〔非公開〕

ＴＲ－Ｍ－００１３

Optimizing and Integrating Static and Dynamic Objects and A Direct-Interactive Dynamic Form Synthesizer

ダーク　ライナース
Dirk　REINERS

１９９６．１１．６

ＡＴＲ知能映像通信研究所

Optimizing and Integrating Static and Dynamic Objects

and

A Direct-Interactive Dynamic Form Synthesizer

by
Dirk Reiners
ATR-MIC
May - Oct 1996

Contents

Chapter 1

Introduction

This report describes the work that I did during my 6 months stay at the Advanced Telecommunication Research Laboratories Media Integration and Communication Labs (ATR-MIC) in Kyoto, Japan. I was sent there from the Fraunhofer Institute for Computer Graphics in Darmstadt, Germany, where I was working in the department for Visualisation and Virtual Reality pursuing a PhD in computer science.

The purpose of my visit was twofold. On the one hand, I was asked to use my expertise in high-quality, high-speed rendering to optimize and integrate different pieces of interactive computer art together with the artists Christa Sommerer and Laurent Mignonneau. As a result of this work the piece 'MIC Exploration Space II - The Garden' described in chapter 3 was created and exhibited very successfully at the ACM's annual computer graphics conference SIGGRAPH in New Orleans from Aug 4th to Aug 9th, 1996, and in an enhanced incarnation at the annual ATR Open House on Nov 7th/8th, 1996.

The second aspect was to jointly develop their artistic ideas in a scientific direction. This was done in the form of a new paradigm for the creation of computer graphics forms that actively encourages the creation of direct-interactive objects, which mediate a very organic and natural interaction, described in chapter 4.

Chapter 2

Acknowledgments

This whole stay was only possible through the support of ATR-MIC's director, Dr. Ryohei Nakatsu, for which I thank him deeply. Furthermore I thank my department head in Darmstadt, Mr. Stefan Mueller, for allowing me to leave my duties for half a year and letting me return. The main force behind all this and the main force driving me through it were my colleagues, friends and supervisors Christa Sommerer and Laurent Mignonneau. They deserve credit for the creative aspects of this work, and I thank them for inspiring me and making me do things I didn't deem possible.

Always important for every kind of stay in a foreign country are the people that take care of the new and unusual aspects of organization and bureaucracy. I thank my department head, Dr. Tsutomu Miyasato, and the planning section of MIC, first and formost Mrs. Kana Itoh for all their support through the numerous problems and questions that awaited me.

And after all the work is done, there is sometimes still a little life left, and the people who helped me fill this piece deserve their place in this work, as they kept me running and active. First and foremost Christa and Laurent as my primary backup, but also Sidney Fels, Armin Bruederlin, Silvio Esser, Kuntal Sengupta and Gert van Tonder, for all the ideas and exchange of ideas between a group of people from 4 continents.

Chapter 3

Optimizing and Integrating Static and Dynamic Objects

Sommerer and Mignonneau have a history of very successful artistic projects in interactive computer art. Two basic types of works can be distinguished: static and dynamic works.

Static works are works in which iteratively a beautiful and very detailed image of a world is created. Examples for this kind of work are Transplant (1995), Anthroposcope (1993) and Transplant2 (1996), which is the basis for the MIC Exploration Space.

In the dynamic works creatures are created according to nature-based rules and they have a chance to move and live in their environment. The environment in these cases in relatively simple (a pool in A-Volve (1994), a black space in GenMa (1996)).

The reason for this is the limited capability even of today's most powerful graphics computers. The static worlds have a very high complexity easily exceeding 10 million polygons and can not be rerendered in real time for every move a creature makes. My task was to find a way to unite these two apparently ununifiable worlds of static and dynamic objects.

The result of this part of my work was the 'MIC Exploration Space II - The Garden' piece which was exhibited very successfully at the ACM's Annual Computer Graphics Conference SIGGRAPH as a part of the art show 'The Bridge', from Aug 4th to Aug 9th, 1996, and in an enhanced incarnation at the annual ATR Open House on Nov 7th/8th, 1996.

To understand the topics involved a little introduction to the setup is warranted (s. fig 3.1).

The system includes two users, each standing on a white carpet in front of a video projection screen and with a backlit screen behind. A camera on top of the projection screen takes the image of each person and this image is used

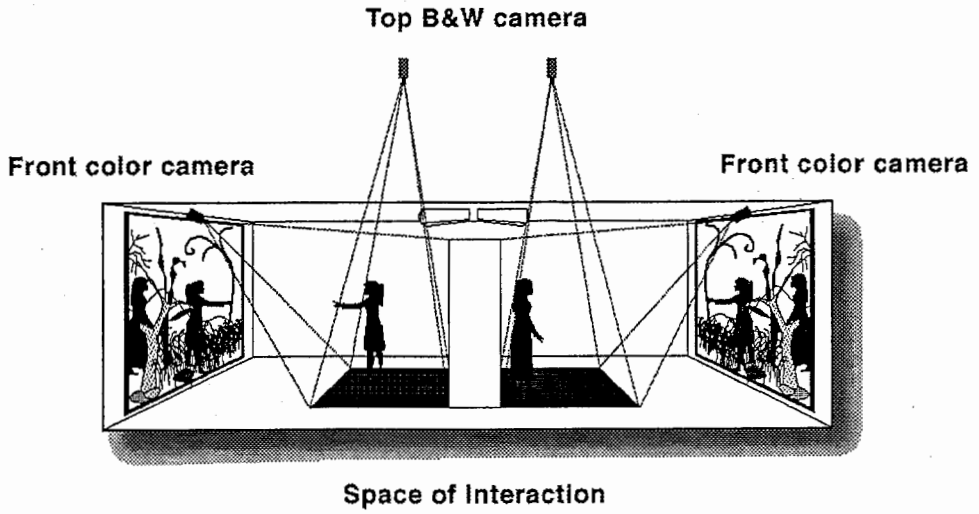

Figure 3.1: Setup of the MIC Exploration Space system ((c) Sommerer&Mignonneau)

to analyze the position and actions of the persons. Furthermore the images of both users are luminance-keyed and sent through Sommerer/Mignonneau's patented 3D-Key system, which extracts the position of the person from the top camera's image, to be integrated with each other and the computer generated image. Fig. 3.2 shows a resulting image. The persons are pixel-accurately keyed into the computer-generated scene and can be in front or behind each other, but this only as a whole, as the depth of the person is assumed to be constant.For the static parts to make sense at all the observer's viewpoint is not changed.

3.1 Basis

As a basis for the integration Sommerer-san and Mignonneau-san created two new systems.

The first is a flower generation system (fig.3.3). The flowers have a stem surrounded by a bunch of leaves and a group of blossoms at the top. There are a number of different kinds of blossom designs and all parts have a random variation built in, so that no two flowers look alike. Furthermore the shape can be influenced by the form and size of the user of the system, so that every person gets his own style of flowers. They are the static part of the system.

