. Like C++ and Java, L is strongly typed. Besides primitive types (**`int`**, **`long`**, **`float`**, **`char`**, **`bool`**), the language allows the definition of array types and object classes. Classes can declare inner classes, constructors, attributes, properties, and methods. In addition, L also supports multiple virtual inheritance, generic classes, operator overloading, virtual methods, and friend classes, among other features.

A complete L grammar description can be found in [Oliveira 2006]. Following, a partial, simplified syntax of class declaration. The symbols `*`, `+`, e `?` denote zero or more, one or more, and optional, respectively. Terminals are written in boldface.

```
ClassDeclaration:
  modifier* class Name BaseClassList?
  ClassBody
BaseClassList:
  : Name (, Name)*
ClassBody:
  { (modifier MemberDeclaration)* }
MemberDeclaration:
  CtorDeclaration
  | MethodDeclaration
  | FieldDeclaration
  | PropertyDeclaration
CtorDeclaration:
  constructor ( ExpressionList? ) CtorInit?
  (Block | ;)
CtorInit:
  : BaseClassInit (, BaseClassInit)*
BaseClassInit:
  Name ( ExpressionList? )
FieldDeclaration:
  Type Name (, Name)* ;
MethodDeclaration:
  Type Name ( ExpressionList? )
  (Block | ;)
PropertyDeclaration:
  property Type Name
  { (read = Name)? (write = Name)? }
```

A property is an instance member accessed like an attribute, but whose value is handled by a getter and/or a setter method. If the property is used as a rvalue in an expression, then the getter is invoked, otherwise the setter is invoked. A property can be read-only (without setter) or write-only (without getter). The other productions are straightforward.

4.1 Scene and Scene Component Classes

The entities of an animation are objects of L classes which are grouped together into the animation system API. The classes that represent a scene and its main components are depicted in the UML class diagram in Figure 2 and commented following.

A `Scene` object is a container of actors, lights, and cameras (due to space limitation the classes `Light` and `Camera` are not commented here). An `Actor` object is defined by a geometric model and a body. A geometric model is an instance of the class `Model` and describes the pose, shapes and dimensions of an actor. It is used

by the renderer to synthesize a picture of the actor and defined by a triangle mesh, vertex normals and vertex texture coordinates.

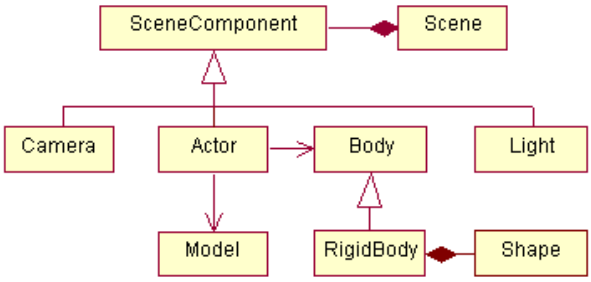

Figure 2: Scene and scene components classes.

An actor body is an object of a class derived from the abstract class `Body` and defines the physical properties of an actor. The class `RigidBody` is a concrete class that encapsulates the specific properties and methods used by the physics engine for dynamic simulation of rigid bodies. The geometry of a rigid body is defined by a collection of `Shape` objects, such as spheres and boxes, which are used by the PhysX to compute the contact points among actors. In general the body geometry is simpler than model geometry, but they can be the same. Figure 3 illustrates on the left the body shapes and on the right the geometric model of a truck.

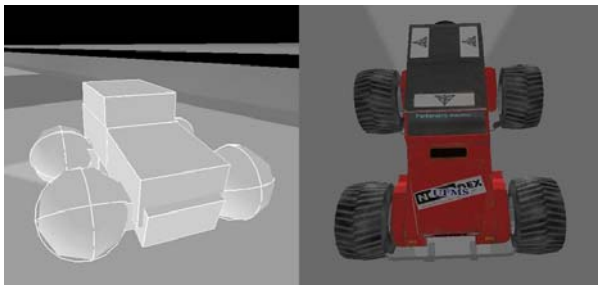

Figure 3: Shapes and model of an actor.

