An Intensional Language for graphics and sound unification

Kees van Prooijen

Abstract

A description is given of a language that combines an original view of an interpreted functional language with a unifying approach to multi-dimensional data as it occurs in graphics and sound environments. With the formal notion of intensionality a method is obtained to combine many aspects of the production and processing in these environments. This language comprises a user-extensible object-oriented type system. Values of any type can transparently take the form of a function on a multi-dimensional, continuous or discrete, domain. In this way a unification is obtained of at least the following: 3D modeling, animation, texturing, image processing, sound synthesis and (scientific) visualisation.

CR Categories: D.3.2 [Programming Languages] Language Classification - Applicative languages; Extensible languages; Specialized application languages; I.3.3 [Computer Graphics] Picture/Image Generation - Display algorithms; I.3.5 [Computer Graphics] Computational Geometry and Object Modeling- Geometric algorithms, languages, and systems; Modeling packages; I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism- Animation; Color, shading, shadowing and texture; I.4.0 [Image Processing] General - Image processing software; I.6.2 [Simulation and Modeling] Simulation Languages; Additional Keywords: Sound synthesis, Image compositing

1 Introduction

In this introduction a summary will be given of the historical origin of the use of intensional principles. Thereafter follows an outline of the rest of the paper.

Intensionality stems from the linguistic expression 'in tense', meaning 'relating to time'. This concept originated in the context of the development of logical systems, in order to be able to construct better semantical models of time related notions as they occur in natural language. In simple logical models, the meaning of a syntactical item is related to sets of individuals. The underlying thought of the intensional paradigm is to make the meaning of such an item a function from time to these sets (or to a relation of these sets). This is necessary in order to fulfill a historical demand made by Frege, who stated that the formal meaning of a sentence should be deducible from the meaning of its constituents. Let's take a simple example:

A former president of the USA.

In order to be able to attach a meaning to this sentence fragment, one can not take the naive approach to deduce this from the simple meaning of 'President of the USA', being an individual. Instead, this meaning must be taken as a function from history to individuals, in order to let 'former' have any sense.

Subsequently the concept was elaborated to comprise other contexts, apart from time. The crucial point is that the semantical strengthening is completely transparent to the syntax.

$\mathcal{IL}$ is the successor to a language on which a range of 3D graphics products was based. (Pronounce $\mathcal{IL}$ as "I, Al", not as "ill") The goal of this former language was the exploitation of the interpreter paradigm, to combine a simple but powerful modeling language with a user-interface written in the same language, as front end for a ray-tracer. In a certain stage of the development of this earlier language, the possibility for writing procedural texture maps was integrated in the language. This means that surface attributes, like intensity and color, of objects become functions of the coordinates of the surface. At first a limited number of primitive parameterized functions was made available, like chessboard and stochastic fractal intensity patterns. To combine and adjust these simple textures, higher-order map functions where introduced, which took map functions as their parameters. In the following example `mp` becomes a 2-dimensional 1/f noise pattern with fractal parameter 0.7, which is subsequently scaled to have values in [0.2,0.8] rather then in [0,1].

```
mp = fract( 0.7 );
smp = scalmap( mp, 0.2, 0.8 );
```

By this time it was realized that it would be more elegant to make maps subject to the standard arithmatical operators and functions as if they were constant numbers. So, this language was adapted to make the following possible:

```
smp = 0.2 + 0.6*mp;
```

In this way the higher-order aspect was replaced by intensionality. In the context of this language it means that values of a certain type can transparently take the form of a function on one (or more) of a number of standard domains. Functions and operators that are applicable to the ground type(s) of these 'intensionally' defined entities, automatically become available for the maps. If the coordinates were made available as primitive maps, the user would be able to create his own maps as well as make any combination of them. The 2D coordinate system was made available with the functions `mu` and `mv`.

```
m = 0.5 + ( sin( 55*mu() ) + sin( 34*mv() ) ) / 4;
```

At first this principle was further elaborated to comprise all forms of texturing techniques. This means that the 3-dimensional domain was added to be able to create so-called solid-textures [6][7], which are functions of a local 3D coordinate system. For this purpose the set of built-in coordinate functions was extended with `mx`, `my` and `mz`. Also bump-maps were added. These are modulations of the surface normal to simulate non-smooth surfaces [1]. To get a closed arithmetical system, bump maps were modeled as complex-valued functions.

So this first intensional system was an embedded part of a graphics language. It was intended for the description of (procedural) texture maps. The possible domains were 2D and 3D continuous coordinate systems and the range value-types were real, complex (bump) and 3D-vector (color).

