Interner Bericht

The fundamental model of Virtual Reality for purposes of Simulation

Alexander Keller

~~224~~/92
225

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-6750 Kaiserslautern

The fundamental model of Virtual Reality for purposes of Simulation

Alexander Keller
225/92

Universität Kaiserslautern
AG Computergraphik
Postfach 30 49
6750 Kaiserslautern

Juli 1992

Herausgeber: AG Graphische Datenverarbeitung und Computergeometrie
Leiter: Professor Dr. H. Hagen

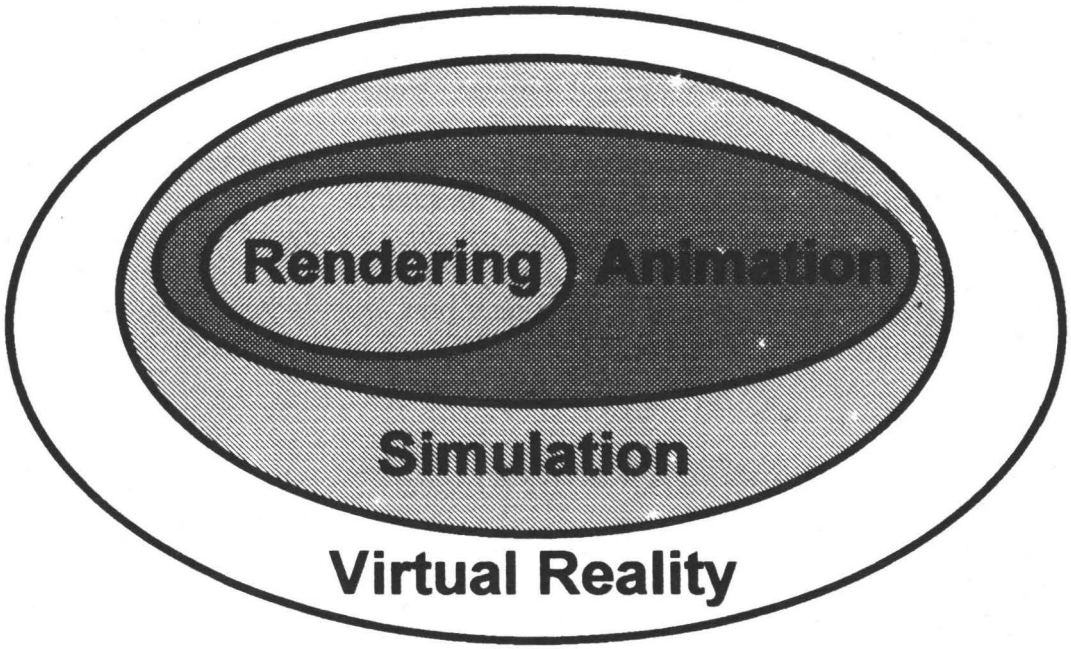
Technical Report

Alexander Keller
June, 12 1992
University of Kaiserslautern
Department of Computer Science
AG Hagen

The fundamental model of Virtual Reality for purposes of Simulation

Introduction

Virtual Reality (VR) is to be seen as the superset of simulation and animation. Visualization is done by rendering. The fundamental model of VR accounts for all phenomenons to be modelled with help of a computer. Examples range from simple dragging actions with a mouse device to the complex simulation of physically based animation.



1. Set structure from rendering to Virtual Reality

Aim

The aim is a model for unification of all animation models used so far. This concept includes realtime interaction as well as high quality rendering of film sequences. Another goal is an easy use by simple construction of own solutions.

Capabilities

The system is capable of running in two modes. In realtime mode strict time requirements have to be met, that is a stepper unit has to complete all simulation and interaction activities in a definite time interval of time and then has to provide a geometric scene description to the visualizer module. The geometric scene description may be passed to several visualizer modules, so that many people are able to consume the visualization from different points of view.

In film mode, the stepper unit simulates without interaction, since no realtime requirements can be satisfied. The scene description is waited for to be rendered completely by a high quality renderer, which then causes the stepper module to simulate the next interval of time. Since the stepper unit is triggered, only one visualizer module can be used.

