Dušan Bezák

Mass Scenes Rendering Framework

Master Thesis

Advisor:
Rastislav Královič

April 2001
## Contents

1. **Introduction** ................................................................. 3
2. **State-of-the-Art** .................................................................. 4
   2.1 Typical mass scene......................................................... 4
   2.2 Instantiating an object ................................................... 4
   2.3 Referencing an object...................................................... 4
   2.4 Special modeling techniques.......................................... 5
3. **Proposed Solution** .......................................................... 6
   3.1 “Parametrically describe” paradigm .................................. 6
   3.2 POV-Ray platform.......................................................... 6
4. **Parametric Object Modifications** ........................................ 8
   4.1 POV-Ray’s random streams............................................ 8
   4.2 Extended random function ............................................. 9
   4.3 The makevalue function syntax...................................... 10
   4.4 The algorithm behind.................................................... 11
   4.5 Alternative................................................................. 13
   4.6 The alternative directive syntax ...................................... 14
5. **Layout Techniques** .......................................................... 16
   5.1 Layout inside an object .................................................. 17
   5.2 Layout on top of an object .............................................. 18
   5.3 Layout on the surface of an object ................................. 18
   5.4 Layout parameters........................................................ 20
   5.5 The macro specification............................................... 21
   5.6 Layout command syntax ............................................... 22
     5.6.1 layout_inside .......................................................... 22
     5.6.2 layout_on_top .......................................................... 23
     5.6.3 layout_surface ........................................................ 23
     5.6.4 layout_surface2 ....................................................... 23
     5.6.5 layout_count ............................................................ 23
     5.6.6 layout_retries ............................................................ 23
     5.6.7 layout_adjust_normal ............................................... 24
     5.6.8 layout_distance ........................................................ 24
     5.6.9 layout_inside_check .................................................. 24
     5.6.10 layout_overlap_test .................................................. 25
     5.6.11 layout_random_stream ........................................... 25
     5.6.12 layout_regular ........................................................ 25
     5.6.13 layout_regular_range ............................................. 26
     5.6.14 layout_sequential .................................................... 26
     5.6.15 layout_size ............................................................. 26
     5.6.16 layout_surface2_accuracy ....................................... 27
     5.6.17 layout_wrap ............................................................ 27
   5.7 The layout global variables .............................................. 27
   5.8 The layout return variable .............................................. 27
6. Complexity and Time Issues .................................................................29
   6.1 Time complexity ............................................................................. 29
   6.2 Memory consumption ................................................................. 30

7. Conclusions and Future Work ..........................................................32
   7.1 Drawbacks ................................................................................... 32
   7.2 Future work ............................................................................... 32
   7.3 Project’s destiny .......................................................................... 32
   7.4 Portability .................................................................................. 34

8. Examples ..........................................................................................35
   8.1 Möbius strip ............................................................................... 35
   8.2 Dwarves mass scene ................................................................. 35
   8.3 Hollow cheese .......................................................................... 36
   8.4 Martians on Mars ..................................................................... 37
   8.5 Flock of birds ........................................................................... 38
   8.6 Mosaic effect ............................................................................ 39

9. References .....................................................................................40
1. Introduction

The computer-based creation of realistic images often requires to generate large amount of similar objects – a mass scene. A lot of very specialized tools have been already developed to cover some particular mass scene types. For a general mass scene, the common modeling and rendering tools offer only the support to copy or clone a specific object and this way to create large amounts of the objects that have to be manually altered to gain some realistic looking scene. There is a serious need to implement some new functionality to simplify the mass scene creation in general.

The basic scene creation process that is provided by the current modeling tools highly depends on the creator’s patience, because he has to act responsibly to all of the elements in the mass scene and he has to do a lot of things manually. This paper suggests an alternative approach. By providing a parametric description of the models, it is possible to generate many different objects (instances) automatically, not by manual cloning and applying changes.

The second step is to put these objects into the virtual scene. Different techniques for automatic object positioning (layout) are proposed. The techniques cover all possible mass scene types, depending on the dimension of the layout problem. The three-dimensional problem means to distribute the automatically generated objects into a given 3D area. The two-and-half dimensional mass scenes are the most common ones – the distribution of objects on a given terrain. The last one, in our terminology a two-dimensional problem is to distribute the generated objects on a surface of a given object.

All the layout techniques use the basic rendering capabilities that each ray-tracing tool provides – to ask if a given point is inside an object and to cast a ray and ask where the object has been hit. The value of this paper is that it uses this functionality in an unconventional manner; it proves that the ray-tracing capabilities could be used also for the scene creation, not only for the rendering.

The results are tested on the Persistence of Vision Ray Tracer (POV-Ray) platform. To fulfill the desired tasks, additional commands for the POV-Ray scene description language have been created. A high degree of the commands’ parameterization has been achieved. Lot of the proposed parameters bring very interesting and useful scene composition amendments. Many testing scenes have been composed and the proposed techniques served perfectly.
2. State-of-the-Art

One of the goals of current computer graphics is to produce images with as much realism as possible. To achieve this, there is not only the need to improve old (or invent new) rendering techniques, but also to enhance the scene. In the recent years hard work has been done to make the modeling tools as comfortable and easy to use as possible. An experienced graphic designer is capable of creating a 3D model of almost any shape he can imagine. However, what good is an excellent three-dimensional model if it is in the scene alone? To create a realistic-looking scene it is necessary to place many entities at proper places. It is the scene composition that often leaves in the spectator the biggest impression. A brief glance at what support is available to the scene designer to make a scene with many objects constitutes an opinion that a mass-scene creation and rendering tool would improve both the quality and the comfort of a 3D-scene design.

2.1 Typical mass scene

Typical scenes that have been considered during the design of the mass-scenes creation tool include: characters in an audience, a flock of birds, a lawn with blades of grass, trees in a forest, a hairy monster, stalagmites in a cave, cars on a parking place.

If the scene author has enough time and patience, he might be able to do a very nice mass scene without any special instrument, but with a great effort. Current modeling tools provide only two techniques to clone an object: instantiating and referencing the particular object. Although they are quite good to model small quantities of objects, they are not sufficient for mass scenes. The aim of this paper is to make the scene creation work much easier – what can be made by the computer should be done by the computer without any painful, long lasting or boring human assistance.

2.2 Instantiating an object

Instantiating is one of the basic operations that are in almost all software products. It is often called “copy & paste”. This facility makes possible to create a lot of instances, to place them into the scene and later change their properties. Some modeling tools provide functions to make the placement automatically – in a row, on the circle, or even some more sophisticated layouts as a spiral, a helix or some grids.

2.3 Referencing an object

Referencing is almost the same as instantiating. The only difference is that if an object is cloned as a reference, after making a change in the original object, all the references are changed too. This has a big advantage if there are changes to be made after the objects have been cloned. E.g. the author forgot to add eyes to a creature that has already been cloned hundred times. After adding the eyes to the master (original) object, all the slaves (references) receive their own eyes immediately.

These two techniques have one main disadvantage – all the changes to the clones are to be done by the designer. The computer only makes the clones that are exactly the same as the original. However, for a realistic look the instances must differ in some details.
Due to the lack of time or patience, the author of the scene alters only a scarce number of the instances. At the first glance the result seems to be quite nice and fulfilling the intention, but after a short observation the objects seem to be somehow periodical and disturbingly regular, the scene is not as realistic as should be.

2.4 Special modeling techniques

A lot of hard work has been devoted in the recent years to some special types of mass scenes and parametric descriptions of objects. The results have usually the same basic scheme – the object to be modeled is considered from two points of view, the first is how an instance of the object can look like, and the second is how the instance behaves in an area or as a part of a whole entity. Then a tool is developed that confirms the assumptions. An example of such paper is [1] where the authors studied plant ecosystems and then rendered scenes consisting of thousands of generated plants in amazing photo-realistic quality. Another example is the modeling of human hair [2] by making a physical model of human hair, and thus generating the whole head.

Also the movie producers have their own special modeling tools that are many times developed exclusively for a desired type of scene. For instance the famous company Pixar developed in 1998 for the movie A Bug’s Life new methods for modeling and animating large crowds of characters, but they keep their technical information in secret. The only know information is that it is an extension of the Renderman standard [3].