After these developments there were several reasons to decide to start a new language. In the old language, implementing new types made it necessary to adapt the C source of the kernel. And when new implicit (built-in) functions were added to the language, it was up to the implementer to make these also available for the texture maps. It was also understood that the same principle that was at the core of the texture generating facilities, could be used to formalise image and time related activities.

In $\mathcal{IL}$ the intensional principle is at the heart of the language. The domains are 1-, 2- and 3-dimensional, both discrete and continuous. All (static) data types can become intensionally defined. Obviously the 1D domain is intended to represent time. For instance 1D dependent matrix valued functions can be used as transformations for procedural animations. Discrete time appears in 3 instances; relating to sound as sample-time and control-time, and relating to animation as frame time. 2D discrete domains have a purpose as basis for images. If necessary there is automatic conversion to a continuous domain by any available interpolation scheme, e.g. when an image is used as a texture map or when two images of different size are composited. The 3D discrete domain is the basis of, for instance, scientific data-sets. Continuous domains are basically abstract mathematical entities, which serve in many ways the desired unification.

It must be noted that intensionality can be regarded as an extrapolation of Perlin's ideas, as first revealed in his Pixel Stream Editor [7], which are also to be found in the texture facilities of so-called shading languages [4]. The implicitness of intensional domains and the applicative style of programming are responsible for the higher order of the language and the stronger unifying power. For instance, a parameterized function can be written to generate a (texture) map. In $\mathcal{IL}$ it it is then possible to call such a function with any map (of the right type) as argument, where in other languages the kind of parameters would be limited to static values.

In database contexts, one sometimes finds the term intensionality with a related but different meaning. It then means that the value of a field is stored as a formula, dependent on other fields, instead of a plain value. Maybe the term 'modal' would be better in that context because of the absence of an (explicit or implicit) intensional domain.

In our context the opposite notion of extensionality could be used to give a new definition of a rendering process:

> extensionalizing a value on a discrete domain, through lazy evaluation of the related intensional dependencies.

Rendering in this sense gives rise to a number of file generating functions in $\mathcal{IL}$. For each dimensionality there is a rendering function for the appropriate type(s); 1D `Real` valued: sound files, 2D `Pixel`: images, 3D `Real`, `Complex` or `Real3` 'scientific' data-sets. The latter two cases (2D and 3D) also bring about a time dependent (animated) version. This kind of rendering is called 'extensional rendering' in this paper.

Rendering in the normal meaning of making (2D) images of (3D) scenes, plays a different role in $\mathcal{IL}$. For this purpose a solid-model type and a link to an associated ray-tracer are implemented in the language. This could be called 'projective rendering', because of the loss in dimensionality. The intensional principle of the language takes care of the texturing and animation of the 3D scenery. To all the rendering functions (in both senses) applies: the result of the function call is an (intensional) value in the language, with all 'civil rights'. File generation is only a side effect. Particularly, the resulting value can directly be assigned to an identifier, and/or be brought into use in other contexts.

```
im = raytrace( model );
```

Other (pre-projection) aspects of rendering (e.g. radiosity related concepts) could, following Kajiya, be captured in the phrase 'equational rendering' [5].

The language characteristics of $\mathcal{IL}$ will only be superficially treated in this paper, in favour of the media-related types and the intensional semantics. Following a short presentation of the

elementary language elements and implemented types, the notion of maps as central part of the intensional paradigm will be treated. Examples will be given in the form of implemented image and sound generating primitives.

2 IL as an interpreted functional language

The interpreter consists of a kernel written in (ANSI) C and an extensible set of C-modules for the implementation of the different types with their associated functionality in the form of implicit operators and functions. $\mathcal{IL}$ exploits fully the consequences of being an interpreted language. An interpreter performs operations according to an input character string. As a consequence, the types for functions and strings are the same in $\mathcal{IL}$. In other words, quoting and function-calling are complementary operations. In the following examples, '}' is the prompt during interactive terminal operation. In this mode $\mathcal{IL}$ reads strings from the console and prints the result of the last evaluated expression.

```
} 4+5
           9
} "4+5"
4+5
} "4+5"()
           9
```

In normal use string-functions will be assigned to identifiers before being called, thus resulting in a more conventional look.

```
} f = "$1 + 6"; f( 88 )
          94
```