In order to read the examples at the end of this section, the classes `Actor` and `RigidBody` are partially listed below. Dots indicate that details are omitted

```
class Actor: SceneComponent
{
  public:
    property String name { ... };
    property Body body { ... };
    property Model model { ... };
    ...
}
class RigidBody: Body
{
  public:
    property Vector3D position { ... };
    property Quaternion orientation { ... };
    property Vector3D linearVelocity { ... };
    property Vector3D angularVelocity { ... };
    property float mass { ... };
    property Tensor3D inertia { ... };
    property Vector3D centerOfMass { ... };
    property List<Shape> shapes { ... };

    void addForce(Vector3D);
    ...
}
```

The class `Scene` is commented in the next subsection, since a scene is also a sequencer.

4.2 Sequencers and Event Classes

There are objects called sequencers that, in conjunction with the physics engine, are responsible to define the changes in a scene during an animation. A sequencer is any set of activities that, when executed sequentially or in parallel, can modify the state of one or more objects of a scene over time. A sequencer can control the movement of actors, lights, and cameras; change model attributes such as colors and textures; create new scene components; apply forces and torques on rigid bodies; start other sequencers; etc. A sequencer is an object of a class derived from the abstract class `Sequencer`, shown in the UML class diagram in Figure 4. The interface of `Sequencer` is listed below.

```
abstract class Sequencer: SyncObject
{
  public:
    abstract void start();
    abstract void exit();

    property float time { read = getTime };
    float getTime();
}
```

The method `start()` begins the execution of the activities of a sequencer. All the started sequencers will be running in parallel in AVM. The method `exit()` terminates the execution of a sequencer. The read-only property `time`, as well as the method `getTime()`, gives the time in milliseconds since a sequencer was started.

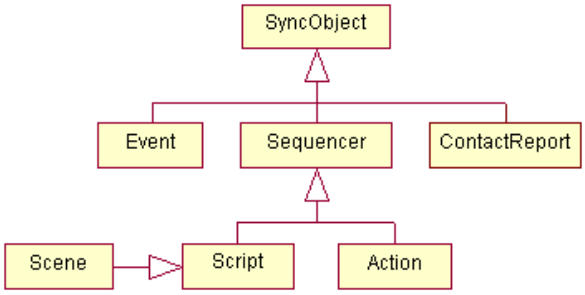

Figure 4: Sequencers and event classes.

A sequencer can be either a script or an action. A script is an object of a class derived from the abstract class `Script`:

```
abstract class Script: Sequencer
{
  public:
    void start();
    void exit();

    int waitFor(float);
    int waitFor(SyncObject, float = -1);
    int waitFor(SyncObject[], float = -1);

  protected:
    abstract void run();
}
```

`Script` overrides the methods `start()` and `exit()` inherited from `Sequencer`. When a script is started, the controller instructs the LVM to invoke the method

run(), which must be override in derived classes. The method implements the activities to be executed by a specific script. The state of a script being executed by LVM is defined as RUNNING. A script terminates when run() returns or when exit() is invoked; in both the cases, the state of the script becomes TERMINATED.

For a RUNNING script, the code in run() will be entirely executed by the LVM in exactly one tick, i.e., during one update cycle (in terms of animation time, it means instantaneously). However, if one of the methods waitFor() is invoked from run(), then the script execution is suspended by the controller until the condition specified by the arguments passed to the method is satisfied. The state of a suspended script is defined as WAITING. A script can wait for a number of update cycles until either a given timeout in milliseconds expires, or one or more synchronization objects become signaled. As soon as the condition is verified, the controller instructs the LVM to resume execution of the method run() in the next instruction after the waitFor() invocation. The state of the script turns back to RUNNING.

