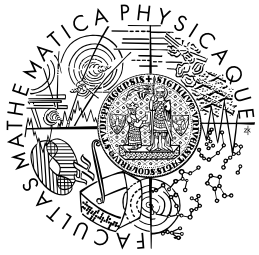

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Štěpán Hojdar

Procedural placement of 3D objects

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	doc. Ing. Jaroslav Křivánek, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my advisor, doc. Ing. Jaroslav Křivánek, Ph.D., for all the guidance and help he has given me, as well as the patience he had. A big thank you to Ondřej Karlík and the rest of the team behind Corona Renderer for both allowing me to work on Corona Scatter and giving me support when I was getting accustomed to their code base.

Last but not least, I would like to thank Ludvík Koutný, for all the helpful insight he has given us, which allowed us to make the product as user-friendly as possible.

Title: Procedural placement of 3D objects

Author: Štěpán Hojdar

Department: Department of Software and Computer Science Education

Supervisor: doc. Ing. Jaroslav Křivánek, Ph.D., Department of Software and Computer Science Education

Abstract: $3D$ modelling in computer graphics often requires placing a big number of objects into the scene. This may be tedious or even impossible if done manually. A few programs exist to perform this task automatically but most of them are either too slow to place the required number of objects or too difficult to use for a non-expert user. We expand the already existing Corona Scatter program which is fast and user friendly but lacks in terms of functionality. We implement scattering regular patterns, scattering along spline objects and the ability to use spline objects to locally modify the distribution properties.

Keywords: $3D$ computer graphics, $3D$ scene modeling, scattering

Contents

Introduction

Computer graphics is a wide field encompassing many different problems, tools and producing many different outputs. In $3D$ graphics a certain pipeline is present. Producing a realistic looking scene starts with 3D modelling of the geometry, continues with realistic materials and lights and after rendering, corrections can be made which are known as post-processing.

During $3D$ modelling, the user often needs to place millions of objects, such as trees, pebbles, etc. into the scene, which is close to impossible if done by hand. However, precise placement of these objects isn't necessary and therefore such objects can be placed by the computer. The best example is covering exteriors with vegetation, which is often almost random – trees or grass patches don't appear in a particular order, though they follow some basic rules (e.g. trees don't usually collide with one another). More examples of these seemingly random distributions include placing animal fur or decorative stone patches in gardens.

However, most things created by mankind follow rules that are immediately visible. Street lamps are placed along roads, with close to constant spacings in between. Cars parked in parking lots are placed into an almost regular grid as are trees in orchards. Houses in cities and estates are also placed regularly with varying degrees of freedom.

The last very important case is distributing a large number of objects to cover an area and then specifying some regions where the distribution should be either modified or avoided altogether. This is best illustrated on a path through a forest. If the path was a hiking trail, it would not go very close to trees but grass would still be present on the trail. The grass would, however, be smaller and not fully grown right on the trail but only becoming normal sized gradually as the distance from the trail increases. If the path was a tarmac road, all vegetation would be gone from its vicinity.

The goal of this thesis is to take the already existing Corona Scatter program – which so far could only distribute instances of a given object in a random fashion – and add more functionality which would allow users to create regular patterns as well as area-defined distribution modifications mentioned above. The two key rules our program should follow are: it should be as intuitive and easy-to-use as possible and we need to make sure the program is capable of distributing large numbers of objects fast and without using up too many resources. To visualize the resulting object placement, we develop a simple standalone application which easily demonstrates the capabilities of Corona Scatter as well as expand the already existing plugin for the 3ds Max application [1] with the new functionality mentioned above.

- Conceptual goals
  - The plugin should be easy to understand and use without the need for extensive tutorials.
  - The program should be capable of placing objects in regular patterns (e.g. a regular grid), with user specified grid size and position.
  - The program should be capable of placing objects along splines (a line, a circle, an arc, etc...) with user specified spacing in between, starting

positions and randomness.

  - The program should be capable of modifying the distribution already generated by Corona Scatter by specifying a helper object which will affect the distribution and size of all objects placed nearby.

- Technical goals

  - The program has to be fast and capable of generating large numbers of instances.
  - The communication of the plugin with 3ds Max will be extended to include loading spline objects.
  - Have automatic testing to make sure the program's behavior does not change and the generated patterns and distributions are correct. This includes both unit testing for parts of the program and complete regression render tests with bitmap comparison.

To give the reader a better picture of what we discuss in this thesis, we now present a couple of pictures. Figure 1 shows an outdoor scene generated using Corona Scatter and rendered in Corona Renderer. Figure 2 shows the same scene as the $3D$ artists sees it in the viewport of 3ds Max during the modelling phase.

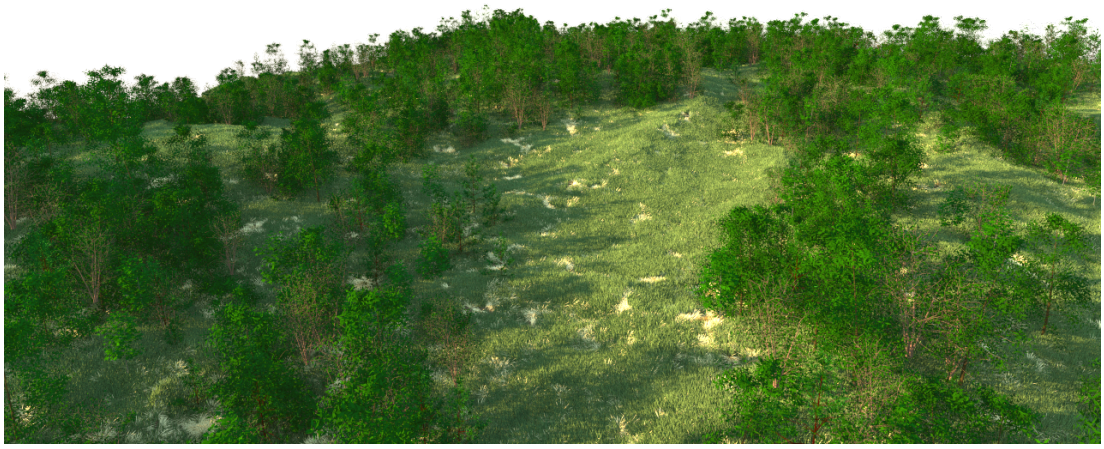

Figure 1: An outdoor scene generated by Corona Scatter.

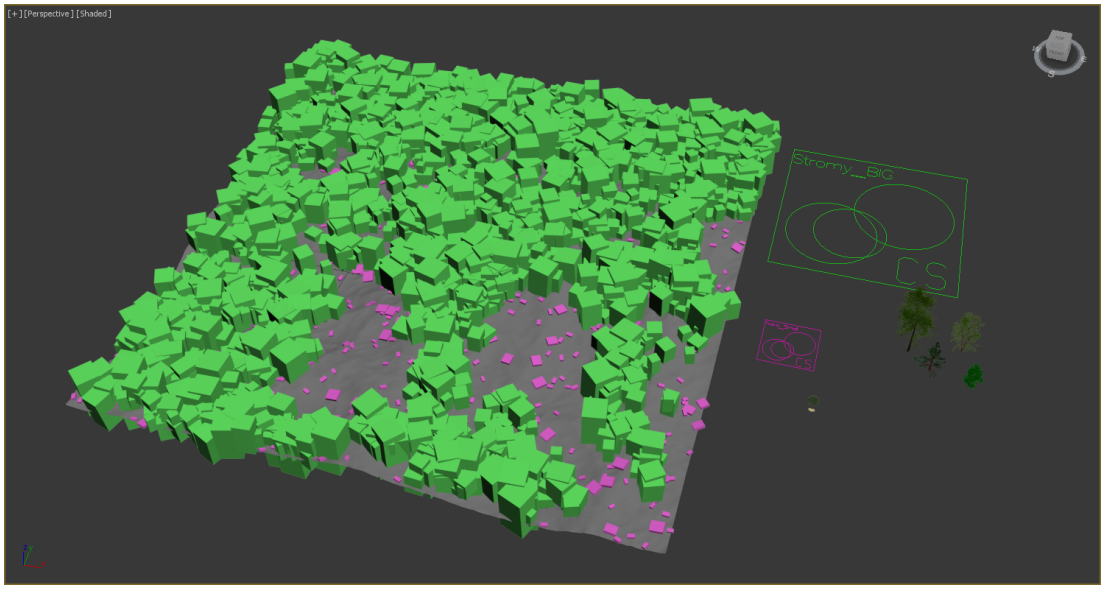

Figure 2: The same scene from Figure 1 as viewed in 3ds Max viewport during modelling.

The following text begins with a brief overview of similar software already on the market. Next we present an analysis of all problems we encountered and solved while developing this thesis. Following the problem analysis is the user and developer documentation which presents all details on the implementation of the solutions mentioned in the analysis. Lastly we include a small breakdown on the program's performance and scalability since both of them are very important.

1. Survey of related work

In this chapter we present a few already existing scattering tools that are currently used by 3D artists. We mention both pros and cons of each of them, hopefully giving the reader a good understanding of what makes a scattering tool better or worse. We rely on this understanding later in the thesis, when explaining our choices in the behavior of our Corona Scatter tool.

1.1 Forest Pack

The first and probably most widely used scattering tool is Forest Pack [2] from iToo Software. This tool is a very professional and advanced scattering program capable of almost anything the user can imagine – from basic random scattering to reducing the size and density of scattered objects with increasing altitude, which closely models real world behavior of vegetation in mountain ranges.

However, this versatility comes at a price – the User Interface (UI) of Forest Pack is very chaotic and confusing to a non-experienced user, spanning multiple long rollouts as illustrated in Figure 1.1. This User Interface layout results in the user not being able to see all controls at the same time and is left to constantly look for the correct rollout to open. This is something we want to avoid. Forest Pack is a plugin only for Autodesk's 3ds Max [1] and comes in two versions – a free trial and the full version. Unfortunately the functionality of the trial version is very limited.

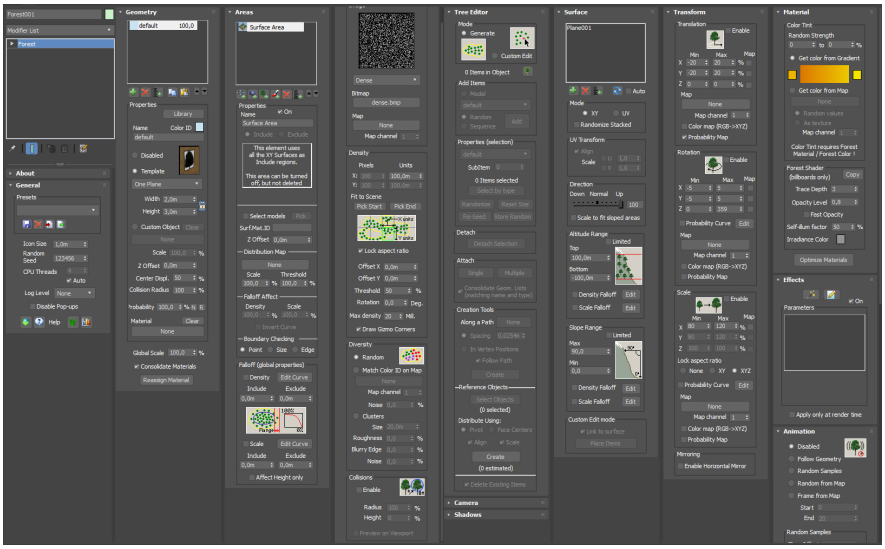

Figure 1.1: Forest Pack Pro user interface.

Examining the features we set to implement, we see that in Forest Pack, scattering patterns are specified by a user-provided bitmap mask (with white spots being places to scatter on a black background). This means that achieving random distributions is problematic but creating new patterns is very easy. Scattering along splines in Forest Pack is handled as scattering in a user defined vicinity of the spline itself. Distribution modification is handled by area modifiers and is simple and intuitive. Users can choose to modify the density of the

distribution or the scale of instances (or both) and can specify the range where the modification should start and end.

1.2 MultiScatter

MultiScatter [3] is another popular scattering plugin for Autodesk's 3ds Max. Even though the development seems stagnant at the moment, this tool is still used by many $3D$ artists. It provides a good UI (see Figure 1.2) and the majority of functions a scattering tool should have.

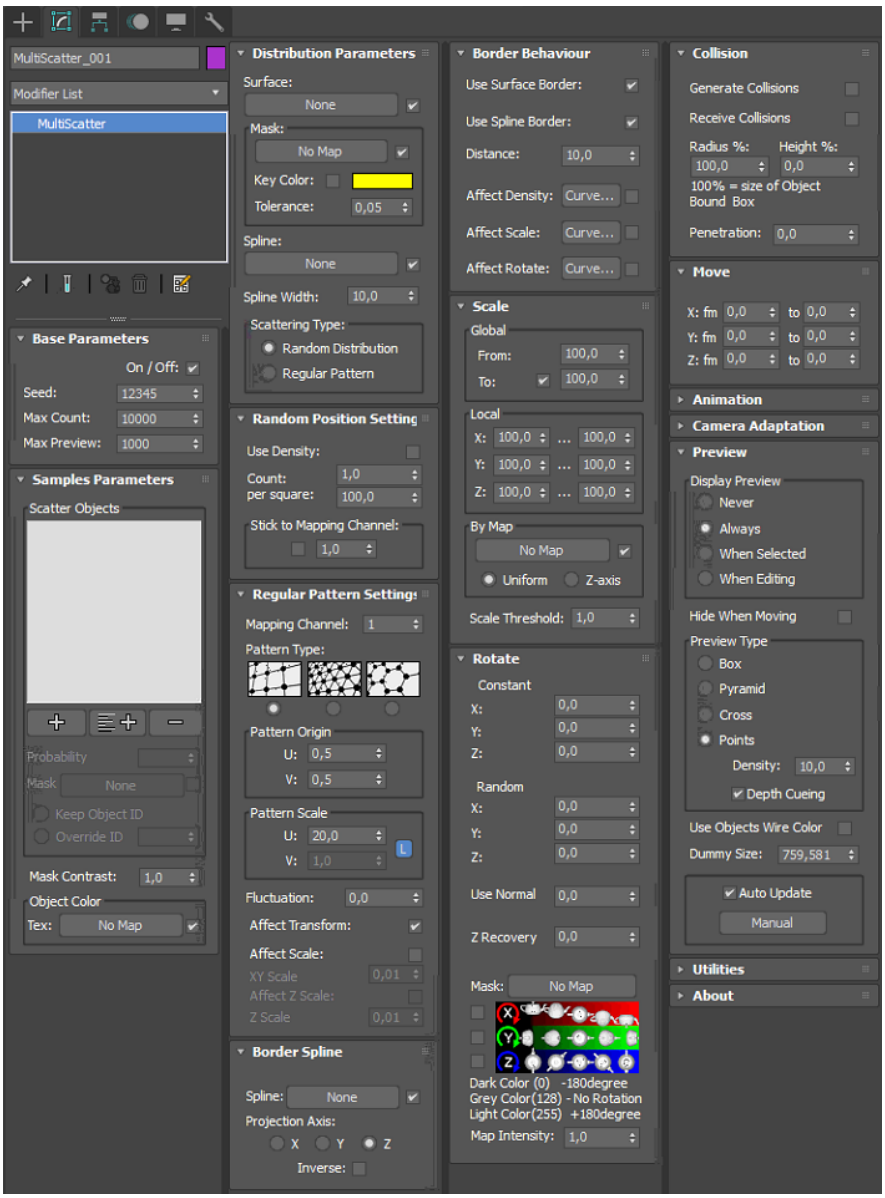

Figure 1.2: MultiScatter user interface.

Let us examine the features similar to the goals of this thesis. Unlike Forest pack, which uses bitmap mask, MultiScatter generates the distribution from scratch, providing both random and regular distribution options on surfaces. The regular option provides the choice of several basic patterns, changing their size

and moving them around. MultiScatter is able to scatter on spline objects, however, instead of scattering directly on the splines, it scatters in their vicinity in the XY plane. Modifying the distribution can be done in two ways. MultiScatter can modify the density and the scale of the scattered objects near one given spline object and near the borders of the object we scatter onto. The ability to define behavior on borders actually seems to be important to users as the Corona team received this feature request multiple times.

We now briefly list a few downsides we found with MultiScatter. First and foremost, the program did not seem as fast as some other alternatives and we experienced several crashes. The UI is cleaner and more understandable than in Forest Pack, however, the UI does not react to the user's settings. This could confuse the user as there may be a situation where the user is changing some settings that do not have any effect on the scattering at the moment and is left confused why changing these controls does not do anything.

1.3 Carbon Scatter

The last scattering software we tried was Carbon Scatter [4]. This plugin works with 3ds Max, Cinema 4D and Maya [5], we tried the trial version for 3ds Max 2016. Carbon Scatter has a completely custom-made UI and does not use any of the UI elements of 3ds Max, creating its own windows instead. This allows a complete customization of UI which makes it very clean and fully customizable by the developers. A screenshot of Carbon Scatter's UI is shown in Figure 1.3.

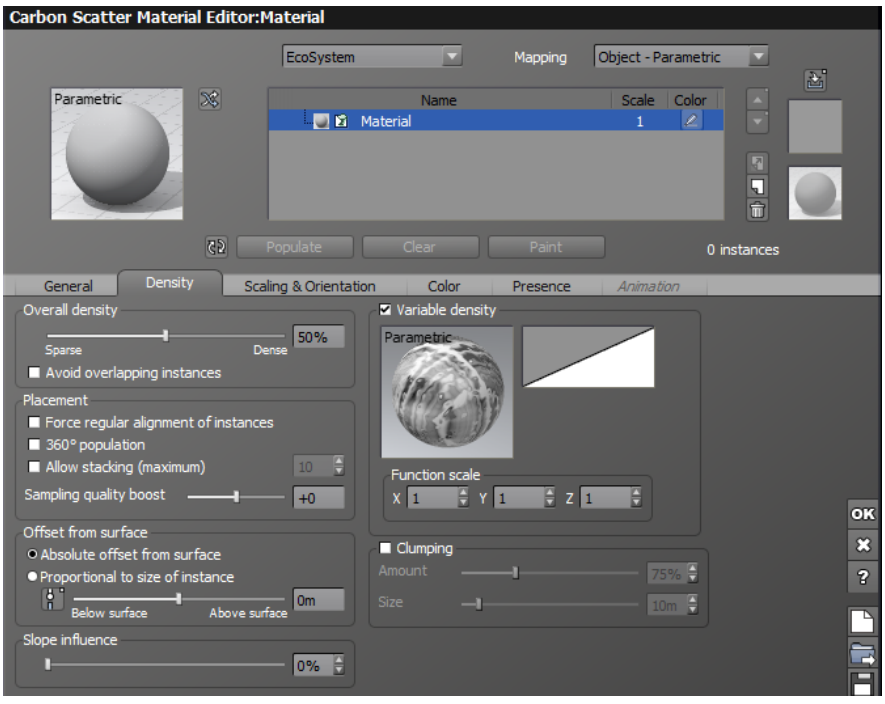

Figure 1.3: Carbon Scatter user interface.

Comparing Carbon Scatter to the other scatter plugins, the distribution patterns are handled in a very unique way in Carbon Scatter. Like Forest Pack, it uses maps to decide where to distribute. However, the maps are generated

using a node editor (very similar to the Slate Material Editor in 3ds Max or the Node editor in Blender [6]). This allows the user to combine a lot of mathematical functions like noises and turbulences to create new patterns very easily. Carbon Scatter also allows the user to paint the instances directly on top of the distribution object with a paint brush tool.

Taking into account how the distribution is handled using the generated maps, it is understandable that the option to scatter in a regular pattern is not present as a UI element and either requires a manually created bitmap (much like Forest Pack) or requires the user to create the pattern manually using the paint tool. We also did not find the option to scatter directly on top of splines. However, much like Forest Pack, an option to scatter in the vicinity of the spline is present allowing the user to specify the width of the region. Excluding regions using splines is possible by using both open and closed splines. The only modification option we were able to find in the trial version was the ability to align scattered instances along a spline object, however, there were no options to scale or reject instances based on their distance to the spline.

1.4 Surface Spread

Surface Spread is a plugin for Cinema 4D [7] – another computer graphics software used by many $3D$ artists. Surface Spread comes within a bigger software package that is able to both scatter vegetation on a surface and generate a realistically looking terrain.

Examining the features we seek to implement, we first of all notice that the ability to scatter in a regular pattern is not present. On the other hand scattering on splines is quite advanced and has many features. The users can scatter along up to two splines and are given the option to blend between the two splines, creating the distribution along a spline that is a convex combination of both user input splines. The users are also given the option to add incremental rotation/scaling of the instances, allowing to transform the instances linearly along the spline. Surface Spread is also able to exclude instances within a specified area and near a specified spline. Additionally the users are able to scale the instances within a specified area. However, these features are separated, which requires the user to create multiple areas if he wishes to both exclude and scale some instances. We also did not find any option to scale the instances based on their distance to a spline object.

Some additional features include the ability to control the distribution based on the altitude or the inclination of the slopes. Surface Spread also has very good support for vertex maps – the users can utilize them to exclude areas from the distribution, scale instances, set the color of the instances, etc.

1.5 Corona Scatter 1.5

Last but not least we mention the previous version of Corona Scatter, as it was released in the last major Corona Renderer release which was the version 1.5 from the 10th of October 2016. The Corona team received positive feedback about the

speed and clean and easy to use UI (Figure 1.4) of the Scatter from their user base.