Nevertheless, all these modeling techniques are quite specialized for a particular type of scene. This paper introduces a technique that is not dependent on the scene type; it is a tool that is applicable on most mass scenes. On the other side it cannot compete (in the image quality) with a very special modeling tool, but this small handicap is counterbalanced by the generality of the approach.
3. Proposed Solution

To fulfill the planned, an environment that permits the creation of perfect 3D models is necessary. It would be of no reason to enhance a technology that has some obsolete drawbacks. We are trying to increase the realism of what today’s computer graphics is capable to do – we want to build on a stable foundation.

3.1 “Parametrically describe” paradigm

The idea is to change the standard clone-modify paradigm to a new one: parametrically describe. The scene designer really needs only to provide the parametric information about the object. Then the computer is able to automatically generate clones of the object according to that parametric information. This scheme enables to create truly mass scenes with vast number of similar objects. Still no two among them will be identical and if they are all put into one scene, the scene could look realistic.

The scope of this paper is focused on still images, not animations. This assumption has been made to fixate on the image quality and not on the mutual object interaction throughout the time. Even though it is possible to create animations using all the techniques proposed, and some animation-supporting features have been created. The creation of so-called static animations – when the objects are only moving on their place is really recommended. For instance if the objects are some characters, they could freely blink, rotate their heads or clap their hands.

While we were choosing the right platform satisfying our needs, we had to keep a lot of aspects in sight. The data describing the models should be extensible to keep the information including what and how could be modified in the cloned model, what are the parameters that the model must comply. The scene composition should be easily regulated. The environment should provide fast and acceptable graphic output. Moreover, there is the demand for a cheap availability of the environment. With these requirements on mind, POV-Ray seems to be an ideal platform for our purposes.

3.2 POV-Ray platform

Because the Persistence Of Vision Ray tracer (POV-Ray) is a source-free application, it cannot be cheaper and it cannot have better availability. For modeling it uses the POV-Ray Scene Description Language [4] that is quite general and allows to simply add new attributes, properties, directives, commands or functions. Binaries and sources of POV-Ray can be found and downloaded for free from the Internet [5]. Nevertheless the quality of POV-Ray’s graphic output is comparable to the best commercial rendering engines.

Our mass scene creation extensions of the standard scene description language could be divided into two categories:

- One object creation – parametric description of 3D model that allows to generate high number of different objects automatically.
- Scene composition – the layout of the objects into the scene. The objects’ distribution should be natural and realistic. On one side it means that it should
be somehow randomly generated, but on the other side also somehow parametrically described to satisfy the given requirements.

For the first category one helper function (*makevalue*) and one directive (*alternative*) have been designed, for the scene composition a new adjustable command (*layout*) is introduced. All these commands are explained in the following two chapters in comprehensive detail.

The expandability of POV-Ray’s functionality is based on the main program’s source code modifications and not on any plug-in technology that is usual in relationship with other modeling tools. The idea is that a developer can download the sources for free, he can study them and see how does it work, and add any functions anywhere, with only one condition – it is a copyrighted freeware and the copyright notice must be left unchanged. Now speaking for my own person, I have to recommend to everyone at least a short look at the source codes, because I have to say that I have learned a lot of programming techniques and much about how to implement some computer graphics algorithms. Making the extensions has been a hard task, because all three commands are totally different; one is a function returning a number, the second is a directive that could occur at any place during the scene source parsing, and the last one is actually an object that could be placed into the scene.

The decision to use POV-Ray is not crucial in any way. It has just been chosen as a preferred platform and served well. We have not intended to make a practical extension of any particular existing software, but to consider mass scenes creation as a process that could be automated. The implementation has been used especially to test the proposed techniques. Almost all the newly introduced commands, results, and methods are applicable for any extensible 3D modeling software product. We suppose that the greatest value of this master thesis is in the algorithms we invented, and the practical realization is just a proof of their applicability. More discussion about the portability of the ideas presented herein is in the chapter Conclusions.
4. Parametric Object Modifications

The basic support for parametric object description comes from the standard POV-Ray’s scene description language. The workflow of POV-Ray image creation is: create the scene source file (a text file), run the parser (that creates internal rendering structures – the scene), start the rendering engine. This allows the straightforward design of parametric objects using macros. A macro represents the objects to be generated many times in the scene. To let them all have different color, the only task is to add the color choice to the macro. The object’s macro is parsed several times and each time a different (random) color is chosen.

4.1 POV-Ray’s random streams

POV-Ray has a simple, and a comfortable built-in support for random values generation. There is a pseudo-random function that uses a well-known algebraic algorithm. On the input is a 32-bit integer number – the last random number generated, and on the output the function generates another 32-bit integer number that will become the input for the next function call. The function body is simple – multiply the input by a well-chosen constant and add another suitable constant. All operations take place in modulo $2^{32}$ arithmetics, which is quite common for 32-bit computers. The particular well-chosen numbers in the functions are 1812433253 and 12345. The proof of their correctness is not important now; assume that the function produces cyclically all 32-bit numbers with desired randomness. The function actually returns a floating-point number in the range between 0.0 and 1.0 (inclusive). The transcript of the random function looks like this:

```c
next_random = prev_random * 1812433253L + 12345L;
prev_random = next_random;
return ((double)next_random) / 0xffffffffUL;
```

The initialization of the stream of random numbers is very important. The numbers should be both random as well as deterministic. The reason is that during the scene preparation if a random generation of some objects is applied, after which the scene author is satisfied with the randomly generated result, the result should remain as it is from that moment forever. Another reason is the animation stability. The POV-Ray is able to automatically re-execute the rendering of multiple images that comes from the same source code with only one little change – a change of the variable named clock. The scene author could define the range and step of this clock variable. For example if the animation should last five seconds at the rate of ten frames per second, the clock starts at 0.0, the step is 0.1 and it ends at 4.9. The source code of the scene should modify the objects’ locations, shapes or any other properties according to the clock. The term animation stability stands for the sameness of the randomly generated attributes in all frames of the animation sequence. This is a vital feature – we do not want to have totally different scenes at various animation frames.

The POV-Ray’s random streams assure this determination of the random number sequences. Why there are more such streams and why is one not enough? The answer is
easy – if we want to turn on or off some features of the scene. For example if the random values are used both to generate locations of some objects and to generate the colors of the objects. After the source code of the scene is complete, comes the search of a good random number initializer – the value that starts the random stream. It is possible to find the initializer for the random locations (by changing it until we are content), it is even possible to find the initializer for the colors, but it is almost impossible to find an initializer for both of them at the same time. Another reason for multiple random streams is that we might want to add some new feature to the generated objects (e.g. a randomly generated size), if there was only one random stream, we will loose the desired and hardly found random locations, because the randomness of the locations depends on the order of the random function calls. Multiple random streams solve all these drawbacks. In the mentioned example we will use separate random streams for the locations, for the colors and for the sizes. A change of the color’s inializer will not alter the locations and sizes of the objects.

The random streams are allocated by a call to the POV-Ray function seed(). The function returns the identifier of the stream that should be placed as an argument to each call of the rand function. Here is an example of how to allocate a new random stream with the initializer 123 with an exemplar call to the random function:

```cpp
#declare stream = seed(123);
#declare value = rand(stream);
```

### 4.2 Extended random function

The production of large amounts of different objects cannot be done without a better random function. The POV-Ray’s built-in random function generates random numbers in the range between zero and one with constant density function; that means each number has the same probability to be chosen. In the real life no measurable parameter of almost any object has such distribution [6]. The basic distribution is the so-called normal distribution, where the density function is the gaussian curve. If the parametric definition of a 3D character requires to generate a number representing the character height, it is much better to use a random number generated with the normal distribution instead of the standard random function. Maybe there is no reason to do such (a bit more complicated) calculation during the generation of only one instance of the object, but it is a real requirement for mass scenes.

What if the scene author wants to have a few small persons and a lot of tall ones? The normal distribution cannot be used. Still there is no reason why the author should not enter the density function that satisfies his intention. That is exactly the first extension of the POV-Ray scene description language – a function that generates a random value according to a given density function and its placement (the center and the dispersion).

To enter the data describing the density function into the POV-Ray scene description language, an array comes to good use. The values in the array represent the density function ordinates; the abscissas are equidistant. Two examples are given – a constant density function, and a gaussian-like density function:
#declare constant_density = array[2] {1.0, 1.0}
#declare gaussian_density = array[9] {0.0, 0.1, 0.5, 0.9, 1.0, 0.9, 0.5, 0.1, 0.0}

The code requesting a random value having the gaussian density function (according to the above example) applied to the height of a human – centered to 170 centimeters and with the dispersion of 30 cm will look similar to this one:

```plaintext
#declare height = makevalue(gaussian_density, 170.0, 30.0, random_stream);
```

Figure 1: The gaussian density function data.  
Figure 2: The makevalue function empirical data.