A synchronization object is one of a class derived from abstract class SyncObject. It represents a signal being expected by a WAITING script in order to have its execution resumed by the LVM. A synchronization object can be in one of two states at any time: signaled and not signaled. As shown in Figure 4, sequencers are synchronization objects. When a sequencer is created and RUNNING, it is not signalized. As soon as a sequencer terminates, it becomes signaled. Therefore, a script can wait for other scripts (and actions) to be terminated before to proceed its execution.

There are other two types of synchronization objects: events and contact reports. An generic event is an instance of the class Event:

```
class Event: SyncObject
{
  public:
    void setSignaled();
}
```

The method setSignaled() is invoked to manually set the state of an event as signaled.

A contact report is an internal event signaled by the AVM whenever a contact between any two actors is detected. The event is an instance of ContactReport:

```
class ContactReport: SyncObject
{
  public:
    static ContactReport getInstance();
    property List<Contact> contacts { ... };
}
```

An animation code should not create a contact report object. The only instance that should be used is maintained by the AVM and whose reference can be obtained with the static method getInstance(). The object maintains a list with the contact points detected at the current tick. Each contact point is represented by a Contact object (see the second example at the end of this section).

An action is a sequencer of a class derived from the abstract class Action:

```
abstract class Action: Sequencer
{
  public:
    void start();
    void exit();

    constructor(float = -1);
    property float lifetime { ... };

  protected:
    void init();
    void update();
    void finalize();
}
```

An action has as a property its lifetime in milliseconds, which is passed as argument to the constructor (a negative real number is considered as infinite). Action overrides the methods start() and exit() inherited from Sequencer. The activities of an action are coded in the methods init(), update(), and finalize(), which can be override in derived classes. The init() method implements an initialization code for the action and is executed once by the LVM as soon as the action is started. Just after the invocation of start() but just before the invocation of init() the state of an action is set to INITIATED. (Note that initialization code can be also written in a constructor, but in this case it will be executed after the creation of the object). Once it is initiated, the state of an action is set to RUNNING.

At each one tick the controller decrements the value of the property lifetime of a RUNNING action. If it results a positive number, then the controller instructs the LVM to invoke the method update() of the action. An action terminates when its lifetime is zero or when exit() is invoked. In this case the action state is TERMINATED and the method finalize() is invoked.

Actions are used to define activities that continually change over time and must be executed in each update cycle, while scripts are used to define a linear sequence of synchronized activities that must be executed just the once in a priori unknown period of time.

A scene is also a script, in the sense that it can be started and define a sequence of activities which can be executed by its components. The interface of Scene is:

```
class Scene: Script
{
  public:
    static Scene getCurrent();
    constructor(float = -1);

    property float totalTime { ... }
    property List<Actor> actors { ... };
    property Camera camera { ... };
    ...

  protected:
    void run();
}
```

4.3 The Animation Language

It is possible to use the L language together the API to entirely specify an animation. However, it is better do it with the animation language. AL has extensions that

makes easier to create a scene, put components into a scene, and define sequencers. The main features of the language are discussed below.

AL introduces properties blocks. It is easier to explain it with an example. Let be the following code fragment:

```
RigidBody body = new RigidBody();

body.position = <0,0,0>;
body.orientation = new Quaternion(0,<1,1,1>);
body.mass = 50;
body.centerOfMass = <0,0,1>;

Actor actor = new Actor();

actor.name = "actor1";
actor.body = body;
```

The code creates a new rigid body and sets its position, orientation, mass, and center of mass. The rigid body is assigned to a new actor named `actor1`. Note that a 3D vector can be defined by an `<x,y,z>` expression, where `x`, `y`, and `z` are **`float`** expressions. Using a properties block, the code above can be rewritten as:

```
Actor actor = new Actor()
{
  name = "actor1";
  body = new RigidBody()
  {
    position = <0,0,0>;
    orientation = new Quaternion(0,<1,1,1>);
    mass = 50;
    centerOfMass = <0,0,1>;
  };
};
```

A property block is an expression defined as an expression followed by a block containing a list of assignment expressions. The grammar production is:

```
PropertyBlock:
  Expression { (AssignementExpression;)* }
```

The value of `Expression` must be a non-null reference to an object O, and the lvalue of each assignment expression in the block must be a property or an attribute of O. The value of a property block expression is itself a reference to O.