As will be obvious from the previous example, the parameter mechanism is like that of a shell or batch language. $ is the 'fetch parameter' operator. It is a genuine unary operator, expecting any integer valued expression. Basic control mechanisms are C-style conditional expressions and (recursive) function calls.

```
} fac = "$1 < 2 ? 1 : $1 * fac( $1-1 )";
} gcd = "$2 ? gcd( $2, $1 % $2 ) : $1";
} fac(7);
        5040
} gcd( 7735, 14586 );
         221
```

As can be seen, boolean as well as integer (these are distinct types) expressions can serve as condition. Also lists can be used, the empty list being regarded as false. The interpreter tests for tail-recursion, so iterative constructs can be implemented without the risk of stack overflow:

```
} for = "$1 > $2 ? return(); $3($1); for( $1+1, $2, $3 )";
} for( 1, 4, "print( exp( $1 ) )" );
 2.7183
 7.3891
```

```
 20.086
 54.598
nil
```

('`nil`' being the notation for the empty list, the default value of '`return`') Being a functional language, the only constituents of $\mathcal{IL}$ are expressions. So 'anything that happens' will always leave a result.

There are mechanisms to link user-defined functions to operators, either new ones or overloading build-in operators. Among the other language features are: a more involved parameter mechanism, global and local identifier scope, protection and checking of global identifiers, a module concept, program-files, comments, debug modes and a generic call-back mechanism for coupling with user interfaces.

3 Types

New types are implemented in C-modules by calling kernel-functions during initialisation time. There are some things all types have in common. They have a name, an integer identification, a size and a number of methods applicable to all of them. When creating a new type in a C-module, the name is given to the kernel. Let that name be 'Mine'. The kernel then creates two $\mathcal{IL}$ identifiers:

- 'Minetyp' is an integer identifier with the value of the internal identification;
- 'Mine' is an implicit function which converts any parameter to the type 'Mine' if at all possible.

```
} Inttyp
           5
} Int( pi )
           3
```

The type identification is very useful in combination with '`typeof`', a function that returns this integer for any parameter:

```
} typeof( "abc" ) == Stringtyp
       T
```

The name of a type can be obtained, as a string, with '`typename`'. As mentioned above this is also the function-name to convert any value to this type:

```
} Realtyp
           6
} typename( 6 )
Real
} typename(typeof(sqrt(-1)))()(8)
 8.0 + 0.0 i
```

Types are defined in independent C-modules and can be linked at will to the kernel when creating an executable for the interpreter.

Types in $\mathcal{IL}$ can be either static or dynamic. Dynamic types are implemented as memory pointers to allocated memory blocks, with automatic maintenance of reference counts. List is a good example of a dynamic type. Only static types can become intensionally defined.

One of the methods that is always defined when a type is called into existence is the one that is invoked by a call of '`print`'. Another one is to copy a value of the type. This is important in the case of dynamic types. Simple assignments of these types copy a pointer and increment a reference count. A copy of a value is created with the operator '`&`'. On top of the normal copy, there exists for some dynamic types another kind of copy. The so called deep-copy, with operator '`&&`', recursively copies sub elements of a value. Again, lists are an obvious example.

Conversions from one type to another are done by coercions. Which method is invoked for an implicit function or operator is determined by the type context. So all arguments have their influence on the choice of method, instead of one argument determining a class. If there is no applicable method for the current context, the kernel tries to create one by coercing the arguments. By this convention it is possible to safely implement new functions and operators with automatic overloading of symbols when necessary. The possible coercion rules towards a target type are defined together with that type in its C-module.

3.1 Lists

Lists are implemented in the usual way with the constant '`nil`' and the functions '`cons`', '`head`' and '`tail`'. There are no type restrictions for list elements.

```
} l = [ T, 2, 3.0 );
```

is syntactic (well, actually semantic) sugar for:

```
} l = cons( T, cons( 2, cons( 3.0, nil ) ) );
```

`@` and `'` are index operators.

```
} l@n
```

is equivalent with calling tail n times.

```
} p = l@3
```

instead of

```
} p = tail( tail( tail( l ) ) )
```

```
} l'n
```

is equivalent with

```
} head( l@n )
```

3.2 Mathematical types

The mathematical types are `Bool`, `Int`, `Real`, `Complex` and `Quat`. Boolean values are written as `T` and `F`. All the usual notations and operators are available. There is a rather exhaustive set of functions for Real and Complex: `abs`, `sqr`, `sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `sec`, `cosec`, `asec`, `acosec`, `sech`, `cosech`, `asech`, `acosech`. They will return complex values for real arguments whenever necessary. As mentioned in the introduction, a special use of complex values is their interpretation as range type for texture bump maps. Quaternions are arithmatically fully implemented. There is a normalisation function to bring them on the 4D 'unit sphere' for the standard interpretation as rotations [10]. For quaternions there are coercions from and to transformation matrices.