Where could the generated numbers be used? Anywhere – for sizes of the objects, for color values, for position coordinates, for pattern modifications, anywhere where a random value is useful.

4.3 The makevalue function syntax

This section contains the syntax of the makevalue function with all the information the user needs to use the function in his self-made code. The makevalue function returns a float number – a random value according to its parameters. The schematic syntax is:

```plaintext
makevalue(  
    <statistics>,  
    <central value>,  
    <dispersion>,  
    <random stream>  
)
```

The statistics parameter is an array representing the density function. It is a linear (one dimensional) array containing normal floating-point numbers. The abscissas of the density function are considered to be equidistant; the ordinates that are actually the numbers in the statistics array could be any non-negative numbers. To create such array from a given density function (called df) a simple POV-Ray code could be used:
# declare nSamples = 10;
# declare stat = array[nSamples];
# declare i = 0;
# declare startingAbcissa = -1.0;
# declare endingAbcissa = 1.0;
# declare abscissaIncrement =
  (endingAbcissa - startingAbcissa) / 
  (nSamples - 1);
# declare abcissa = startingAbcissa;
# while (i < nSamples)
  # declare ordinate = df(abcissa);
  # declare stat[i] = ordinate;
  # declare abscissa = abscissa + abscissaIncrement;
  # declare i = i + 1;
# end
// the stat array contains the desired data

The central value is the value to which this statistics is centered. The dispersion is exactly one half of the width of this statistics. To place the statistics to a desired place, count these values and provide them as the arguments of the makevalue function. The values generated by the makevalue function are in the range <central value – dispersion, central value + dispersion>. There could have also been the possibility to provide for example the lower value and the width of the statistics, but we thought that it would be only confusing and that this type – the centering of the density function is the most useful.

Figure 3: The explanation of the makevalue function parameters – the central value and the dispersion

The last parameter ‘random stream’ indicates the random stream that is used to generate the desired value. The necessity to use the streams for random number generation has already been described.

4.4 The algorithm behind

There has been no special requirement on the integral of the density function. Normally (in most mathematical statistics theories) that integral should be equal to one, because it is expected that the union of all disjunctive events that could occur has complete probability. In other words the sum of all probabilities of the disjunctive events should
be exactly one. We do not need to require such user discomfort. If the integral is not equal to one, it can be automatically scaled to fit that assumption. The first step of the algorithm is to count the integral of the whole density function. This can be made directly by a linear cycle stepping through all the abscissas, counting the small areas of the trapezoids. If we have the integral value, we might consider the integral of the density function to be equal to one by scaling the density ordinates using the inverted value of the acquired integral.

The second step is to generate a standard random number (called $r$) in the range between zero and one inclusive. The number $r$ will determine the integral of the density function that is determined by the sought random number $c$. In the mathematical language:

\[
\int_{a}^{b} df(x) \, dx = r \int_{a}^{b} df(x) \, dx
\]

The most important last step of the algorithm is to find the value $c$ in the above equation, where all other data are given. It cannot be done mathematically, because we do not know the inverse function to the density function $df$. The density function has to be divided into the segments determined by the abscissas where the function becomes linear and easily integrable. The algorithm comes straightforward – having the value $r$, overstep the trapezoid between two abscissas if its area is smaller than the $r$. The step consists of two operations – increase the abscissa by the step between two abscissas and decrease the $r$ by the size of the trapezoid’s area. Do these steps in a cycle until an area larger than the $r$ is found. Then the problem is transformed into an algebraically solvable equation

\[
\int_{a_i}^{c} df(x) \, dx = r \int_{a_i}^{a_{i+1}} df(x) \, dx
\]

where $a_i$ and $a_{i+1}$ are the neighboring abscissas, $r$ is the original random number decreased by the trapezoidal areas of previous abscissas, $df(x)$ is the linear density function, and $c$ is the variable we are searching for:

![Figure 4: The easily solvable case – linear density function on one segment.](image)

The value of $c$ could be expressed by the following equations (the substitution of the integral’s density function by the respective linear function with the elimination of the right side, and the solution of the obtained quadratic equation):
\[
\left[ \frac{k}{2} x^2 - k a_i x + o_i x \right]_{o_i}^c = \frac{k}{2} c^2 - k a_i c + o_i c - \frac{k}{2} a_i^2 + k a_i^2 - o_i a_i = r
\]

where 
\[
k = \frac{o_{i+1} - o_i}{a_{i+1} - a_i}
\]

\[
d = \sqrt{(o_i - k a_i)^2 - 2k \left( \frac{k}{2} a_i^2 - a_i o_i - r \right)}
\]

\[
c = -\left( o_i - a_i k \right) \pm d / k
\]

The correct \( c \) is that one of the possibly two solutions, that lies between the neighboring abscissas \( a_i \) and \( a_{i+1} \).

### 4.5 Alternative

The second randomization of the macro representing the parametric object comes from the need to make decisions. Suppose, (for instance) that it is needed to decide whether a person will wear short or long trousers. Perhaps it is possible to describe the trousers length by the makevalue function with proper settings, but if there is the need to make totally different objects for the trousers, it has to be determined which type of the trousers to use. An alternative directive is our second contribution to the POV-Ray scene description language.

Usually the scene designer has the feeling of the percentage – the partial amounts of how many of the generated objects should comply a given condition. In the above example it could be stated that (for example) sixty percent of the people would have long trousers, and forty percent the short ones. An alternative directive added into the macro could express it in this way:

```plaintext
#alternative (random_value)
    #case (0.60) longTrousersMacro()
    #case (0.40) shortTrousersMacro()
#end
```

The alternative could be used wherever a decision has to be applied – for color selection, for the choice of object type, to choose any optional orientation. It is advantageous to use the alternative in the macro of the generated object to determine which object should be actually generated. A typical example occurs while modeling a common food – a letter soup, where the soup contains different soup elements:

```plaintext
#macro soupElement()
    #alternative (random_value)
        #case (0.86) letter()
        #case (0.08) carrot()
        #case (0.06) parsley()
    #end
#end
```
Most of the soup elements are the letters – 86 %, generated by the macro named letter. Then there are 8 percent carrots and the 6 percent rest consists of parsleys. The complete soup image can be found in the following chapter.

4.6 The alternative directive syntax

The alternative directive could be used anywhere in the code.

```plaintext
#alternative (<generator>)
   #case (<amount>) data for this case
   ...
   #case (<amount>) data for this case
#end
```

The amounts should be non-negative numbers. The sum of all amounts should be equal to the maximum possible number that is allowed to become the generator. Typical use of the alternative directive generator is a call to the standard random function with an argument of some random stream identification. Notice, that now we prefer a random function with constant probability density function.

There is no need to describe the algorithm that is behind the alternative directive implementation in detail, because there is nothing special about it. Just a few lines had to be added to the parsing mechanism. At the beginning read the generator value in the parameter into an internal variable (let’s call it \(v\)), and iterate through the cases. Each time read the amount and subtract it from the internal variable \(v\). If the \(v\) is positive, pass the case, if not (\(v\) is zero or negative), the case that should be parsed have just been found. Do the parsing of this case and skip all other cases up to the end directive.

The basic usage of the alternative is with the generator in a form of a random value gained by a call to `rand(random_stream)` and the amounts are actual probabilities that the particular case will be chosen. The generator could be simply multiplied by 100 and the amounts could be written exactly as percentages.