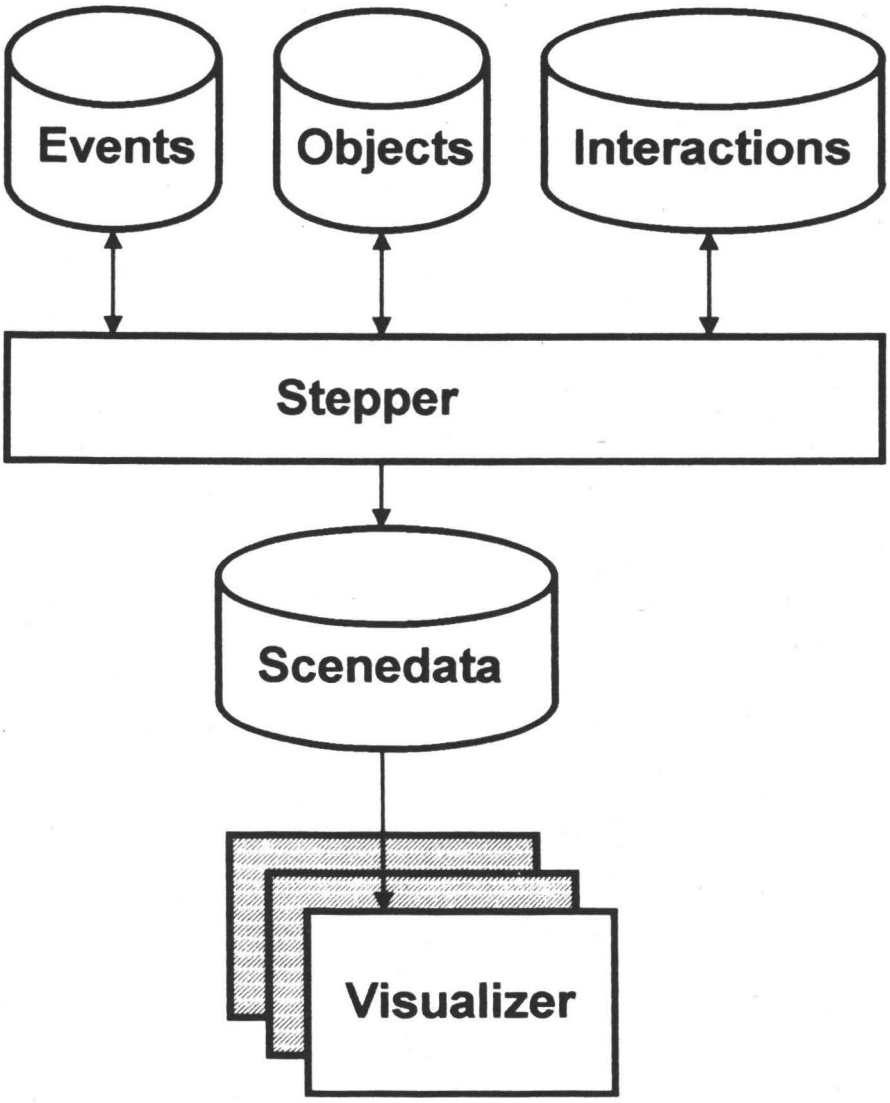
Overall structure

The visualizer modules are separated completely from the stepper unit controlling the simulation. This is an advantage, since the visualizer can be exchanged in an easy way and visualization is separated from simulation. This way graphics and mathematical models do not interfere. The stepper module and the visualizer also easily can be distributed over separate computers. Here, one problem is the network synchronization of several visualizers, for examples one for the right and left eye of a Virtual-Reality-headset, because mall differences in synchronization may desturb human reception of the scene displayed.

The event resource always contains the event which is next to happen. It also contains planned events which should take place at a specified moment. This way scripted or static animation and dynamic simulation are combined. The system then is capable of both.

The interactions resource contains methods for simulating interaction of the objects. This interactions also include constraints.