A variant of the property block can be applied to add elements into a collection. To the AVM a collection is any object whose class derives from abstract class `Collection`, such as `Vector` and `List`. Let be the code fragment:

```
Scene scene = new Scene();
Actor actor;

// create actor1 and add it into the scene
actor = new Actor();
actor.name = "actor1";
actor.body = new RigidBody();
...
scene.actors.add(actor);
// create actor2 and add it into the scene
actor = new Actor();
actor.name = "actor2";
actor.body = new RigidBody();
...
scene.actors.add(actor);
```

Using property blocks and the add-into-collection variant, the code can be rewritten as:

```
Scene scene = new Scene()
{
  actors
  {
    // create actor1 and add it into the scene
    new Actor()
    {
      name = "actor1";
      body = new RigidBody()
      {
        ...
      };
    };
    // create actor2 and add it into the scene
    new Actor()
    {
      name = "actor2";
      body = new RigidBody()
      {
        ...
      };
    };
  };
};
```

An add-into-collection variant is an expression defined as an expression followed by a block containing a list of expressions. The grammar production is:

```
AddIntoCollectionVariant:
  Expression { (Expression;)* }
```

The value of the first `Expression` must be a non-null reference to an object O which is expected to be a collection of objects of a type T. The value of each expression in the block must be a reference to an object of the T type. The value of an add-into-collection expression is itself a reference to O.

Property blocks and the add-into-collection variant give a cleaner appearance to descriptive parts of an animation code, like in PSCL [2006].

Another extension of AL is anonymous classes. For example, suppose that the user wants to declare a new class derived from `Scene`, override the script method `run()`, and starts a single instance, i.e., a singleton, of the class. The code in L is:

```
class MyScene: Scene
{
  protected:
    void run()
    {
      // create some actors
      // starts some scripts and/or actions
      ...
    }
}

(new MyScene()).start();
```

If only one instance of `MyScene` is created, then the explicit declaration of the class can be avoided. The following AL code creates and starts a singleton of an anonymous class derived from `Scene`:

```
new Scene() class
{
  protected:
    void run()
    {
      // create some actors
      // starts some scripts and/or actions
      ...
    }
}.start();
```

In the example above, the keyword **class** after the **new** expression denotes that a new instance of a class derived from Scene will be created. The anonymous class body follows the keyword **class**. The message start() is then sent to the singleton. The syntax is:

```
AnonymousNewExpression:
  new Name ( ExpresionList? ) class ClassBody
```

In addition, AL declares the following keywords:

- **run**: when used in the body of a class derived from Script denotes the header of the method run().
- **init**, **update**, and **finalize**: when used in the body of a class derived from Action denote the headers of the methods init(), update(), and finalize(), respectively.

If an action has to execute some activities at given instants of its lifetime, a **switch_time** statement can be used inside the **update** block, as illustrated below.

```
class MyAction: Action
{
  constructor(float lifetime);

  init
  {
    // start code comes here
  }
  update
  {
    switch_time(time)
    {
      from 0 to 2000:
       // do anything
      at 5000:
       // do anything
      from 1000:
       // do anything
      to 7000:
       // do anything
      from 1000 for 6000:
       // do anything
      for lifetime:
       // do anything
    }
  }
  finalize
  {
    // exit code comes here
  }
}
```

A **switch_time** statement takes as argument a float expression, typically the local time of the action or the scene total time. In the statement block, code can be associated to time intervals which are specified by **at**, **from-to**, **from**, **to**, **from-for**, and **for** conditions. The AMV executes the code of all conditions that are satisfied at the current updated cycle.