To mention a real-life example, we have chosen the pigment randomization – the choice of a color. There are two basically two possibilities, the first one is to call a random function for each of the red, green and blue color ingredients, or the second one – to use the alternative for some specified colors. The first one has an advantage of possibly many different colors; the second’s advantage is that it will generate exactly the desired colors and not some gray mixtures. We recommend using the first one basically for the choice of color tone; and the second one for some predefined colors – e.g. car colors of one manufacturer can only be chosen from a few samples:

```plaintext
pigment{ color
   #alternative (rand(randomColorStream))
   #case (0.1) Red
   #case (0.1) Magenta
   #case (0.1) White
   #case (0.1) Green
   #case (0.1) Cyan
   #case (0.1) Brown
   #case (0.1) Blue
```
One big advantage of the alternative is that it is a directive. It can be used wherever in the code, also breaking the code bounding brackets. Only one rule must be obeyed—the directives cannot be crossed. They could be nested, but a nested directive must end before the outer does. A syntactically remarkable code that follows shows the use of choosing some totally different objects—for example to build up some scene including a heap of garbage and apply an object modifier to all of the possibly generated objects by writing just one line of code—the rotate command at the end. Any other object modifiers could be also there, such as the texture, finish, normal definitions, object matrix transformations (translate, scale, rotate), and others.

```csharp
#macro any_element(size)
#alternative (100*rand(randomStream))
#case (30)
sphere {
 <0,0,0> size/2
 #case (25)
torus {
   size/2 size/5
   rotate x*90
 #case (20)
 box {
   <0,0,0> <size,size,size> translate -size/2
 #case (10)
 cone {
   <0,-size/2,0> size/1.5 <0,size/2,0> 0
 #case (15)
cylinder {
   <0,-size/2,0> <0,size/2,0> size/2
 #end
   rotate y*rand(random_stream)*180
 } #end
```

15
5. Layout Techniques

A lot of mutually different objects could be generated with a great help of the functionality described in the previous chapter. A unifying macro could be written that is able to generate all of these objects with desired abundance. In other words, all of the objects could be generated by a single macro that will be invoked several times. Each time the macro is invoked (and parsed) a new so-called macro object is generated.

The object’s 3D model is generated in its own coordinate system. The center of the coordinate system (the point with coordinates \( <0, 0, 0> \)) is considered to be the origin of the model’s space. We agreed to a convention that the objects to be generated into the space (as described later) are offset so that their center meets with the coordinate system’s origin; the objects to be placed on a surface are offset to stand on the origin point. The POV-Ray is using a left-handed coordinate system, with the x-axis heading to the right, the y-axis heading to the top and the z-axis heading towards the back. If we imagine a bounding-box around the macro object, its left-lower-front point having coordinates \( <x0, y0, z0> \) and the right-upper-back point described \( <x1, y1, z1> \), the conventions could be interpreted this way: if the object is meant to be floating in the air, this equation should hold:

\[
<x0+x1, y0+y1, z0+z1> = <0, 0, 0>.
\]

If the object is going to stand on the ground, these formulas should be true:

\[
<x0+x1, y0, z0+z1> = <0, 0, 0>
\]

\[
y1 > 0
\]

The validity of these formulas is only recommended, not necessary. If the origin of the object has improper position offset by a vector \( v \) from the recommended origin, the only difference would be that the resulting scene would have these objects translated by the same vector \( v \) from their recommended locations. A simple example is if we model a mass scene of characters and they have their origin in their center of mass, after placing them on a terrain into the scene they would be ‘plugged’ waist-deep into the ground. The rules are only a recommendation, because it might be useful to have the origin at some different position, e.g. a tree should have the origin somewhere in the middle of the trunk, but above the roots, just where the tree-trunk begins.

The next and quite a difficult task is to put these macro objects into the scene – to find the proper reference points where the generated objects will be placed. A reference point is an expression for the point, which becomes the origin for the macro object in the mass scene. According to our terminology, we are putting the macro objects onto the respective reference points. A general mass scene consists of many objects placed on somehow randomly chosen locations – it is impossible to find any regularity in the distribution of people in a crowd.

Three different techniques have been developed and are introduced in this paper. The techniques are dependent on the dimension of the layout problem. The basic three-dimensional problem means that the reference points should be distributed in some predefined space. A logical question arises: How to describe the given 3D space? The POV-Ray environment gives a straightforward answer: by a 3D object described using the scene description language.
The same idea could be applied to a two-and-half-dimensional problem – to find the reference points on top of a terrain. The description of the terrain could be again any standard 3D object.

Usually (in mathematics) a two-dimensional problem is easier than a more dimensional. In the layout task, the two-dimensional problem means to find the reference points on the surface of any 3D object. It is quite hard indeed, because the 3D models are not described as standard meshes, but they can be any CSG compositions of non-elementary primitives. Anyway, the problem is only named as a two-dimensional. The surfaces we are dealing with, do not have to be planar, they could have any (or almost any) shape.

5.1 Layout inside an object

Given are a 3D object (to become the skeleton for the layout – the so-called layout object) and a parametrical description of the object to be generated many times (the object’s macro – the macro object). The task is to find the reference points in the layout object to place the objects generated by the macro. The ray-tracing capability of POV-Ray gives us the opportunity to call a function inside to ask if a (3D) point is inside the layout object. The algorithm is quite straightforward: generate random points inside the bounding-box of the layout object, for each randomly generated point ask if it is inside the layout object and if the test succeeds, a reference point has been found.

The probability that a random point from the bounding box of an object is inside the actual object equals to the volume of the object divided by the volume of the bounding box. If the object almost fills the entire bounding box space, it is much easier to hit it as if it was only a small slice of something narrow. But the reversed value of the probability fraction represents an average case of how much points must be discarded before a reference point is found. The process of searching and discarding is limited either by the number of macro objects to be placed, or by the number of consecutive unsuccessful tries. The scene creator knows the size and shape of the layout object and he is able to guess this probability number and according to it he can decide to increase the number of tries so that it would become bigger than the reversed probability value.

Figure 5: An example of the layout inside technique based on the idea of Juraj Gottweis. The layout object is a conical object representing the soup. The objects in the soup are automatically generated by the soupElement macro described above.
5.2 Layout on top of an object

The input data specification is the same as in the layout inside, but now the generated reference points should be on the top of the layout object. This is the most common layout request – the basic association when talking about mass scenes is a crowd of people. And people generally stand on a ground. During the scene preparation the author knows the exact 3D model of the layout object (e.g. a street, a forest terrain, etc.) where he wants to have an automatically generated crowd.

Again the ray-tracing capabilities are used. The rays are not fired as usually (from the eye through the projection plane), but they start somewhere above the layout object, and they are heading directly downwards. If the ray hits the terrain object, a reference point has been found to place an object (to use the macro to generate an object automatically).

It might seem to be strange to use the ray casting also in the parsing phase (not only during the actual rendering), but it is again a clever use of a universal technique that should work on all types of objects, because all objects should be able to be visualized using the same ray-tracing methods. The probability for a ray to hit the object is again based on the division of the top-view area by the area of the bounding rectangle. If the fraction in a particular case is too small, an increase of the algorithm retries should be considered.

To summarize the algorithm: find the bounding-box of the layout object, generate random points on the top face of the bounding-box, for each of these randomly generated points make a ray directed downwards (in the proper coordinates it means the vector \(<0, -1, 0>\)), use the ray-casting method to find the intersection of the ray and the layout object. If a hit occurred, the reference point has been found, otherwise repeat the process again.

![Figure 6: Characters generated on the terrain using the layout on top technique.](image)

5.3 Layout on the surface of an object

The input data specification is the same again – the macro of the object to be generated many times, and the object that will serve as the skeleton of the layout (the layout object). The task is to find a random point on the surface of the layout object with constant density function. The constant density function requirement is very important, because there is the need of uniform distribution of the generated reference points on the layout object’s surface. Why? A typical surface mass scene – a hairy monster, could answer this. It is expected that the density of the hairs of the monster is the same
everywhere on its body. If the reference point’s distribution were not uniform, there would occur some areas with higher hairs presence.

An algorithm to find the random reference points on an object’s surface is again based on the ray-casting technique. Two random points on the surface of a bounding sphere are generated, their connection constitutes a ray (that is actually fired from the first point heading towards the second), and the first intersection of the ray with the layout object is taken as a reference point. The reference points do not have necessarily the uniformity property that has been demanded (in all cases when the layout object is not a sphere), but it works quite fast. To obtain the uniform reference points distribution, a simple check for the relative reference point distances can be applied – if a ray in the algorithm has hit a point, it becomes a reference point only if the distance to all other reference points is greater than a given value.

A big disadvantage of this method is that if the layout object has too concave surface (e.g. a vase) the probability of hitting a point on the inner surface of the object is too small. In most situations, this is not a serious drawback because the concave parts of objects have usually limited visibility (if it is hard for the rays to get into the vase, it is also hard to see there). However, to solve this handicap together with the true uniform reference points’ distribution, another method has been developed.