Corona Scatter in version 1.5 was able to randomly place objects on surfaces and into the bounding volumes of mesh distribution objects. The number of objects placed was either directly specified by the user or calculated from a density parameter which specified the number of objects placed per unit area. Additionally, the distribution could also be modified by providing a bitmap. The color of the bitmap would then affect either the probability of generating the instance or the scale of the generated instances (with black being 0% chance to generate an instance here or scaling the instance to 0% and white being 100%). The users were also able to randomly set the translation, rotation and scaling of all scattered objects – for example setting that all objects scattered will have their scale between 40% and 70% (random for each individual object) and will be offset by exactly 1 meter upwards. For previewing the Corona Scatter results in the viewport window of 3ds Max [1], the users could choose to display only a certain percentage of scattered objects and set a few options specifying how to draw instances – a simple dot, the bounding box of the scattered object, a full mesh of the scattered object, etc.

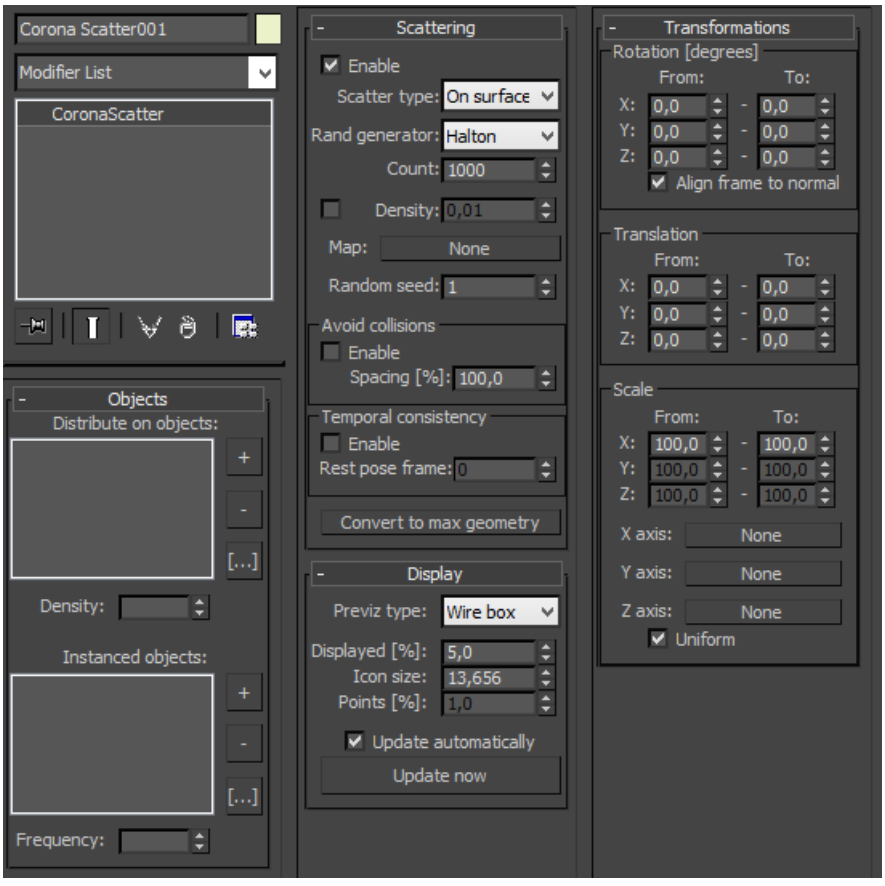

Figure 1.4: Corona Scatter 1.5 user interface.

2. Problem analysis

In this chapter we discuss all three features we set to implement – regular pattern scattering, scattering along spline objects and distribution modification using spline objects in this order. We describe the behavior we chose for each feature, explain all of our choices as well as present solutions to all problems we found during the development.

2.1 Regular pattern scattering

By scattering in a regular pattern we mean placing the *instances of the scattered object* (we will call them just *instances*) on the *distribution object* in a way that some rules are observed. The simplest example would be placing houses in a housing estate. A regular grid is strictly observed, with the individual houses almost identically placed. However, more difficult cases arise – what if we were to scatter on an aligned plane or a sphere? Which rules determine a "regular pattern"? Which direction should the regularity of the pattern be observable from? These questions lead us to the first major choice.

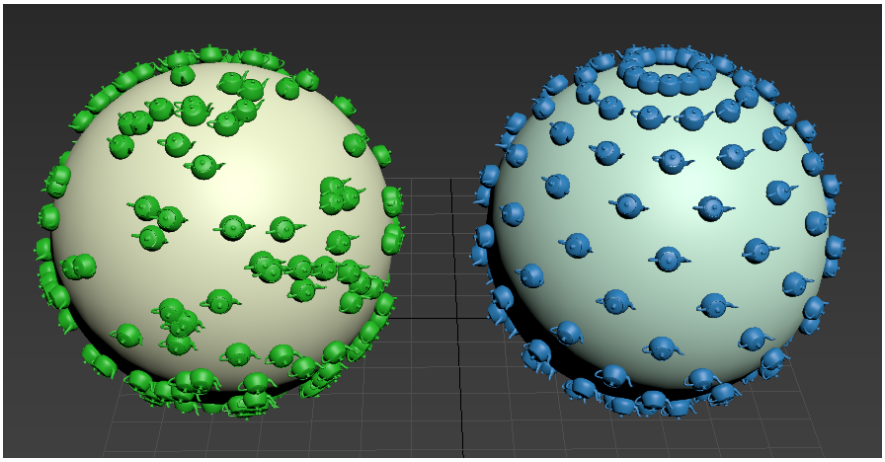

Figure 2.1: A demonstration of what we would like to achieve in this section. The left sphere is an example of a random distribution while the right sphere is an example of a regular distribution.

2.1.1 Choosing a method

Projection

One possible solution is scattering the pattern along a projected axis. This means we would need to project the distribution object's faces along an axis defined by the user and then place the regular pattern on this newly acquired object. This method yields a simple answer to our previous question – the pattern is most observable when looking along the chosen axis. The reader can imagine this method as if the instances appeared in a plane above the object in a regular pattern and then fell until they hit the object, falling in the direction of choice.

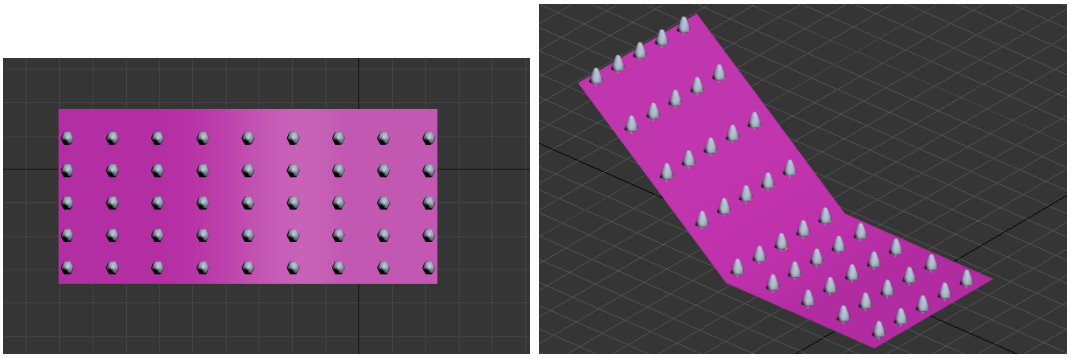

Figure 2.2: Pattern is regular when viewed along "up" axis, pattern becomes less regular when viewed from a different angle.

Let's examine the behavior of this method. The feature we liked about this method is how easy it is for the user to start scattering. If we consider that the vast majority of regular scattering will be done by projecting into the XY plane (i.e. the "ground"), which seems like a reasonable default value, the input from the user in the beginning is very limited, which suits our needs for intuitive program behavior.

However, due to the projection deformation the pattern will be regular when viewed from the axis of projection, but depending on the viewing angle might look less regular than we would like. This is best illustrated on a slope – the steeper the slope, the bigger the spacings between each row would be as shown in Figure 2.2:

This behavior makes scattering on more complicated geometry like pyramids very tricky since, in most cases, each face of the distribution object mesh will be aligned with the axis of projection slightly differently and the pattern will look differently on every face of the mesh.

Another issue we found using this method is regular scattering on the whole surface of the distribution object. Trying to create a sphere with regularly placed spikes using this method is close to impossible which is a limitation we would like to avoid.

Lastly, even though we mentioned the limited user input that is required as a desirable feature, we felt this method was a bit too simplistic. The user couldn't change the input in any other way than altering the axis of projection limiting the user more than they would have liked.

UV space

To describe this method, we will first need to define a UV space and a geometric object's mapping into it. In computer graphics, UV space is a $\mathbb{R}^2$ space generally used for texture mapping – since textures are $2D$ objects (usually images), a way to map them onto a $3D$ geometric object has to be provided. Mapping textures is done by unwrapping the object's surface onto a $2D$ plane, creating a so called UV mapping (much like maps of the Earth in cartography). UV mapping is, in most cases, provided as per-vertex UV coordinates for each triangle of the object's mesh. We note that sometimes a W coordinate is added too, making the space $\mathbb{R}^3$ – a so called UVW space used for example for 3D textures. An example of

sphere mapped into the UV space is shown in Figure 2.3.

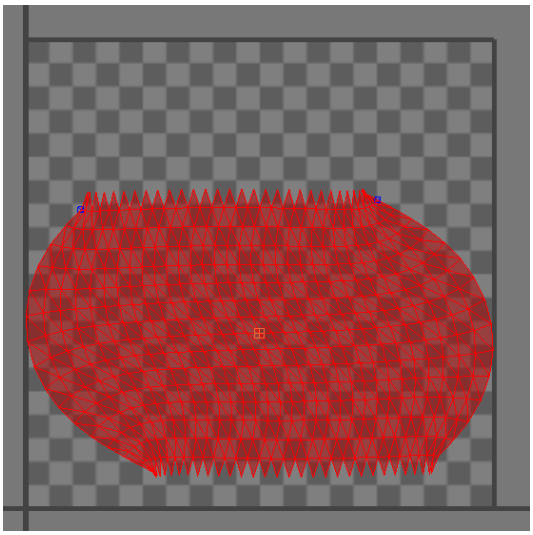

Figure 2.3: Example UV mapping of a sphere.

Now for the regular pattern method. Provided we obtain a UV mapping of the distribution object, we can generate the regular pattern in the UV space and then map it onto the distribution object afterwards.

First thing to notice is this process is slightly more complicated than the projection and requires the user to put a considerable amount of effort before the scattering can begin – providing the UV mapping. In return, the user gets a lot more control over how the pattern should be scattered, since by changing the UV mapping he directly influences the scattered instances. This can include scaling the mapping up to make the resulting instances closer together, deforming the mapping to deform the regular pattern, etc.

Conclusion

Choosing between the two, we decided on implementing the **UV mapping** option, mainly because it allows the user to control the resulting pattern a lot more. In many modern modelling programs like 3ds Max [1] or Cinema 4D [7] the UV mappings of simple objects are generated automatically and the programs contain state-of-the-art tools that make creating the mapping fairly easy even for complicated geometry. This means that the creation of the UV mapping is not a big hurdle and will not deter any potential users as the user base is accustomed to UV mapping. Meanwhile an expert user will be able to finely control the object placement by altering the mapping.

Regarding the computational power needed to execute both versions, it is true that the simple projection is probably easier and therefore faster, but we found that UV mapping still provides results fast enough for the user not to notice any difference.

2.1.2 User input description

Now that we have decided on our general approach, we will present and briefly describe user input into the Corona Scatter. We already mentioned that the

resulting distribution of instances can be modified at will by the user by changing the UV mapping. However, this indirect means of manipulating the instance distribution could be clumsy at times, and therefore we also want to provide a way for the user to change the distribution without completely changing the UV mapping. The parameters that we give the user at her/his disposal are:

- The choice among three basic patterns: regular grid, running grid and hexagonal grid, as shown in Figure 2.4.
- Pattern scale in both U and V axis, which allows the user to change the spacing between "rows" and "columns" of generated instances.
- Pattern jittering (in %), which is the amount of randomness that gets added to each instance in the grid, making the grid completely random at 100% and regular at 0%.
- Pattern offset which makes the pattern shift in the U or V axis. The instances wrap around after getting pushed out of the pattern's edge creating an endless pattern illusion.
- The option to enable planar mapping which is computed by the Scatter itself.

These controls, when combined with the already existing parameters like collision avoidance, should provide the user with a high level of customizability without being overly complicated. All of this input is presented to the user in a form of a clean and compact User Interface which we will discuss later in the Chapter 3.

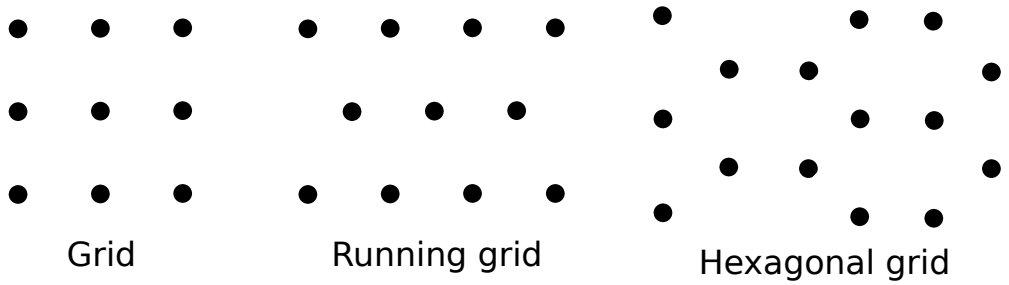

Figure 2.4: Example of all three regular patterns we decided to provide.

2.1.3 Our algorithm

So far, we have illustrated what we would like to achieve and chose an approach we thought was best. We also described the way the user will interact with our program. We should now have a good understanding of what we would like to achieve. We now present the algorithm which we designed to scatter instances in a regular pattern on multiple distribution objects:

Definition: An *axis-aligned bounding box* (AABB) of an object in N-dimensional space is a tuple of points defining a box that fully encloses the object.

Algorithm 1: Scatter objects on multiple mesh objects in a regular pattern

1 Load all distribution objects
2 Load user configuration
3 **for** *each distribution object* ***O*** **do**
4 Find the axis-aligned bounding box **B** of **O** in the UV space
5 Generate the chosen pattern in the bounding box **B**
6 Apply pattern offset and jittering, taking care to wrap instances that overflow
7 Go point by point and map each generated UV point to all triangles it belongs to, placing an instance on each of them (this step involves inverting the UV map)
8 Meanwhile, make sure we stay under the maximum number of instances, exit early if we would generate more instances than the user specified limit
9 **end**
10 If the user has enabled collision avoidance, remove all instances that collide with any previously generated instance

Loading the distribution objects and the user configuration for triangle meshes was already handled by the existing Corona Scatter, so we do not describe it here.

Generating the pattern (step 5 of Algorithm 1)

We want to generate a regular pattern in the bounding box of each distribution object in the UV space. However, we want the program to behave as if the bounding box was only a window through which we look at an infinite pattern in the UV space. This will allow us to respect any transformation of the UV mapping, as we can just transform the window in the same way. For example, if the user translates all the distribution object's triangles in the UV space, we translate the window in the same direction, which will move all generated instances. This creates the illusion of an endless pattern.

While generating the pattern we also need to decide how the user will specify spacing of instances in the UV space. There, again, are several options – the simplest one, and the one we eventually decided to use, is to take the user input as absolute coordinates in the UV space. This provides the user with the most transparent control over the scattering, since more experienced users will be able to set the value however they like and then modify the UV mapping to achieve the desired outcome by scaling, rotating or otherwise modifying the UV mapping of the distribution objects. The second option is to interpret the user input value as the percentage of the width or height of the distribution object's bounding box in the UV space. This may look more intuitive to a less experienced user since setting the spacing to a value of 10% will always yield 10 instances in each row/column. However, this option also means that if the user changed the scale of the UV mapping, nothing would happen. Again, we felt that this behavior would be neither expected nor desired by a more experienced user and opted for

the absolute spacing described above, which did not seem too complicated for a beginner but provided a bit more utility to an expert user.

Offsetting the pattern (step 6 of Algorithm 1)

We already mentioned that translating the distribution object in the UV space (moving the window over an infinite pattern) should offset all generated instances accordingly. In the "User input description" section we also mentioned we want the user to be able to set additional offset directly from the UI. We now need to come up with a method to wrap the instances correctly after they exceed the boundaries of the bounding box of the distribution object in the UV space, which effectively achieves the infinite pattern illusion. To implement this, we need to find the first point at which each pattern starts to repeat itself. We then need to align the distribution object's bounding box in the UV space to the nearest greater point of repetition. After that, we only need to wrap around such instances that exceed the boundary of the bounding box by at least the length of the spacing. This corresponds to the idea of the user looking through a small moving window (the extended bounding box) as shown in Figure 2.5.

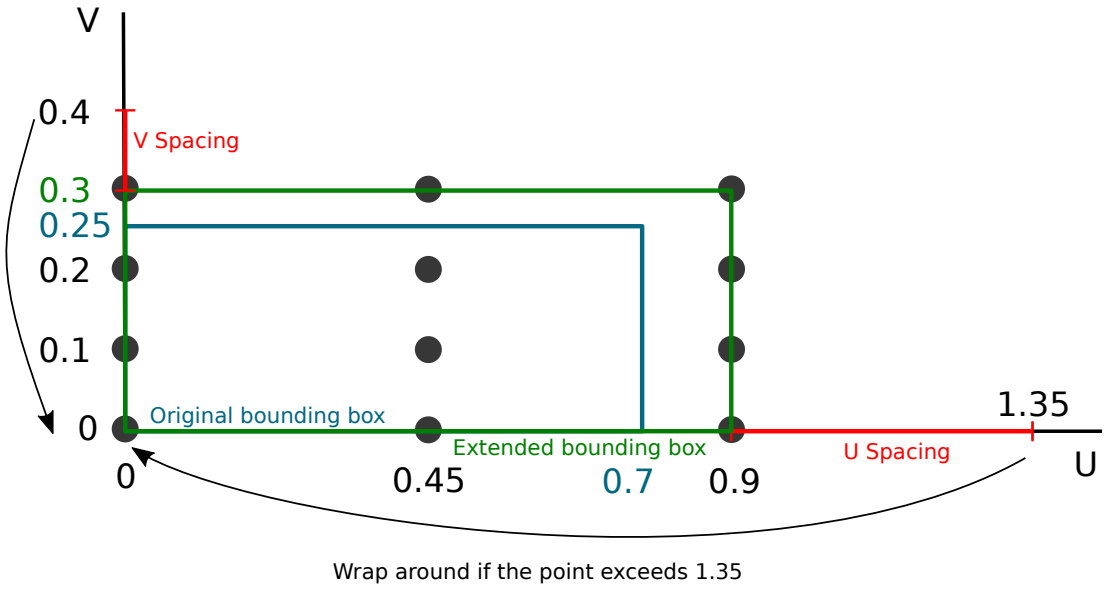

Figure 2.5: Demonstration of aligning the bounding box to the point of repetition in each axis. Spacing is set to 0.45 in the U axis and 0.1 in the V axis.

Placing the instances on distribution object triangles (step 7 of Algorithm 1)

Having generated the grid, we can now start placing instances onto the distribution objects themselves. For that, we need to map the points generated in the UV space onto their corresponding world space positions. Unfortunately, we only have a mapping from the world space into the UV space (specified as per-vertex UV coordinates for each triangle), but not the other way around. To find the inverse of the UV mapping, i.e. to map point in UV space to world space, we use a computer graphics data structure called Bounding Volume Hierarchy:

Definition: *Bounding Volume Hierarchy* tree is a rooted tree built over a number of geometry objects allowing fast searches of all objects (e.g. triangles) containing or intersecting a geometry primitive (like a point or a line). This is accomplished by constructing a tree dividing the objects in each node into subtrees with their associated bounding boxes, keeping the closest objects together, separating the farthest.

We build the Bounding Volume Hierarchy over triangles in the UV space. That is, the BVH is built in a 2D space and the triangle vertices correspond to their per-vertex UV coordinates. The BVH can efficiently answer queries in the following way: *for a given point in UV, return all triangles that contain this point.* This is the key to this algorithm as it provides us with means to identify which points in UV lie in which UV triangles of the distribution object. Knowing this, we can map the point from the UV space to the world space using *barycentric coordinates.*

Definition: *Barycentric coordinates* of a point in a triangle are three coordinates A, B and C, are equal to the ratio of areas of PBC, PAC and PAB respectively to the area of ABC.

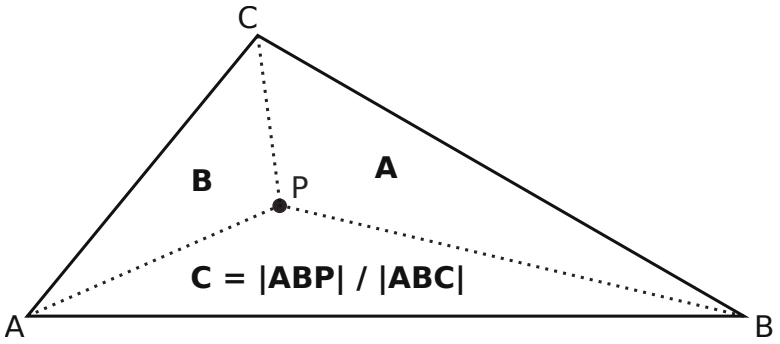

Figure 2.6: Barycentric coordinates of point P in the triangle ABC.