To create and start the MyAction one writes:

```
/* The MyAction constructor is invoked.
   The argument is the action lifetime. */
Action action = new MyAction(10000);
// The MyAction init block is invoked.
// The update block is executed at each tick.
action.start();
```

As an alternative, a **start** expression can be used to create and immediately start a sequencer:

```
Action action = start MyAction(10000);
```

4.4 Examples

This section is concluded by presenting two examples that illustrates how to create very simple scenes, scripts, and actions. The first one is a scene whose initial state is defined by a stack of box-shaped actors on a ground plane and a sphere on the stack. When the scene is started, the gravity acts and the sphere falls over the stack. Next, the script shoots twenty new spheres in the direction of projection each two seconds. In parallel, an action is started to rotate the camera around the scene and to create a new stack of boxes at a given instant. Figure 5 shows some frames.

```
// Add a ground plane into scene.
void createGroundPlane(Scene scene)
{
  ...
}
// Add a stack of boxes into scene.
void createBoxStack(Scene scene)
{
  ...
}
// This is the animation main function.
void main()
{
  // Create a scene and start its script.
  start Scene() class
  {
    /*
     * Create an actor whose body is a sphere.
     * Since the actor does not have a model,
     * its body is used for rendering.
     */
    Actor createSphere(Vector3D position)
    {
      Actor sphere = new Actor()
      {
        body = new RigidBody()
        {
          shapes
          {
            new SphereShape()
            {
              globalPosition = position;
              radius = 1;
            };
          };
        };
      };

      actors.add(sphere);
      return sphere;
    }

    // This is the script.
    run
    {
      // Create initial actors (Figure 5(a)).
      createGroundPlane(this);
      createBoxStack(this);
      createSphere(<0,6,0>);

      // Start an action to move the camera.
      start Action(this, 42000) class
      {
        Scene s;

        constructor(Scene s, float lifetime):
          Action(lifetime)
        {
          this.s = s;
        }

        update
        {
          // Rotate the camera continuously.
          s.camera.azimuth(0.5);
          /*
           * The switch_time statement below
           * is not really necessary and can
           * be replaced with an if statement.
           */
```

```
        switch_time(time)
        {
          /*
           * Create a new stack at time 32s
           * (Figure 5(e)).
           */
          at 32000:
            createBoxStack(s);
        }
      }
    };

    for (int ball = 0; ball < 20; ++ball)
    {
      waitFor(2000);
      createSphere(s.camera.position)
      {
        linearVelocity = s.camera.DOP * 60;
      }
    }
  }
};
}
```

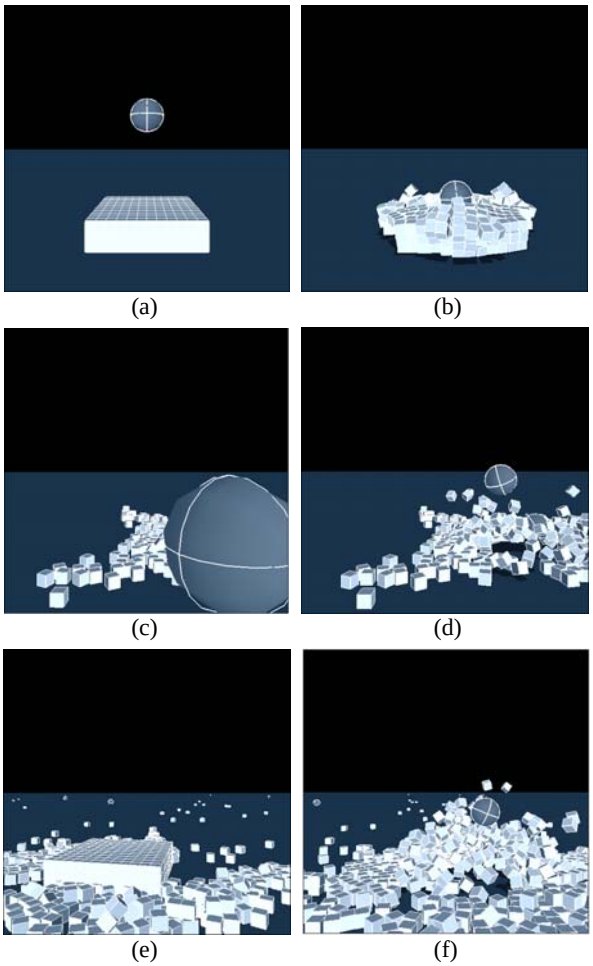

Figure 5: Frames of example 1.