3.3 Vector types

All vector types necessary as base classes for the geometrical types and the intensional domains are implemented.

`Int2`	≡ int[2],	discrete 2D points
`Int3`	≡ int[3],	discrete 3D points
`Real2`	≡ real[2],	2D points
`Real3`	≡ real[3],	3D points or homogeneous 2D lines
`Real4`	≡ real[4],	homogeneous 3D planes
`Intl2`	≡ int[2][2],	discrete 2D 'box'
`Intl3`	≡ int[2][3],	discrete 3D 'box'
`Reall2`	≡ real[2][2],	2D line vector representation
`Reall3`	≡ real[2][3],	3D line

Besides common operators for addition and scalar multiplication, they also have the `'` index operator. There are also matrix types for the representation of linear transformation in all dimensionalities.

3.4 Geometrical types

A complete set of 2D and 3D geometrical types (points, lines, planes and transformations) along with their constructions is implemented. Some examples:

```
} x = p1 * p2
```

dot-product of two points

```
} p = p1 ** p2
```

cross-product of two 3D points

```
} x = k ^ l
```

angle between two lines, or between a 3D line and a plane

```
} l = m |+ n
```

line through two points, or a 3D plane through a point and a line

```
} t = rotate( 37.5, l )
```

transformation matrix of a rotation around a line

```
  etc.
```

Special attention is deserved for the possibility to raise a matrix to a real power:

```
} t1 = t ^ 0.3
```

This works for any transformation! It is the basis for interpolation of transformations on the matrix level. Again it must be noted that because of the coercion mechanism, quaternions can be freely mixed with matrices in calculations. For 1D (time) manipulation there are also 1D transformations implemented.

3.4.1 Points as colors

Besides as geometrical coordinates, Real3 can also be interpreted as representing colors. In this respect there are a couple of functions: `rgbtohsv`, `hsvtorgb` and `nyuvtorgb`. The first two are usual conversions for rgb and hsv (hue, saturation, value) representations, except that they incorporate a correction to avoid first order discontinuities in hue transitions. The third one interprets its argument to lie in a normalised YUV space, normalised in the sense that each value in the interval `( [0,1], [-0.5,0.5], [-0.5,0.5] )` represents a distinct color.

3.5 Pixels

The implemented pixel type consists of 4 unsigned bytes, red, green, blue and 'alpha'. Coercions to and from Real3 (colors) and Real exist, as well to Complex. The last one is useful for bump maps stored in file. The usual compositing-algebra is implemented [9].

4 Maps

Map is the type of intensionally defined entities. They are functions of one or more of the standard domains to any one of the static $\mathcal{IL}$ types. They are normally not evaluated in the interpreter context but when invoked in some kind of rendering operation, like ray-tracing, image processing or sound synthesis. An exception to this is the function '`evalmap`', which explicitly evaluates a map after setting the domain with '`setmapdom`'. All functions and operators for a certain type are automatically applicable to maps of that type, delivering maps as result. So the syntax for creating maps is exactly the same as for normal expressions. But when one of the arguments is a map, the result is a map, which is a 'compiled' version of the expression instead of the evaluated result.

The type a map evaluates to is obtainable with '`typeofmap`'.

4.1 Map domains

4.1.1 Continuous domains

There are 3 continous map domains, one for each of the relevant dimensionalities. They are available through implicit functions: `m1`, `m2` and `m3`. Some possible interpretations are:

`m1()` global time

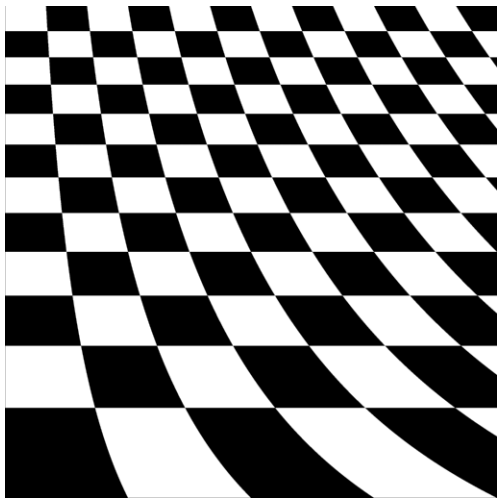

Figure 1: Chess figure

`m2()` normalised object surface for texturing
`m3()` space for solid textures

The following is an example which will produce a chess-board pattern when used as a texture map (operator precedence as in C; `+` higher then `&`):

```
} chs = Int( 8*m2()'0 ) + Int( 8*m2()'1 ) & 1 ? 0.0 : 1.0;
```