Figure 2.6 illustrates the concept of barycentric coordinates. The inverse mapping (i.e. the mapping from the UV space to the world space) is realised by calculating the barycentric coordinates of the UV point in the UV triangle and then combining the world space coordinates of the triangle verticies with the coefficient of each vertex equal to its barycentric coordinate. The full algorithm for the inverse mapping from UV space to world space is shown in Algorithm 2.

Algorithm 2: Inverse mapping from the UV space to the world space

```
for each point p in the UV space do
    for every UV triangle t this point lies in do
        Find the barycentric coordinates of p in t
        Find the triangle t' in world space which corresponds to t in UV space
        Barycentrically interpolate the vertices of t', yielding the instance
          position p' in world space
    end
end
```

The last problem we are facing is a special case – a point in the grid can lie on the vertex or an edge of the triangle. The BVH tree will correctly return both triangles and with the algorithm as written above, we will place an instance on both of them, resulting in a doubled instance. Simply making sure we only place one instance for each point is not good enough either – it is not uncommon to have faces overlap in a UV mapping. A simple cube is a good example as the most simple mapping maps all faces to $[0, 1]^2$. The best solution we found was: For each UV point $\mathbf{p}$, remember all edges we placed instances on. If we find that we need to place another instance corresponding to that same UV point on the same edge, we discard it instead.

We have now finished describing scattering regular patterns using the UV space mapping. We now continue with the second feature we set to implement – scattering along splines.

2.2 Scattering along splines

One feature Corona Scatter was missing that was requested by many users is scattering along splines, where by "spline" we mean a line-segment object approximating a general curve in R^3. This feature would allow users to scatter e.g. lamp posts near roads or create a link chain. Defining the distribution with a spline shape instead of a triangle mesh has many advantages – processing a spline takes a lot less resources than a comparably smooth triangle mesh surface, the placement is a lot more precise and convenient. In this section, we walk the reader through the development process of this feature.

Figure 2.7: A demonstration of trees scattered along an arc.

2.2.1 Requirements and user input

Based on the input gathered by the Corona team from their user base, we decided on implementing spline scattering with both random and regular options. The random version would scatter on the whole length of the spline, placing instances at random, being affected only by a random seed and the number of instances to generate. Regular mode would require the user to input a spacing which it would keep in between each instance, scattering from the beginning to the end, leaving any unused space at the end. Both modes would provide an avoid collisions

option. The user input is very limited which suits our needs since all of the mentioned parameters are easy to understand with no unexpected behavior.

However, when we consulted this design with one of Corona team's most trusted and expert users, Ludvík Koutný, we came to the conclusion that the general user base of Corona Scatter would appreciate one mode with the option to linearly blend from random to regular a lot more. The other input we received was that the user should be able to explicitly set the spacing between instances on the spline – this allows the user to, for example, place a lamp post every 10 meters, which is close to a real world situation.

We also wanted to have the random version generating the instances with a uniform distribution and provide the user with the ability to offset all instances on the spline in the same direction.

While developing this feature, we released a preview version in one of Corona Renderer's daily builds [8] that was tested by a few dedicated Corona users and we received more user feedback – the first instance should always be placed at the start of the spline and the last instance should be, if possible because of the spacing set, at the very end.

2.2.2 The algorithm

Considering all these requirements on how this feature should behave, we designed Algorithm 3. Let us now explain each step of the algorithm.

Algorithm 3: Distribute instances along a spline object

Data: The distance *spacing* between each instance, the amount of randomness *jitter*, and the amount by which we should offset all instances $offset$

1 **for** *each spline* S **do**
2 Let $N = \lfloor \frac{spline\ length}{spacing} \rfloor$ be the number of instances to generate
3 Scattering first on a $[0, 1]$ interval, calculating new spacing: $s = \frac{spacing}{spline\ length}$
4 Generate points p_i for $i = 0 \dots N - 1$, r a random number for each instance, $p_i = i \times spacing + jitter \times r + offset \bmod spacing$
5 Discard all points that exceed the interval boundaries
6 Map the points from the interval $[0, 1]$ back onto spline S
7 **end**

Scattering first into $[0, 1]$ instead of the spline (step 3 of Algorithm 3)

When designing the algorithm, we decided to first scatter the instances into the interval $[0, 1]$ and map them onto the spline afterwards. While this is not completely necessary for the algorithm, we chose to include this level of abstraction to allow implementing further changes or features a bit more easily. One feature we thought of that could be easily added into the Corona Scatter in the future because of this abstraction is allowing the user to alter the mapping by providing a custom curve. This would allow the user to specify different densities along the spline, making the distribution more or less dense in places. However, we decided that this feature would be, at least if introduced without the users having any experience with scattering along splines, too overwhelming and decided to omit

it. But since we left the abstraction form of the $[0, 1]$ in place, implementing this should be an easy task for the developers at the Corona team in the future.

Instance placing (step 4 of Algorithm 3)

The scattering method we decided on was to divide the interval $[0, 1]$ into intervals exactly $\frac{spacing}{spline\ length} = new\ spacing$ wide. To seamlessly blend from the random and regular distribution we added an input parameter which we call jitter which affects the length of the random movement for each instance – at $jitter = 1$ the length would be the whole interval, at $jitter = 0$ there would be no movement at all. Since we want to place the first instance at the beginning of the spline, the first interval needs to start at $-new\ spacing/2$. Similarly, we need to allow the last interval to go beyond 1 because we want to make sure an instance can be scattered at the very end. An example of such distribution is shown in Figure 2.8.

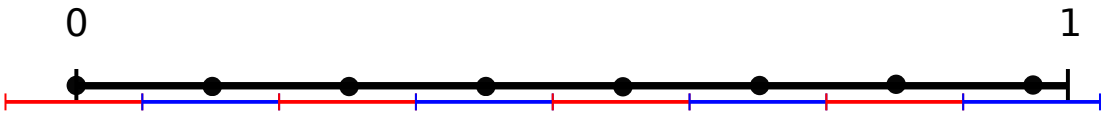

Figure 2.8: An example of a distribution on the interval $[0, 1]$ placing 8 instances with a bit of space left at the end.

Notice that, in Figure 2.8, the line did not fit the 8 instances exactly, therefore a small bit of space was left at the end. This does not matter since the users can set the spacing so the instances fit perfectly.

While the offset parameter looks similar to our offsetting problem in "Regular pattern scattering" section of this chapter, notice that we cannot implement it the same way. We cannot modify the interval $[0, 1]$ (mainly make it wider) as that would break the level of abstraction that we designed it for. Our offsetting method is based on one simple observation: Let o be the user-specified offset. If, instead of offsetting all instances by o, we offset the instances by o mod $spacing$, at most one instance (the last one) will get pushed out of $[0, 1]$ for any given o. This will give the user the illusion that the instances are actually moving around, which is exactly what we wanted.

Behavior at the end points (step 5 of Algorithm 2)

We need to discuss the behavior at the end points for two reasons – the offset and the jitter parameters. We have already shown that if we discard the instances as they exceed the interval's boundaries, the offset will work exactly as we want.

We now need to find a solution for behavior at the end points for the jitter parameter. From the user's point of view, both offset and jitter should behave the same way because we do not want to confuse the user. This means that when the instance leaves the $[0, 1]$ interval, it should disappear, as it does for the offset.

However, we would also like to have the instances follow a uniform distribution. Fortunately, the distribution as we described it now is uniform. By definition, we have 100% percent chance we will generate an instance in all intervals.

Half of the first interval will always be outside $[0, 1]$. This means that if we discard all instances outside $[0, 1]$, the first interval has a 50% chance to generate an instance. Because its length is half of the length of the others, this keeps the distribution uniform. The last interval behaves very similarly.

This means that discarding the instances outside $[0, 1]$ behaves consistently from the user's point of view and generates a correct uniform distribution.

Instance orientation

When scattering on splines, we also need to consider the resulting instances' orientations. When we were scattering on mesh surfaces, the orientation of each instance was set to the face's normal. Since the segments of spline objects do not have a unique normal, we need to find a method to reliably orientate all instances in a way that is both consistent and intuitive for the user.

A good starting point is the vertical axis of the world space (we will call it the Z axis). While the other axes X and Y carry some meaning, they are, as far as the user is concerned, generally interchangeable. The vertical Z axis on the other hand is very important and we should take extra care to ensure that all instances are aligned to the Z axis by default.

However, we would like the user to be able to orientate the instances in such a way which respects the curvature of the spline. Figure 2.9 shows an example of such an orientation. This would require the user to set a *reference direction* vector – the orientation of the first instance on the first line segment of the spline. Each other instance would then be oriented in a direction resulting from rotating the reference direction with respect to the spline's curvature.

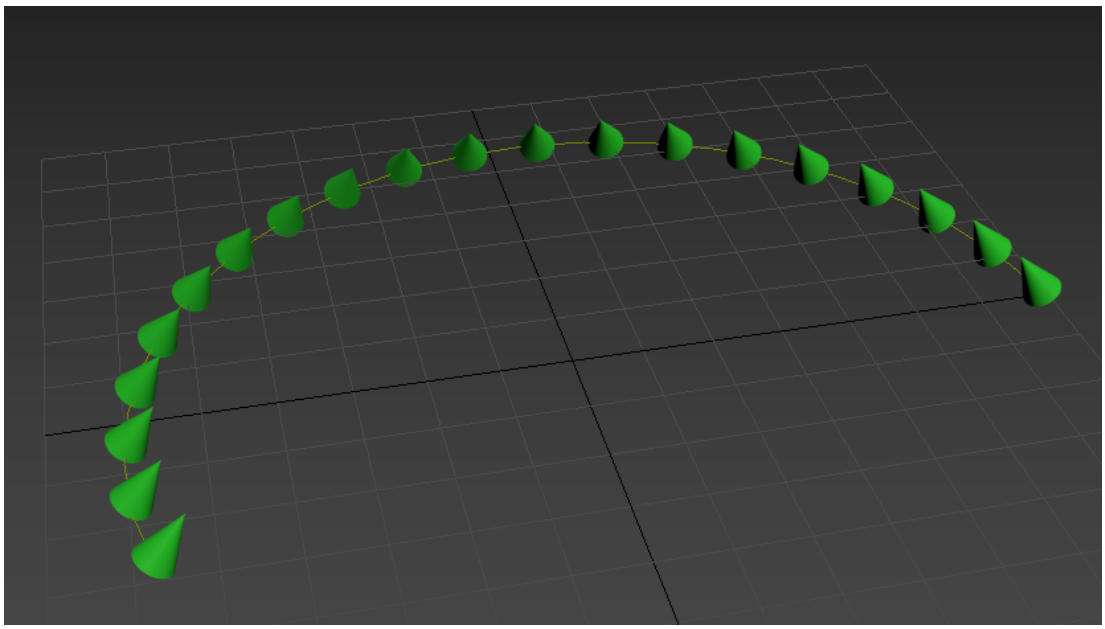

Figure 2.9: An example of instance orientation that respects the curvature of the spline.

The first problem we had to solve was figuring out how the user should input the reference direction variable. We settled on what 3ds Max [1] calls a "Pivot

point". A pivot point is a property of each object and describes an orthogonal coordinate system. It can be moved, rotated and scaled as desired with no effect on the object itself. Since each spline has its own pivot object we decided that the vertical Z axis of this pivot would be a good default reference direction.

While looking for a solution for respecting the curvature of the spline we discovered the Bishop frame [9] which describes a particle moving along a curve in a 3-dimensional space. While these formulas are defined for differential geometry which is not what we are dealing with (as all the splines are in the form of line segments), we realized we could use this concept and apply it in our case too:

We create an orthogonal coordinate system at the beginning of the spline from the two vectors that are available to us: the spline's unique tangent t at the starting point and the user input reference direction r. Calculating the cross product of r and t and normalizing all three vectors gives us an orthonormal coordinate system. Now going segment by segment, we rotate the coordinate system according to the spline's curvature, obtaining a direction in which we should orientate all instances on this segment.

Algorithm 4: Rotation of an input vector along a spline object

Data: Reference direction r for the first segment, an array of line segments S
Result: Array of directions D, where D_i corresponds to the orientation of all instances on ith line segment

$D_0 = r$ – instances on the first line segment should be oriented in the reference direction
for *each segment* S_i **do**
    Subtract the segment's end points to receive its tangent T_i
    The angle which we need the matrix to rotate by is $cos\theta = \frac{T_{i-1} \cdot T_i}{\|T_{i-1}\|}$
    The axis which we rotate around is $a = T_{i-1} \times T_i$
    Compute the rotation matrix M from T_{i-1} to T_i
    Use M to rotate D_{i-1} to obtain D_i
end

For the algorithm to be complete, we need to explain how to obtain a matrix performing a rotation around a certain axis by a certain angle. The matrix we use is a slightly modified (the diagonals are simplified) version of the matrix derived in section 9.2 of this doctoral thesis [10] and is as follows:

$$\begin{pmatrix} {a_x}^2(1-cos\theta)+cos\theta & a_x a_y(1-cos\theta)-a_z sin\theta & a_x a_z(1-cos\theta)-a_y sin\theta \\ a_x a_y(1-cos\theta)-a_z sin\theta & {a_y}^2(1-cos\theta)+cos\theta & a_y a_z(1-cos\theta)-a_x sin\theta \\ a_x a_z(1-cos\theta)-a_y sin\theta & a_y a_z(1-cos\theta)-a_x sin\theta & {a_z}^2(1-cos\theta)+cos\theta \end{pmatrix}$$

Notice that instead of rotating the whole coordinate system, we actually just need to rotate the reference direction r, as we are not using the third axis for anything.

This algorithm solved the issue and produced consistent results even when scattering on more complicated spline objects.

We briefly describe a solution we tried which ended up not being consistent enough. We describe it mainly because it looked very promising and we did not think of the unwanted behavior prior to seeing it happen. The solution was based

on the cross product – taking the reference direction (passed as the Z axis of the pivot point of each spline as discussed above) and calculating its cross product with the tangent of the spline at the point of the instance's placement. For open splines that were not too curved, this worked very well, however, we quickly ran into a problem – the cross product operation is sensitive to the orientation of the two vectors becoming positive or negative depending on the angle between them. This proved to be a problem on closed or almost closed splines like circles where it produced results as shown in the Figure 2.10.

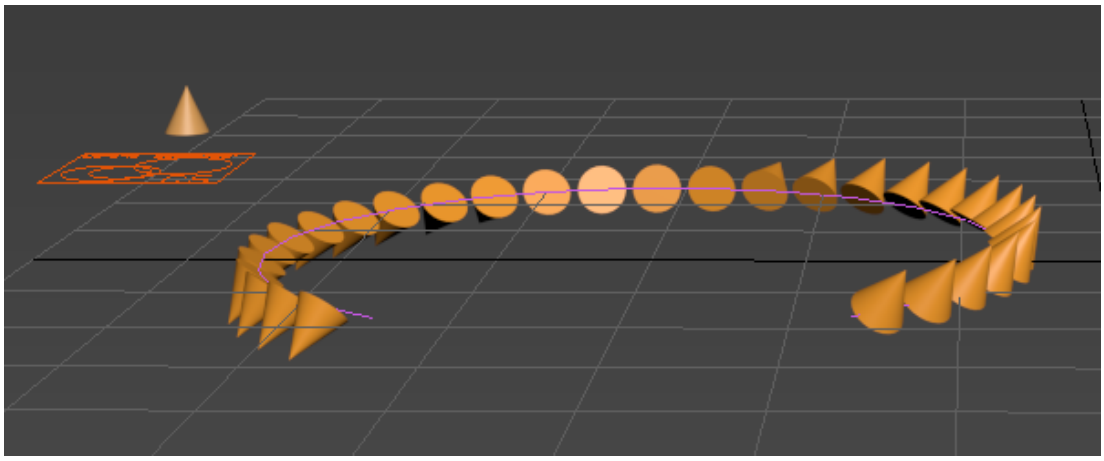

Figure 2.10: An example of the instance orientation algorithm based on cross product failing.

In this section of the chapter we covered scattering directly along spline objects in $3D$ space. In the upcoming section, we describe utilizing splines to modify the distribution on mesh distribution objects. These two are separate features of Corona Scatter and should not be confused.

2.3 Distribution modification using splines

One of the Corona Scatter's most requested features by the user base of Corona Renderer was the ability to modify the distribution locally in the vicinity of a spline object (we will call the splines *modification splines*). The distributed instances would either be completely eliminated or for example have their size reduced more and more as they get closer and closer to the projection of the spline on the distribution object. As we mentioned in the Introduction chapter, this functionality is very handy for creating for example paths or roads through some vegetation. Figure 2.11 illustrates what we would like to achieve.

This problem can be separated into two cases that differ quite substantially. The behavior of these modification splines should change depending on the type of the spline – the spline can be either closed or open. If the spline is open, the modification should be happening in a certain distance, or a "band" around the spline itself, on both sides of the spline. If the spline is closed, we would like it to discard all instances that lie inside or on the edges of the spline and modify the instances outside of the area as shown in Figure 2.12.

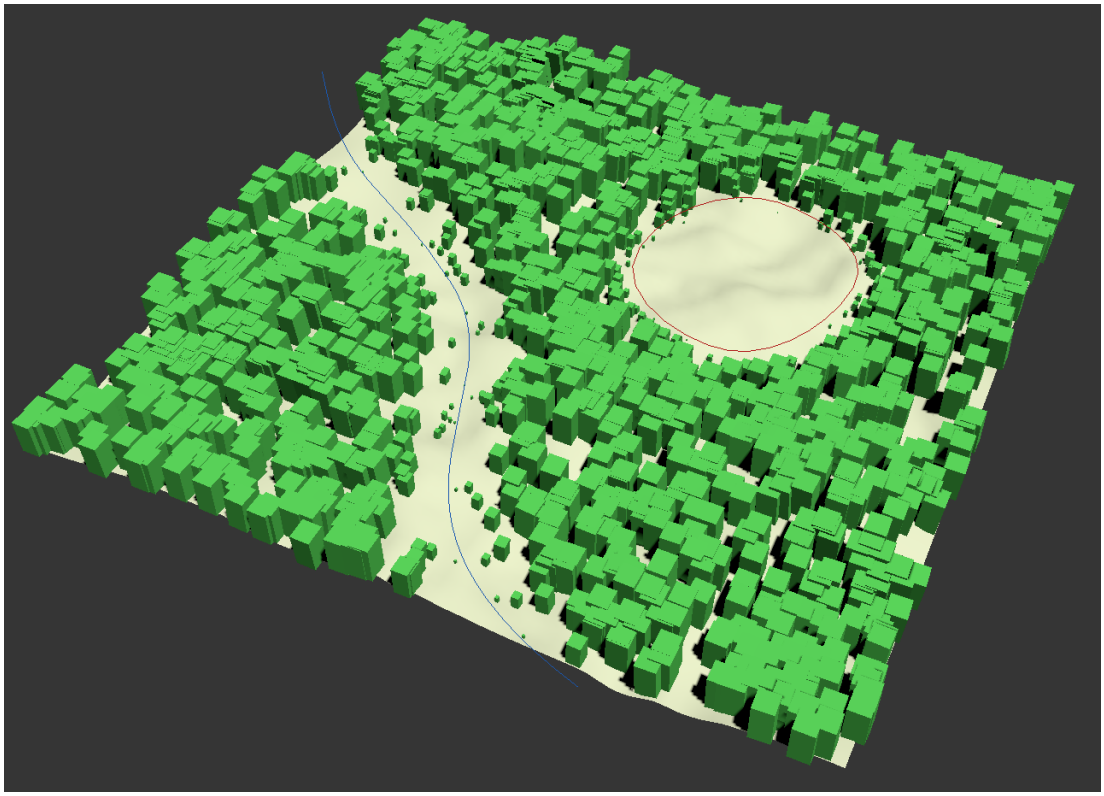

Figure 2.11: An example of a distribution being modified by both an open spline (blue line on the left) and a closed spline (red circle on the right).

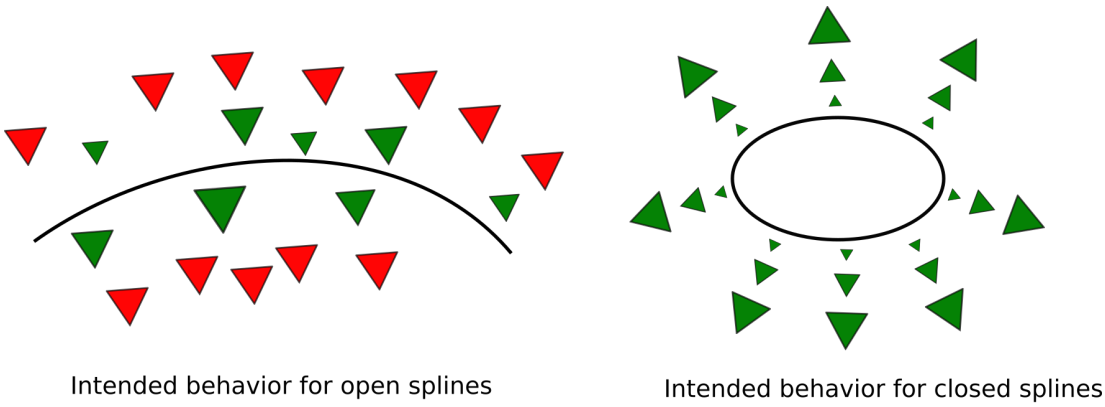

Figure 2.12: The intended behavior of both open and closed splines.