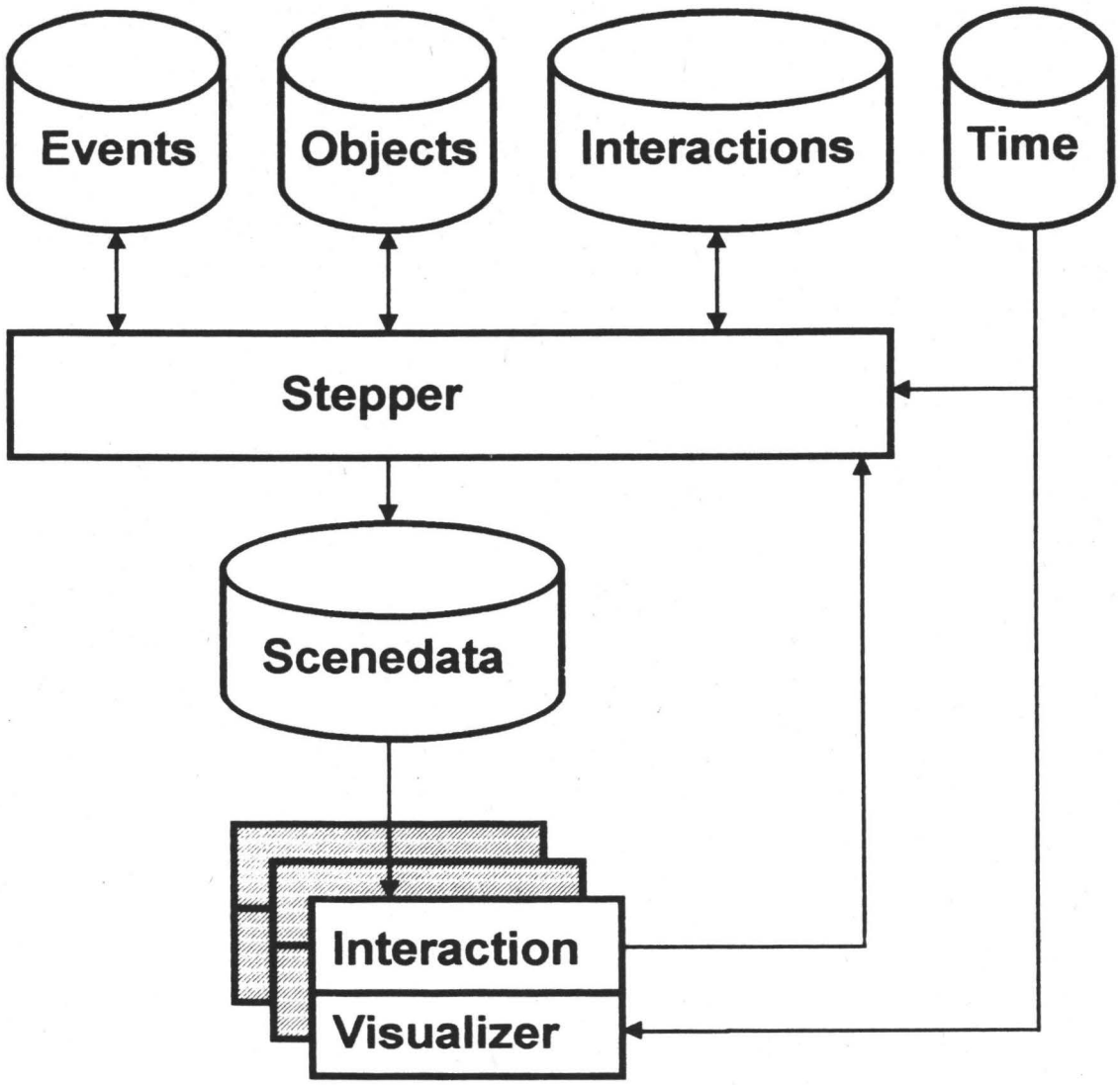
Filming and editing are separated strictly. Each run of the stepper unit from a start time up to an end time produces one piece of film or a scene. Blending and cutting should be done afterwards. Although it principally is possible to include such actions in the model, this should be avoided to keep structures and use easy.