The second part is an insect generation system (fig.3.4).There are two

Figure 3.2: The combined and keyed image ((c) Sommerer&Mignonneau)

different kinds of insects generated:

- fly- or bee-like insects with a short body and one pair of small wings
- dragonfly-like insects with a long body and two pairs of long and thin wings

The insects are built using a random variation of different parameters for body and wing shape and size. The shape and size of the body is used to calculate the flight behavior of the insect, so that no two insects look alike or fly alike. These are the dynamic objects in the system.

My task was to optimize the existing programs, which from a GL standpoint were already pretty good, and integrate them into one system.

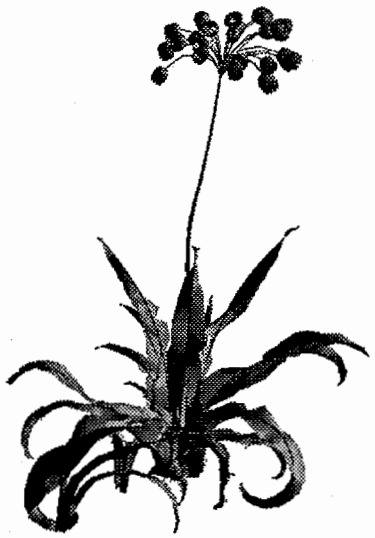

Figure 3.3: An example output of the flower generation system

3.2 Optimizing

As the whole system as to be shown on SIGGRAPH had to be optimized, there were two distinct parts to work on: graphics and the main program.

3.2.1 Graphics

The basis programs, as given to me by Mignonneau-san, already made effective use of the underlying graphics hardware.

He used a low number of polygons to generate high quality output by calculating good normals and using smooth shading. The lighting model was chosen to be computationally cheap, but with good results (non-local lighting with only two directional light sources). The polygons were send using triangle strips, so no unnecessary vertices had to be transformed.

The target machine was an SGI infinite Reality, so this suggested some changes in the structure. The infinite Reality is Silicon Graphics latest flagship and has unprecedented calculation power (the combined peak performance of the iR's geometry subsystem is 2 GFLOPS), much higher than the main processor's floating point power, so it made sense to offload some work onto the graphics board. This mainly included the normalization of the normals used for lighting, which requires a square root and a division, two extremely costly operations, which for highest graphics performance should be avoided, but in our case the work had to be done anyway and was not

Figure 3.4: Some insects generated by the insect system

going to be repeated (as the plants are only rendered once), so it made sense to utilize the graphics hardware for it.

The second consequence of an infinite Reality being the machine was to allow me to mix OpenGl and IrisGL. Officially this is not possible, and it is not possible for most SGI machines. For the newest machines like Impact, infinite Reality and O2, IrisGL is set as a layer on top of OpenGL and the machines no longer have special microcode for IrisGL. Thus if one is very careful about the OpenGL state it is possible to mix the two. This proved to be very important, as some of the features needed for integration (s. sec.3.3) did not work at the maximum performance or not at all in the IrisGL emulation the infinite Reality employed.

A different strategy was used for the insects. As they are constructed from essentially rigid parts (body and wings) which only move as a whole they could be put into GL objects, thus being optimized by the graphics library to be rendered at the full processor speed. The same could have been done by using my own, optimized rendering loops from Y, but that would have meant to completely change the program and was considered to not be worth the effort.

3.2.2 Main program

The second part to be optimized was the main program.

The whole system would facilitate two users and consist of three machines:

- an SGI Indy R5000 running pFinder for the first person
- an SGI Indy R5000 running pFinder for the second person
- an SGI Onyx for the simulation and the image generation

3.2.2.1 pFinder

The pFinder system was initially developed at MIT and has been customized and improved for the use in this installation by Roberto Lopez of ATR International. He also did the communication part between the Indys and the Onyx.

To do this communication he used the RPC mechanism, as it is well established, easy to use and automatically supported by the IRIX operating system. To prevent unnecessary waiting for data from pFinder (which can be expected at a rate of 20-25 Hz, while the main program was targeted to be run at 30 or 60Hz), non-blocking I/O using the select system call was used.

While the select call is indeed non-blocking, it can still take a lot of time, up to 7 ms or more. This might not seem to be a lot of a time for many problems, but for interactive graphics applications, which are targeted to an update rate of 60 Hz, it is. At 60 Hz a new image has to be generated every 16.6 ms, so if 7 ms are spent waiting, there is not enough time left to complete the image in time. This results in a stuttering movement on screen and is very disturbing.

The remedy was to put the communication task into a separate thread. This thread would sleep most of the time only to awake when new data from pFinder was received. This data is then transformed into the format needed by the main program and the main program is notified via a flag located in shared memory that new data has arrived. On the next pass through the main loop the main program would just switch a pointer to the new data and notify the pFinder thread, again via a shared memory flag, of that fact. Thus the next time new Finder data arrives, the former data can be overwritten and the memory reused. This forms a ping-pong buffer, an important construct for efficient multi-process data communication.

The result was, that the separate thread does not noticeably contribute to the system load ($< 1\%$ CPU load), and at the same time the communication part of the main program does not appear in the profile, so in the end the

communication with the external pFinder processes is effectively free, a very pleasing result.

3.2.2.2 Simulation and Graphics

The main loop of the program divided itself into two distinct parts: the simulation of plant growth or insect flight and the rendering of the simulated image.

These two tasks invited themselves to be distributed across multiple processors, as the demo machine was expected to have at least two processors, just as the development machine available to me at ATR.

Passing data between the two tasks was again done using the ping-pong buffer methodology described in 3.2.2.1. A complication of this process was the constraint imposed by the IrisGL, that only one thread can access a graphics window, so all graphics actions had to be done by the rendering thread. To keep as much flexibility as possible in using and extending the program without tying it too much into the one given situation a FIFO command queue was set up in shared memory between the two threads, so that the simulation thread could keep on simulating and acting without having to wait for the rendering thread. Furthermore this allowed to insert synchronization as an explicit command, allowing to reduce synchronization, which always is costly, to a bare minimum.

3.2.2.3 Results

The results of these combined efforts were encouraging. The pFinder communication turned out to be free, compared to a noticeable amount of time and possibly frame drops before the splitting.

The graphics results were less apparent, as the plants grow frameless and thus objective comparisons are difficult. Subjective comparisons did point to a noticeable speedup, which inclines it to be 1.5 or higher. For the insects, which have defined frame boundaries the speedup was measurable to the extent that about twice as many insects could be rendered at the same framerate.

3.3 Integrating

An important new target of this work was the integration of the static and the dynamic objects into a joined image.

The problem is how to do it. Rerendering the static objects is not possible, as they can easily contain more than 10,000,000 polygons. Even theoretically,

i.e. using benchmark numbers, this would take a whole second on an infinite Reality, which is not even remotely interactive.

But what is constant, no matter how complex the scene, is the number of pixel. Thus using an image based integration, if feasible, would allow the integration independent of scene complexity.

One additional complication is the threedimensional nature of the scenes that are handled here, i.e. the information needed to perform hidden surface removal has to be saved and restored, too, otherwise the insects would not correctly interact in depth with the plants. Luckily the hidden surface removal technique employed by the Silicon Graphics hardware is the Z-buffer and it can be accessed much like the color buffers. Thus copying of color and Z-buffer was chosen for the integration.