The second algorithm gives better results but is (a bit) slower. It finds the reference points on the surface of any object (or almost any – except some fractal-based surfaces when it is hard to talk about a ‘surface’) with uniform distribution. The idea is that random points are generated anywhere in the bounding-box of the layout object. For each of these points a very small (smaller than the smallest bend of the layout object’s surface) sphere is considered (let’s label the sphere’s radius \( q \)). On the surface of this sphere a random point is chosen (called \( s \); let the sphere center be called \( c \)). Test whether \( c \) is inside and \( s \) is outside the layout object. If not, re-execute the search. Otherwise a ray is fired starting from the point \( s \), targeting the center \( c \). If the ray hits the layout object’s surface on the line between \( s \) and \( c \) (and it should hit, because \( s \) was outside and \( c \) was inside), the point that was hit is a reference point. This algorithm provides reference points with uniform distribution on the surface of the layout object.

It is hard to express the probability of a successful reference point finding. The first algorithm constitutes the ray as a connection of two randomly generated points on the surface of a bounding sphere. Even the question of a truly random point (with uniform probability) on a surface of a sphere is not trivial. The first idea – to create a random point inside the sphere’s bounding cube, make a half-line starting in the sphere’s center crossing that random point and where this half-line intersects the sphere’s surface is a random point on the sphere – is not correct. The uniformity request is not satisfied, because the points round the intersection of the cube’s diagonals with the sphere’s surface have higher probabilities. The reason why is simple – the length of the vector (of points inside the cube) that is able to generate such point (at the corner) is \( \sqrt{3}r \) where \( r \) is the radius of the sphere that is equal to one half of any edge of the cube. And the length of a vector of points that are able to generate the point \( <0, 0, 1> \) is only \( r \). The correction of the random-point-on-a-sphere algorithm is easy, as a modification of the previous: After the random point inside the cube is chosen, a test is made whether it is also inside the sphere. If not, it is discarded and another one is generated again. If it is inside, the algorithm continues as before.
The probability for a random ray to hit the layout object is an average of all probabilities for a random ray with a fixed beginning. And if we are dealing with a random ray, which has its origin fixed to some point (on the surface of the bounding sphere), the probability of successful hit becomes the fraction of the area that is filled by the layout object (in the projection looking from the given point) divided by an area of the bounding circle. Having this on mind, it is easy to determine if the increase of the algorithm retries is rational.

The probability of successful reference point finding in the second algorithm largely depends on the radius of the small spheres – the value $q$. The probability decreases with the third power of the $q$ decrease – if $q$ is changed to its half, the probability changes to its eighth. It also depends on the successful finding of the point $c$ in the inside of the layout object – it has been commented in the description of the inside technique.

![Figure 7: The layout object.](image1)

![Figure 8: A hairy monster. The result of the layout on the surface technique.](image2)

### 5.4 Layout parameters

After a reference point is generated, the object creation macro is called to create the object to be placed at that reference point. Basically, there is nothing that prevents the macro objects from mutual overlapping. An instrument that forbids the overlapping is the minimum distance parameter. If a minimum distance is provided, for each new reference point that is generated, the distance to the nearest reference point is taken (it is the minimum distance to all reference points) – and it is checked if it is greater than the given value. If not, the reference point is discarded and the respective generation algorithm continues. After a given number of unsuccessful retries the algorithm stops with the output that no more reference points could be found. A test if two objects are really overlapping could also be made, but it takes much longer time and is not hundred-per-cent successful (see the description of the layout_overlap_test parameter below). There are also other useful parameters, all of them described later together with their usage.
Another type of parameterization is to allow the object’s macro to be called with additional environment describing parameters. The macro has the possibility to alter the generated objects depending on

- the location in the scene (the reference point). The macro object is able (for instance) to change its color or size according to the location. If the scene is a crowd of people, the persons (or their heads) could be rotated according to their locations to look at some special place (e.g. the audience looking at the ball in a tennis match).
- the ground orientation (the normal of the ground). This parameter describes the elevation of the ground at the reference point. It is useful (for instance) to rotate the hairs of a monster so that the roots of the hairs are perpendicular to the layout object’s surface.
- the color of the layout object at the reference point. This is very useful to generate the macro objects with different behavior at different places in the scene. Typical example is a meadow with flowers, where the flower-types are chosen according to the texture of the meadow base object – if the texture is some gradient transition, one side of the meadow will have a majority of one type of flowers, the other side will have flowers of the second type. It is also possible to alter the height of the grass blades according to a well-chosen texture.

Also one important output parameter has been developed: The macro could output a boolean value determining if it wants to be generated (with the proposed settings at the proposed reference point) or not. For instance in a scene with characters on a terrain it is not very common to have a character on a high slope where the elevation is too big. The object’s macro can check the input parameters and decide that the proposed reference point is not very suitable for the macro object to be generated there. It outputs false and the reference point generation continues with another try.

5.5 The macro specification

The macro generating an object for layout purposes could be any POV-Ray macro satisfying these regulations:

- The macro could have zero or one parameters. If there is one parameter it is considered to be the size of the object to be generated – the size value could be set by the parameter layout_size.
- The macro must generate at least one object. If it does not generate any object the parsing of the macro follows in the code of the layout command and throws an error. If it generates more objects, only the first one is parsed and then the macro is terminated ignoring the rest.
- The macro could read or modify any global variables. Take into account that the #declare directives inside the macro are parsed only until the end of the first object, and then ignored.

In the subchapters that follow there are all possible layout parameters with their usage, defaults, behavior and respective algorithms.
5.6 Layout command syntax

The layout command is actually an object – it is a union of the generated macro objects.

```
layout {
  <object_for_layouting>
  [layout_inside
    layout_on_top
    layout_surface
    layout_surface2]
  [layout_count <count>]
  [layout_retries <count>]
  [layout_adjust_normal <ratio>]
  [layout_distance <distance>]
  [layout_inside_check <number_of_checks>]
  [layout_overlap_test <number_of_tests>]
  [layout_random_stream <random_stream_id>]
  [layout_regular <start> <2-3_vectors>]
  [layout_regular_range <axis>, <from> <to>]
  [layout_sequential]
  [layout_size <size>]
  [layout_surface2_accuracy <accuracy>]
  [layout_wrap]
  <"macroname">}
```

The order of the optional parameters is arbitrary. The object for layout must be the first parameter of the layout object. The macroname string must be placed at the very end of the layout object definition, no other layout specific commands should follow. It is possible to add any object modifier commands afterwards, such as the object matrix transformations (the translate, rotate, scale commands) or the texture commands. For example if we want to do a layout of stalagmites on a floor of a cave, we have an expectation that the stalagmites’ texture will fluently join the floor’s texture without any disturbing transition. This can be easily done this way: the stalagmites macro definitions will not have any textures, the global texture will be provided at the end of the layout command definition that will be inherited by the stalagmites as well as by the floor.

5.6.1 layout_inside

The generated objects will be placed inside the given layout object. This is the default parameter – it is active if no layout type is given. It applies the first algorithm using the inside method on the layout object. It tries to distribute as much macro objects as set in the layout_count parameter, but the actual number may be lower – depending on the other parameters and on the possibility of successive distribution. Notice that only the macro objects’ coordinate system origins are guaranteed to be inside the layout object, the actual models generated by the macros need not always be completely inside. See layout_inside_check parameter for details on the true insideness of the objects.
5.6.2 layout_on_top
The generated objects will be placed on the top surface of a terrain object using the ray-casting method on the given layout object. The algorithm works as in the layout_inside object type, but using different layout technique – trying to distribute as much macro objects as possible, but accomplishing all of the parameters.

5.6.3 layout_surface
The generated objects will be placed on the surface of the given layout object using the ray-casting technique for random rays, that were constitutes as a connection between two random points on the layout object’s bounding sphere. This algorithm is useful especially for objects having inner surfaces that want to be left out of the layout process.

5.6.4 layout_surface2
The generated objects will be placed on the surface of the given object using the second technique – the inside of an object method in combination with the ray casting. It distributes the macro objects uniformly on the layout object’s entire surface, including concave patches or hidden holes inside the object.

5.6.5 layout_count
This constant determines the maximum number of possibly generated objects. The default value is 100. All of the layout type algorithms use an iterative search for suitable reference points. If a randomly chosen reference point is considered to be suitable, the macro of the macro object is invoked and the counter of already generated objects is incremented. If the try has not succeeded, the algorithm has to try again, but increases another counter – the counter of unsuccessful retries. To avoid making an endless loop, a check on the unsuccessful retries counter is made – if it exceeds some given value, the main cycle terminates with the message that it is not possible to generate as much objects as requested.