2. The fundamental model of Virtual Reality

Realtime model

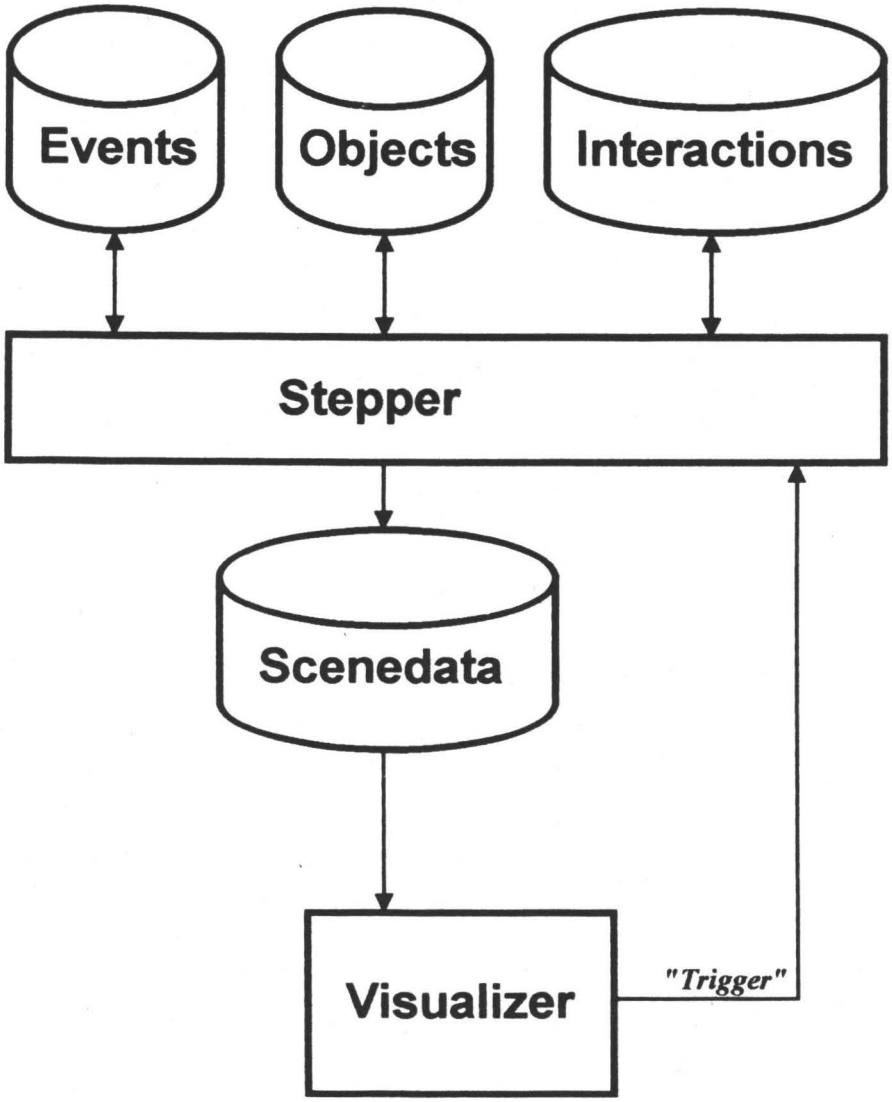
In realtime mode the stepper module is periodically triggered by a time device. The visualizer modules are triggered by the same clock and in consequence are running synchronously to the stepper unit. Live interaction with the stepper unit is done via an interaction unit in the visualizers. So each visualizer can determine its own viewpoint by the space position and orientation of a VR-headset. In addition it can propagate this data to the stepper modules for interaction with the objects.



3. Realtime system structure

Changes for triggered rendering

In triggered mode realtime requirements are ignored. That way a high quality rendering can be done for film production. When one frame is completed, the stepper unit is triggered updates all objects up to the next moment where a picture is to be taken. As before the system stops if the time interval of simulation is vanished.