The next example demonstrates how to use the contact report event. The class `CheckContact` below defines a script that, once started, waits until a contact between `a1` and `a2` occurs.

```
class CheckContact: Script
{
  public:
    Actor a1;
    Actor a2;

    constructor(Actor a1, Actor a2)
    {
      this.a1 = a1; this.a2 = a2;
    }

    run
    {
      ContactReport r;

      for (r = ContactReport::getInstance();;)
      {
        waitFor(r);
        for (Contact c: r.contacts)
          if (c.isBetween(a1, a2))
            return;
      }
    }
}
```

The script of the scene is very simple. It creates forty times a sphere and a box, applies a force on the sphere in order to put it in collision route with the cube, and waits for the contact. In parallel, an action moves the camera in the direction of the positive z-axis. Figure 6 shows some frames.

```
// This is the animation main function.
void main()
{
  /*
   * Create a scene and start its script.
   * The scene lifetime is 42s.
   */
  start Scene(42000) class
  {
    constructor(float totalTime):
      Scene(totalTime)
    {}

    // Add a sphere into scene.
    Actor createSphere(Vector3D position)
    {
      ...
    }
    // Add a box into scene.
    Actor createBox(Vector3D position)
    {
      ...
    }

    // This is the script.
    run
    {
      // Create the scene ground.
      createGroundPlane(this);
      // Start an action to move the camera.
      start Action(camera) class
      {
        Camera c;

        constructor(Camera c)
        {
          this.c = c;
        }

        update
        {
          c.pan(<0,0,10>);
        }
      };

      float z = 0;

      for (int i = 0; i <= 40; i++, z += 3)
      {
        // Create a sphere and a box.
        Actor s = createSphere(<2,0,z>);
        Actor b = createBox(<-2,0,z>);

        // Apply a force on the sphere.
        s.body.addForce(<-550,0,0>);
        /*
         * Wait for a contact between the
         * sphere and the box and repeat.
         */
        waitFor(start CheckContact(s, b));
      }
    }
  };
}
```

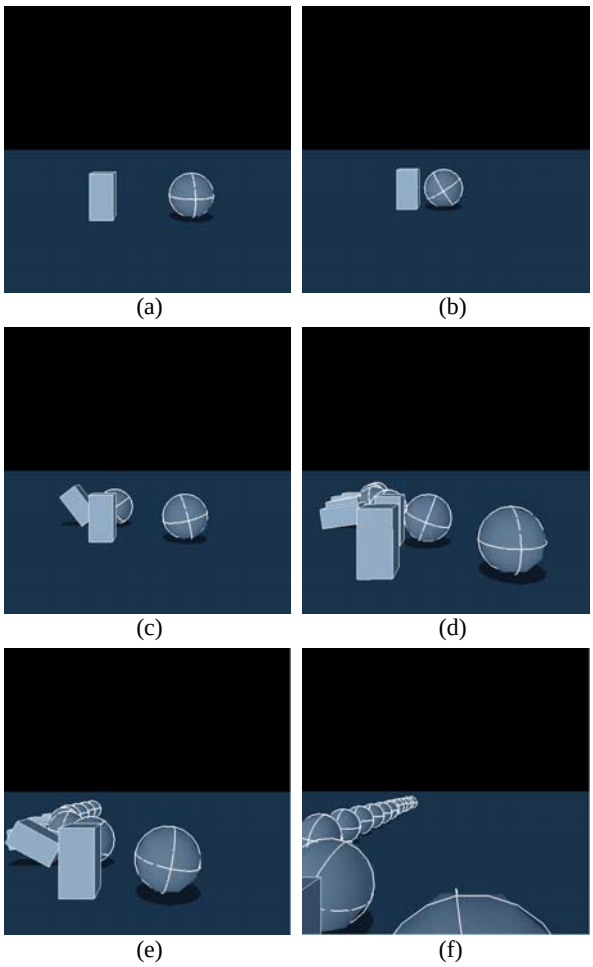

Figure 6: Frames of example 2.