5.6.6 layout_retries
This number determines the maximum number of tries required to find a reference point. The iterative technique trying to find a reference point will stop if the number of unsuccessful retries reaches the layout_retries number as described above. The default value is 1000. It is not recommended to lower this number. In such situation the layout algorithm might generate incomplete results – because some suitable reference points might be omitted. On the other side, a rise of this number decreases the speed of the layout algorithm, but might find the reference points that it has not been able to find with the default value. It has almost unobservable change of the outcome if the increase is not big enough. In some cases of unlucky random initial values the layout algorithm might terminate prematurely, and leave the result incomplete just because an unfortunate random sequence generated too much unsuccessful tries consequently. Then the increase of this value is reasonable. Do not hesitate to change it directly up to 10.000; an increase by tenths or hundreds will not be sufficient.
5.6.7 layout_adjust_normal

The parameter determines the amount how much the normal of the surface on which the generated object is placed changes the object’s orientation. Value 0 means – leave the object as it was generated without any additional rotations, value 1 means – orient the object so that its upward vector will be aligned with the normal. Any other number between zero and one might also be used, resulting in an interpolating rotation. It is a convenient way to adjust the macro objects rotation so that they stand on the ground. This parameter is useless with the layout_inside type because there are no normals to which the orientation could be adjusted. The default value is 0 – no normal adjustment.

5.6.8 layout_distance

This parameter determines the minimal distance between two generated reference points. It is assured that no two reference points will be closer than this distance. The algorithm simply discards a reference point that has been generated by the main distribution algorithm, but violated the minimum distance condition. In case of the layout_on_top layout type, the distance is not counted between the reference points, but between the ray sources that generated these reference points. This special check serves greatly for the animation stability – e.g. imagine a crowd scene of characters shifting on a terrain; if the distance was checked on the ground, it could cause totally different layouts for the animation phases if some character came into the minimum distance range of another character just because he changed his altitude.

5.6.9 layout_inside_check

This parameter works only with the layout_inside type. It determines the number of points to be generated inside a newly created macro-object to test for being inside the big layout object. If a point inside the macro object is found that is not inside the layout object, the macro object is discarded. The value approximately 50 is enough for a simple check, but the macro objects still might have small ledges that would not be inside. To force a true insideness, keep this value above 1000. By default the check is not used.
5.6.10 layout_overlap_test

This parameter allows a test for mutual macro objects overlapping. It determines the number of points to be randomly generated for each newly created macro-object to be tested for being inside all other (already generated) macro-objects. Recommended values are approximately 50 for standard test, more than 1000 for a true test. By default this test is not applied – the objects could overlap.

![Image of tori demonstrating layout_overlap_test](image)

Figure 10: A demonstration of layout_overlap_test – 50 tori tested for mutual overlapping using the test value 200.

5.6.11 layout_random_stream

This parameter identifies the random stream to use for the generating of the object locations – the reference points. If this parameter is not used, the layout algorithm uses an internal instable random stream that will change for each execution of the source code parsing. It is a very useful parameter for animation stability or similar layouts generation (e.g. the same crowd on different height-fields).

5.6.12 layout_regular

The regular layout is a parameter for making regular grids of macro objects. Given are: the starting point (of a virtual coordinate system) and two or three vectors (depending on the dimension of the problem) that describe the coordinate system. Only the grid points (points that come from some integral multiplicands of the vectors) of this coordinate system are allowed to become reference points. This parameter is allowed only in conjunction with layout_inside or layout_on_top. To make a regular square grid layout over a terrain (for instance an army of soldiers), provide two perpendicular vectors – the first determining the distance between rows, the second meaning the distance between columns.
Figure 11: A parking place – a composite of two regular layouts.

5.6.13 layout_regular_range
This parameter is dependent on the layout_regular layout type. In regular layouts, linear combinations of the given vectors that describe the axes of the virtual coordinate system are made to find a reference point. The vectors are multiplied by scalars that are chosen from the range that is given by this parameter. The code

\[ \text{layout\_regular\_range \:} x, \: 0 \: 9 \]

means that the first vector will be multiplied only by values between zero and nine inclusive. Default values for all axes are –10 and 10.

5.6.14 layout_sequential
Normally, the regular layouts mean only that the macro objects are on the grid points, but on any grid point that is found. To assure that all grid points in a given area will be occupied, a sequential regular layout generation is required. The keyword layout\_sequential means that all the grid points will be tested in a loop according to their given ranges. The inner loop iterates the first of the vectors.

5.6.15 layout_size
The size of the generated objects passed as the argument of the macro. The default value is 1. If the macro does not accept any arguments, this parameter becomes useless. It is a good habit to create the object macros with the size argument. Thus it allows the remote changing of the size – the developer is able to play around only by adjusting the layout parameters; he does not need to search for the macro definition in the code.
5.6.16 layout_surface2_accuracy

This parameter describes the accuracy of the layout_surface2 technique. It is ignored if it is used with different layout types. The accuracy is the radius (the value $q$ as mentioned in the second layout surface algorithm) of the small sphere that is generated for each inside point found. It defaults to approximately one twentieth of the maximum bounding box edge size. Realize, that a decrease of this value increases the algorithm’s accuracy, but significantly decreases its speed.

5.6.17 layout_wrap

The distance comparison is being wrapped by the layout object’s bounding box. This might be useful for cyclic animations, when a crowd of characters moves on top of a terrain. If the crowd is generated on a rectangular area cut from the terrain, the same crowd could be generated on a similar area that follows the first one on one of its sides. (To generate the same crowd set the layout_random_stream value to the same random seed again.) The wrapped comparison will assure that the minimum distance check will work also between the two crowds. By default this flag is not activated.

5.7 The layout global variables

The macro object is able to use some of these global variables:

```plaintext
#declare layoutPosition=<0,0,0>;
#declare layoutUp=<0,1,0>;
#declare layoutColor=rgb <0,0,0>;
```

The layoutPosition and layoutUp are declared as three-dimensional vectors, the layoutColor is declared as a color. To use the global variables they must be declared (with any values, but they must be the correct type) before the layout command takes place in the parsing of the code. The declarations above would serve fine for this purpose. The layoutPosition variable contains the so-called reference point. The layoutUp contains the normal of the surface at the location of the reference point – the vector that is looking up from the point on the surface. The layoutColor should contain the pigment element of the layout object’s texture at the reference point, if some texture for that object has been entered.

5.8 The layout return variable

The macro object could return a value in a layoutSuccess variable:

```plaintext
#declare layoutSuccess=0;
```

If the object macro does not want to be generated at the suggested place, with the proposed settings, or due to any other reason, it could set layoutSuccess is zero as shown in the code above. The macro still has to return any object that will be discarded immediately, but it must obey the rules for macros generating objects. Our suggestion is to generate a simple object in such case – for example a sphere.
Figure 12: Forest on a river’s bank. The probability of the macro object returning positive layoutSuccess depends on the color of the surface (the layoutColor variable).
6. Complexity and Time Issues

6.1 Time complexity

When talking about mass scenes the computational complexity is very important. The most relevant is the time complexity. The time necessary to create a nice image could be divided into three exclusive parts: the time to prepare and to design the scene, time for the parser to read the source and to build the internal structures, and the time that the rendering engine needs to render the image.

The first time – needed for the scene composition has been reduced significantly. The scene composer does not have to mark for each of the generated objects the place where to put it. The boring long-lasting work surely belongs to the computer. Also the commands for parametric object description let him focus on the quality of the modeled object instead on wasting his time by thinking how to write the desired commands.

The execution of all proposed commands takes place in the parse time. Although the parse time is much shorter that the other two times (see the table 1 below), the shortening of that time is still not a worthless job. During the scene preparation the designer needs to preview the image (at low resolutions) many times, where a longer parse time is very annoying.

To accelerate the layout minimal distance comparison, some types of hash-tables are applicable. As the reader has certainly noticed, all the layout techniques work this way: generate a random entity (a point or a ray) and try to use the entity to find a reference point. If the reference point has been found, it was a good choice, but if not (the point has not been inside or the ray did not hit anything), the quest for a reference point has to be repeated. To avoid these unlucky choices, some space partition trees could be made that would assign correct probabilities to the tree nodes. Thereby a random entity (the point or the ray) will have lower probabilities to be generated in the regions where the chance to find a reference point is lower.