We require that in both of these cases the modification happens not based on the distance to the spline but based on the distance to the spline's projection onto the distribution object. This projection will be limited to the choice of one of X, Y or Z world space axes. We will discuss this limitation in detail in the "User input" section that follows.

2.3.1 User input

Let us briefly reiterate on the ways the user will be able to modify the distribution and explain each spline parameter. Each modifying spline will have the following settings:

1. Near region distance
2. Far region distance
3. Scale multiplier
4. Density multiplier
5. Projection axis X/Y/Z

Near and far region distances

To allow the user to control the scaling and density of near instances a bit more finely, we decided to divide the vicinity of each modification spline into two regions, which we called the *near* and *far* region, as shown in Figure 2.13:

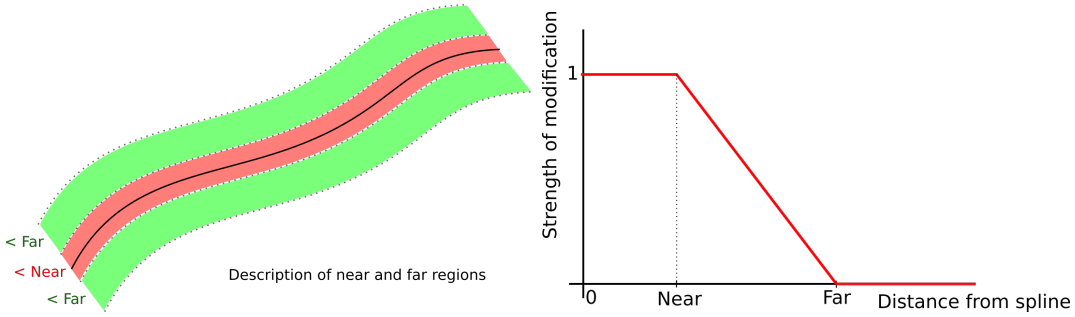

Figure 2.13: The image on the left illustrates the concept of *near* and *far* regions. The graph on the right illustrates the dependency of the strenght of the modification on the distance from the spline object.

These two regions differ in behavior in the following way: in the *near* region, no instances are allowed at all and all will be excluded without being modified. In the *far* region, the instances get modified linearly based on their distance from the near region border, as seen in the graph in Figure 2.13.

We gave the user the ability to control both the width of the red *near* region and the width of the green *far* region directly from the UI, with both being able to be set to any number from $[0, \infty)$. We think this allows the user sufficient creative freedom while still being very clear.

Scale and density multiplier

We already mentioned that the two main things the distance from the modification spline should affect are the density of the distribution and the scale of each instance. However, we wanted to give the user more control over these two modifications than just simple turning on or off. We found that allowing the user to set a weight of the modification was a good compromise. This allows the

user to set two numbers d and s to any value in the range of $[0, 1]$ determining how much this spline affects density or scale respectively. When both are set to 0, the modification spline only excludes all instances in the near region (as per near region's definition) and doesn't modify instances in the far region at all. Setting s to 1 will make the modification take full effect – instances will get scaled to very small when approaching the near region and to almost their full scale when approaching the borders of the far region. Setting d to 1 will reject more instances in the far region, with each instance's rejection likelihood being linearly dependant on its distance from the modification spline. See Figure 2.14 for illustration.

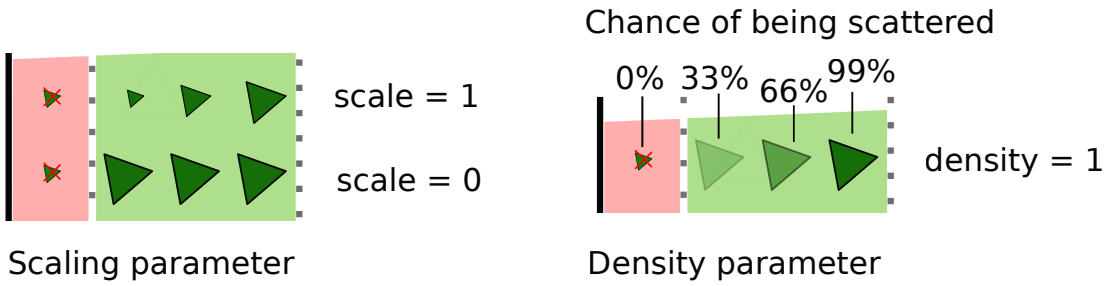

Figure 2.14: The left image illustrates the effect of the scale multiplier, the image on the right illustrates the effect of the density multiplier.

Projection Axes

We decided that the projection will be limited to the choice of one of X, Y or Z world space axes. The main reason behind this decision was that we wanted to strike a good balance between comfortable handling and user limitation. Allowing any arbitrary axis would not slow the program as much, however, it might slow down the user considerably. This is because setting the axis in the UI would probably need to be handled using the pivot point as we did in scattering along splines. We did not like this because unlike while scattering along splines where the pivot point is used to achieve a specific rotation, the choice of the projection axis is very important and should be highlighted to the user as much as possible – without the need to navigate to an entirely different UI tab. Meanwhile limiting the choice to X, Y or Z allows us to present three buttons and let the user choose simply by clicking on one of them. This is a lot more comfortable for the user, although it limits the user. Upon discussing it with our expert user Koutný, we were told that projecting along a different axis than X, Y and Z is very rare and not worth the extra effort both for us and for the user. This made us choose to limit the projection axis to one of X, Y and Z. We chose the vertical Z axis as the default value since we think it will cover the majority of all use cases as this feature is most often used to create special regions on flat surfaces, such as roads, paths, parking lots and pull-ins.

2.3.2 The algorithm

Now that we have a good understanding of what we would like to achieve, we present the algorithm that we implemented to take spline modification objects and modify the already existing scattering based on the parameters:

Algorithm 5: Distribution modification in the vicinity of splines

```
1  Project the distribution objects' triangles T to get T'
2  for each modification spline s: do
       // project each spline onto all distribution objects
3      Project the spline along the chosen axis to get s'
4      for each segment g of s' do
5          Find the triangles T'' of T' intersecting g
6          For each triangle t of T'', find the line segment g' ⊂ g such that t ∩ g = g'
7          Find the barycentric coordinates of the endpoints of g' and interpolate g'
             back to the world space to get g projected onto T
8      end
9  end
10 for each instance do
       // rejection sampling
11     Calculate the distance from each projected spline, modifying the scale and
         density ratio
12     If the instance is in the near region, reject it
13     Randomly decide to keep the instance or reject it, with chances equal to the
         density ratio calculated based on distance
14     If we didn't reject the instance, scale it by the scale ratio and keep it in the
         distribution
15 end
```

Notice that projecting segments of a spline on a $3D$ mesh along a certain axis means finding all triangles that the segment intersects in the projection along the given axis. This means that, in reality, the problem is actually $2D$, as the projection reduces the dimension of both the mesh and the spline. We therefore need to firstly project the whole mesh of all distribution objects, then project all modification splines. After that we find the projection of the spline onto the mesh in $2D$ – all line segments that are the results of the projected spline intersecting the projected triangles. Lastly, we interpolate the spline back to $3D$ world space. Let us examine each step:

Projecting the distribution objects (step 1 of Algorithm 5)

Since we decided to project the splines only in X, Y and Z axes, projecting the distribution object is straightforward – simply taking all the triangles in the meshes and setting one of the coordinates to zero. Since we want to avoid doing this projection too many times, we want to project all distribution objects in all axes that will be used right in the beginning (e.g. if the user has 3 modification splines, 2 of them are projected in Z and 1 of them is projected in Y, we want to project the distribution objects in Y and Z). We are going to use this projection in the following part of the algorithm.

At this point, the algorithm will be different for open and closed splines. This is because as we already stated, we would like for the behavior to differ in these cases. For open splines, we want to modify the distribution in its vicinity. For closed splines, we would like to reject all instances that lie inside the area and modify all instances that lie near the modification spline's borders. The 3ds Max [1] application allows us to do this easily, because open splines get passed to our plugin as a line segment soup, whereas closed splines get passed as a simple triangle mesh that has the closed spline as its boundary. We start with open splines.

Projecting open splines onto the distribution objects (step 2 of Algorithm 5)

Having projected the distribution objects meshes, we now need to create new splines – the projections of the user input modification splines onto this new mesh. By projecting the splines along their respective axes chosen by the user, we reduce this problem into a 2-dimensional problem in the form of *"For a given line segment L and given set of triangles T, return all line segments that are created by intersecting L and T."*. After solving this problem, the only thing we need to do is placing the $2D$ line segments returned back into the 3D world space, which yields the projections of input splines.

Finding the intersections of the projected mesh triangles and the projected splines consists of two parts. We need to find which triangles intersect which line segments of the spline and we need to be able to calculate the new line segment resulting from this intersection.

Finding triangles intersecting line segments

To find which triangles intersect which line segments, we use the Bounding Volume Hierarchy tree which we discussed in the "Regular pattern scattering" section of this chapter. However, since the standard BVH structure answers queries about points lying in triangles, we will need to modify the traversal of the tree since the query we are looking to solve is *"For a given line segment, find all triangles the line segment intersects."*.

The most important feature of the traversation method is speed since there will be one BVH traversal per line segment and each of these traversals will require roughly $log(N)$ tests within the BVH. Once the traversal is complete, we check if the line segment intersects the triangle which was returned by the BVH tree.

The intersection of a bounding box and a line segment is a very similar problem to intersection of a bounding box and a ray, a topic researched very extensively in computer graphics, thanks to Ray Tracing renderers. The only modification we need to make limiting the ray's "length" to the length of our segment, which is trivial. The book Physically Based Rendering [11] provides us with a fast method of checking whether a ray intersects an axis-aligned bounding box:

The axis aligned bounding box (AABB) in 2-dimensional space gives us two regions – the stripe areas between the bounding box's boundary lines for both the X and Y axis. The method calculates the intersection of the ray with each of these regions as a parametric interval of the ray and then intersects these intervals, if the interval is non-degenerate, the ray intersects the AABB, otherwise it does not. See Figure 2.15 for illustration, the pseudocode follows:

Algorithm 6: Intersection of a line segment and a bounding box

Data: Parametric ray : $point + direction \times t$, where t is from $[0, 1]$, Bounding box: pointMin, pointMax

```
1  for each axis x,y do
       // Find the intersections with the stripe for each axis
2      tNear = (pointMin[axis] - point[axis]) / direction[axis]
3      tFar = (pointMax[axis] - point[axis]) / direction[axis]
       // tFar is the second intersection, tNear is the first
          intersection
4      if tNear > tFar then
5          swap(tNear, tFar)
6      end
       // Intersect [tNear, tFar] with [t0,t1]
7      t0 = tNear > t0 ? tNear : t0
8      t1 = tFar < t1 ? tFar : t1
9      if t0 > t1 then
           // The is degenerate, there is no intersection
10         return false
11     end
12 end
13 return true
```

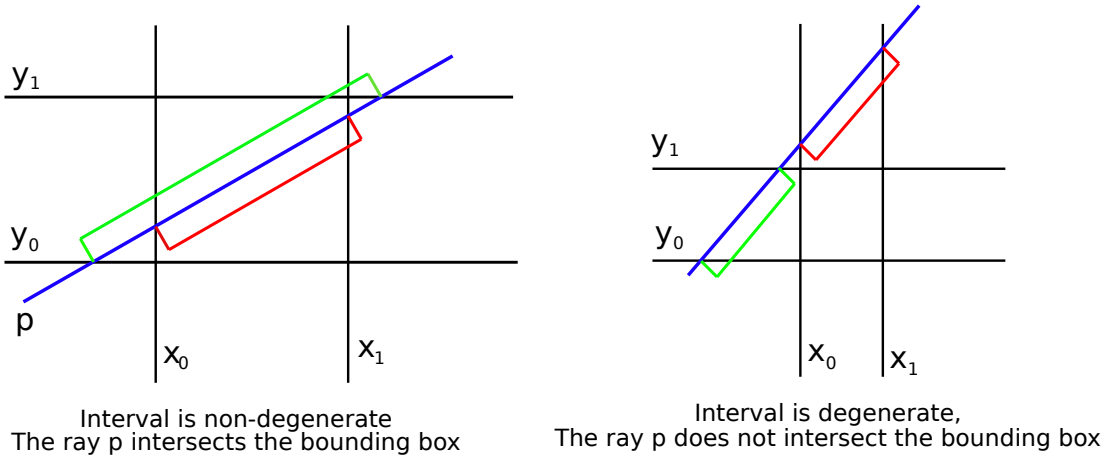

Figure 2.15: The left image illustrates the case when the line and the bounding box intersect. The right image illustrates the case when the line does not intersect the bounding box.

This method allows us to traverse the BVH very quickly and find all projected triangles that intersect the segment.

Finding the line segment resulting from the intersection of a triangle and a line segment

How do we find the part of the segment that intersects with the triangle? We found that the quite simple method of intersecting the segment with each of the triangle's sides provided a method fast enough for our needs and did not need to be improved in any way. Therefore we simply intersect the triangle's sides one by one with our segment, remembering the point of the intersection, in the end

we get two points if the intersection is a line, or just a single number if the line intersects the triangle in its vertex. These are the endpoints of the resulting line segment.

Transforming the points back to world space

Having now found all line segments that intersected with the triangles of the meshes of distribution objects, we need to transform these from the $2D$ projection space back to the $3D$ world space. We can accomplish this transformation exactly as we did in "Regular pattern scattering" section of this chapter. For each segment inside a triangle, we calculate the barycentric coordinates of the segment's end points and then barycentrically interpolate the triangle's world space vertices using the calculated barycentric coordinates. This way, we receive the projected spline segment in the world space.

Calculating distances from open splines, rejecting the instances (step 10 of Algorithm 5)

Now that we have projected both the distribution object and the modification splines, all that is left to do is calculate the distance to each of the projected splines for each instance and modify the instance accordingly. Finding the closest object to a point is another type of query the BVH tree is very good for. We build a BVH tree above each projected spline and then for each generated instance we use the BVH to find the closest spline segment to the instance. We then calculate the distance of this instance to that segment. We repeat this for each single projected spline object receiving multiple distances.

We decided that the scale modification will be multiplicative between the different splines – if two splines would shrink an instance by 50% each, the resulting scale of the instance should be 25%. This means that we will multiply the coefficient across all splines that the instance fall's in the far range of and scale the instance at the end.

The density modification should follow similar multiplicative rules but we will need to generate a random number for each instance, to see if we should reject it or not. If the random number generated for this instance is lower than the density coefficient we got by multiplying across all splines, we keep the instance. Otherwise the instance will get rejected.

Projecting closed splines onto the distribution objects (step 2 of Algorithm 5

As we already mentioned, a closed splines is, for our purpose, equal to a triangle mesh representing the surface with the spline as its boundary. This means that we need to solve the problems we already solved for line segments again, this time for triangles.

The whole proccess of projecting the area of the close spline onto distribution objects starts with projecting the distribution objects themselves (step 1 of Algorithm 5), which we already discussed above. Having projected the distribution objects, we now need to project all triangles of the closed spline's area. After that, we need to find all triangles of distribution objects that intersect with the triangles of the closed spline's area in the projected space. We then calculate all

triangles that result from this intersection – the projected area. Finally, we map the projected area back into the world space. Let us now discuss the difficult steps of this process.

We already know that projecting the closed spline area's triangle along the X,Y or Z axis is as simple as leaving out the respective coordinate.

Having accomplished this, we now need a reliable way to find all distribution objects' triangles T_D that intersect with each triangle of the closed spline area triangles T_A. We again use the BVH tree and a custom way to traverse it. This time, the input query has the form of *"For a given triangle $t \in T_A$, find all triangles $T_O \subset T_D$ such that $\forall t' \in T_O : t \cap t' \neq \emptyset$"*. To save speed during the traversal, we only check the intersection of the bounding box of triangle t with the nodes of the BVH tree instead of checking the intersection of the triangle t itself with the bounding boxes of the nodes. Using this traversal, the BVH tree will quickly return all such triangles T_O.

Now we need to find the new triangle mesh that is the result of t intersecting each triangle $t_d \in T_O$. To find the intersection we use the simple method of intersecting each side of the triangle t with each side of the triangle t_d and checking if any verticies of t lie in t_d and vice versa. We found that this simple method was quick and suited our needs perfectly. This yields a number of verticies that are the intersection points of the two triangles.

Having calculated all verticies in which triangles t and t_d intersect, we now need to construct a triangle mesh using these verticies – *triangulate* them. An observation that the result of the intersection of two triangles can only be a convex mesh will help us greatly. Notice that triangulating a convex set of verticies can easily be done by sorting these verticies clockwise (or counter-clockwise) and then going in a clockwise order and adding a new triangle consisting of two neighbouring verticies in the sorting and a center of mass of the whole set. Figure 2.16 illustrates this process.

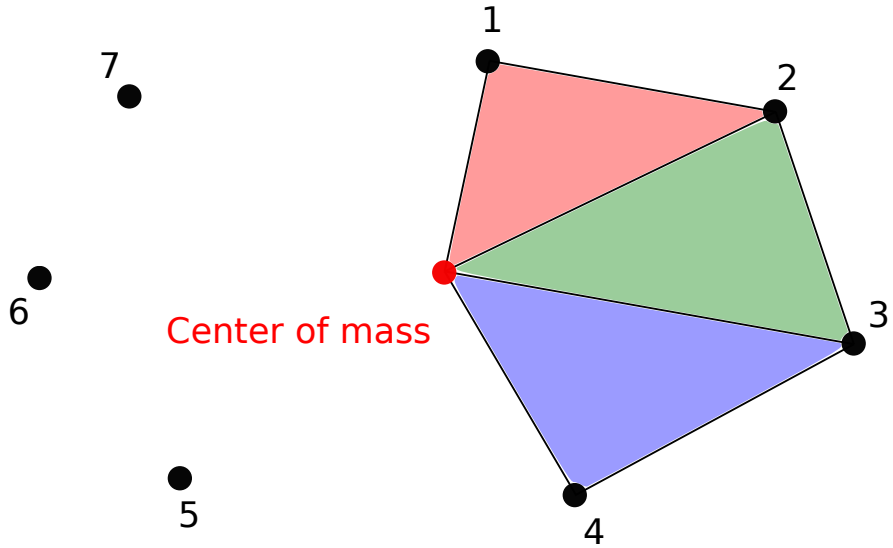

Figure 2.16: An illustration of an incomplete clockwise triangulation process. The verticies are numbered according to their position in the clockwise sorting.

This calculation leaves us with the last and least of the problems. Inverting the projection and transforming the triangles back to world space can be done

exactly the same as before – using barycentric coordinates. This means that while calculating the intersection points, we also remember the barycentric coordinates of each of the points with respect to the triangle t. After we have constructed the new triangulated mesh, we then use the barycentric coordinates of each point to transform each point back to the world space.

Calculating distances from closed splines, rejecting instances (step 10 of Algorithm 5

As we already mentioned at the beginning of this chapter the funcionality of closed splines is two fold – we should reject all instances that lie in the projected area and modify the instances that lie outside and close to the area's borders.

We start with rejecting instances that lie inside the area. This, again, is an easy task for the BVH tree structure. We build one BVH tree above each projected area. For each instance we then check if the instance's position (a point in the $3D$ world space) is inside any triangle of the projected area. If it is, we reject the instance, otherwise we keep it.

The second part of this section – modifying the intances that lie close outside will use the same BVH tree, however, we will use the same type of query as we did for solving the same problem for open splines. We will ask the BVH structure to return the closest triangle to the instance's position and modify the instance depending on the distance to this triangle. This requires us to be able to calculate the distance from a point to a triangle. We found that taking the minimum of the distances to each of the triangle's sides had the best ratio of precision and speed. While this method is not as fast as approximating the distance (for example by only calculating the distances between the point and the verticies of the triangle and taking the minimum) we cannot lose much precision as the excluded area might look deformed due to precision errors.

This concludes the section examining distribution modification using splines, as well as the whole Problem Analysis chapter. In the next chapter we closely examine the program's implementation as well as all the tools and libraries we used.

3. Implementation

The structure of the Corona Scatter code is divided into a few logical pieces, namely the Scatter Core, the Scatter plugin for 3ds Max [1] and our OpenGL viewer application. Scatter Core is a class that holds all the scattering functionality. The Scatter 3ds Max plugin uses the Scatter Core to scatter while providing and managing the UI in 3ds Max. Our OpenGL viewer is very similar to the Scatter plugin – it is a simple OpenGL application that can showcase the functionality of Corona Scatter. It has a simple UI where the user can set all settings relevant to this thesis as well as provide custom geometry to scatter on.

To avoid overwhelming the reader, we first briefly describe our simple OpenGL viewer application and its structure, as well as its connection to the Corona Scatter Core itself. This will neatly illustrate what Corona Scatter's API looks like and what data the program needs to function properly. After that, we describe the Corona Scatter Core itself in more detail with the complete data-flow chart to highlight where the data provided through the Scatter's API flows next and how it is processed to achieve all the goals we listed in the two previous chapters. We will then briefly discuss the Scatter 3ds Max plugin, how it works and what we changed in this thesis.