An open question is the place where the background copy of the image should be stored. The simplest solution would be to transfer the data into main memory and transfer it back from there. The drawback of this solution is the limited bandwidth of the main memory-graphics connection. A better and faster solution is to keep the data in the graphics subsystem itself, thus the bottleneck on the way to main memory is circumvented.

For the color buffers there are several alternatives where to store them, e.g. in the accumulation buffer. This is not the case for the depth buffer, though. Depth buffer data can only be copied to another depth buffer, thus an area with a depth buffer had to be found. The easiest and chosen solution was to just double the window height (s. fig 3.5) and use the free area as the backup store. As the output used is NTSC at a format of 646x486 there is enough space on a standard 1280x1024 screen to handle two copies of the image.

To reduce the necessary amount of copying the course of the usage of the program was split into two phases. In the first phase the plants would grow without any insects flying around. So in this phase no copy is necessary.

After a specific action by the user is executed (e.g. touch the flowers) a copy would be made . Then the insects would appear and fly through the plants. For every frame the background copy has to copied to the foreground, then the insects have to be rendered and using the Z-buffer are correctly integrated into the scene. Thus only one screen-screen copy has to be made per frame.

Of course it would be possible to have growing plants and flying insects at the same time. To do that one would have to restore the image, let the plants grow one cycle, store the image and then draw the insects. This idea was abandoned for different reasons. First it would have overwhelmed the user with action in the scene that he would hardly be able to control at the same time and second the second copy operation would be too expensive.

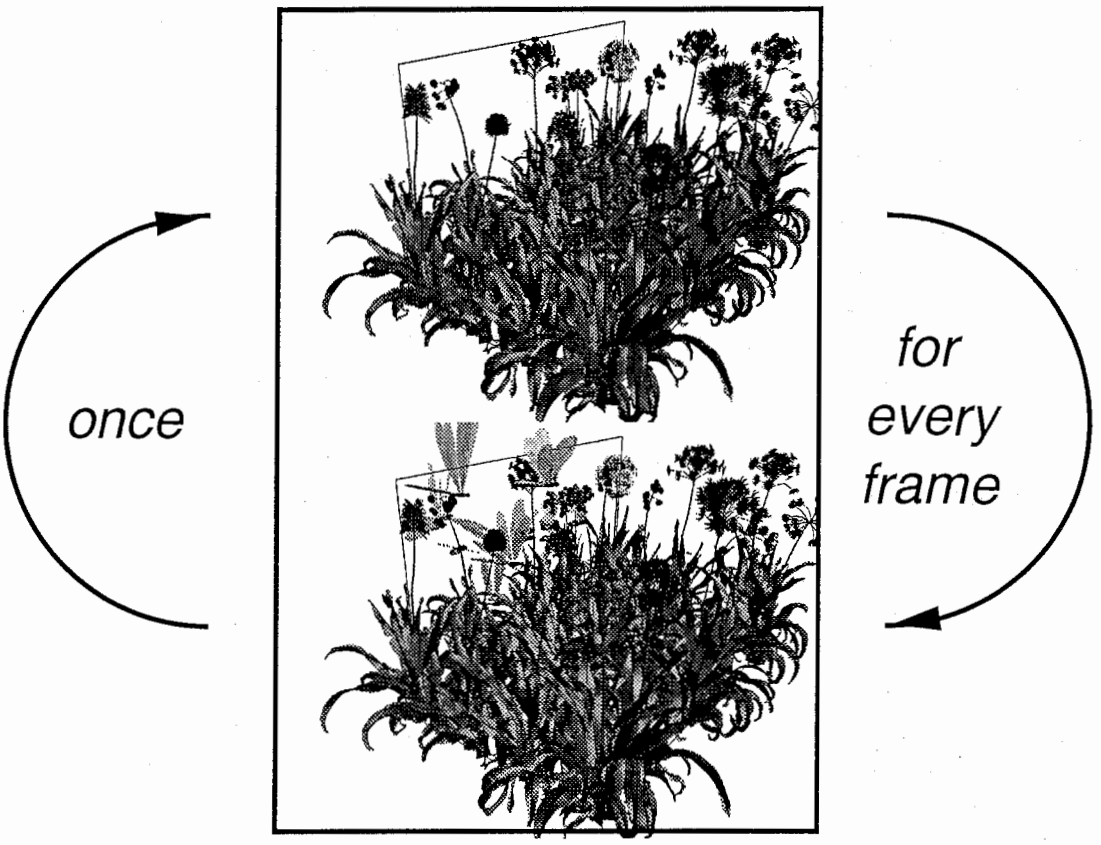

Figure 3.5: A double height window is used to store the background copy of the image

It turned out that the copy operation takes a significant amount of time. About 40% of the available rendering time are used up by the copy operation. For the new quality of the experience this can be accepted, as the infinite reality is fast enough to render interesting dynamic objects in the remaining 60%. The second copy however would have reduced the amount of time available for the dynamic objects to 20%, and that would have severely limited the expressive power of the dynamic objects, thus this solution was not used.

3.4 Insect Behavior

After the integration and optimization was done the basic behavior of the insect had to be refined to show the capabilities that pFinder offers.

For the first version shown at SIGGRAPH two basic interactions were

implemented. First after being born the insects would fly around the persons body and follow him whereever he went (for simplicity the male pronoun is throughout this paper). The distance that the insects deemed to be a save distance was influenced by the speed of the person's arm movements. If the person moved too vigorously the insects would find it too dangerous to stay close and would go to the other person.

The reason why the person's movements are dangerous for the insects is the second interaction. Insects that bump into the persons body or into his hand (determined from the body centroid's position or the hands' positions by using a screen-space aligned elliptical catch range) would stop their wings, fall to the floor and die.

To counter the diminishing number of insects in flight resulting from insects being caught they were given an opportunity to reproduce. Two insects getting close enough to each other would create an offspring. To prevent a stream of offspring from two close flying insects each pair was allowed to mate only once. And to prevent the created child to immediately create new children with its parents children were assumed to be immature for a limited amount of time.

The resulting interaction was very active, as a swarm of 50 insects flying around a person at 60 Hz creates quite a commotion on the screen. One problem was that the catching and dying of insects was very hard to notice, as they moved fast and the stopping of the wings would result in them falling down into the plants' leaves and disappear. It was interactive nonetheless, but some people did show signs of disbelief about the amount of information extracted from the camera image.

Thus for the ATR Open House 1996 the interaction model was changed.

The insects are not be as shy anymore. After they are born they still choose a person as a target and fly around him. But when he raises his hands they slow down and come close to the hands, as if being fed. And indeed they only reproduce while being fed.

They follow the person's hand whereever he moves, unless the movement is too fast, and only for a limited time. After that they accelerate and fly around the person's body again, for a limited time until the hands are raised again.

This interaction turned out to be much more direct, as the persons could directly notice their hand being detected and followed. By the time of this writing the Open House has not yet happened, but first tests with users show a very positive response.