The render time acceleration is a hard task and out of the scope of this paper. There is one suggestion how to save some render time (and quite a lot of memory): to count the distance from the camera to the object being generated by the object’s macro to reduce the model quality according to this distance. It is sufficient if the models in the front (near the camera) have higher detail, and the others at the back have lower detail. However POV-Ray makes for the rendering some sort of scene object trees, thereby the render time rises by the logarithm of the scene complexity – the number of objects in the scene. In other words it almost does not matter if there is one thousand of objects in the scene or ten thousands.

As we have chosen POV-Ray, it works on almost any platform and it is possible to accelerate the parse and render times by a migration to a faster or remotely mastered computing machine.

The following table shows the times it took to produce the image during the scene preparation process. The designer had to re-execute the rendering several times at low resolution and only after he was fully satisfied he produced a big image. Another technique often used by the designer is to render only a section of the image. The
parsing time really needs to be cut to minimum, because the rendering time (in case of a small section of the image) has been reduced significantly.

The table contains parse and render times comparisons of the preview rendering. A subset of scenes presented in this paper (including the examples chapter at the end) that were worth to mention are present in the table.

<table>
<thead>
<tr>
<th>Scene</th>
<th># of elements</th>
<th>Parse Time</th>
<th>Render Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letter soup</td>
<td>1500</td>
<td>0:00:05</td>
<td>0:01:35</td>
</tr>
<tr>
<td>Characters on a terrain</td>
<td>50</td>
<td>0:00:01</td>
<td>0:00:05</td>
</tr>
<tr>
<td>Hairy monster</td>
<td>45000</td>
<td>0:04:08</td>
<td>0:01:03</td>
</tr>
<tr>
<td>Cars on the parking place</td>
<td>200</td>
<td>0:00:00</td>
<td>0:00:08</td>
</tr>
<tr>
<td>Möbius strip</td>
<td>800</td>
<td>0:00:06</td>
<td>0:00:10</td>
</tr>
<tr>
<td>Dwarves mass scene</td>
<td>1200</td>
<td>0:01:04</td>
<td>0:11:00</td>
</tr>
<tr>
<td>Holbw cheese</td>
<td>300</td>
<td>0:00:07</td>
<td>0:06:24</td>
</tr>
<tr>
<td>Martians on Mars</td>
<td>1000</td>
<td>0:00:28</td>
<td>0:00:49</td>
</tr>
<tr>
<td>Flock of birds</td>
<td>150</td>
<td>0:00:03</td>
<td>0:00:46</td>
</tr>
<tr>
<td>Mosaic effect</td>
<td>2800</td>
<td>0:00:03</td>
<td>0:00:05</td>
</tr>
</tbody>
</table>

Table 1: The parse and render time comparison, executed on an AMD K6-2 350 with a preview resolution (320 x 240 pixels) without the antialiasing.

Another possibility to accelerate the layout command is to save the reference points that have already been found into a file and reuse them the next time the parser is started. This is the way that a lot of scene designers (who do not have our layout command) are using – they create a file containing the list of locations for the objects and include it into the scene description. While the file is external, it is easily editable and a very valuable feature is that any external software can regenerate it.

6.2 Memory consumption

A large number of macro objects causes naturally the allocation of very much memory for the internal structures. All of the scenes presented in this paper have been rendered on a computer with 128 mega bytes, sometimes allocating the whole memory. There are two basic options how to go beyond this limit, how to create much larger scenes. The easiest is to find a computer with more memory; the second opportunity is to improve the layout command. Currently, the layout command generates a standard POV-Ray’s CSG union object. But it could have made a new type of objects – a new ‘layout’ object that would implement the necessary functions as all regular objects do. These functions are:

- All_Intersections. The method finds all intersections with the given object that interfere the given ray.
- Inside. Returns true if (and only if) the given point is inside the object.
- Normal. Returns the normal of the object’s surface in one of the points that have been found by the All_Intersections function.
- UVCoord. This method is not standard to POV-Ray, but appears in MegaPov (described later). It finds the \( u \) and \( v \) texture mapping coordinates for a given point (that should be on the object’s surface obtained by the All_Intersections method).
- Copy, Destroy. Simple self-explaining functions.
- Translate, Scale, Copy, Transform, Invert. The functions for the object’s transform matrix modifications.

The task is to implement all these functions in case a new object-type is being added. To make a simple starting implementation is easy – just redirect all these function calls to the functions of the union object. We tried to do this, and it really worked, but it was significantly slower comparing to the direct union object. Nevertheless, we are now dealing with the memory consumption, and this should be really the way to conserve a lot of memory. Assuming that we have the full control of the layout methods, we could make a technique called on demand macro object generation. During the parsing process only the reference points would be generated, and the macro objects would return only some bounding boxes of their actual future model. In the main image creation phase, when the ray-tracing would call some of the functions of the layout, it will interpret the functions according to their type:
- All_Intersections. The bounding boxes of the macro objects hit by the given ray would be taken, the respective macro objects would be truly generated (not only as bounding boxes), and the method would be called again – this time for the already existing macro objects.
- Inside. The test for insideness would again invoke the true macro object generation only for the macro objects that interfere the given point.
- The same holds for the Normal and UVCoord methods.
- The copy and destroy methods are simple, nothing special about them.
- All the other methods could only modify the transform matrices of the already generated macro objects, as well as the to-be-generated macro objects.

This on demand technique would slow down the rendering process, but would allocate only the macro objects that are visible. The question is whether this is the right trade-off between the time and space complexity; if it is not better to focus only on the rendering time acceleration.
7. Conclusions and Future Work

This master thesis has introduced a general technique that allows to create and to render mass scenes. It is impossible to make something like a mathematical proof of completeness (that it can be used with all mass scenes), but the technique is at least helpful in the process of mass scene creation. It has been tested with many various scene types, and it showed to be very easy to use and conductive.

The implementation – an extension of the standard POV-Ray has been tested on both Windows and Linux platforms. It seems to be working quite fast and reliably. If some bugs are detected in the future, we will try to correct the code as soon as possible.

7.1 Drawbacks

It must be said that the proposed layout techniques do not consider any interaction between the placed objects. An instance is a crowd scene where a person holds one hand of another person. This small interaction can be avoided by making not only one-character macro objects for the layout, but also a couple should be a macro object. There are still some scenes (like soap bubbles on the water) that need a real interaction between the objects (bubbles) to modify the object shape. To cover these scenes a small modification could be made – the objects will not be really created in the layout process, but only the reference points will be placed into some array and the object creation macro will have the possibility to use this array, to look around and see where and what is around. E.g. a soap bubble will know its surroundings and according to them will modify its shape.

7.2 Future work

The future work includes the creation of additional layout parameters according to the requirements of some special mass scenes, where the set of current parameters would be insufficient. To achieve some state of completeness maybe also the opportunity to create other regular layouts should be added, e.g. some multidimensional grids in polar coordinates. Another research path could be the possibility to create truly dynamic animations.

7.3 Project’s destiny

The Persistence of Vision Ray-tracer (POV-Ray) [5] is being developed by volunteers interested in computer graphics and particularly image rendering in their spare time. The core of the rendering engine is based on DKBTrace 2.12 written by David K. Buck and Aaron A. Collins in the beginning of 1990’s. The POV-Team has been established and is doing a very meritorious job for today’s computer graphics by maintaining the official releases of the POV-Ray in cooperation with Christopher Cason (who ported the software on the Windows platform). The idea to provide the free sources has proven to be very progressive, because the programmers are encouraged to add extensions to the standard POV-Ray easily. The last official version of POV-Ray has been named 3.1 (actually 3.1g due to some bug fixes) and has been released in 1999. A lot of extended custom versions have been made since. One of them is especially worth our attention. It
is the MegaPov [7] maintained by Nathan Kopp who added a lot of valuable and sophisticated functionality, as radiosity, photon maps (to render true reflective and refractive caustics), true meshes and other. The current MegaPov version number (in April 2001) reached 0.7 – that means it is the seventh edition. In September 2000 the current version of MegaPov (it was the version 0.5 that time) has been considered as the foundation of a new official POV-Ray that will be probably named 3.5. A discussion about all the enhanced features of MegaPov has been made and the outcome is a list of extensions that are really recommended to be imported into the new official release. The main reasons why the MegaPov is not considered to be an official version are that the new features are many times only experimental and there is a high probability of bugs. To avoid and repair the bugs it needs only time and patience, but to promote an experimental enhancement to an official POV-Ray feature, a comprehensive discussion is required. It must be viewed from many sides:

- Is the syntax of the newly added commands satisfying the given formal standards? The scene description language should be homogeneous.
- Are the new commands adding some new functionality? If not, are they at least simplifying some tasks? Otherwise there is no reason to use such command.
- Are the new commands generalized enough? Do they work with all possible objects or other commands? The scene description language should be consistent.