3.1 Corona Renderer code base

Before we dive into the documentation itself, we want to point out a few features of Corona Renderer's code base to prevent future confusion when we start to discuss the implementation. The developers of the Corona team are developing their code base very actively and they keep it to very high standards. This includes developing their own functionality and prioritizing it over the functionality provided by the standard C++ library [12].

The most striking new features for the user of Corona Renderer's API are the custom container classes like Stack, Array, StaticArray, etc and geometry types like Vector2, Pos (a representation of a point in $3D$ space) and Dir (a representation of a direction in $3D$ space). Being accustomed to the custom container range is important since some parts of the API, Corona Scatter included, expect their input data to be placed in these containers. All geometry types have very efficiently implemented operations to work with them.

Other features of the Corona code base include a custom parallel Scheduler class which makes creating and managing parallel tasks very easy, as well as a plethora of computer graphics specific data structures like a BVH tree we discussed in Chapter 2.

3.2 OpenGL viewer

When we started implementing this thesis, we wanted to have an easy, quick and reliable way to preview the output of Corona Scatter. We decided on implementing a small and clean application that would simply scatter on the geometry it receives and allow us to set the settings that are relevant for this thesis and

not much more. This also allows the reader of this thesis to try Corona Scatter himself, without the need to install the whole 3ds Max program.

Additionally, during the later stages of development we found out that implementing the OpenGL viewer has one very significant upside. The Corona team allowed us to profile the Scatter Core using Intel's VTune Amplifier [13] which allowed us to inspect the program's weakest points that are slowing it down. Unfortunately, the VTune software is not able to measure the performance of the Corona Scatter plugin in a running 3ds Max. However, measuring through the OpenGL viewer application worked flawlessly and we were able to locate and speed up a few of the program's hotspots.

In the end, the OpenGL viewer proved to be a handy tool for profiling the Scatter Core as well as a user-friendly program that can demonstrate the capabilities of Corona Scatter.

3.2.1 Using OpenGL

We start the OpenGL viewer application documentation with libraries we used to implement the OpenGL standard. Since OpenGL is a standard that tries to be as clean of any OS (Operating System) specific details as possible, we need a library that will provide us with the basic setup to utilize the capabilities of OpenGL – mainly a window with an OpenGL context to draw in and some event handling (like mouse click events, keyboard presses, etc.). We decided to use the **GLFW** [14] library. While many more libraries exist (see [15]) and the differences for our very limited use would be minimal, we found out during the development that GLFW was the most used window library in different tutorials and other resources, making it really easy to learn to work with this library.

Since OpenGL implementation is usually provided by the hardware manufacturers, if we tried to code it using this code directly, we would end up with very platform-specific code. To avoid this, the The OpenGL Extension Wrangler Library (**GLEW**) [16] exists. This library determines the platform which the code is currently running on and manages the tedious task of binding the correct function names to their implementations on the actual hardware.

To provide the user of the viewer application a simple way to change the Scatter configuration, we used the **AntTweakBar** [17] library which creates a simple and clean UI directly in the OpenGL drawing window.

3.2.2 Loading geometry

Since the Corona Scatter needs at the very least one distribution object and one instanced object to be able to scatter, we need to be able to load 3D meshes into our viewer. We chose the **Wavefront .obj** format which is a very simple and user-friendly format of exporting geometry objects. Wavefront .obj is a widely supported format in modern applications like 3ds Max [1] or Blender [6]. It is a text/plain type format which makes it very easy to parse into our viewer. Wavefront .obj also supports line segment objects which is crucial for our needs.

3.2.3 Class structure

As we already mentioned, we will use the OpenGL viewer as a simple introduction to the Corona Scatter's API. The viewer itself is a rather simple application consisting of only 4 simple classes, as shown in Figure 3.1.

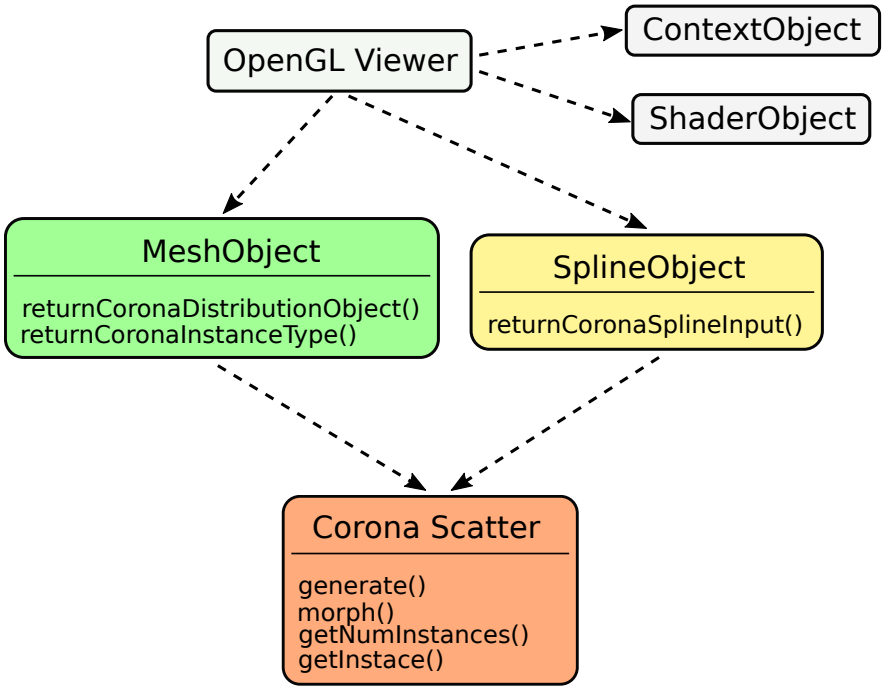

Figure 3.1: A class diagram of the OpenGL viewer application.

Context Object

This class is used for storing and accessing the current state of the program. It only contains static members as there should always be only one current state. Storing the data this way is required because we want to access the data both from the main draw loop and from different callback events we receive from GLFW and AntTweakBar. These events do not provide any references to the `main()` function's context, so we cannot store the data in the main function's stack, even though it would make the code a bit cleaner.

Shader Object

Even though our program's focus lies somewhere else, we still need to utilize at least very basic shaders. This class provided by the brilliant tutorial from learnopengl.com [18] provides us with a simple way on loading and applying shaders.

Mesh Object

This class serves two purposes – drawing a triangle mesh in OpenGL involves filling different buffers and similar objects (vertex buffer object, vertex array object, etc.). Doing this manually for every object when we need to draw it can

become very tedious. This class loads the buffers automatically and when we need to draw the mesh, only one binding is required (binding the mesh object's VAO).

The second and main purpose this class serves is parsing the mesh from the Wavefront .obj format and converting the mesh into data compatible with Corona Scatter when needed.

Spline Object

The functionality and purposes of this class are exactly the same as those for the Mesh Object class, only done for spline objects. This means that both parsing and the overall data stored is very different, however, conceptually these classes are almost identical.

3.2.4 Generating with Corona Scatter

Having explained all classes in the Viewer itself, let us now examine the steps required to make Corona Scatter generate some instances for us and how to retrieve them once the generation is complete. Examining the API of the Scatter Core class, we see three main public methods:

1. `void generate(...);`
2. `void morph(...);`
3. `const Instance& getInstance(const int i) const;`

`Generate()` is the method we should call when we are scattering instances for the first time, while we can use `morph()` once we have already generated some instances and wish to transform one of the distribution objects. Once we generated some instances, we can retrieve them one by one by calling the `getInstance()` method. Getting a closer look at the call of the method `generate()` method:

```
void generate(const Stack<InstanceType>& instanceTypesInput,
Stack<DistributionObjectInput>& distributionObjectsInput,
const Config& config,
const Stack<SplineObjectInput>& splineObjectsInput,
const Stack<SplineModifier>& modificationSplines,
const Stack<AreaModifier>& areaMesh);
```

We can see a handful of unfamiliar types which we now explain in more detail:

Stack<T>

Having already discussed the code base of Corona Render, suffice to say Stack<T> is one of their custom data structures. It is a dynamically allocated array with stack-like behavior very similar to std::vector and is generally used as such.

InstanceType

A class representing one instanced object. The key data points are the bounding box of the instanced object's mesh in local space, the transformation from

local space to world space and the probability of this instance type being scattered.

DistributionObjectInput

A class representing a distribution object. This object requires the full triangle soup of its mesh in the local space, the transformation from local space to world space and the density of this distribution object. The density represents the chance that an instance will be scattered on this distribution object while scattering randomly.

The user can also specify something called mesh morphing. Morphing is the way the Corona Scatter handles animation of the distribution object – if no morphing is supplied, the scattering is redistributed every single time frame. This, however, can lead to some instances appearing or disappearing during animation. If morphing is provided, the already scattered instances will be translated to new positions depending on the transformation of the triangle they were scattered on, but the sampling will not be redone. This eliminates the risk of instances "popping" – appearing and disappearing rapidly each animation frame.

The last optional parameters are maps affecting the density of the distribution on this distribution object and the scale of the scattered instances. These maps modify the values based on the color at each pixel, with white being 100% and black being 0%.

SplineObjectInput

Similarly to the other two input objects, this object represents the distribution splines. Each spline has its geometry in the form of `LineSegment` objects, which is a simple wrapper containing the endpoints of the spline. A matrix that realises the transformation from local space to world space, the optional spline morphing – which functions exactly same as for `DistributionObjectInput` class. The last parameter is the `referenceDirection`, which is a vector in $3D$ pointing in the direction which the instance at the beginning of the spline will be aligned to, as was discussed in Chapter 2.

Config

This is an object representing all Corona Scatter settings. The distribution can be customized via the substructure *scatter*, modification of each instance after scattering via the substructure *transforms* and controlling the collision avoidance via the substructure *collisions*.

Having correctly filled these data structures, we call the `generate()` method of the Scatter Core object of the Scatter. The scatter will generate the objects and store them in Scatter Core's `instances` member. The Scatter Core API then provides several methods to retrieve the generated instances mainly: `int numInstances()` and `const Instance& getInstance(int i)`. The user can also retrieve the instance types that were passed to the Scatter in a similar fashion.

3.3 Corona Scatter

We now describe the internal structure of Corona Scatter. Figure 3.2 provides a class diagram featuring the main classes used in the program. We then describe

the behavior, public functionality and important implementation details of each class.

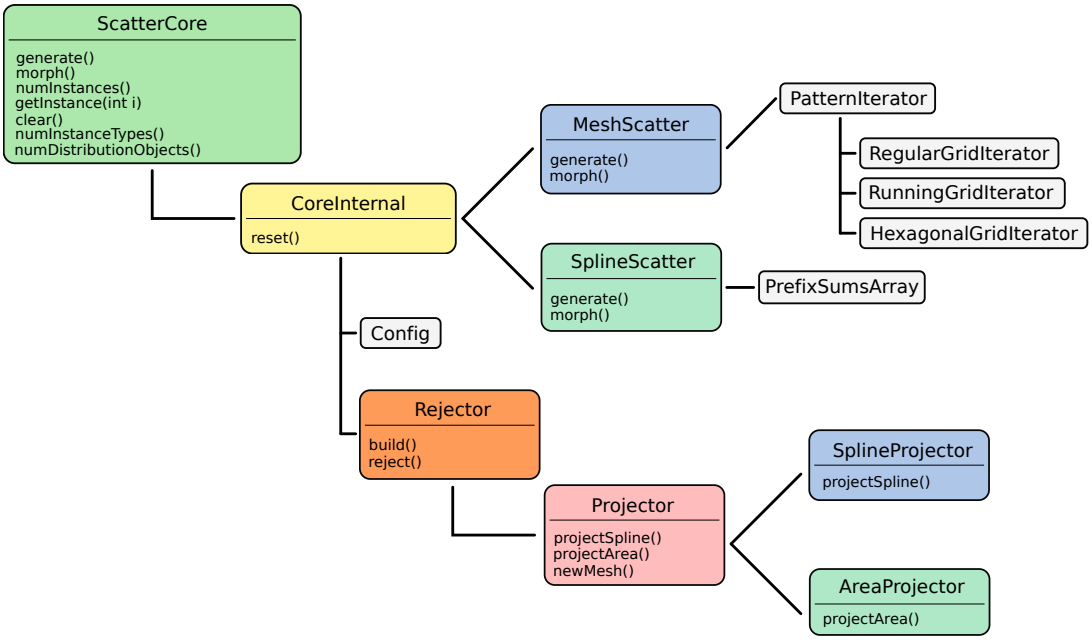

Figure 3.2: A class diagram of the Corona Scatter Core.

3.3.1 ScatterCore

This class is a part of Corona API released to the public. Because of this, the main purpose of this class is to provide the Corona Scatter functionality while hiding as much implementation and data members as possible.

However, as this class is the only interface to access the Scatter and its results, this class holds all instances that get generated and makes them available via the `getInstance(int i)` method.

The class holds an instance of a CoreInternal object, which contains the core of the implementation.

3.3.2 Config

The second and last class of Corona Scatter that is provided in the Corona API released to the public. This class is a data-holding type without any functionality. This class holds all Corona Scatter settings which are required for calculating the distribution. The parameters are divided into logical blocks such as *scattering*, *transformations*, *regularScatter*, *splineScatter*, etc.

3.3.3 CoreInternal

As we mentioned above, this class is the central hub of the Corona Scatter functionality. It holds the Config object as well as both SplineScatter and MeshScatter objects. This class then provides methods that calculate the complete scattering of instances, namely `generateRandom()`, `generateUvGrid()` and `generateonSplines()`. These methods get called by `Core::generate()`.

3.3.4 MeshScatter

This class contains the implementation surface scattering, both random distribution and generation of regular patterns using UV coordinates. Since we did not modify the random distribution feature in this thesis, we do not discuss it here.

The class holds the distribution object geometry, separated into two members: a Corona::Stack (we will call it a Stack from now on) of DistributionObject objects and a Stack of DistributionObjectTriangles which contains all triangles of all distribution objects. This is because having the data together in this fashion makes the memory accesses faster, since the program does not jump all over the memory space, which makes it more friendly towards the CPU caches.

DistributionObject is the internal representation of properties of the whole distribution objects such as density maps, scale maps, the bounding box of the distribution object and its transformation to world space. It also holds the index of its first triangle in the Stack of DistributionObjectTriangles and the number of triangles. This provides quick and easy access to the object's triangles when needed.

DistributionObjectTriangle holds all geometry data of each triangle – its position in the world space, all of its UVW mappings, the probability of scattering on this triangle, etc.

Mesh scatter implementation details

Let us now describe in more detail some important implementation details. As we already mentioned in Chapter 1 in the section about Corona Scatter, the existing user base of this program appreciates how fast the scattering is, so it is very important for us to generate the regular patterns fast. To accomplish this, we took care to implement the pattern scattering carefully.

The first measure that was very much required was making the algorithm run in a parallel environment, fully utilizing all of the computer's available logical processors. This is very important since Corona Renderer is a CPU based renderer, therefore it can be expected that its users have state-of-the-art CPUs with large numbers of logical processors available. We also need to take great care to make sure the scattering finishes exactly the same for every kind of hardware configuration. This is because Corona developers are constantly working on distributed rendering software which allows rendering the scene on many connected computers at the same time. We need to ensure that the resulting distribution generated by Corona Scatter is the same on each of these computers, since otherwise the resulting images would differ and combining the resulting renders would yield wrong results.

Our implementation utilizes the custom Corona scheduler which provides us with the ability to start the generation on as many threads as we would like. The parallel tasks are divided into jobs which are passed to each thread. The jobs are continuous strips of the grid pattern with constant size for each thread. We will discuss the choice of the size these jobs more in Chapter 4 of this thesis. Due to the consistency requirement we need to merge these continuous strips of the scattered pattern into the memory in the correct order, as we might have to cut the scattering short if we would generate more instances than the user-set *limit* variable. If that happens, we need to make sure the instances are in a defined order as this could cause undefined behavior due to racing of the threads.

These parallel tasks are calculated repeatedly until we either reach the maximum point of the bounding box we are scattering in, or we reach the *limit* setting. The Corona Scheduler ensures that starting the threads wastes as little time as possible.

Notice that while generating the grid the way we described, we often need to start generating from a different point than the start. While we could pre-generate the whole grid before placing the instances, we found out that while generating millions of instances, the data would get large and the memory allocations would slow the program down. We decided to implement an iterator for each pattern that would generate the grid as is needed during the algorithm run. The abstract class **PatternIterator** is the base class for all three of our iterators (one for each regular pattern). The iterator has a `getNext()` method that returns the next point in the grid and the iterator's constructor provides means to advance the iterator from the beginning of the grid, skipping an arbitrary number of points in the grid. This ensures that each thread can start generating at its starting point fast, without the need to "unwind" the iterator first.

3.3.5 SplineScatter

Similarly to MeshScatter, this class is responsible for generating the distribution on spline objects as we described in Chapter 2, in the section Scattering along splines. Like MeshScatter, the class keeps a Stack of DistributionSplineSegments which each contain the geometry information of the all splines' segments and DistributionSplineObject which contains the information specific to each spline.

When we were designing the parallelization of the distribution to speed the process up, we found out that a simple greedy algorithm to split the work for all cores is already as fast as we would like. The algorithm calculates how many instances should roughly be distributed by each thread in the ideal case: $countperthread = instances/numberofthreads$ and each thread then one by one greedily takes spline by spline looking to reach or exceed this number. This way of scheduling should perform very well if the user scatters a small amount of instances per spline but scatters on a lot of splines, while performing a bit worse if the user scatters a large amount of instances on a single spline. However, we felt that the first case is much more likely than the second one – for example, when creating a shipyard scene, the user will need to create multiple short link chains more often than one very long link chain.

An important class utilized to map the points from the interval $[0, 1]$ as we described in Chapter 2 is the **PrefixSumsArray**. This simple class receives an array of numbers, computes their prefix sums and then allows queries to answer which interval the queried number lies in. This is very handy when trying to map a point to a position on a spline – we need to find the correct segment to place the instance on, which is this exact type of query since we know the position on the spline in percent, the length of the spline and the length of each of the spline's segments.

3.3.6 Rejector

This class provides the functionality needed to reject instances based on their distance to a projected spline. Rejection is done via *rejection sampling*, which means that after the instances are scattered, using SplineScatter or MeshScatter, we iterate through the instances again, one by one calculating the distances to the projected objects and rejecting the instance if it is too close to an open spline or inside a closed spline, modifying its scale and density if it is not.

The rejection sampling is simple to parallelize, since the rejection of each instance is dependant on the projected splines only. Here we utilize the maximum number of threads, dividing all the instances between the threads equally as there are no problems with neither one instance being dependant on another nor multiple-PC-consistency. This is because the distance of each instance from each spline does not change with the number of logical processors available, giving us the same result every time.

Apart from rejection, the rejector has to also be able to project the given distribution mesh and modification splines. This functionality is separated into the class Projector, which handles the projections.

This class also contains two functors **PointSegmentDistance** and **PointTriangleDistance**. These are important as they get utilized in the rejection. As we discussed in Chapter 2, we needed to find the closest record from a BVH tree (either a line segment or a triangle) to a given point. This requires us to define a functor calculating the squared distance between a point and the record. **PointSegmentDistance**, as the name suggests, calculates the squared distance between a point and a line segment, while **PointTriangleDistance** calculates the squared distance between a point and a triangle. The distance between a point and a triangle was originally computed as the minimum of the distances between the point and its three sides (interpreted as line segments) and so each call of **PointTriangleDistance** basically called **PointSegmentDistance** three times. However, we found out, thanks to the profiling we did with Intel VTune software [13], that this approach is actually slowing the program down. We noticed that since the three distance calculations do not depend on each other, we could compute them simultaneously. Of course, doing this on different threads would probably not provide any speedup, but the Corona code base provides a custom wrapper implementation over *SSE instructions*. These are SIMD (single instruction, multiple data) type instructions which allow us to perform the same operations on up-to four floats at the same time. Since we only really require three at the same time (one for each side of the triangle), this suited our needs perfectly. Changing the computation to the simultaneous version using SSE instructions gave us roughly 35% faster performance.

Projector

This class provides the functionality to project splines on a given triangle mesh geometry. For this class to function properly, the geometry to project on needs to be given first, using the `newMesh()` public method. When the geometry to project on is given, the user is required to specify along which axes this geometry should be projected in, multiple axes can be chosen. This class then provides two public methods `projectArea(object, axis)` and `projectSpline(object, axis)` which project the given object (area or

spline) along the specified axis. Note that the axis has to be one of the axes specified in the `newMesh()` call.

Since projecting the geometry in `newMesh()` can take a considerable amount of time, we wanted to have to do this projection only once. On the other hand, the algorithms for projecting splines and areas have different implementations, therefore should be separated. This is why we created two more classes, that realize the projection of areas and splines themselves – SplineProjector and AreaProjector. These classes are used to calculate the projection itself in `projectArea()` and `projectSpline()` methods of the Projector class.

This class also utilizes two custom iterators of the Corona::Bvh class – **SplineIterator** and **TriangleIterator**. These iterators answer the special queries we discussed in Chapter 2. **SplineIterator** returns all triangles which have bounding boxes intersecting the given line segment. **TriangleIterator** returns all triangles which have bounding boxes intersecting the bounding box of the given triangle.

SplineProjector and AreaProjector

Since the classes differ only in the implementation but their usage and API is very similar, we will discuss both at the same time. Upon construction, these classes require a ProjectionData object, which holds all information of the objects these projectors should project on. This data is created in the method `Projector::newMesh()` and then stored in the Projector class. This means that if the users of this code require projecting splines or areas onto another triangle meshes, they should always use the whole Projector class, instead of just SplineProjector or AreaProjector.