5. Executing an Animation

The execution of an animation involves the collaboration of all components of the AVM, as summarized in this section.

The controller is the component responsible for orchestrating the steps of the execution. For that, it owns a set of script context queues and action queues. An element of a script context queue contains a reference to an active (i.e., `RUNNING` or `WAITING`) script, and a `Context` object related to the method `run()` of the script. A context is a structure with all information is needed to the LVM to be able to resume the execution of a function from a specific point in the object code. This includes the address of the next instruction to be executed and a reference to the function stack frame, among others. The controller has two context script queues:

- `RSQ`, which has an element for each `RUNNING` script started in the animation;
- `WSQ`, which has an element for each `WAITING` script. An element of `WSQ` also maintains the remainder sleeping time and a list of references to the synchronization objects for which a script is `WAITING` for.

An element of an action queue contains a reference to an action. There are three action queues: `IAQ`, `UAQ`, and `FAQ`, which have an element for each `INITIATED`, `RUNNING`, and `TERMINATED` actions in the animation, respectively. In addition, the controller owns a list `SEL` of references to the signaled synchronization objects at the current tick. The steps of execution of an animation are outlined below.

Step 0. The AVM loads the object animation file and looks for the main function. If this one is not found, the AVM throws an exception that aborts the execution of the program. Otherwise, the AVM asks to LVM to begin the execution of the main function byte-code.

Step 1. The execution proceeds until the method `start()` of a scene is invoked. The first instruction generated by AL compiler for every method that starts a script is `halt`, which is interpreted by the LVM as an order to suspend the execution of the current function. Since a `Scene` is a `Script`, the LVM stops. Next, the controller creates a new script context queue element containing a reference to the started scene and the LVM current context, and puts it into `RSQ`. The scene becomes the current scene and the current tick is set to zero. If another type of sequencer except a `Scene` is first started, then the AVM throws an exception that aborts the execution of the program. At least a scene should be started from the main function.

Step 2. While the total time of the current scene is not zero and the context script and action queues are not empty, the controller carries out the update cycle for the current tick, steps 3 to 9.

Step 3. For each element `S` into `RSQ`, the controller asks to LVM to resume from the corresponding script context. As a consequence, the LVM executes the method `run()` of the corresponding `RUNNING` script, which can start others scripts and actions and signal events. If a new script is started, then the LVM halts and a new element for the new script is created and put into `RSQ`. If a new action is started, `start()` of `Action` puts into `IAQ` a new action queue element containing a reference to the new action. If an event is signaled, `setSignaled()` of `Event` adds into `SEL` a new element containing a reference to the event. The LVM continues the execution of `run()`until:

- the method returns or `exit()` is invoked. In this case, the script is `TERMINATED` and the element `S` is removed from `RSQ`;
- a method `waitFor()` is invoked. As in the case of a method `start()`, the first instruction of a `waitFor()` halts the LVM. Next, the controller removes `S` from `RSQ` and puts into `WSQ` a new element containing a reference to the script, the current context, and, as given by the arguments passed to `waitFor()`, the timeout and the list of references to the synchronization objects for which the script will be `WAITING` for.

Step 4. For each element `S` into `WSQ`, the controller verifies if the timeout is zero or the synchronization objects for which the corresponding script is `WAITING`

for are in SEL. If it is true, then S is removed from WSQ and a new element for the script is put into WSQ, i.e., the script wakeups and turns back to the RUNNING state. Otherwise, the timeout is decremented.

Step 5. For each element A into IAQ, the controller asks to the LVM to execute the method init() for the corresponding action. A is moved from IAQ to UAQ.