4. Triggered system structure

Datastructures

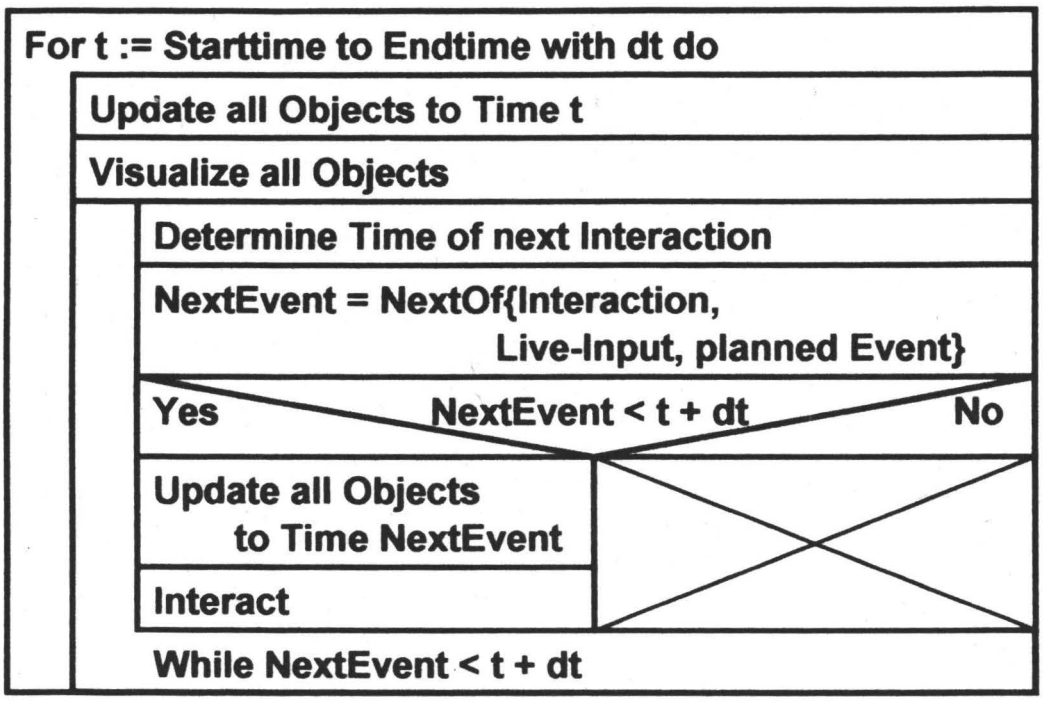
The concept of expandable records with attached routines (objects) is used extensively and guaranties the easy use of the simple application procedure interface (API).

The object ***OBJECT*** is the base type of all objects used in the simulator. It keeps its state by a time t. An object is updated to a new time using the ***UPDATE*** method. This method completely describes the behaviour of an animated object, e.g. movement. An instanciation of its geometric shape is done via a ***VISUALIZE*** call.

An ***INTERACTION*** object describes the kind of interaction of the objects listed in the "objects"- list. One routine (***TimeOfNextEvent***) supplies the moment of the next interaction starting at the current time t. When this interaction is selected next, it can be executed calling ***INTERACT***, which changes the objects' states according to the interaction.

The object ***SIMULATION*** is derived of an ***INTERACTION***. It so consists of a list of all objects in the system, a current time t, a list of all interactions and planned

events. The procedure ***STEP*** is called with the start and end time of the simulation to be run and a parameter indicating the interval of creating a geometric scene description. To produce a simulation the procedure ***STEP*** is called with the desired parameters.