3.5 Summary

A new integrated interactive computer graphics art piece was created. The basic elements were designed by the artists Christa Sommerer and Laurent Mignonneau. These were two separate programs to generate two different kinds of the aesthetic forms of plants and creatures they are known for. My task was to integrate the two parts graphically as well as programmatically together with a client library to access pFinder data and optimize the system to get the best possible performance.

The optimization consisted of slightly adapting the given programs to the target machine infinite Reality and splitting the program into three threads: pFinder communication, simulation and rendering, and connect the threads with a fast communication link. This was done using shared memory and ping-pong buffers.

The integration was done on an image basis, as the complexity of the scene prohibited a geometrical integration. The image integration is still expensive, but it is feasible and leaves enough room for expressive dynamic objects.

After the integration and optimization was done the existing basic flight model was enhanced to give a better feedback to the user and show the capabilities of the used pFinder software.

The result was exhibited to the public at two events, ran stable and fast and got generally only positive comments.

Chapter 4

A Direct-Interactive Dynamic Form Synthesizer

The second part of my work done at ATR was the attempt to utilize the creative ideas and concepts of the two artists I was working with in a scientific context. As a result we designed and prototyped the Direct-Interactive Dynamic Form Synthesizer described in this chapter.

4.1 Idea

Sommerer/Mignonneau's pieces are very beautiful and aesthetically pleasing, but the programs used to create them are unique, and they have to be rebuilt for a new piece. The idea was to create a new system that could be used as a basis for a more generalized approach.

One main target was the creation of fully dynamic forms, i.e. every aspect of the form was to be changeable at any time. Furthermore the design of direct-interactive objects and behaviors was to be encouraged. What do I mean by direct-interactive?

Two basic concepts for the creation of interactive programs are rule-based and expression-based.

The rule-based system, unless fuzzy logic is applied, is decision based: a rule is active, and depending on the rule being active or not an action is performed or not. Thus the changes are usually very abrupt, as soon as a magic border is crossed something happens. If this is not very carefully balanced it can easily surprise the user and destroy his confidence in the understanding of the system, resulting in lower subjective usability.

The expression-based system expresses the actions to be taken as an arithmetic expression of the user's actions. This allows the response to be con-

tinuous and thus much more natural. Physical constraints make all actions occuring in nature continuous, thus discontinuous actions are perceived as less natural. Expression-based systems, via the use of continuous expressions, allow a very gradual and natural response to the user's actions. Furthermore the response is not necessarily limited to a specific zone of response, it can be ubiquitous due to the possible continuous decrease of reaction depending on the user's distance to the point of action.

Direct-interactivity is the usage of an expression-based system for multiple users. It is a very small step to make the expressions based on the actions of two different users, still retaining the degree of continuity and ubiquitousness. Thus the interaction between the users is very direct. They always interact in a consistent way, their interactions just change the degree of resulting actions. Thus no mediator is needed, the direct interaction between the users is used to control the whole system.

Based on this an expression-based system to support the interaction of two or more users was designed. As it was supposed to be used an experimental platform and toolkit for further exploration it had to be very flexible, powerful, modularized and easily extendable. An important target was a system supporting interactive design, as during the design phase constant changes to the used expressions and the values used to weight the expressions was expected. This automatically implies the need for a checkpoint/restore facility, as redoing the whole interactive design for every run of the program is not feasible. Of course the system also had to support full performance rendering, otherwise I couldn't bear seeing my name on it.

4.2 Concept

Based on the requirements a new concept for dynamic form generation was developed. The nomenclature is based on an abstracted tree structure, as this was context it was developed in. It takes ideas from discussions with Mignonneau-san and Sommerer-san, based on the experience in developing their artistic programs over the years.

The structure consists of the following elements:

- Rings
- Ring Operation Sets
- Tubes
- Trees

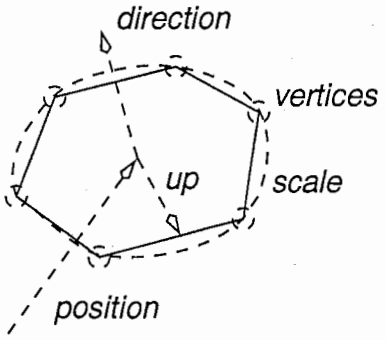

Figure 4.1: Ring structure

- Operators

4.2.1 Rings

The ring is the lowest level form element (s. fig 4.1).

It's basic defining attribute is an array of vertices. These don't have to lie an a plane, although it is the natural way to think about it. Each vertex has an associated 3D position and an associated color as well as texture coordinates. For consistency reasons all the attributes are stored as a 3 dimensional vector of floating point values. Even if the texture coordinates only use two of these values, a three dimensional vector makes sense in the context of operators (s. sec .4.2.5).

Apart from the individual vertices there is a set of attributes that influence the ring as a whole. The ring has a position vector to define where in space the ring is located. To define it's orientation a single vector for the direction of the X-Z plane of the vertices is used. The remaining degree of freedom is covered by a second directional vector indicating the direction of the vertices' Y-axis and thus called up.

To simplify modifications of the whole ring without having to resort to changing every single vertex (for reasons becoming apparent in sec. 4.4) two additional attributes are introduced. One is a vector to define scale values that are applied to the ring as a whole, the second is a color value for the ring as a whole. This color value is only used when there are no vertex colors, so it is a sort of fallback value (if not even a ring color value is defined the natural color of the material is used).

The ring is little more than the collection of attributes described here. It can be manipulated by a set of ring operations, which is the last attribute a ring has.

4.2.2 Ring Operations

A single ring in itself is not very useful. Neither is a static ring that doesn't change.

Thus rings can be manipulated by ring operations. The ring operation type is a collection of operators of each ring attribute. The exact specification of operators is given in sec. 4.2.5, here it might suffice to say that they have a vector input and a vector output and can do some kind of calculation.

Any element of a ring operations set can be empty, in that case the output is not changed. For the whole arrays of vertices/vertex colors/vertex texcoords there is only one operator which is in turn applied to every element of the array. All valid operators are applied, thus a ring color operator is executed even if it's result will be ignored because there is also a vertex color operator.

After the vectors that define the orientation of the ring in space (dir & up) have been changed by their operators they might not be orthogonal or unit length any more. If only one of them was changed it is assumed that the other one should be adjusted, thus it is renormalized and made orthogonal to the first vector again. If this is not the desired result, a dummy copy operator can be inserted for the other vector, in that case the two vectors will be left alone.

Ring operations are used in different places, thus it made sense to create a separate object for them. Their most important use is to connect the rings of a tube.

4.2.3 Tubes

The tubes are the basic structure element, and they are the only rendered element.

They are a container for rings, thus they have an array of rings. In the standard case these rings are connected to each other with polygons to form the graphical image of the tube. This is just the default case, however, any other interpretation is possible. For example the wire tube connects corresponding vertices on a ring with lines to form a set of lines following the tube.