Step 6. For each element A into UAQ, the controller verifies if the lifetime of the corresponding action is not zero. If it is true, the controller asks to the LVM to execute the method update() for the action, and updates the properties time and lifetime. Otherwise, or if exit() was invoked from update(), A is moved from UAQ to FAQ.

Step 7. For each element A into FAQ, the controller asks to the LVM to execute the method finalize() for the corresponding action. A is removed from FAQ.

Step 8. The controller clears SEL and asks to the physics engine to perform the simulation for the time step corresponding to the current tick. If any contacts among rigid bodies are detected, the controller sets the state of ContactReport as signaled and adds a new element for the event into SEL.

Step 9. The total time of the current scene is updated and the current tick is incremented. If it is a multiple of the render resolution, the controller asks to renderer to make a scene frame, which, depending on the application, can be immediately exhibited or sent to the animation file linker.

6. Concluding Remarks

This paper presents a new programming environment for visualization of dynamic simulation of rigid bodies. An animation is specified in an animation language (AL) in terms of objects that represent a scene and its components and sequencers and events. A sequencer is an control object responsible to changes scene states over time. A script is a sequencer representing a linear sequence of activities that must be executed just the once, from begin to end. Scripts can synchronize each other and wait for events to be signaled. An action represents a sequence of activities that must be continuously executed each update cycle.

An animation is executed by an animation virtual machine (AVM). The physics engine is the component of the AVM responsible for dynamic simulation. The version of the animation system addressed in the paper uses the AGEIA PhysX for rigid body simulation. Due to complexity of the system and space limitations was not possible to give details how the PhysX is integrated to AVM.

Possible extensions of the system include: a computer-human interface for graphical description of animations; extension of the language to support behavior rules for characters; just-in-time translation of part of the bytecode of an animation to native code in order to improve execution speed; use of graphics processing unit (GPU) for physics. Currently the authors are developing a new physics engine to be integrated to the programming environment for real-time, interactive simulation of rigid and elastic bodies.

References

AGEIA, 2006. *PhysX SDK documentation* [online]. Available from: www.ageia.com/pdf/PhysicsSDK.pdf [Accessed 15 March 2006].

BRAUN M., AND FORMELLA, A., 1995. AniGraph: a data structure for computer animation. *In Proceedings of Computer Animation, April 1995.* IEEE Computer Society Press, 126-137.

DEVILLERS, F. AND DONIKIAN, S., 2003. A scenario language to orchestrate virtual world evolution. *In: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, San Diego*. Eurographics Association, 265-275.

FORMELLA, A. AND KIEFER, P.P., 1996. AniLan: an animation language. In data structure for computer animation. *In Proceedings of Computer Animation, June 1996.* IEEE Computer Society Press, 184-189.

IERUSALIMSCHY, R. 2006. *Programming in Lua*. Lua.org,

OLIVEIRA, L.L., 2006. *Um sistema de animação baseado em dinâmica de corpos rígidos articulados.* Dissertação de mestrado, UFMS (in Portuguese).

PERLIN, K. AND GOLDBERG, A., 1996. Improv: A system for scripting interactive actors in virtual worlds. *In Proceedings of SIGGRAPH 96.* ACM Press, 205-216.

PSCL, 2006. *The physics script language* [online]. Available from: www.physicstools.org/docs/pscl [Accessed 17 July 2006].

REINOLDS, C., 1982. Computer animation with scripts and actors. *In Proceedings of the 9th annual conference on computer graphics and interactive techniques, Boston, 1982*. ACM Press, 289-296.

SNYDER, J.M., WOODBURY, A.R., FLEISCHER, K., CURRIN, B. AND BARR. A.H., 1993. Interval method for multi-point collision between time-dependent curved surfaces. *In Proceedings of SIGGRAPH 93*. ACM Press, 321-334.

ZELEZNIK, R.C., 1991. An object-oriented framework for the integration of interactive animation techniques. *In Proceedings of the 18th annual conference on computer graphics and interactive techniques, 1991*. ACM Press, 105-112.