Now, `chs` is directly usable as intensity parameter for an attribute in a solid model. In fact, using scalar multiplication of Real2 and because integers are converted to real when used as intensity, it is also possible to write:

```
} d = 8*m2(); chs = Int( d'0 ) + Int( d'1 ) & 1;
```

The following is a function which gives a chess-map with the number of fields as parameter:

```
} chf = "d = $1*m2(); Int( d'0 ) + Int( d'1 ) & 1";
```

This function can now be called with any real value, and the result can be used as a map. Instead of with a plain constant:

```
} vchs = chf( 8 );
```

it can also be called with a map as parameter:

```
} vchs = chf( 4 + 8 * m2()'1 );
```

giving as result figure 1.

4.1.2 Discrete domains

As 1-dimensional domains are normally used as time-representation, there are 3 discrete versions of them, reflecting the different uses of 'sample-time'. These are:

`m1da()` animation time, frame rate
`m1dc()` audio control time, the rate of sound parameters
`m1ds()` audio sample rate

And for the 2- and 3-dimensional cases:

`m2d()` images
`m3d()` volume data

4.1.3 Domain conversion

Most of the time, when mixing discrete and continuous domains, the system does what you would expect it to do, namely, converting to and from the relevant domains. To this end there are some quantities which exists globally as well as associated with individual maps. Their value can be obtained or changed with functions of the same name:

`dbnd2 : Int2` bounds of m2d in relation with m2
`dbnd3 : Int3` bounds of m3d in relation with m3
`frqa : Real` framerate, default 25
`frqc : Real` audio control rate, default 100
`frqs : Real` audio sample rate, default 44100

So when, for instance, you multiply a (discrete) image with a (continuous) noise, the noise function is automatically sampled (evaluated) at points corresponding with the image pixels, converting the image space to the square unit interval. Apart from this there is a complete generic way to switch domains with '`trfmap`'.

```
} m1 = trfmap( m, d )
```

The result is the map `m`, but with `d` as new domain, where `d` can be a map itself. The following example is a function to convert a 2D map to a 3D solid map with a spherical projection:

```
ballprj = "m = m3(); x = m'0; y = m'1; z = m'2;
           u = vatan2( y, x );
           r = hypot( y, x );
           v = 2 * vatan2( r, -z );
           trfmap( $1, real2( u, v ) )";
```

`$1` is a 2D map, dependent on `m2`. So, the second parameter of `trfmap` is of type `Real2`. This expression is the new domain. It is solely dependent on `m3`, so will be the result map. The trigonometric functions with `v` as prefix are normalized versions, in the sense that their values and arguments are supposed to lie in the interval [0,1].

When maps on discrete domains are evaluated on continuous domains (perhaps as an automatic step in a conversion to another discrete domain), the values are to be interpolated. The way this is done can be set with '`polmode`'.

- 0 — constant interpolation (no interpolation)
- 1 — linear interpolation
- more modes can be implemented

For the linear case the function '`linpol`' must be implemented for the relevant types as a function that interpolates between two values of that type according to a third real argument in [0,1]:

```
} linpol( 4.2, 5.7, 0.4 )
 4.8
```

`linpol` is implemented for: `Real`, `Real2`, `Real3`, `Pixel`, `Complex`, `Quat`, `Transform`.

4.2 Modifying maps

A map can always be looked upon as a function with a number of maps or values as arguments. With the function '`setmap`' each of these arguments can be replaced with another value or map at a later instant. The third argument of `setmap` is the index of the parameter of the map that is to be replaced. The default is 0 (for the first argument).

```
} mp = m1() / 2;
```

Now `mp` is a map with two arguments, the first being `m1()`, the second (with index 1) the constant 2, coerced to `Real 2.0`. With the index operator `@` these arguments can be extracted.

```
} mp @ 1
 2.0
```

After:

```
} setmap( mp, mr()'1, 1 );
```

`mp` is defined as:

```
} mp = m1() / mr()'1;
```

5 Extensional rendering

5.1 Image rendering

An image can be generated with:

```
} im = writeimage( fname, m )
```

`fname` is a filename to be associated with the image. `m` is a map, probably dependent on `m2` and/or `m2d` of the pixel type, or coercible to it. the resulting map im will be a `m2d` dependent Pixel valued map. Here is an example to distort and artificially color an image. (figures 2 and 3)