How are tubes generated? The basis for the tube is it's template ring. When the tube grows, this ring is copied and appended to the tube. The attributes of the ring are generated from the attributes of the former last ring of the tube via the ring operations stored in the ring template. Thus the attributes of the new ring depend on the history of the tube and don't have to be calculated completely new, a relative operation is enough. They can

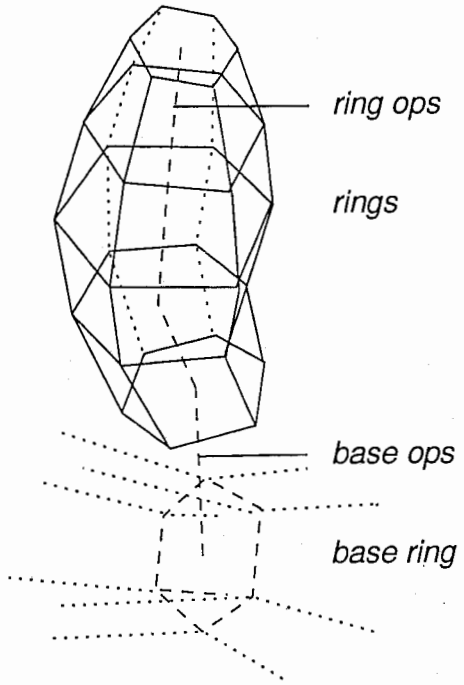

Figure 4.2: the tube structure

be, though. It is in the operator's freedom to completely ignore the input.

Growth itself is also controlled by an operator. The growth operator is evaluated after all the rings of the tube have been evaluated and the first element of it's return value decides, if the tube has to get longer or if rings should be removed from the end of the tube.

Dynamics and movement come into the picture through iteration. For every frame the tube is regenerated in the same fashion it was created in the first place. Beginning from the base the ring operations are applied to all the rings in a sequential way, thus the whole structure of the tube can be changed at any time.

With the capabilities covered so far, a single dynamic tube can be created. A lot of interesting things can be done with a single tube, but a lot more interesting things result from the combination of a number of tubes in a branching fashion.

To do that a decision has to be made, when and where to create new branches. In the interest of continuity it makes sense to connect new branches to an easily identifiable unit whose position can be easily obtained. Thus it makes sense to link the decision and position to individual rings. A tube has a branch operator, a binary decision that is checked during the update of the tube if a new branch should started here and now. If this decision evaluates

to true, a new branch is created.

If the new tube was just created at the place where it used to be at the time of creation it would become disconnected as soon as the parent moves. Keeping them connected might become arbitrarily difficult for complex movements of the parent. To simplify this important operation the new tube has a connection to it's parent, the base ring. The base ring is a reference to the ring in the parent. Together with a special ring operation set it is used to update the first ring in a tube. Thus by just copying the postion from the base ring to the first ring the child tube will always stay attached to the parent tube, no matter how the parent tube moves or deforms. If the base operation set also honors the direction of the base ring, the child tube can also move following the moves of the parent. But of course it can also completely ignore any information handed down and detach itself from the parent moving freely.

To ease individualisation the tube also has an operator that is only executed once, at the time the tube is branched from it's parent. This operator can be used to initialize variables specific to this tube, e.g. an initial rotation constant that should be different for every tube.

To organize the tubes into units the trees are added.

4.2.4 Trees

Trees are a pretty shallow wrapper around an array of tubes. They have a creation operator, that decides whether a new instance of this kind of tree should be created and a tube template used to start the tree with.

4.2.5 Operators

The operators are the basic active element, and they make this system direct-interactive.

An operator is a small procedure with a defined interface. It has one vector valued input and one vector-valued output. Additionally the context in which this operator is invoked is available, holding information about which ring in the tube this operation is for, how far this is away from the root of the tree etc.

On top of these an operator can have an arbitrary number of parameters. These do not change during the run of the program, only for interactively changing the structure of the targetted result. The parameters can have a number of different types: floats, integers, strings and vectors. And the most important parameter is a reference to an operator. Using this mechanism

operators can be cascaded, allowing arbitrarily complex expressions to be constructed.

Some examples for operators:

- constant: just returns a constant value to its output
- global value: e.g. the mouse position or the user's position in the space. This is the place that allows interaction with the user. And this also makes the extension to two or more users obvious (as long as the number is fixed): you can just access each user's data and combine them into a common result
- context value: e.g. time or the position with a tube. This allows operations depending on where within the evaluation chain the operator is. Typical applications for time are as input to oscillators (s. below), mostly in combination with the index in the tube, to create a movement that depends on time and moves along the length of the tube
- addition / subtraction / multiplication / division: basic arithmetic operations
- component selection / splicing: to mix different inputs componentwise into one output
- oscillation, sinusoidal or sawtooth: for cyclical changes
- rotation around/to a vector: to orient structures according to inputs
- color space conversion: to convert a color from HSV to RGB space or vice versa
- commandline: shorthand notation for complex arithmetic expressions which otherwise take a lot of operators to construct
- etc.

As operators are an essential part of the flexibility of the system, they have to be small, easy to write and to change, without the need to change other parts of the system, and if possible without having to recompile the system.

4.3 Realization

The described concept has been prototypically realized.

As the whole concept is very object-oriented and easy extendability was important, C++ was chosen as the implementation language. Each of the described elementary types was realized as one (or more, where sensible) classes. On top of that a set of templated support classes for an automatically generated user interface (s. sec. 4.3.1) were built.

Special care was taken to ease the addition of new operators, as this is an action that is expected to be executed very often. To add a new operator only the file of the new operator has to be created. No other files have to be touched, the system just needs to be recompiled (even though that may change in the future (s. sec. 4.5)). Thus apart from the small programming overhead described in sec. 4.3.1 no additional knowledge about the system is needed, just copy an example operator and change it. In this way simple operators can be added in less than 10 minutes.

It was tried to keep the system efficient. To achieve that goal a fast library for the very often needed vector/matrix functions was written. Furthermore in all inner loops data is passed around by reference and is never copied when it is not needed.

To achieve the goal of full performance rendering the data structures were optimized for the infinite Reality. All rendering data is stored in directly accessible stripes. The consequence of this is a duplicated storage of the vertices, but it allows the usage of OpenGL's VertexArray extension to render them with very little load on the main system. Furthermore the already known iR datatypes (short normals, byte colors) were used.

Another step in optimizing the system was the use of a pipelines architecture. In this case the pipeline has three stages:

- simulation
- normal calculation
- rendering

The first stage is the main simulation and interaction loop. This stage handles all user interaction and interfacing aspects as well as the simulation and behavior of the objects.

For a system like this, where all the objects can completely change all their vertices for every frame a step that was not so important in the MIC Exploration Space system needs more attention: the calculation of normals. Good quality normals are important for interactive systems, as they allow

the use of a smaller number of polygons when used with Gouraud shading. But the calculation of normals is an expensive operation, as it has to be done for every vertex and takes a crossproduct and 6 vector additions. This can easily become one of the most expensive parts of the system, thus using a separate thread for this calculation is useful.

The rendering thread in the system is not very intelligent yet. There is no command pipeline between the different processes, as all of them are working on the same types of data and every frame is exactly the actionwise, thus a complicated flow control is not needed. It is however possible, and needed, to synchronize the three threads at specific points.