5. Pseudocode of the *STEP* procedure

```
//****************************************
// Module: Simulator
// Date  : 26.04.1992
//****************************************

#include "C:/C/Lists.cpp"
#include "C:/C/BSP.cpp"

typedef double Time;

const Time Infinite = 1e10;

class Object
{
public:
      Time TimeOfObject;

      Object(Time t)
      {
            Update(t);
      }

      ~Object()
      {}
```

```
        virtual void Visualize()
        {}

        virtual void Update(Time t)
        {
                TimeOfObject = t;
        }
};

class ObjectNode : public Node, public Object
{
public:
        ObjectNode()
        {
        }
};

class Interaction
{
public:
        Time t;
        List Objects;

        Interaction()
        {
                Objects = NULL;
        }

        ~Interaction()
        {}

        virtual Time TimeOfNextEvent()
        {
                return(0.0);
        }

        virtual void Interact()
        {}
};

class InteractionNode : public Node, public Interaction
{
};
```

```
class Simulation : public Interaction
{
public:
      List Interactions, Events;

      void Step(Time Start, Time End, Time TimeStep)
      {
            Time time, NextEvent;
            Node * Walk;
            Node * Check, * NextInteraction;


            for (t = Start; t < End; t += TimeStep)
            {
                  Walk = Objects.Nodes;
                  while (Walk)
                  {
                        ((ObjectNode *) Walk)->Update(t);
                        ((ObjectNode *) Walk)->Visualize();
                        Walk = Walk->Next;
                  }

                  NextEvent = t + TimeStep;
                  do
                  {
                        Check = Interactions.Nodes;
                        while (Check)
                        {
                              if ((time = ((InteractionNode *)
                                    Check)->TimeOfNextEvent()) <
                                          NextEvent)
                              {
                                    NextEvent = time;
                                    NextInteraction = Check;
                              }
                              Check = Check->Next;
                        }

                        if (Events.Nodes)
                              if (((InteractionNode *)
                                    Events.Nodes)->t < NextEvent)
                              {
                                    NextEvent = ((InteractionNode
                                          *) Events.Nodes)->t;
                                    NextInteraction = Events.Nodes;
                                    Events.Nodes = Events.Nodes
                                                ->Next;
                                    // delete node afterwards
                              }
```

```
                        if (NextEvent < t + TimeStep)
                        {
                              Walk = Objects.Nodes;
                              while (Walk)
                              {
                                    ((ObjectNode *) Walk)
                                          ->Update(NextEvent);
                                    Walk = Walk->Next;
                              }
                              ((InteractionNode *)
                                    NextInteraction)->Interact();
                        }

                  } while (NextEvent < t + TimeStep);
            }
      }
};

//***************** Example pseudo code
class Sphere : public Object
{
public:
      Vector Position, Speed;
      real Mass, Radius;

      Sphere(Vector p, Vector v, real m, real r, Time t)
      {
            Object :: Update(t);
            Position = p;
            Speed = v;
            Mass = m;
            Radius = r;
      }

      virtual void Visualize()
      {
            // draw mesh or simple point
      }

      virtual void Update(Time t)
      {
            Position += (t - TimeOfObject) * Speed;
            Object :: Update(t);
      }
};

class Wall : public Interaction
{
public:
      Wall()
      {
            Interaction::Interaction();
      }
```

```
        ~Wall()
        {}

        virtual Time TimeOfNextEvent()
        {
              // determine time of collision with wall
        }

        virtual void Interact()
        {
              // reflect sphere at wall
        }
};
```

6. C++ fragment of an implementation (only base types)

The Visualizer

The visualizer module reads out an geometric picture-description supplied by the stepper unit. Its viewpoint can be determined by live input or vectors provided by the stepper. In realtime mode synchronization is done by a timer device and multiple visualizers for many users or VR-headsets can be used, opposite to the filming mode, where the visualizer triggers the stepper module and so only one visualizer (in this case a high quality renderer) can be used. The existence of visualizers support the complete separation of image display and mathematical model (simulation).

Caching techniques

This techniques are nothing new, but have to be used consequently in order to reduce the complexity of the simulation activities. These techniques do apply most to the data transfer between the stepper and visualizer modules. Since this communication can run over a network in realtime mode, the amount of data has to be reduced as to be able to display complex environements. So objects which do not change need not to be transferred.

Problems

One problem of this model is, that so far it cannot model the phenomenon of motion blur. For solving this problem a lot of information about ths objects' motions have to be passed to the visualizer module.

Another problem is the consistency of the resources. Events may delete objects which are registered in some interaction. That way access violations can occur.

Examples

Mouse dragging
Particle Chamber
Scripted animation

<<< to be expanded >>>