(`/*` at the begin of a line begins a comment until the end of the line)

```
/* open a stored image
   imag = openimage( "screamhead" );
/* get the image in memory for efficiency
   readimage( imag );
/* cast the pixels to real valued colors
   x = Real3( imag );
/* set the global image-size to the size of this one
   dbnd2( dbnd2( x ) );
/* get the intensity value (eq 0.299 * x'0 + 0.587 * x'1 + 0.114 * x'2)
   y = yval(x);
/* make a function to add a 2D-noise distortion to the domain
```

Figure 2: screamhead

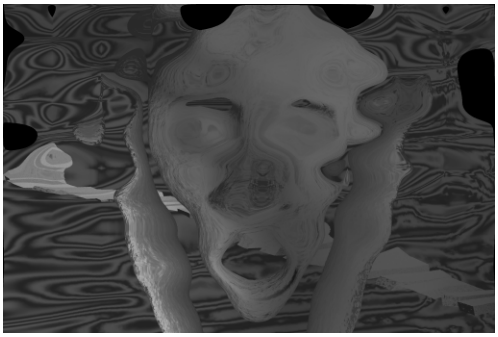

Figure 3: distohead

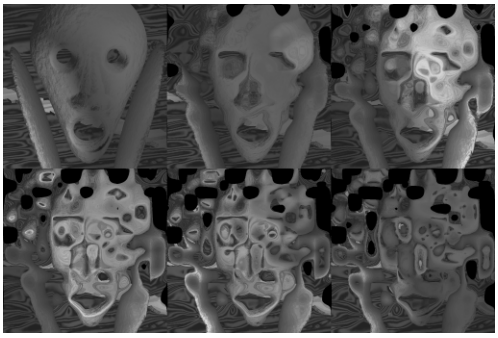

Figure 4: anihead

```
   distm = "m2() + $1 * dnoise( m2(), 0.1 )";
/* a function to distort an image, the image and the amount of distortion
/* are the parameters
   disto = "trfmap( $1, distm($2) )";
/* a function with hue as parameter with the intensity also determining
/* the color saturation
   huecol = "hsvtorgb( real3( $1, y, y ) )";
/* the actual map for the new image;
/* the hue is produced by a noise.
/* the distortion is made proportional to the height in the image
   dis = disto( huecol( noise( m2(), 0.9 )), 0.1*(m2()'1) );
/* render the resulting image
   writeimage( "distohead", dis );
```

If m is also time dependent, the following function can be used:

```
} anim = writeanimage( fname, m, startfr, endfr )
```

An animated sequence of images will be generated, from frame number `startfr` to frame number `endfr`. So if we parameterize the process above:

```
disf = "disto( huecol( $1+noise( m2(), 0.9 )), $2 * m2()'1 )";
writeanimage( "anihead", disf( m1(), 0.5*m1() ), 0, 25 );
```

The result is an animated coloring and distortion. (figure 4 gives frames 0, 5, 10, 15, 20 and 25 of the resulting sequence)

5.1.1 'Modeling' images

To render an image which is a combination of a (large) number of linearly deformed, likely smaller, images, the function `rimage` and it's animated form `ranimage` are effective solutions. Instead of one image-defining map, as `writeimage`, they take a list of image definitions as argument. Each element of this list is a pair (2 element list) consisting of a 2D transformation and a map.

```
po = real2( 0, 0 );
pm = real2( 0.5, 0.5 );
tf = "scale( 0.5, po ) * rotate( frandom( -pi, pi ), pm )";
```

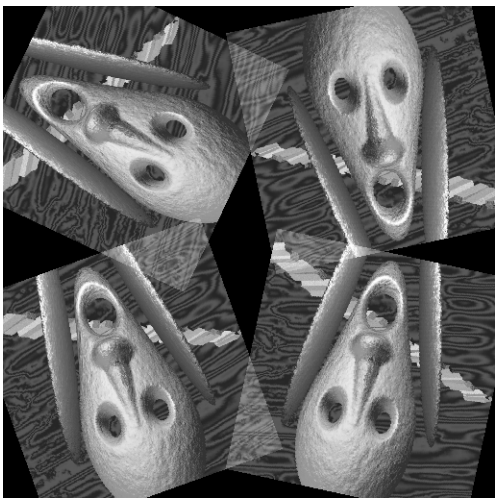

Figure 5: fourhead

```
t0 = tf();
t1 = translate( 0.5, 0   ) * tf();
t2 = translate( 0.5, 0.5 ) * tf();
t3 = translate( 0, 0.5   ) * tf();
iml = [ [t0, imag), [t1, imag), [t2, imag), [t3, imag) );
rimage( "fourhead", iml );
```