As this is a very dynamic system, in which lots of objects are created and destroyed, memory management is a special problem. But as there is three stages deep pipeline behind everything, the objects that are deemed unused in the simulation are still needed in the rendering two frames later. Thus the memory can not be freed so easily. To prevent usage of objects in the rendering that have not yet gone through he normal calculation stage the renderer keeps a list of new objects in contrast to the list of active objects.

Newly created objects stay on the new object list for 2 frames before they are rendered, thus the security of having been correctly handled by the normal calculation stage is given.

For the deletion of objects there are three KillBuffers. The actual KillBuffer contains the objects that have been marked 'to be deleted' by the application three frames ago. Before the next frame is started, these objects are deleted, as they are old enough to be safely destroyed.

To correctly handle these lists the three processes have to be synchronized and cannot work on the data while the organizational changes are made. This is a potentially expensive operation like every synchronization, even more in this case, as three processes have to be synchronized. Thus this synchronization is only executed if the NewList or the KillBuffer have elements on them, otherwise it can be ignored.

4.3.1 User Interface / File I/O

As the design of a direct-interactive system is a very iterative process it should be done using an interactive system to eliminate turnaround times. As an interactive system it also needs a way to to save and restore the state during the work to be able to create and keep a work over multiple sessions. On the other hand the target was to create a very flexible and easily extendable system, so that the creation of a new user interface for every new operator was not an option.

To solve these problems the attribute types of all the objects in the system were restricted to a limited set:

- integers (32 bit)
- enumerated values
- single precision floating point values
- character strings
- threedimensional single precision floating point vectors
- vector arrays
- pointers to the basic object types:
  - operators
  - rings
  - ring operation sets
  - tubes
  - trees
  - materials
  - textures

For all these types an ASCII printing format was designed. For all the pointer types this was done using a template, thus the coding expense is minimalized while keeping type security. Furthermore for each of these types a small user interface element was designed using the FORMS library.

Each of the objects that want to be saveable or manipulable by the user interface have a new attribute: an array of structures describing the fields this object has, giving a name, the type and the offset of the fields from the start of the object.

From this information an ASCII version of the object can be created. An example would look like this:

```
Operator PosTo2 sub
```

```
{
        sub1 User2Center
        sub2 value      {
                        context in
                        direction get
                        name ''pos''
                        input 0x0
                        }
}
```

This text describes an operator named `PosTo2` which is a `sub` operator. it has two attributes, `sub1` and `sub2`.

Both of them are operators, which can not be seen from the code itself, but is known to the system by the declaration done by the `sub` operator. `sub1` just given the name of the operator to be used, in this case `User2Center`, `sub2` uses an anonymous operator which is saved inline. It is a `value` operator which gets the the value `pos` from the `in` context and has an `input` attribute that is not used. `in` and `get` are enumerated values, while `pos` is a character string.

This is exactly the format that is printed to a file to be read again in a later session to restore the state of the system. To do that the objects have to be written in the order that allows to dereference at the time of loading. To make this possible empty objects can be inserted as forward references (this is done automatically on saving).

The same format is also used by the user interface to communicate with the system. In this case the objects can be identified using a pointer, so that names do not have to be resolved by search, which might take a measurable time when many objects are loaded. On top of that it also allows to help debugging the system by printing the commands passed between user interface and the data management.

The user interface is constructed from the field description in a similar manner. An example of an automatically constructed user interface looks like fig. 4.3.

This example user interface displays an operator similar to the one in the text example, though slightly different. The left part shows all available operator instances. The top bar shows the type of the operator (`sub`), by clicking it new operators of this type can be created. The two arguments are inline operators, which are indicated by being slightly indented. The arrows pointing to the right symbolize a way to exchange this argument for any other of its type (here only operators). After pressing this button it stays pressed and any other operator, be it in the Instance list, the argument of

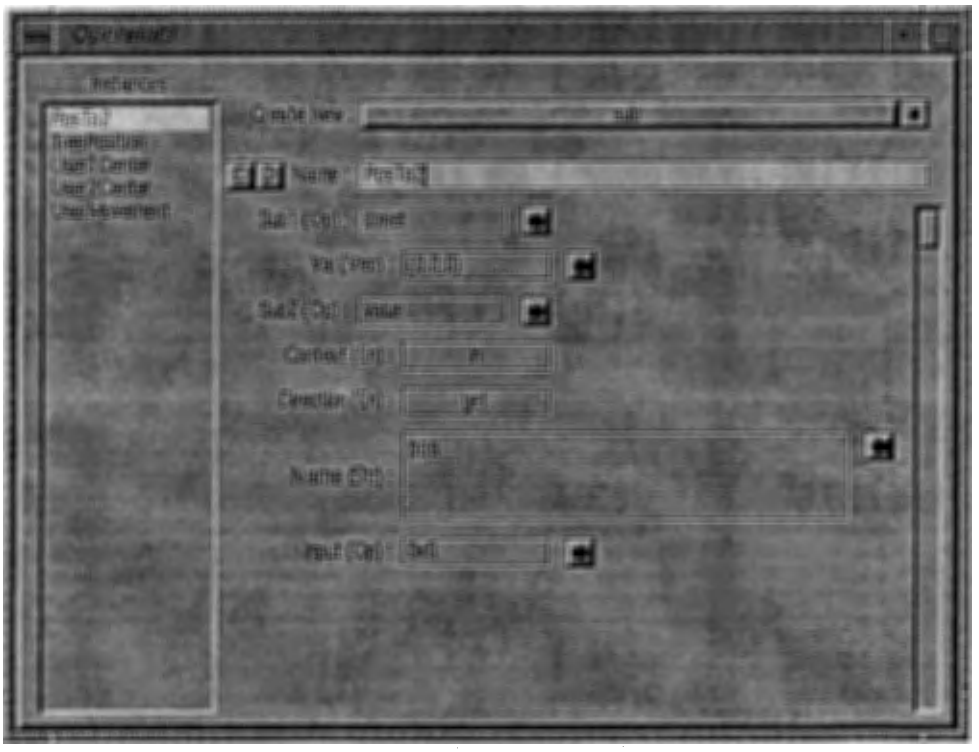

Figure 4.3: example user interface

another operator or onthe Create New list can be transfered to the button's field.

The inward arrow on the constant's value has a different meaning. As an important part of creating an interactive scene is the adjustment of constants a special user interface has been created for this purpose: the hotlist (s. fig. 4.4). Different constant values can be collected in a hotlist. From here they can be manipulated using different interfaces appropriate to vector values (the main constant value type). It uses automatic border adjustment for the styles having limited ranges (slider/dials). For textual input the range is of course free, and when switching back to a limited range style the range is adjusted to the actual value.

4.4 Results

A number of different interactive dynamic forms have been created. Fig.4.5 shows a typical working screen. It shows some dynamic forms that orient themselves according to the position and movement of the red tube, which symbolizes one user and is linked to the mouse for testing purposes. Some

Figure 4.4: an example hotlist

windows showing ring operator sets and tubes for changing the rules governing the object's behavior, as well as a hotlist for some constants are visible. Any number of windows of any type can be active at the same time.