After being constructed, both classes provide a public method to project a spline/area onto the geometry objects stored in the ProjectionData object.

3.4 3ds Max integration of Corona Scatter

After implementing all the new features into the Scatter Core, we, of course, needed to carry these changes over to the 3ds Max plugin. We added a whole new way of scattering (along spline objects), new way to scatter of surfaces (in regular patterns) and a whole new way how to interact with the already existing distribution (distribution modifications using splines), this meant that we need to change the UI significantly.

3.4.1 Keeping the UI clean

We already mentioned that to make the User Interface (UI) clean and easy to use, we need to hide or at least disable all controls that at the moment will not affect the distribution if changed (for example, setting the regular pattern spacing when generating randomly will not have any effect). 3ds Max is a windows application and the developers can interact with its UI using a mixture of 3ds Max API and Windows API. All UI design is handled by using Microsoft's resource files – **.rc** and then binding the controls to the appropriate variables. We will not go into the details here as the system is complicated and 3ds Max API [19] can explain it a lot better. The important information for us is that we are able to easily control each individual control button (e.g. a text box, a spinner box, etc.) via

both 3ds Max API calls and Windows API calls and so disabling each control when it was not needed was not difficult. This behavior can be seen in Figure 3.3, as the middle rollout named "Surface scattering" has all options in the "UV map" section disabled, because random scattering is enabled and chaning these controls would have no effect.

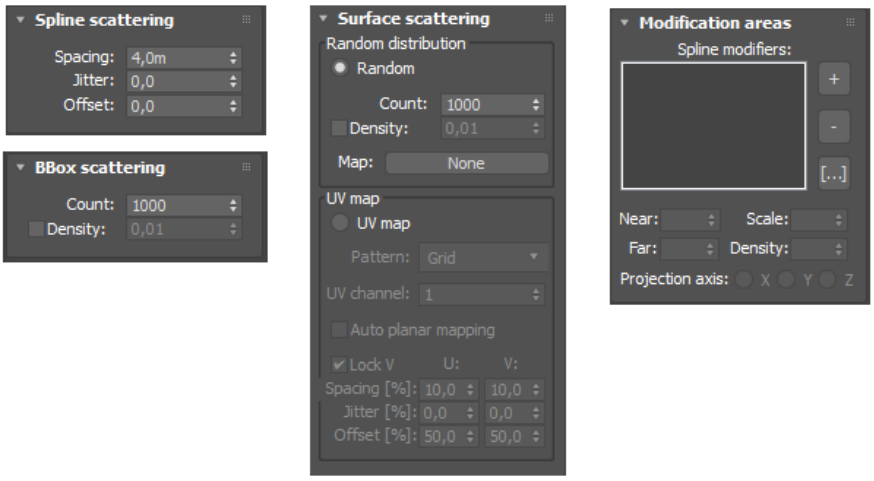

Figure 3.3: This images showcases the four new rollouts we added to Corona Scatter. As we mentioned, only one of "Spline scattering", "Surface scattering" and "BBox scattering" is visible at a time. "Modification areas" enable the user to use our distribution modification feature.

The second thing we wanted to make sure was working properly was to hide as many controls as possible. This is most important for the settings that should not be visible at the same time because only one of them will be useful which are the settings for the three main scattering types: $1D$ scattering along splines, $2D$ scattering on surfaces and $3D$ scattering in the bounding boxes. We decided to make a separate rollout for each of these types, consisting of all relevant settings. This required to modify the original "Scattering" rollout as seen in Figure 3.4. We then implemented a system that would hide the two of these rollouts that were not actively used at this time. These rollouts can be see in Figure 3.3.

3.4.2 Loading spline objects from 3ds Max

The second major thing we needed to implement was loading spline objects from 3ds Max into the Corona Scatter Core. The Scatter did not use spline objects for anything in the versions 1.5 and prior and when it got passed a spline it only checked if the spline could be converted to a triangle mesh (this can be done for closed splines) and if not, would discard the object. However, thanks to 3ds Max API's robust type system, retrieving the spline geometry was not problematic.

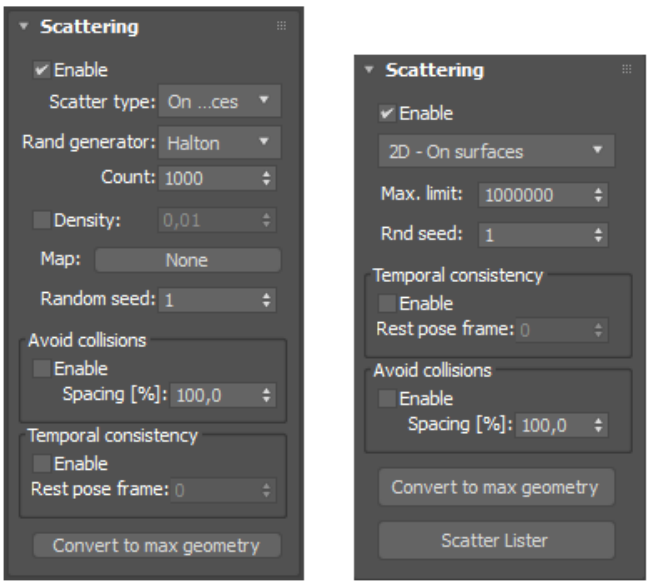

Figure 3.4: This image illustrates one rollout of the old Corona Scatter 1.5 on the left. The rollout on the right is the new UI we designed.

3.5 Autotesting

3.5.1 Unit tests

Unit tests are very important for each piece of software and Corona Scatter is no exception. We include unit testing for many of Corona Scatter's classes. For unit testing, we chose the Google Test [20] library made by the company Google. It is an open source unit testing framework written in C++, which perfectly suits our needs.

3.5.2 Render regression tests

Another type of tests utilized by the Corona team are tests that require designing a whole scene. These scenes are then rendered by the Corona Renderer and compared pixel by pixel with a saved reference image. These tests are slower than unit tests but can reveal more intricate bugs that are hard to catch in unit testing.

Having discussed the implementation of all the features we set out to develop in this thesis, we now move on to the next chapter where we showcase some results of our work – scenes that are now possible to create using Corona Scatter. We also test the performance of the implementation to confirm that the program is reasonably fast.

4. Results

In the first section of this chapter we showcase scenes that were made using Corona Scatter's new features we implemented in this thesis. In the second section of this chapter we examine the program's performance.

4.1 Implemented features

In this section, we showcase a few images that were modelled with the new features we implemented in this thesis.

Figure 4.1: A demonstration of regular pattern scattering and scattering along splines. Image taken from Corona 1.6 release post [21].

Figure 4.1 utilizes three different Corona Scatter objects. The buildings in the image were generated using the regular pattern distribution feature, placing them into a simple and fully regular grid. The debris on the ground was also generated as a regular pattern, however, the amount of randomness was set high. This ensures that the debris will be spread very uniformly and strange situations are less likely to arise. An example of such situation would be generating a demolished building in the image which was not surrounded by any debris (this could happen if we generated using the random distribution). The third element of the image is the foliage. The foliage is distributed along splines that copy the edges of the roads in the image. Each piece of foliage is then slightly moved around. This makes the foliage appear around the roads very regularly but gives it the right amount of randomness to prevent it from looking too artificial.

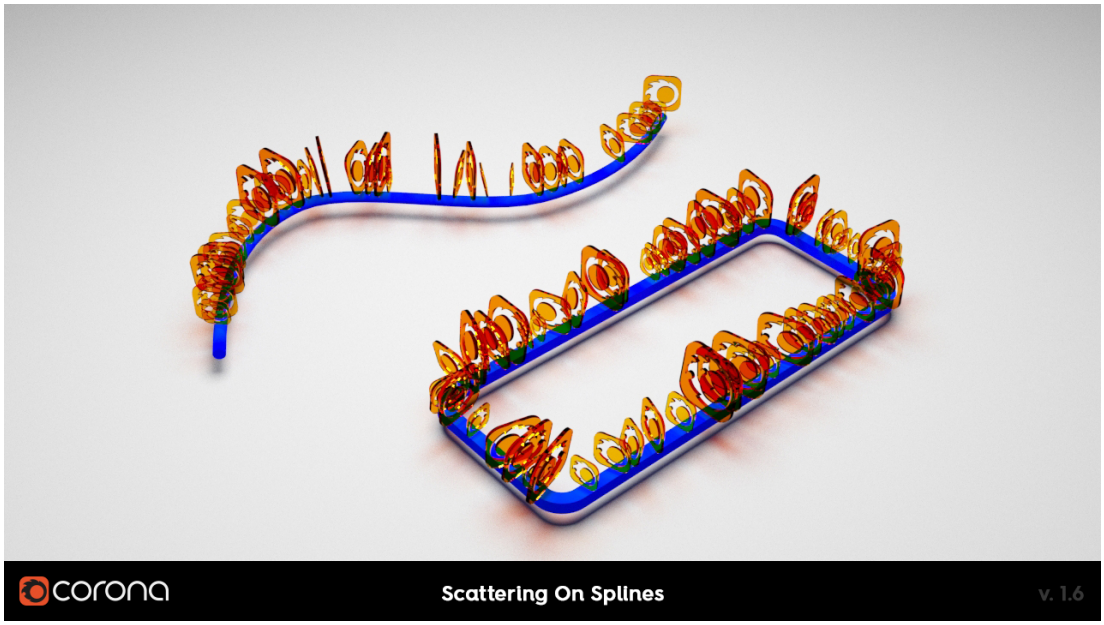

Figure 4.2: Image illustrating our scattering along splines feature. Image taken from Corona 1.6 release blog post [21].

Figure 4.2 is a simple demonstration of scattering along curved splines.

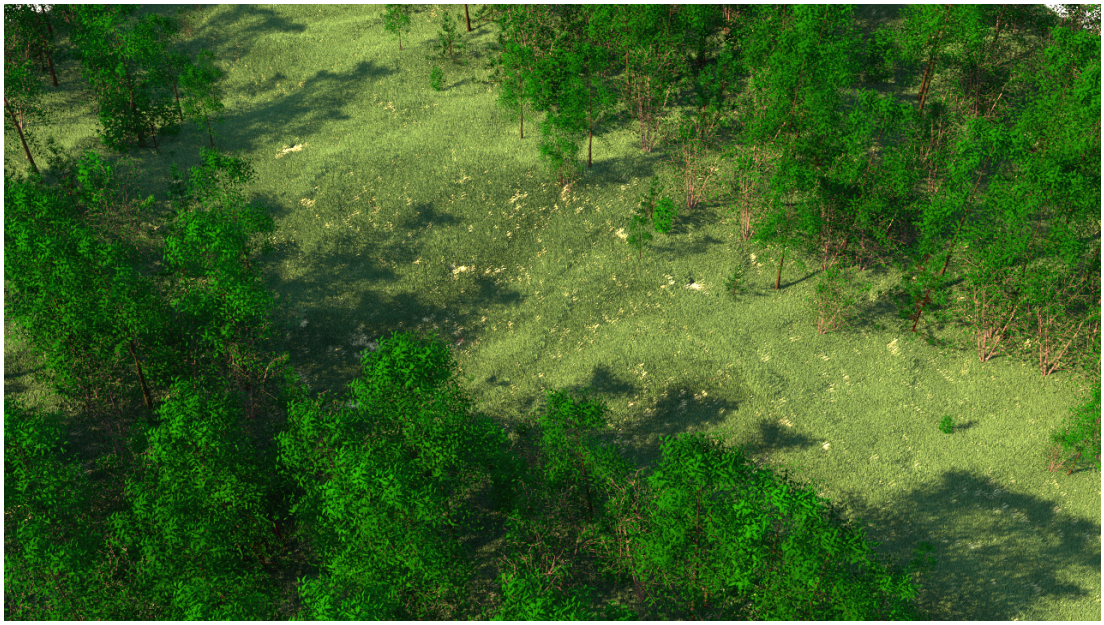

Figure 4.3: Image illustrating modification of the distribution near a spline to create a path through the forest.

We both modelled and rendered Figure 4.3 to demonstrate the distribution modification feature we implemented. This scene utilizes two Corona Scatters. The first simply scatters two types of grass (fresh green patches and old yellow patches) on the whole terrain. The second Corona Scatter object scatters four different versions of tree models over the grass. This tree distribution then gets

modified by one open spline, carving out a path in the forest. As you can see in the bottom right corner, the near region (as we discussed in Chapter 2) is very thin which allowed the two small trees in the bottom right corner to be scattered. The trees along the border of the forest are slightly smaller which should make the forest look less artificial.

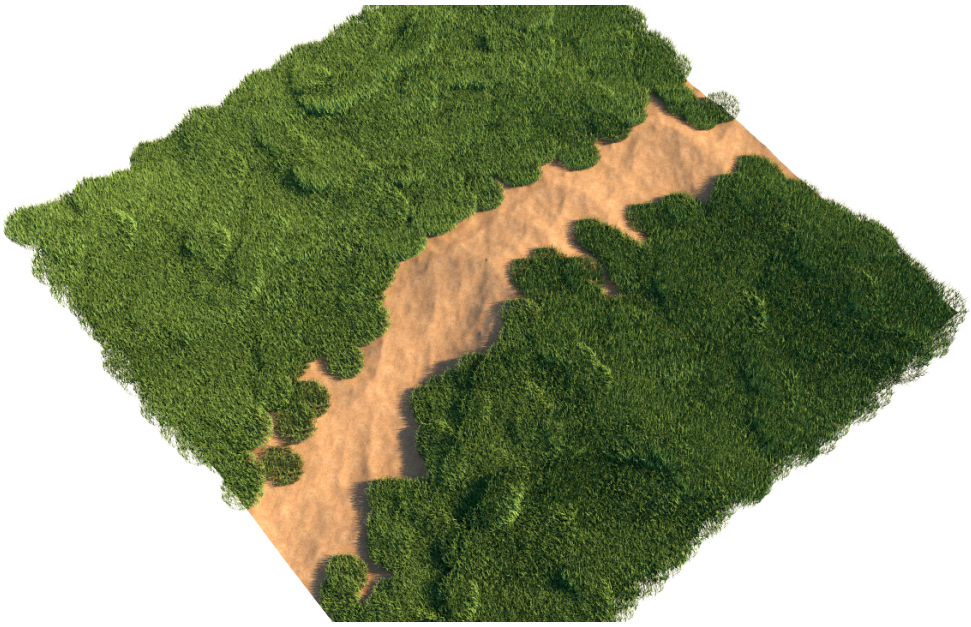

Figure 4.4: Using the spline modification feature to create a small frequented footpath on a grassy terrain.

Figure 4.4 is another example we created for the purposes of this thesis. The image is a close-up shot of modifying grass distribution to create a simple, frequented path on a grassy terrain. It should illustrate another possibility of using the distribution modification feature which, when combined with the image above skillfully, should create a very realistically looking terrain. Unlike in Figure 4.3, we only allowed a small amount of scaling and most of the path was created by rejecting the grass patches along the path.

Please note that both Figure 4.3 and Figure 4.4 are images that were created for the sole purpose of demonstrating the features of this thesis and are not meant to be examples of the best work that can be achieved using these features.

4.2 Performance measurements

In this section we present a brief summary of the program's performance. First, we explain the methods used to measure the time.

All these measurements were taken on a desktop PC with the following hardware components:

- Microsoft Windows 10 Pro 64-bit
- Intel Core i7-4790 (4 physical cores, 8 hyper-threads @ 3.60Ghz)

- 32GB RAM, 4×8GB at 1866 MHz

The times were measured using a macro present in the Corona namespace called `MEASURE_TIME_ELAPSED`. This macro is very simple to use – the only thing required is to call it at the beginning of the function we want to measure and when the function returns, the time elapsed is printed out in the debug console.

The code was compiled in Microsoft Visual Studio 2015 Community using the following optimalization options:

- Full optimalization (/Ox)
- Inline function expansion: Any suitable (/Ob2)
- Enable Intristic Functions (/Oi)
- Omit frame pointers (/Oy)
- Enable string pooling (/GF)
- Floating point model: Fast (/fp:fast)

While measuring, the Microsoft Visual Studio program was connected to a running instance of Autodesk's 3ds Max. The plugin was loaded into the 3ds Max program and the actions then executed using the plugin UI controls. We constructed a scene for each test case, which was loaded into 3ds Max and measured several times. The results were then averaged and are presented in the following tables.

4.2.1 Random scattering

Even though we did not develop this part of Corona Scatter in this thesis, we measured the random distribution speed, since it provides good reference times which we can compare to our performance. Since the existing users of Corona Scatter liked the speed of the random distribution, we should not exceed the time required to scatter randomly by much.

The scene we measured on consisted of one Plane object which we scattered on (size: 150×150 units, 4 width and length segments, resulting in 32 triangles), one Box object which was the scattered object (sized $2.79 \times 3.58 \times 6.8$ units) and one Corona Scatter (limit was set to $25M$, and random count was set to the value in the instances column of Table 4.1, density setting was not used, random seed was set to 1). The scattering times for the random distribution are presented in Table 4.1.

Table 4.1: Scattering on the surface randomly

Instances	Random scattering
250 000	35.36 ms
1 000 000	143.29 ms
2 042 041	304.75 ms
5 166 529	803.76 ms

4.2.2 Regular pattern scattering

The first feature we implemented in this thesis was scattering in regular patterns using the UV mapping of distribution objects. This feature was expected to be slower than random scattering due to multiple facts. Firstly, the inversion of the UV mapping is costly as it requires us to build and traverse a BVH tree. Secondly, we need to have the generated instances sorted in a defined order – the order they appear in the regular pattern as we described in Chapter 3. This requires us to constantly move the created instances in the computer's memory, which slows the program down.

The scene used while measuring the first column of Table 4.2 – "Regular (32 triangles)" consisted of one Plane object which we scattered on (size: 150×150 units, 4 width and length segments, resulting in 32 triangles), one Box object which was the scattered object (sized $2.79 \times 3.58 \times 6.8$ units) and one Corona Scatter (UV map setting was turned on with the parameters: grid pattern, UV channel set to 1, offset was set to 50, jitter was set to 0, limit was set to $25M$, the spacing was set to 0.2 for $250K$ instances, 0.1 for $1M$, 0.07 for $2M$ and 0.044 for $5M$, random seed was set to 1).

The scene used while measuring the first column of Table 4.2 – "Regular (45000 triangles)" was composed of the same Box object which was scattered as the scene above, however, to avoid precision errors that would arise if we subdivided the small 150×150 plane too much, we increased the size of the plane to 950×1308. We also increased the scaling of the UV mapping using the "UVW Map" modifier in 3ds Max, where we set the width to 3776 and the length to 5200. This meant that we were able to then set the subdivision to 150 segments in both width and height, resulting in 45 000 triangles. The settings of Corona Scatter were: grid pattern, UV channel set to 1, offset was set to 50, jitter was set to 0, limit was set to $25M$ and the spacings were set to 0.063 for $250K$ instances, 0.031 for $1M$ instances, 0.022 for $2M$ instances and 0.014 for $5M$ instances.

The results are presented in Table 4.2.

Table 4.2: Scattering on the surface in a regular pattern using UV coordinates

Instances	Regular (32 *triangles*)	Regular (45000 triangles)
250 000	77.00 ms	99.43 ms
1 000 000	276.70 ms	360.35 ms
2 042 041	588.14 ms	624.47 ms
5 166 529	1 400.39 ms	1 451.95 ms

As we can see in the Figure 4.5 which compares the speed of random distribution and regular distribution, the program is slower than the random distribution. However, the slowdown is not very significant, since the users do not require to generate more than 2000000 instances that often. The goal of the Corona team is to have the program finish within very few seconds even for the extreme cases such as 5000000 instances, which we definitely fulfilled.

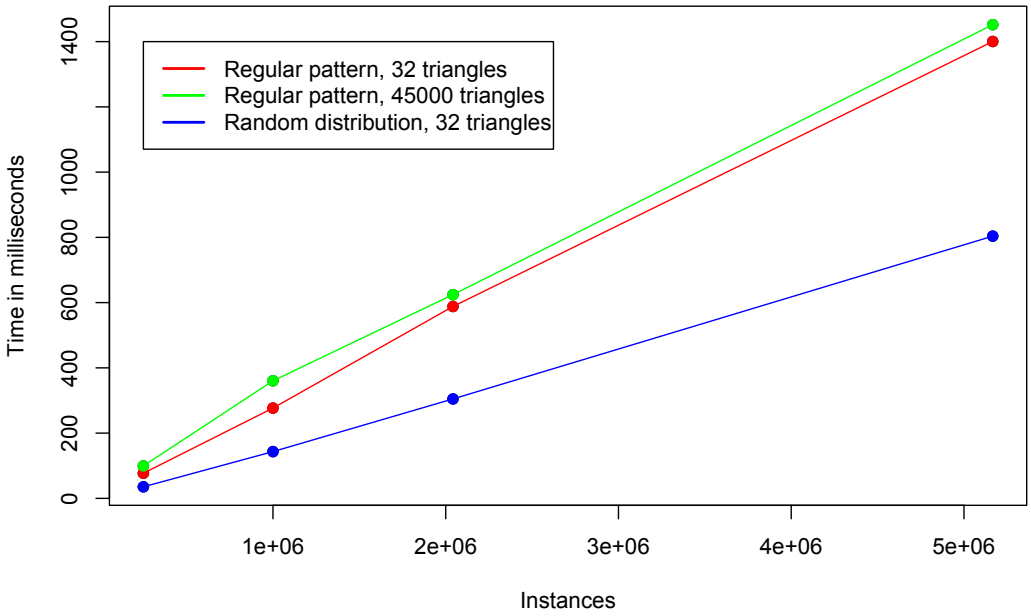

Figure 4.5: A graph showing the dependency of time taken to modify instances on the number of instances. The red and green lines represent regular pattern scattering (Table 4.2) while the blue line represents random scattering (Table 4.1).