(figure 5)

5.2 Imposing hierarchy

The principle from the previous section of dividing the domain in linear deformed subparts, is even more essential for imposing a hierarchy on the time domain. First it must be made clear that the list of 2D intervals, as used in `ranimage`, can form the outcome of a hierarchic structure of transformation combinations. It is not the purpose of the language to impose such a structuring organisation; on the contrary, it's aim is to provide enough elements for the user to implement any kind of structuring she wishes. The principle characteristic of the use of 2D transformations is their ability to represent any combination of linear deformations, and the possibility to invert them for the efficiency of the rendering process. This makes it possible to sort the intervals in their resulting order. In the case of time, one expects to be able to organise dependencies in a non-linear fashion. To be able to do so, the structure of a so-called timeframe is recognised in the language. It is defined as follows:

A timeframe is a list. If `nil`, it is equivalent with time as defined by `m1()`. Otherwise it is to be interpreted in the time as defined by it's tail, which is a timeframe.

This interpretation is defined by it's first element, which is a list of one or two elements. The first one is a 1D transformation, defining an interval. The optional second element is a `m1` dependent, `Real` valued map, which defines a deformation function of the time on the established interval. This function has to be monotonic on the unit interval, in order to make it possible for the system to numerically invert it, and to make sure that subsequently defined

intervals remain intervals.

5.3 Sound synthesis

In analogy with `rimage` there is a function implemented to calculate a 'sound image'. The result is a Real valued, `m1ds` dependent, map, generated from a number of `m1` and `m1ds` dependent maps. This function is '`raudio`' (render audio) and its arguments are a filename and a list. (the possibility to generate multi-channel audio is disregarded for the moment)

```
} m = raudio( name, lau );
```

The role of the 'interval defining' transformations in `rimage` is taken over by timeframes. Apart from sound-generating (`Real` valued) maps, it is also possible to indicate 'programmed' executions of the `setmap` mechanism. As sound synthesizing maps can be memory intensive, e.g. when involving delay units, it is not feasible to define a map for each interval. An alternative option is to use a map for a hierarchically covering interval and changing it's parameters over the smaller intervals. In this context one can think of the map as a musical instrument and playing the notes as varying parameters. The local time inside the specified time-interval is available with the implicit map function `lt`. This is just for ease of use, because any timeframe can be evaluated from anytime else by the map function `tfmap`, with the specific timeframe as parameter.

There is a number of functions dedicated to special sorts of sound synthesis. A simple example is '`fsin`', a sine generator which takes a sample and a frequency as argument.

```
} fmp = fsin( m1ds(), frq )
```

is an efficient alternative for

```
} pi2 = 2*arg(-1);
} fmp = sin( pi2*frq*m1() );
```

The following is an implementation of a FM synthesizer [2]. The arguments are a single carrier frequency and a list of modulators, each consisting of a modulating frequency and a modulation index.

```
fmgen = "fsin( m1ds(), rfmgen( $1, $2 ) )";
rfmgen = "carrier = $1; modlist = $2;
          modlist == nil ? return( carrier );
          modfreq = modlist'0; modidx = modlist'1;
          rfmgen( carrier + modidx*modfreq*fsin( m1ds(), modfreq ),
                  modlist@2 )";
```

5.4 3D Rendering

The logical equivalent of the sound and image generating primitives discussed so far, is the generation of a 3D (voxel) data set like the ones used in medical and scientific visualisation. This is done with the function `r3D`, again with a filename and a list. Each element of the list is a pair consisting of a 3D transformation and a map. The animated counterpart is `rani3D`.