The system lends itself to the creation of organic forms like the ones shown, but much more abstract forms are also possible.

But the static images don't do the forms right, as their main strength is the complete freedom for change they possess.

4.5 Future

Different forms have been created, and some problems arose during the design that point to areas for future work.

The user interface for inlined anonymous operators takes quite a lot of screen real estate. For moderately complicated expression the screen is already not high enough. Of course a vertical scrollbar has been employed to make bigger expression feasible, but the construction and management of these expressions is difficult. Thus for the more complex expressions a command line operator was used, that can handle arithmetic expression in

Figure 4.5: a typical working screen

standard written form, has some shortcuts (e.g. for context values, global values or constant operators) and knows how to call any other operator by creating temporary internal anonymous operators. One approach to solve this problem in a more intuitive and graphical would be a graphical tree display of expressions with editing possibilities, similar to the tree widget available from SGI's ViewKit library.

The system can be made even more flexible and easier to extend by supporting dynamically loadable operators using dynamic shared libraries, available on SGI. This would obviate the need to recompile the system when integrating new operators. Even if it only takes some minutes, it still disturbs the work flow and it is thus a good idea to prevent it.

As any interactive system is never fast enough the question how to speed this system up is an interesting area for further work. It turns out that the simulation is the main bottleneck. Thus a future work area would be the usage of multiple threads for the simulation. This raises a lot of synchroniza-

tion and locking issues and is definitely going to be difficult in a system as flexible as this one.

Another approach to the problem would be the creation of an expression compiler that creates customized throwaway operators that are dynamically compiled and loaded during the runtime of the program. This would allow the compiler to optimize whole expression and not only the small operators. This is expected to achieve a major speedup.

As this system is designed to be extended even in very basic areas a wide array of direction for extensions is available. Some ideas that come to mind are the inclusion of a particle system metaphor. These particles would have some primitive acceleration-speed-movement behavior that is executed for every frame, while the more complicated target following or interaction behavior is only executed once for a number of frames for every particle. This would allow a big number of particles without adding an intractable workload on the simulation. Another idea would be to more fully use the data pFinder is able to deliver. For example the silhouette of the user can be extracted from pFinder and used as the form of a ring.

Last but not least this is just a very low-level system. For more complex interactions a higher level of control is desirable. This was not the topic of my work, but will be needed for complex interactions.

Chapter 5

Extending

The following is a list of the program's files with an explanation what they do. .c and .h files are C-Code, .cc, .hh and .icc files are C++ code, with .icc being inline functions. These can be moved into the main modules as separate functions (e.g. for debugging) by defining OUTLINE. Unless otherwise mentioned for evey .cc file there are corresponding .hh and .icc files, and .h files for every .c file.

- RCS/: Revision Control System repository for older sourcecode version (s. RCS(1))
- pfinder/: Lopez-san's pfinder client source code
- objs/: the object codes
- ii_files/: SGI CC's instance information files repository
- examples/: example data files
- GNUmakefile, Makedepend: makefile and dependencies. GNU make (/usr/local/bin/make) has to be used to compile the system.
- defs.h, defs.hh: general definition needed in every module and convenience fundtions to add attributes and access function to a C++ class
- dll.cc: code for templated doubly linked lists and iterators for them. To use it just derive your class from dllElemC<class>.
- dynarr.cc: code for templated dynamically growing arrays and iterators for them. This is just a template class, usage should be obvious and it is used every throughout the system, so examples of usage are easy to find.

- vector.cc, matrix.cc: high-speed threedimensional floating point vector and (4x4) ((4x3) used) matrix library. The speed is mostly achieved by providing most functions in a way that does not create new objects but rather manipulates the values of already existing objects, preventing the allocation of temporary objects. This might not be as convenient as the standard overloaded C++-operators (which are provided, too), but for simple operation the difference is acceptable and the speed increase is definitely worth the effort.
- string.cc: simple character string handling functions, nothing fancy
- time.cc: time handling and arithmetic.
- cfb.c, picture.c, ppm.c, rgb.c, tiff.c, tiffdef.h, tiffio.h, yuv.c, libtiff.a, mem.h: image loading/ saving routines for a variety of formats. These routines are taken from the 'Y' system (which in turn took them from the 'Genesis' system), so the coding and naming conventions are different and some conversion macros have been defined in picture.h to make them compile.
- forms.h, fd_inter.fd, fd_inter.c, fd_inter_cb.c, libforms.so.0.81, libforms.so: the FORMS interface library and the definition files for the user interface, designed with fdesign.
- port.cc: serial port handling object. Handles blocking/ non-blocking and binary/ASCII IO
- container.cc: container for vector values or references that can be accessed by name or index. Also has a generalVectorContainerC class that can be extended at runtime.
- gl_window.cc: OpenGL window/widget handler class. Also cares about redraws, resizes, input (mouse/key). Because it handles the redraws it also has to take care about the pipeline synchronisation issues.
- render.cc: this module contains a number of classes for renderable objects. renObjC handles multibuffering for pipelining, renGLVertexC is a simple wrapper around a vertex's attributes, renTubeC is the main rendered object, being able to render every combination of attributes and materials/textures.renTubeVertexIteratorC is used to change the vertices of a renTubeC by allowing to iterate over all the tube's vertices taking care of multibuffering etc. renWireTubeC is an example for a new rendered object based on renTubeC. It renderes it's vertices connected by lines.

- material.cc: wrapper class for OpenGL material state, including texture references (s. texture.cc).
- texture.cc: wrapper class for OpenGL texture state. Also responsible for loading the textures using the image loading routines described above.
- cim.cc: the Class and Instance Manager. This is the central database manager class. It defines a set of templated superclasses for types that want to handled. As the expressive power of C++ is not quite enough to handle all aspects of this, a couple of macros are defined, too, which should be included into the class definition. See the use of the manager in operator.cc and the individual operators for examples.
- io.cc: Input/Output handling. Defines the IOFieldC class that is used to handle the single fields that are supposed to be written/read. Also defines the IOFieldSetCclass, which is part of every object type that is written/read.For the handling of the different types of an object kind (e.g. all the different operators) the IOTypeContainerC clas is used.
- global.cc: defines the globalC class, which handles all user interface variable aspects (Mouse, pFinder, Mignonneau-san's camera interface). All the data is available through the globals globalC object. Furthermore the main loops are methods of this object.
- if_fields.cc: the different fields used byt he automatic user interface. Individual fields know how to change the field it stands for and where in the interface panel they are. Hotlist fields are a special case that is only used in a hotlist (s. if_hotlist.cc).
- if_types.cc: templated class ifTypeWinC is used to create the interface panels for the different types of objects (rings, tubes, etc.).
- if_hotlist.cc: special type of container for ifHotlistFieldCs that knows how to attach to a field and can grow and shrink acfcording to the numer of active hot fields.
- if_main.cc: main interface part. Handles the pull-down menus, File IO and some window callbacks.
- operator.cc: define the operatorC superclass for all different operators.
- ring.cc: defines the ringC and ringOpC classes as described in sec. 4.2.1 and 4.2.2.