If all the rules are accomplished, a recommendation to import the command to the standard official POV-Ray is made.

Another very significant point of view is the quality of the pictures produced using the new features. A suitable place to put ray-traced images to be criticized and reviewed by people interested in computer graphics is the Internet Ray-Tracing Competition (IRTC) [8]. Each two months a topic (a theme) is chosen and published on the competition pages and anyone around the world could make and submit a computer-generated image. The major competition rules are – the rendering of the image must be done using only a rendering tool (and not altered manually using any image enhancing program) and the image should comply the given topic. The image should be accompanied by a short description how it has been created, which techniques have been used. The competition is supported by the POV-people; most of the submitted images are rendered using POV-Ray. The images are rated in three different categories – the artistic merit, the technical merit, and the concept, originality and interpretation of the given theme. Everyone who submitted an image should also vote all the images (except his own). Each of the voters gives three ratings and any comments to all the images. We took part in this year’s round, which had the topic ‘worship’. The actual image is in the section examples at the end of this thesis. We were pleased that the rating of our image was quite good, especially the technical merit. Also the responses encouraged us to publish the mass scene rendering techniques. Currently we are discussing the syntax and the parameters of the commands and testing the functionality, in some time we will consider it complete and reliable and we will try to include it into the MegaPov’s next edition, hopefully the 0.8. A simple merge with the MegaPov 0.7 sources, that we made as a test, seems to be working fine, but has not been much tested yet. We hope that this master thesis will not become only a heap of printed-paper, but also a usable scene
generating and rendering tool that will be freely available to anyone interested in computer graphics.

7.4 Portability

This mass scenes rendering framework is not limited to the POV-Ray platform. It can be easily ported to any other 3D environment, where the adding of some functionality is possible, e.g. the possibility to make own plug-ins.

Let’s consider (only theoretically) the 3DStudio [9] with its plug-in technology in comparison to our POV-Ray solution. The 3DStudio is a commercial product; the sources are secret and protected. To allow the creation of plug-ins, they release a software development kit (SDK), with a well-documented library of classes with only header files and prototypes that are available to be derived. A plug-in (written in MS Visual C++ 6.0) typically derives one of the recommended classes depending on the plug-in type, implements the necessary functions and adds the new functionality to them. The types of the plug-ins are:

- Procedural objects
- Object and edit modifiers
- File import and export
- Procedural materials, textures
- Shaders, renderers, image filters
- Lot of other special types

To implement our makevalue function as a plug-in, making an object modifier could do it. The makevalue parameters would be the properties of the modifier and the modifier would be added to the modifiers queue of the object or property (classes INode and Animatable in 3DStudio Max SDK terminology) that would generate the random number according to the parameters when prompted. The cloning of an object that has this modifier in the queue will result into a new clone that is different from the original!

The alternative could also be done this way. We will loose the behavior of a directive, because we are not dealing with a scene language that is parsed, but it would work by almost the same principle.

A procedural object plug-in could be made in place of the layout command. The procedural object will represent the whole set of the generated so-called macro objects. The access to the ray-tracing functions (question if a point is inside and the ray intersect function) is granted by the 3DStudio so the functionality could be made the same way as it is in POV-Ray.
8. Examples

8.1 Möbius strip

The Möbius strip was created as a union of cylinders. The centers of the cylinders are on one big circle; each cylinder rotated a bit more than the previous one around the axis tangent to the big circle in the cylinder’s center. The whole strip was sliced into eight parts. An appropriate rectangular area had been cut into a union of three adjacent slices. On top of this area the characters were distributed using the layout_on_top technique with the parameter layout_adjust_normal set to 1.

For correct mutual layout meets, the layout_wrap parameter had been activated. The characters are only simple entities made of six scaled spheres, but it is enough to demonstrate the scheme. Nevertheless they are so small that the effort put into their enhancement would not be very significant.

An animation of moving characters has also been made, it can be found on the enclosed compact disk. The movement creation was quite unusual – not the characters were the moving parts, but the ground had to be moved backwards, and only after the layout has been made, the objects could be returned to the proper place.

It sounds strange, but the same layout technique had to be used on both sides of the strip, although it has only one...

Figure 13: The layout on top technique applied to a Möbius strip. Based on the idea of Rastislav Královič.

8.2 Dwarves mass scene

It is a mass scene in the meaning of large amount of objects, as well as a mass service that takes time currently. This is the picture that has been submitted to the Internet ray-
tracing competition (as referenced in the chapter 7). There is still a lot of work to be made (especially regarding the textures and lighting), but there was no more time to improve the scene and at last it shows the mass scene techniques responsibly.

The cave walls have been created using a layout of stalagnates on the top of a wall that has been lied down temporarily for this process. Throughout the cave hall are distributed stalagmites and the dwarves using a layout on the top of the floor object. A check for their relative distance is used to ensure they are not overlapping. The layoutPosition variable is considered for two reasons – the distance from the central object, and the rotation towards the diamond. The first reason assured that nobody would interfere with the big stalagmite in the middle, because everyone wanted to be far enough (if a reference point has been generated too near, the layoutSuccess returned false). The second reason – the rotation angle calculation has been made to turn the dwarf so that he was looking towards the gem.

Another trick was the use of layoutUp variable to test whether the slope of the ground to place an object is enough horizontal. If not, the object was discarded (using the layoutSuccess), otherwise the dwarf has found his proper place.

Figure 14: Dwarves Mass Scene. Created using automatic layout of dwarves and stalagmites.

8.3 Hollow cheese

The distribution of holes is as simple as the distribution of normal objects. The result of the layout algorithm is actually a union of the macro objects. The union can be put into the scene as any other object. It can also be merged, subtracted or even intersected with
another object or objects. The cheese model is a simple object that served as the layout object for the distribution of holes on its surface. Actually two types of holes have been used – the big ones and the small ones. One hole is a blob structure made of randomly chosen spheres. Firstly, the big holes are distributed on the layout object’s surface. The second step is the distribution of the small holes on the surface of a special layout object that is a subtraction of the original cheese and the union of first holes. Thus it is possible for a small hole to be cut also into one of the big ones. The resulting hollow cheese is the difference of the original cheese object and the two layouts.

Figure 15: Hollow cheese. Based on the idea of Zuzana Rjašková. A demonstration of the possibility to distribute holes using the layout on the surface technique.

8.4 Martians on Mars

The Martian characters are taken from the Möbius strip example, the planet is an isosurface based on a bumps pattern [10]. The layout_adjust_normal parameter has been set to one, so that they stand on the ground according to the slope of the ground at the place of their reference point. The layout_overlap_test is activated to ensure that no part of two characters share the same area in the 3D space.

There is also the possibility to enhance the image simply by adding some water, that would have a different color, and with the help of layoutColor variable test whether to generate a Martian on a given reference point or return layoutSuccess set to zero. This would result into a planetoid where the Martians would be placed only on the continents and not on the oceans and seas.
Figure 16: Martians on Mars, distributed on a surface of the asteroid-shaped planet.

8.5 Flock of birds

Figure 17: An insight into a flock of flying birds, which are distributed into a large box using the layout_inside technique with a relative distance check.
The birds are a good example of parametrically described objects. A lot of parameters vary on them – the size, shape, color, and the animation phase. This scene has also been animated: the birds are waving their wings and tail, change their positions and blink some of their eyes. The actual layout here is very simple, just to fill an area before the camera.

8.6 Mosaic effect

A mosaic effect has been achieved by a regular layout of intersecting spheres. A sphere surrounded and partially intersected by six other spheres with the same radius constitutes a hexagonal shape. These hexagons are composing the two dimensional regular layout with the coordinate system described by two vectors containing a 60 degrees angle.

Figure 18: Original image used as a pigment texture for the mosaic-effect.

Figure 19: A simple mosaic-effect – regular triangular grid of spheres with their color gained from the underlying texture.

Figure 20: The altitude of the macro objects modified according to the color of the original image.

Figure 21: The macro objects – the spheres with modified color and altitude. A compilation of figures 19 and 20.
9. References