Worker job size parameter

One parameter of the regular pattern distribution is the size of the jobs given to each parallel worker which we mentioned in Chapter 3. It indicates how many points in the UV space each worker will process before the worker stops, we linearize the generated instances, check for exceeding the maximum instances limit and run the workers again. This parameter is very important, since setting it too low will make us waste a lot of time on waking the threads up over and over, while setting it too high will make us generate a lot more instances than the user requires, potentially taking a very long time or freezing the user's computer. In Figure 4.6 we show the dependency of the time it takes to generate 2042041 instances on the size of the worker jobs.

The scene we measured on consisted of one Plane object which we scattered on (size: 150×150 units, 4 width and length segments, resulting in 32 triangles), one Box object which was the scattered object (sized $2.79 \times 3.58 \times 6.8$ units) and one Corona Scatter. The Corona Scatter's settings were: UV map turned on, grid pattern, spacing set to 0.07, offset set to 50, jitter to 0, limit set to $25M$ and random seed set to 1. During the experiment, we added one UI element controlling the size of the worker jobs, which we removed after the measurement was done as the job size is something we do not want the user to be able to modify. This unfortunately means that the experiment cannot be replicated without modifying and recompiling the code. Should the reader want to replicate this measurement it can be done by changing the value of the `batchSize` variable in the `scatterUvw()` method.

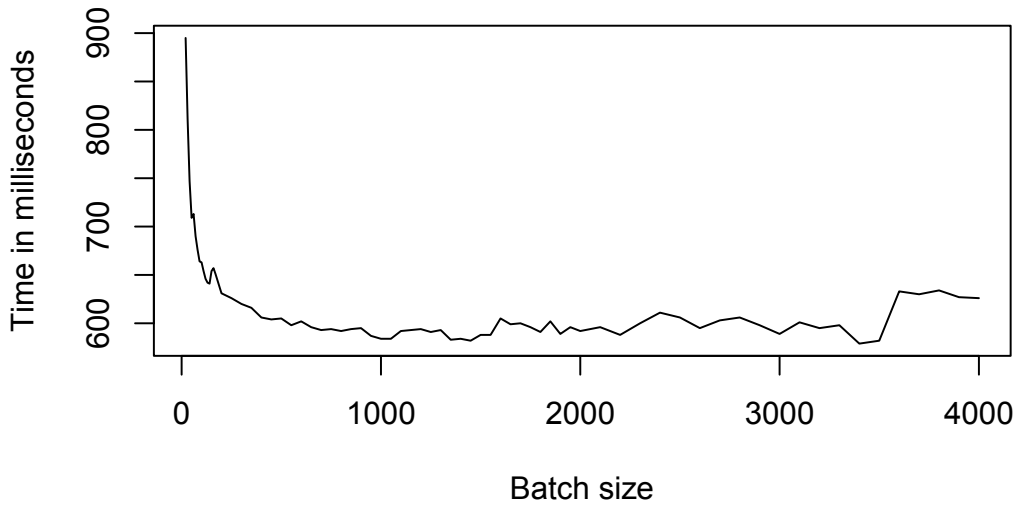

Figure 4.6: Time needed to generate the same amount of instances while changing the size of the jobs.

As we can see in Figure 4.6, the performance increases up to roughly 1000 worker job size, after which it starts to behave randomly with a small amplitude. We chose the number 1000 as it seemed to perform well and it well represented the balance between waking the threads too often or generating for too long.

Dependency on the number of threads

Since Corona Renderer is a CPU based renderer, its user base tends to use very high-end CPUs in their computers. This is why the scalability of our algorithms with the number of the CPU cores is an interesting information. The other very important reason why we want to make our code as scalable with the number of CPU cores as possible is the fact that since recent years, increasing the speed of CPUs is mainly being done by increasing the number of CPU cores, since we do not have the technology to increase the speed of the individual cores much further.

We present a measurement in two cases – when scattering on a simple plane and when scattering a on a plane that is heavily subdivided, which is often required when creating realistically looking terrains. We limited the number of threads Corona Scatter ran on by modifying the Scheduler object we talked about in Chapter 3. Therefore this measurement cannot be replicated without modifying and recompiling the Corona Scatter code. The distribution object for "simple geometry" was a simple Plane object with size 150×150 units, 4 width and length segments, resulting in 32 faces. The distribution object for "complicated geometry" was a Plane object with size 950×1308 units with the UV mapping scaled up using "UVW Map" modifier of 3ds Max with the width set to 3776 and the length set to 5200. This plane was then subdivided so it had both 150 width and length segments. This resulted in a plane with 45 000 triangles. The scattered object was in both measurements a Box object with sizes $2.79 \times 3.58 \times 6.8$ units. Corona Scatter was set to UV map distribution, spacing was set to 0.07 for the simple geometry and to 0.022 for the complicated geometry, jitter set to 0, offset set to 50, limit set to $25M$ and random seed set to 1. In both measurements,

2 042 041 instances were generated. The results of this experiment can be seen in Table 4.3 and are illustrated in Figure 4.7.

Table 4.3: Using different number of hyper-threads for scattering a regular pattern with 2042041 instances.

Threads	Simple geometry time	Complicated geometry time
1	1 332.33 ms	1 873.29 ms
2	851.82 ms	1 132.50 ms
3	702.53 ms	907.45 ms
4	623.12 ms	783.80 ms
5	727.45 ms	920.73 ms
6	608.53 ms	778.17 ms
7	610.03 ms	759.00 ms
8	528.69 ms	660.62 ms

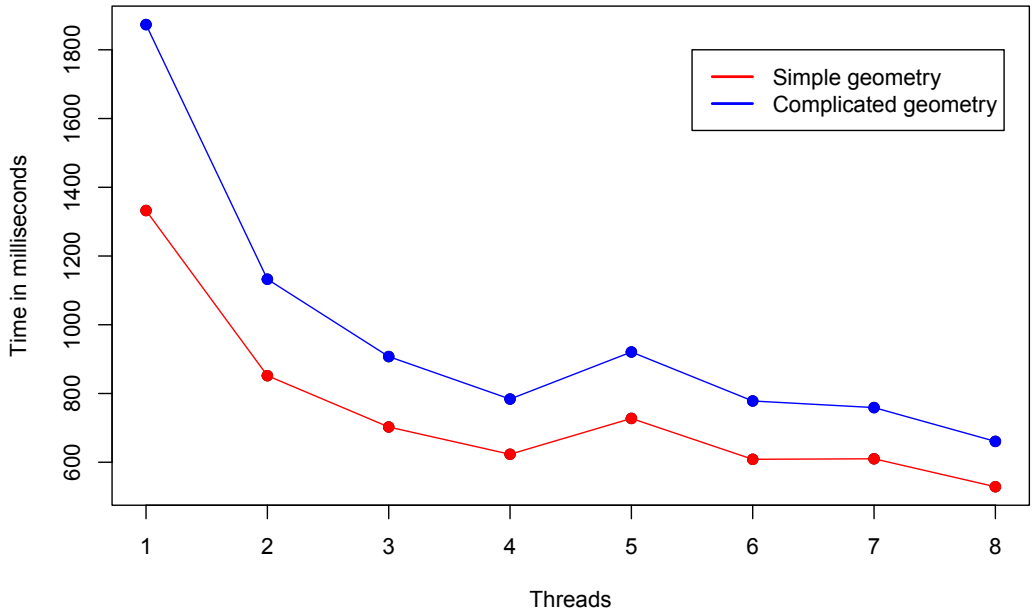

Figure 4.7: A graph showing the dependency of time taken to modify instances on the number of instances. The graph visualizes Table 4.3.

As we can see, the speedup is not as significant as one would have hoped – in the ideal situation, four CPU cores would do the work four times faster (possibly up to five times faster, due to the hyper-threading mechanism). This is often not the case though and as we already mentioned, the fact that we need to sort the instances into a defined order taxes the memory and slows the performance down. However, as we can see, we still managed to speed the program up by rougly 300% using four physical cores with 8 hyper-threads. Regarding the strange performance peak at five threads, our assumption is that it was caused by the CPU's

hyperthreading mechanism which did not function optimally when only one core was hyperthreaded and the others were not. However, since the production code always runs on maximum threads available (in this case it would be eight), these inconsistencies should not happen to any user using Corona Scatter.

4.2.3 Scattering along splines

One advantage of scattering along splines directly that we mentioned, was the fact that scattering along splines takes less computation time. There were three objects in the scene during the measurement. The scattered object was in both measurements a Box object with sizes $2.79 \times 3.58 \times 6.8$ units. The Corona Scatter object was set to generate on splines, the jitter and offset parameters were set to 0. The spacing parameter was set to 0.2 for $250K$ instances, 0.05 for $1M$ instances, 0.025 for $2M$ instances and 0.01 for $5M$ instances. The spline we scattered along was a Circle spline with parameters: radius 12642.74 units, interpolation was set to 6 steps, with the "Optimize" checkbox turned on. This spline had 28 segments. We then measured the same scene once again, this time with the interpolation of the Circle spline set to 50, which produced a spline with 204 segments. The results in Table 4.4 confirm our suspicion since we see an improvement over the random distribution.

Table 4.4: Scattering along one spline object

Instances	Along spline (28 segments)	Along spline (204 segments)
251 915	28.44 ms	52.08 ms
1 007 660	116.96 ms	164.03 ms
2 019 482	206.68 ms	312.63 ms
5 038 298	554.33 ms	756.31 ms

Table 4.4 shows the timings of scattering a very large amount of instances on one spline. We think this case is rather unlikely to happen often in a real-world usage of Corona Scatter but chose to include it as it shows the raw performance well. This is why we also included a second measurement run. Table 4.5 shows the performance when scattering on four spline objects to achieve the same amount of instances placed. There were four splines in the scene, each had the radius of 3443.77, the interpolation set to 6 steps with Optimize turned on. The parameter of Corona Scatter were identical to the case above, as was the distributed Box object. This use case is closer to the method of parallelization we selected in Chapter 3 which results in the scattering being faster – this is because we thought that the users will be scattering small number of instances on a lot of spline objects much more often than a large amount of instances on one spline object and designed the algorithm this way. Note that we left out the case with 250000 instances because the times were too small to measure reliably.

Table 4.5: Scattering along four splines

Instances	Scattering along four spline objects
250 000	—
1 034 792	72.20 ms
2 069 583	143.22 ms
5 173 955	352.49 ms

The relation between Table 4.4 and Table 4.5 is illustrated in Figure 4.8. We see that increasing the smoothness of the spline from 28 segments to 204 only increased the time required by 50%. Additionally, setting the interpolation parameter to anything above 20 is not used often, as there are almost no visual differences beyond this point. This measurement shows that our spline algorithm scales very well with the increasing smoothness of the spline.

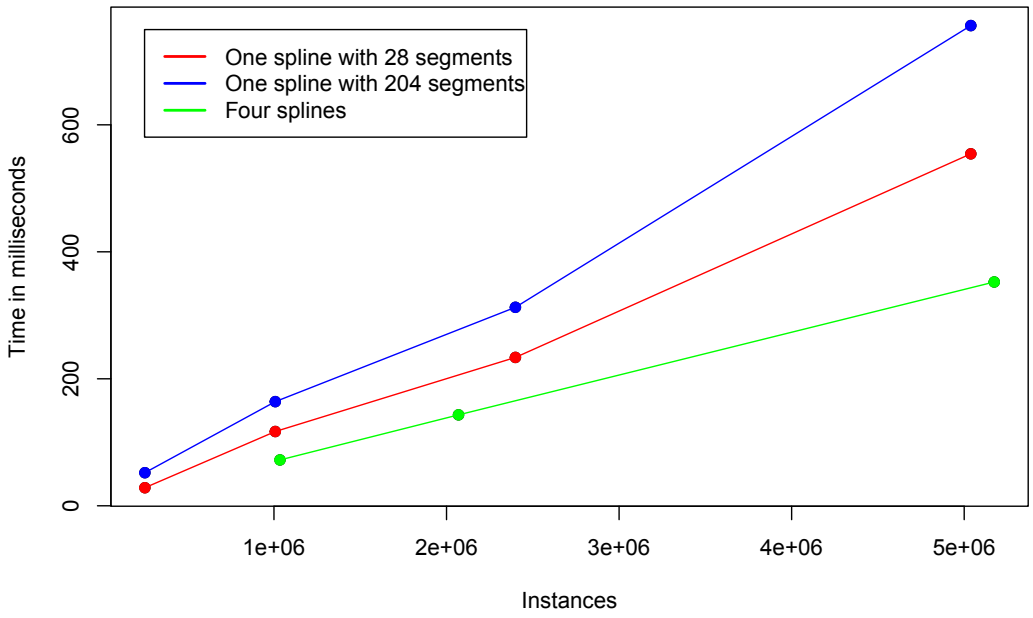

Figure 4.8: A graph showing the dependency of time taken to scatter along splines on the number of instances. The red and blue lines represent the Table 4.4 while the green line represents Table 4.5.

4.2.4 Distribution modification using splines

Now we present the performance of the instance rejection/modification based on the input spline objects. Because the modification takes place after generating the instances, the "Scattering" column is included in the performance table and signifies the time it took to generate the number of instances using the **random** distribution method. Since this process unfortunately requires modifying a large amount of instances – therefore modifying a large amount of memory, we split the measurement into two parts. "Rejection sampling" is the time it takes to iterate through all instances and calculate the modifiers for each instance based on its

distance to the modification objects. "Total rejection" is then the total amount of time spent doing the modification (including the "Rejection sampling" column) – the remaining time is spent on building the BVH tree over the distribution objects and accessing and modifying the instances in the computer's memory. Table 4.6 summarizes the performance of modification using open splines, while Table 4.7 does the same for closed splines. Figure 4.9 compares the values from the "Total rejection" column of both tables.

Table 4.6: Modifying instances using open splines

Instances	Scattering	Rejection sampling	Total rejection	Total
250 000	35.85 ms	6.06 ms	18.39 ms	55.09 ms
1 000 000	167.31 ms	32.00 ms	80.07 ms	248.26 ms
2 000 000	304.13 ms	54.18 ms	144.42 ms	449.95 ms
5 000 000	836.89 ms	155.22 ms	393.00 ms	1 230.89 ms

Table 4.7: Modifying instances using closed splines

Instances	Scattering	Rejection sampling	Total rejection	Total
250 000	36.46 ms	13.08 ms	25.19 ms	62.38 ms
1 000 000	149.19 ms	51.73 ms	96.08 ms	246.19 ms
2 000 000	304.48 ms	103.67 ms	191.14 ms	496.48 ms
5 000 000	779.15 ms	260.79 ms	478.47 ms	1 258.91 ms

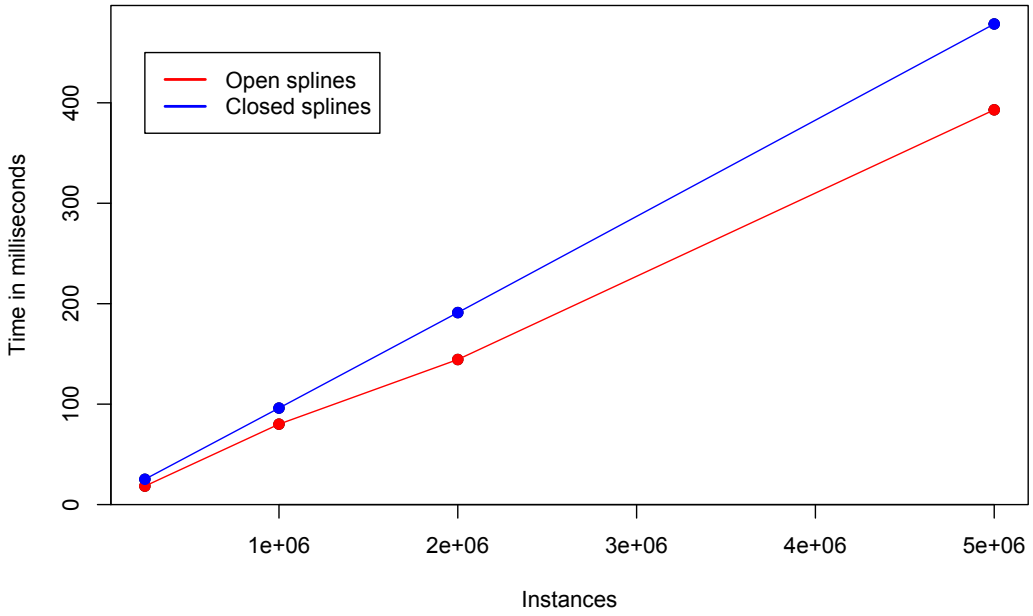

Figure 4.9: A graph showing the dependency of time taken to modify instances on the number of instances. The blue line represents Table 4.7 while the red line represents Table 4.6.

As the reader may have noticed, the times in "Scattering" columns differ from what we measured in the Random scattering section by a few percent. This is because since the processes are parallel, their runtime is affected by how the operating system decides to schedule them. For example, even though the computer has eight hyper-threads, when we launch our parallel computation, the operating system could only start seven threads (or any other number), because some other processes, unrelated to Corona Scatter are running on the eight thread. This would mean the eight thread will get started a little later, which can show in the measurements as these 5% inconsistencies.

4.2.5 Comparison with Forest Pack Pro

We now compare the performance times we achieved with the program Forest Pack Pro [2] which we discussed in Chapter 1. These times and instance numbers were taken from Forest Pack's "Statistics" window. Since Forest Pack does not feature the ability to precisely specify how many instances we want to generate, the numbers of instances generated in Forest Pack do not perfectly correspond to the numbers we measured in the sections above. This should not matter much though, because the comparison should only serve as a rough demonstration. The times for Forest Pack Pro are shown in Table 4.8.

Table 4.8: Forest Pack Pro performance

Instances	Time
1 015 676	1 677 ms
2 095 518	3 567 ms
2 926 644	5 238 ms
4 453 883	7 876 ms
4 730 760	8 494 ms

As we can see, the measured times indicate that Forest Pack Pro is significantly slower than Corona Scatter. However, Forest Pack only generates the distribution when the user wishes to render the scene, while Corona Scatter recalculates the distribution in real time. Both of these options have their advantages and disadvantages so each user has to decide which of these options suits him better.

Conclusion

In this thesis, we set out to improve the already existing Corona Scatter plugin for Autodesk's 3ds Max [1]. We needed to ensure that the program is easy to use with a User Interface (UI) that is as clean, concise and responsive as possible. We also needed to implement the solutions effectively because the program would be used to generate millions of instances.

4.3 Achieved goals

We implemented three features that were most requested by the already existing user base and which we thought would help make Corona Scatter a tool suited to more $3D$ modelling cases apart from randomly scattering vegetation, while also expanding on the already existing capabilities.

We ensured that both the UI and the behavior of the program is of great quality by communicating with a Corona expert user Ludvík Koutný. Later in the development cycle we also communicated with the users after releasing the Corona Scatter in Corona Renderer daily builds [8] and received more feedback about the behavior of the program along with a few bug reports we were not able to find during the quality assurance phase.

We implemented scattering in regular patterns using the UV mapping, which allows the user to scatter instances in a regular way. We also implemented scattering along spline objects allowing the user to scatter instances along the spline object with a custom spacing in between each instance and allowing to set a custom amount of randomness. These two features help to expand the scope of $3D$ modelling cases Corona Scatter can be used for – users are now able to create housing estates or city centres by placing buildings regularly with custom spacing between each row and column. Users are now also able to line city streets with lamp posts or create alleys by scattering lamp posts and trees along splines.

Expanding the already very advanced capabilities of Corona Scatter for scattering randomly, which is most commonly used when scattering vegetation, we added a feature that modifies the instances placed in the vicinity of spline objects and omits instances inside closed spline objects. This feature gives the user more control over the generated distribution and allows the user to modify the vegetation near footpaths, roads or simply to create a glade in the middle of a forest.

As we have shown in Chapter 4, all these features are implemented efficiently giving the user real-time responses to every input change and are capable of scattering large amounts of instances very quickly. While developing, we designed regression tests that ensure that further changes in Corona Scatter will not affect any already existing scenes in a negative way. We also developed unit tests for all parts of Corona Scatter that function as a unit which both prevent small mistakes caused unknowingly and could help to locate a bug should one or more regression tests fail.

We also developed a small application to preview the features of Corona Scatter, which is able to load all geometry inputs of Corona Scatter and then demonstrate the scattering capabilities along with the ability to change all parameters

relevant to this thesis and allows the user to observe the Corona Scatter output.

4.4 Further improvements

It is fully expected that the Corona team will continue its work on Corona Scatter, both by adding features that we did not select to implement (or that we have found during the development) as well as increasing its speed.

The most room for speed improvement in the parts that we programmed would be the regular pattern scattering and distribution modification features. This is because, as we mentioned, our implementation accesses and modifies a lot of memory. Even though we did not find a better solution, if this part was improved, the speed of these features would increase.