- tube.cc: defines the tubeC class as described in sec. 4.2.3.
- tree.cc: defines the treeC class as described in 4.2.4.
- inter_test.cc: the main program. Just a main loop to create the interface and call FORMS.
- operators: this directory contains the code for the different operators. Additions to this directory will automatically be added to the system on the next compiler run.
  - veclen.cc: returns the length of it's input
  - value.cc: read/write a value from the in, out, local, global or own context
  - sub.cc: subtract it's two inputs
  - splice.cc: takes the first element of it's first input, the second of it's second and the third of it's third to create the output
  - seq.cc: executes it's four operators in sequence
  - select.cc: set's its output to the ind's element of it's vector parameter
  - scale.cc: scale the input parameter by scale
  - sawtooth.cc: runs a sawtooth oscillator between start and stop at a speed of phase cycles per second.
  - rotate.cc: rotates in around/using vector angle degrees.
  - ref.cc: empty demo operator
  - random.cc: craetes a random vector between base and base + amplitude
  - ramp.cc: runs a ramp oscillator between start and stop at a speed of phase cycles per second.
  - print.cc: print's op's value and copies it to the output.
  - param.cc: access one of the parameter values like time,phase or generation.
  - osc.cc: runs a sinusoidal oscillator between start and stop at a speed of phase cycles per second.
  - logicop.cc: connect it's inputs by logical operations
  - lc.cc: linear combination of it's inputs

- hsvtorgb.cc: convert's betwwen the HSv and RGB color spaces.
- glob.cc: access a global value like mouse or user position
- generic.cc: a generic operator that calls a user-supplied function. This operator can not be saves and restored!
- dist.cc: calculates the distance between it's points.
- copy.cc: just copies input to output.
- constcomp.cc: compares a constant value to an operator.
- const.cc: just sets the output to a constant value.
- comp.cc: compares two operators.
- cline.cc: commandline. Can handle arithmetic expressions involving constants and operators. Vector comparisions are handled as the logical and of componenwise comparisions and are written using standard C comparison syntax. Comparison of only the first vector element is written with the following symbols (the meaning should be obvious): <<, >>, >>=, <<=, === and !==. The understood arithmetic operations are +, -, *, /, ^ (exponentiation), % (modulus), and the C ?: operator. Vector constants should be written as [0,0,0].Other operators can be called as <opname>(<parameters>). Special cases exist for the reading of constant operators (write without ()), in-context, out-context, global or parameter values (write as in|<val>, out|<val>, global|<val> resp. par|<val>). The expression is compiled and optimized using constant folding and constant decision reduction.
- clamp.cc: clamps value to not be lower than low and not be higher than high.
- angle.cc: returns the angle between it's two params
- add.cc: adds it's two inputs.

Chapter 6

Appendix

6.1 Operator Superclass Header

This is the header describing the attributes and abilities of the operator superclass, the most important class in the system.

```
// operator:  Operations on vectors aided by
vectorValueContainers

#ifndef _OPERATOR_HH
#define _OPERATOR_HH

#include <iostream.h>
#include <iomanip.h>
#include "defs.hh"
#include "vector.hh"
#include "dynarr.hh"
#include "dll.hh"
#include "container.hh"
#include "io.hh"
#include "cim.hh"

//
// abstract base class for operators
//

class operatorC;

// global operator class and instance manager
extern cimGlobalC<operatorC> opGlobals;
```

```
class operatorC : public cimElemC<operatorC,operatorC>
```

```
{

    public:

        static const char *kindName;    // kind name
of this operator

        class paramsC   // Parameters wrapper class to
ease extension and reduce
                        // # passed parameters
        {

            public:

                paramsC(void);

                paramsC(    vectorContainerC *
_outcontext,
                            vectorContainerC *
_incontext,
                            vectorContainerC *
_localcontext,
                            float _time = 0,
                            float _localtime = 0,
                            float _phase = 0,
                            float _maxphase = 0,
                            float _globalphase = 0,
                            float _generation = 0,
                            float _subphase = 0,    //
index for points in
                                                    //
single ring (for shape
                                                    //
and color)
                            float _maxsubphase = 0
                            );

                paramsC( paramsC & _ops );

                void set(   vectorContainerC *
_outcontext,
                            vectorContainerC *
_incontext,
                            vectorContainerC *
_localcontext,
                            float _time = 0,
                            float _localtime = 0,
                            float _phase = 0,
```

```
};

typedef operatorC * operatorP;


// doubly linked lists definitions

typedef dllC<operatorC> operatorDllC;
typedef dllIteratorC<operatorC> operatorDllItC;


// dynamic array definitions

typedef dynArrC<operatorP> operatorPArC;
typedef dynArrIteratorC<operatorP> operatorPItC;


// include inline functions (if wanted)

#ifndef OUTLINE
#include "operator.icc"
#endif

#endif
```

6.2 An Example Operator: Addition

The header file:

```
// add: add operator, add its 2 operator inputs

#ifndef _ADD_HH
#define _ADD_HH

#include "defs.hh"
#include "operator.hh"

class addOperatorC : public operatorC, public
cimElemC<operatorC,addOperatorC>
```

```
{
    public:

        void operate(   vec3fC & out,
                        vec3fC & in,
                        paramsC & params
                    );

        addOperatorC( const addOperatorC & _op );
        addOperatorC( const operatorP _add1 = 0, const
operatorP _add2 = 0);


        cimMembers( operatorC, addOperatorC, "add",
opGlobals);

    private:

        operatorP add1, add2;
};

typedef addOperatorC * addOperatorP;

#ifndef OUTLINE
#include "add.icc"
#endif

#endif
```

The source file:

```
//
// add operator
//

#include <stddef.h>
#include "add.hh"


#ifdef OUTLINE
#include "add.icc"
#endif
```

```
cimInitElem( operatorC, addOperatorC,
operatorC::kindName, "add", opGlobals );

void addOperatorC::initClass( void )
{
    iofields.addField( IOFieldCoperatorE, "add1",
offsetof(addOperatorC,add1) );
    iofields.addField( IOFieldCoperatorE, "add2",
offsetof(addOperatorC,add2) );

    opGlobals.addType( addOperatorC::typename,

&cimElemC<operatorC,addOperatorC>::create,

&cimElemC<operatorC,addOperatorC>::create,

&cimElemC<operatorC,addOperatorC>::change );
}

addOperatorC::addOperatorC( const addOperatorC & _op )
    :   add1( _op.add1 ), add2( _op.add2 )
{
}

addOperatorC::addOperatorC( const operatorP _add1,
                            const operatorP _add2)
    :   add1( _add1 ), add2( _add2 )
{
}

// action

void addOperatorC::operate(     vec3fC & out,
                                vec3fC & in,
                                paramsC & params
                           )
```

```
{
    if ( add1 && add2 )
    {
        vec3fC tmp;

        add1->operate( tmp, in, params);

        add2->operate( out, in, params);

        out.inc( tmp );
    }
}
```