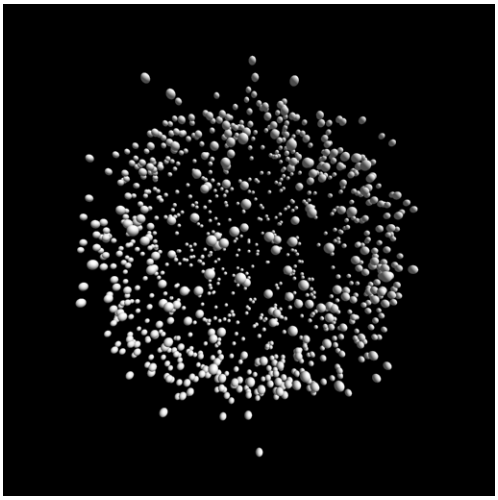

Figure 6: generated density

6 Projection rendering

The methods of modeling and rendering as employed in the language will not be described in detail in this paper. $\mathcal{IL}$ currently works with a ray-tracer and associated CSG (Constructive Solid Geometry) models. Important is, that surface and volume properties are implemented in a 'shade tree' fashion [3], with the possibility to map any parameter. So any defined map of (or coercible to) the proper type, defined in the language, can be used in a model as e.g. a texture definition. Also at any hierarchic place in the model where a transformation is expected, a time dependent map is recognised for animation purposes.

The render module is able to visualise generated voxel sets. Also it is possible to render continuously `m3()` based maps in a 'hypertexture' manner [8] or by generating iso-surfaces.

7 Density simulation

This is an example how new functionality can be added under the intensional paradigm. The function `density` generates a list of coordinates, distributed over the proper dimensionality, according to a `Real` valued map, which is interpeted as representing the required density in space. The coordinates are generated by a simulated Poisson process and integration of the density function over the proper domain. The arguments of `density` are the density map and the interval where the simulation has to take place, in the form of a line representation.

```
xyz = m3();
r = hypot( xyz );
r = r > 1.0 ? 1.0 : r;
box = real3(-1,-1,-1) |+ real3(1,1,1);
pl = density( 3000*sin(pi*r)^50, box );
```

After this, `pl` will be a list of `Real3` points, grouped around the origin in a shell (figure 6).

8 Differential equations

Another example is solving an ordinary differential equation with initial value. For this purpose a new primitive map is available; `v3` is a function giving the velocity vector as a 3D domain. It is now possible to formulate the general set of differential equations:

$$\dot{x} = f(x, v, t)$$
$$\dot{v} = g(x, v, t)$$

Let `m3()` represent x, `v3()` represent v and `m1()` represent t, then the functions f and g can be formulated as maps `mv` and `ma` resp. The solution can be gotten with:

```
} mp = integrate( ma, mv, x0, v0 );
```

Where `x0` and `v0` are `Real3` constants indicating the initial values. The result `mp` is a `Real3` valued, `m1da` dependent map giving the solution for x.

Two special versions are recognised:

```
} mp = integrate( ma, x0, v0 );
```

for

$$\dot{x} = v$$
$$\dot{v} = g(x, t)$$

and

```
} mp = integrate( mv, x0 );
```

for

$$\dot{x} = f(x, t)$$

9 Conclusions

Intensionality is explained as a principle by which most multi-dimensional data manipulating activities can be unified. An implementation of this paradigm is discussed in the context of an interpreted functional language, in which many common problems are consistently solved; such as conflicts between continuous and discrete domains, and combinations of discrete domains of differing sizes. A formalisation of a class of rendering processes is established which extends over all common phenominal dimensionalities. Also the connection with standard 3D modeling, visualisation and animation is explained. This may lead to new insights in the linking of procedural methods for texturing, animation and sound synthesis.

References

[1] Blinn, James F. Simulation of wrinkled surfaces. *Computer Graphics*, 12(3):286–292, August 1978.

[2] Chowning, John. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the Audio Engineering Society*, 21(7):526–534, 1973.

[3] Cook, Robert L. Shade trees. *Computer Graphics*, 18(3):223–231, July 1984.

[4] Hanrahan, Pat and Lawson, Jim. A language for shading and lighting calculations. *Computer Graphics*, 24(4):289–298, August 1990.

[5] Kajiya, James T. The rendering equation. *Computer Graphics*, 20(4):143–150, August 1986.

[6] Peachey, Darwyn R. Solid texturing of complex surfaces. *Computer Graphics*, 19(3):279–286, July 1985.

[7] Perlin, Ken. An image synthesizer. *Computer Graphics*, 19(3):287–296, July 1985.

[8] Perlin, Ken. Hypertexture. *Computer Graphics*, 23(3):253–262, July 1989.

[9] Porter, Thomas and Duff, Tom. Compositing digital images. *Computer Graphics*, 18(3):253–259, July 1984.

[10] Shoemake, Ken. Animating rotation with quaternion curves. *Computer Graphics*, 19(3):245–254, July 1985.