While working on Corona Scatter we received multiple feature requests which we have written down and which are up to the Corona team to discuss and implement in the future. These include:

1. Limiting the visibility of the instances to the camera field of view, which reduces the computational power needed to preview the distribution in 3ds Max viewports.

2. To allow the users to paint instances directly on top of the distribution objects (much like in Carbon Scatter as we have discussed in Chapter 1).

3. The ability to change the level of detail of the meshes of the instanced objects depending on the distance from the camera, which would save a lot of time during rendering, as we do not need to render high quality building models that are very far away from the camera.

4. Border behavior, which would allow to modify the instances that are close to the borders of the distribution object in a similar way to how we currently modify instances in the vicinity of spline objects.

5. Restrict the distribution depending on the altitude – allow the user to set an interval $[low, high]$ and scatter instances only on such triangles of the distribution objects that are higher than low and lower than $high$.

6. Restrict the distribution depending on the slope of the triangles which would prevent scattering on triangles that are angled too steeply, simulating the lack of trees on very steep mountains.

Bibliography

[1] Autodesk. 3ds Max. http://www.autodesk.com/products/3ds-max/overview, [Online; Accessed April 8, 2017].

[2] Itoo software. Forest Pack – The Scattering tool for 3ds Max. https://www.itoosoft.com/forestpack.php, [Online; Accessed April 8, 2017].

[3] Visual Dynamics. MultiScatter. http://www.multiscatter.com, [Online; Accessed April 8, 2017].

[4] e-on software. Carbon Scatter. http://carbonscatter.com/, [Online; Accessed April 8, 2017].

[5] Autodesk. Maya. https://www.autodesk.com/products/maya/overview, [Online; Accessed April 8, 2017].

[6] Blender Foundation. Blender. https://www.blender.org, [Online; Accessed April 8, 2017].

[7] Maxon. Cinema 4D. https://www.maxon.net/en/products/cinema-4d/overview/, [Online; Accessed April 8, 2017].

[8] Render Legion. Corona Renderer Daily Builds. https://coronarenderer.freshdesk.com/support/solutions/articles/-5000570015-daily-builds, [Online; Accessed April 8, 2017].

[9] Richard L. Bishop. There is more than one way to frame a curve. *The American Mathematical Monthly*, 1975.

[10] Ian R. Cole. Modelling CPV. 2015. https://dspace.lboro.ac.uk/2134/18050.

[11] Greg Humphreys Matt Pharr. *Physically based rendering: from theory to implementation. 2nd ed.* 2010.

[12] Standard C++. https://isocpp.org/, [Online; Accessed April 8, 2017].

[13] Intel. Intel® VTune™ Amplifier 2017. https://software.intel.com/en-us/intel-vtune-amplifier-xe, [Online; Accessed April 8, 2017].

[14] GLFW – open source, multi-platform library for opengl. http://www.glfw.org/, [Online; Accessed April 8, 2017].

[15] Khronos group. OpenGL – Related toolkits and APIs. https://www.khronos.org/opengl/wiki/Related_toolkits_and_APIs, [Online; Accessed April 8, 2017].

[16] GLEW – The OpenGL Extension Wrangler Library. http://glew.sourceforge.net, [Online; Accessed April 8, 2017].

[17] AntTweakBar – GUI library to tweak parameters of OpenGL programs. http://anttweakbar.sourceforge.net/doc/, [Online; Accessed April 8, 2017].

[18] Joey de Vries. Learn OpenGL. https://learnopengl.com, [Online; Accessed April 8, 2017].

[19] Autodesk. 3ds Max 2017 Documentation and help. http://help.autodesk.com/view/3DSMAX/2017/ENU/, [Online; Accessed April 8, 2017].

[20] Google. Google Test. https://github.com/google/googletest, [Online; Accessed April 8, 2017].

[21] Render Legion. Corona renderer 1.6 for 3ds max released. https://corona-renderer.com/blog/corona-renderer-1-6-for-3ds-max-released/.

List of Figures

List of Abbreviations

BVH	Bounding Volume Hierarchy
CScatter	Corona Scatter object in 3ds Max scene
GUI	Graphical User Interface
OS	Operating system
SIMD	Single instruction, multiple data
SSE	Streaming SIMD Extensions t
UI	User Interface

Attachment 1 – User documentation

OpenGL viewer

Navigating the scene

The users can navigate the scene in a few ways. Moving the camera around can be achieved by clicking the right mouse button and dragging. This moves the camera in the its local X and Y axis. Scrolling with the mouse wheel zooms the scene in and out from its current position. Finally dragging the scene with the middle mouse button rotates it around the origin point $(0, 0, 0)$. The OpenGL viewer application User Interface is illustrated in Figure 4.10.

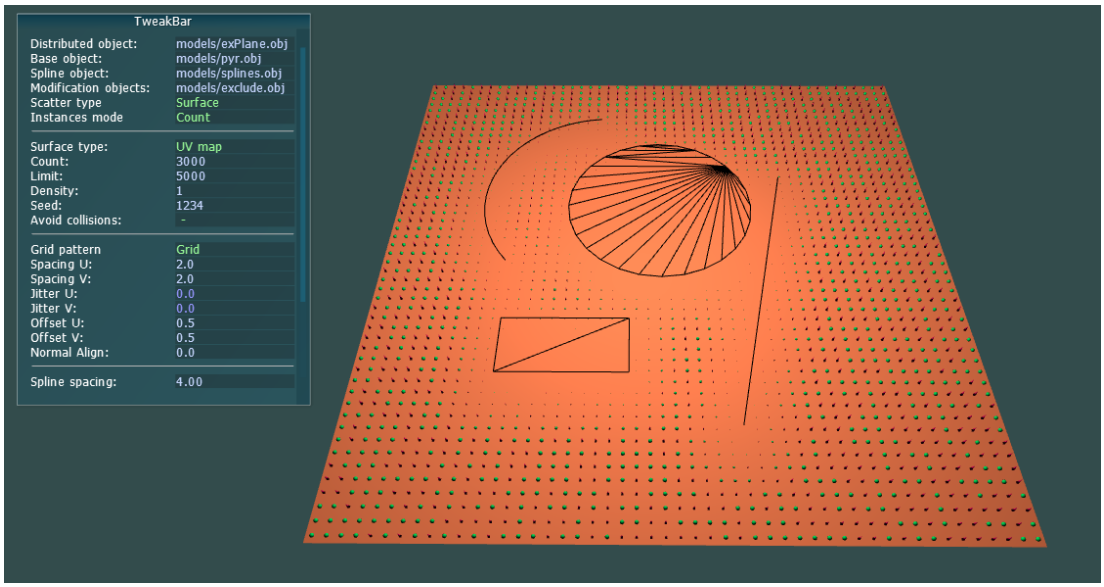

Figure 4.10: A screenshot showcasing the UI of the OpenGL viewer application.

Loading objects

The user can change the distribution objects in the viewer by providing his own scene in Wavefront .obj format. To change the mesh distribution objects, which we scatter onto, the user should enter the scene's path into the "Distribution objects" text field in the UI of the viewer. To change the spline objects we distribute along, the user should enter the scene's path into the "Spline objects" text field. Finally, the user can change the distribution modification objects by entering the scene's path into the "Modification objects" text field. The distribution modification scene should contain spline objects which the user wants to have interpreted as open splines and full triangle meshes for the objects which the user wants to use as area modification objects (which is how the 3ds Max plugin behaves for closed splines).

The user can also change the instanced objects in the following way: a path to one instanced objects can be provided in the "Instanced object" text field. If "Additional instance type" is disabled, this will be the only instanced object to scatter. If the checkbox is enabled, a basic sphere gets added.

Note that these paths should be relative to the position of the .exe file. The recommended way is storing the scenes in the "models/" directory that is already initialized with some basic scenes. Leaving one of these paths empty will result in no corresponding objects being loaded into Corona Scatter.

Setting the parameters

The user can change all the parameters of the Corona Scatter program that are important to demonstrate for this thesis directly from the viewer User Interface. The parameters are divided into sections based on their behavior. There are also several parameters that affect the behavior of the viewer which we will now discuss in more detail:

Randomize seed

Different from the "Random seed" parameter of the Corona Scatter, if this parameter is set to true, the Scatter's seed will be randomized with every re-scattering. Setting it to false will keep the current seed set in the Scatter.

FPS limit

While generating a large number of instances a large strain can be put on the user's computer. This setting limits the times the view is re-painted each second. The default value of 60 should provide the user with smooth experience while not putting too much strain on his system. However, if you are experiencing stutters and freezes, we recommend setting this limit down to 30 or lower, depending on your hardware.

Redistribute – hotkey 'Enter'

This button will instantly redistribute the scattering while re-drawing the viewport.

Using custom shaders

A more experienced user can also provide custom OpenGL shaders if he wishes to alter the look of the scene in the viewer. To change the vertex or fragment shader, copy your shader into the "shaders/" folder and rename it to either "fragment.txt" or "vertex.txt", depending on which shader you would like to replace. Keep in mind that respecting the current shader structure is necessary. Also note that using different shaders may lead to the program behaving unexpectedly, crashing or freezing.

Corona Scatter 3ds Max plugin

In this section, we briefly describe the *User Interface* (abbreviated as *UI* in the following text) of the Corona Scatter plugin for 3ds Max [1].

Installing the Corona Scatter plugin

Installing 3ds Max plugins is as simple as copying files to the right directories. The plugin itself is one file, named `ScatterPlugin.dlt`. This file needs to be copied to the `plugins\` subfolder of the 3ds Max root directory which could, for example, look like this `C:\Program Files\Autodesk\3ds Max 2017\`. The root folder is easily found as it contains the executable `3dsmax.exe` which is used to run the program. After copying the file `ScatterPlugin.dlt`, the reader also needs to copy the other `.dll` files we supplied next to the `ScatterPlugin.dlt` file directly to the root folder of 3ds Max. More detailed instructions will be present in the form of a read-me file in the digital attachment on the DVD.

Note that this way of installing Corona Scatter is only required for the purposes of this thesis. When users install Corona Renderer using its installer, the Corona Scatter is automatically installed alongside it as well. This means that the "normal" means of installing Corona Scatter is very easy but since we, for the purposes of this thesis, provide the Corona Scatter as a stand-alone plugin, special actions are required.z

Creating a Corona Scatter object

Creating a Corona Scatter from the 3ds Max UI is done by clicking Create → Corona → CScatter, as illustrated in Figure 4.11.

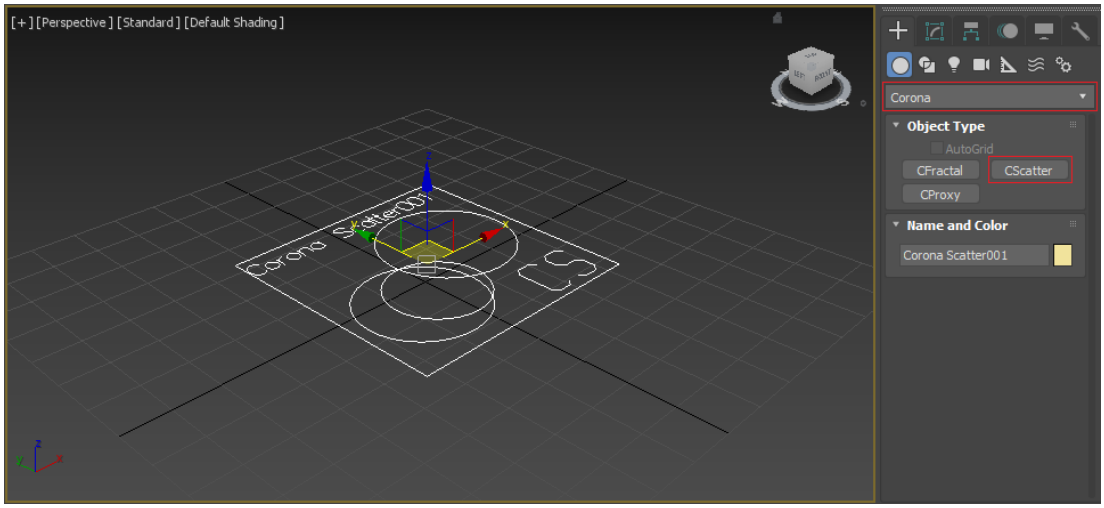

Figure 4.11: A screenshot showcasing creating a CScatter object in the 3ds Max program.

User Interface

Having created the CScatter object, clicking the tab *Modify* provides us with the interface as shown in Figure 4.12. We now go through each UI panel and briefly explain the controls. The UI of the Corona Scatter plugin for 3ds Max is separated into rollouts – small windows that contain the UI controls. This

allows us to divide the controls into logical sections of controls that affect similar settings. The rollouts of the Corona Scatter plugin are:

- **Objects** allows the user to specify the distribution objects and the instanced objects.
- **Scattering** rollout contains all settings that are relevant for all types of scattering.
- **Spline scattering** contains settings for scattering on splines.
- **Surface scattering** contains settings when scattering on surfaces, mainly the choice of random and regular scattering using a UV map.
- **BBox scattering** contains the settings for random distribution in bounding boxes.
- **Viewport display** allows the user to change the distribution preview in the 3ds Max live viewport.
- **Transformations** allow the user to translate, rotate and scale each individual instance randomly.
- **Modification areas** allow the user to add modification splines that will affect the distribution, as we discussed in Chapter 2.

We now discuss each rollout in more detail.

Objects

The two lists in this rollout specify which objects to distribute on (*distribution objects*) and which objects should be scattered – *instanced objects*.

Distribution objects each have a Density setting which indicates the ratio of instances placed on this distribution object in comparison to the other distribution objects. For example, for two objects A and B, if A's density is 1.0 and B's density is 2.0, B will have roughly two times more instances placed on its surface than A. Note that this parameter only takes effect while scattering randomly on surfaces or in the volume of the bounding box and thus does not affect any features we implemented in the thesis.

Each instanced object has a frequency parameter which, similarly to the density of distribution objects, specifies the likelihood of scattering this instance instead of others. Setting the frequency of A to 1 and frequency of B to 3 will result in B being scattered roughly three times as much as A.

Scattering

This rollout is filled with general scattering parameters – parameters that affect all types of scattering and are not specialized in functions. The parameters are:

- Type lets the user choose what type of objects Scatter should scatter on.
  - On splines – choosing this will make the Corona Scatter scatter along splines, exactly as we discussed in Chapter 2, in the section Scattering along splines

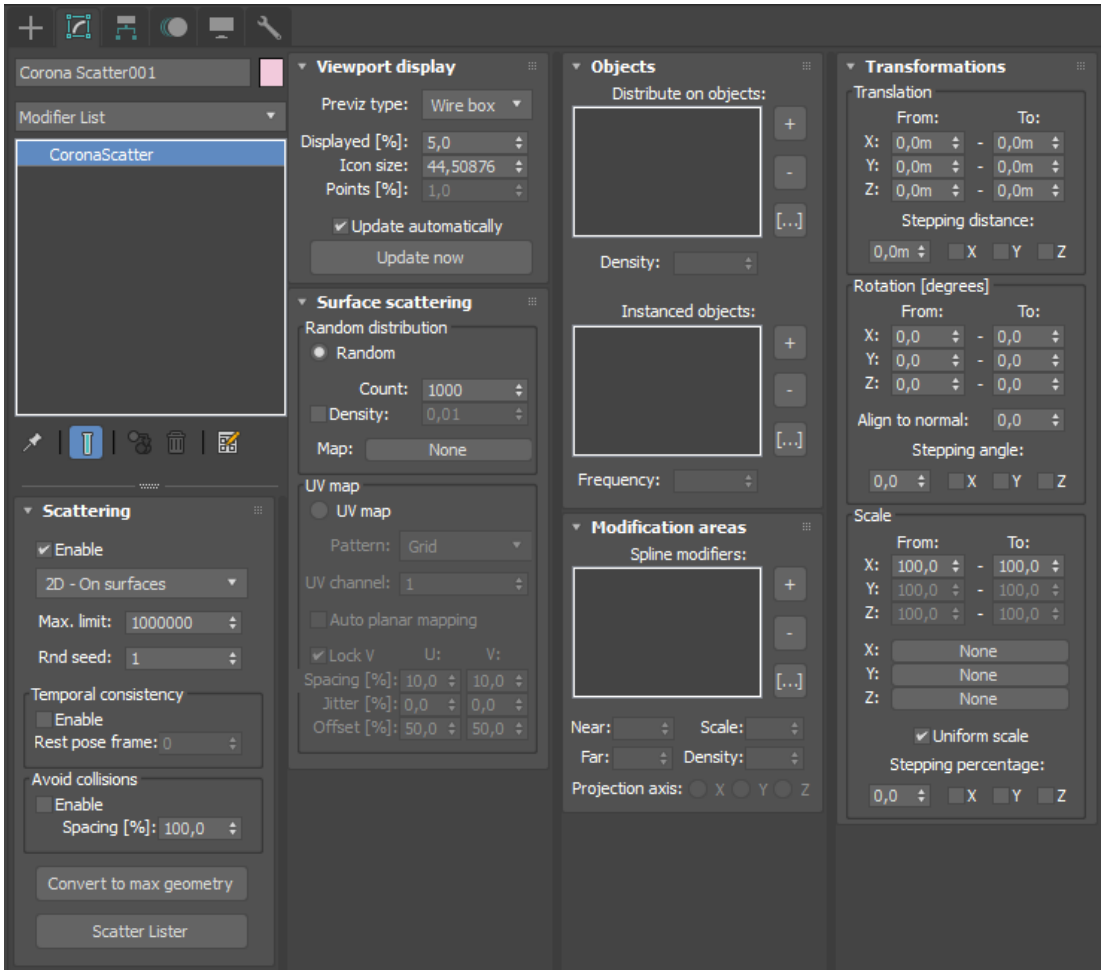

Figure 4.12: A screenshot showcasing creating a CScatter object settings.

  - On surfaces – while this mode is selected, the user has the option to choose either random scattering or scattering regular patterns using the UV space
  - In BBox – this mode will scatter randomly in the bounding volume of the distribution objects.

- Limit is the safety net for how many objects the CScatter should generate at maximum. This is very crucial since setting the wrong parameter can lead to billions of instances getting generated, which could result in crashes or freezes.

- Rnd seed is the random seed for the random generators used when scattering randomly.

- Temporal consistency is a setting used when the users are animating a scene with a CScatter object. Enabling this ensures that the CScatter will not re-scatter during the animations which prevents instances appearing at random at some frames.

- Avoid collisions allows the user to enable or disable generation of instances

that collide with each other.

- Convert to max geometry – this button takes all the instances generated by Corona Scatter and converts all the instances into separate geometry objects.

Only one of the following three rollouts appear at the same time, depending on which Type chosen in the Scattering rollout.

Spline scattering

This simple rollout allows the user to set all three parameters of spline scattering we discussed at in the Problem Analysis in the section Scattering along splines. The three parameters *Spacing*, *Jitter* and *Offset*.

- Spacing sets the distance between each instance scattered on the spline.
- Jitter sets the amount of randomness in the distribution.
- Offset moves all instances along the spline by the same distance.

Surface scattering

This rollout is divided into two sections *Random* and *Uv map* corresponding to the two modes which the users can scatter on surfaces. To keep the UI clean and simple, we disable all controls in the section that is currently not used. The parameters are:

- Random
  - The **Count** and **Density** spinners work hand in hand with one being active at the same time. This allows the user to either precisely specify the number of instances to scatter (using the Count parameter) or set the density of the distribution and let the CScatter calculate the required number to scatter.
- UV map
  - **Regular pattern** lets the user choose one of the three patterns we chose in Chapter 2.
  - **UV channel** specifies the channel which the CScatter should take the UV mappings of geometry objects from. The Planar checkbox is checked, the UV mapping is calculated by CScatter itself as by projecting the geometry object to the XY plane.
  - **Lock V** checkbox disables setting parameters for the V coordinate and keeps the values for U and V the same.
  - **Spacing in** % determines the spacing between rows and columns of the pattern in absolute UV coordinates times 100.
  - **Jitter in** % specifies the amount of randomness added to the regular pattern.
  - **Offset** specifies the translation added to each generated instance in the pattern.

BBox scattering

The settings for BBox scattering are very similar to the random section in Surface scattering. Again, one of Count or Density is used to determine the number of instances to scatter in the volume defined by the distribution objects' bounding boxes.

Viewport display

This rollout allows the user to change the way the generated distribution is previewed in the viewport window of 3ds Max.

- **Previz type** allows the user to change the way how each instance is displayed. The options are: None, Dot, Box, Wire box, Full and Point cloud.
- **Displayed in** % modifies how many instances are displayed in the viewport.
- **Icon size** sets the size of the CScatter object icon.
- **Points in** % determines how many points are shown per instance and is only active if Previz type is set to Point cloud.

Transformations

This rollout is used to apply transformations to the instances post-scattering separately in each world axis. The user can add any amount of translation, rotation or scaling to each instance. The amount is specified in the form of an interval $[From, To]$. There also is an option to divide this interval into discrete steps by setting the appropriate Step spinner. For example, setting the rotation interval to $[0, 90°]$ and setting $step = 30°$ will rotate each instance by one of $0°, 30°, 60°, 90°$.

An important setting here is the **Align to normal** spinner. This allows the user to align the instances to either the -Z (inverse world vertical axis) when set to -1, the normal of the distribution object when set to 0 or Z (the world vertical axis) when set to 1. Since the value can be set to any number from $[-1, 1]$, numbers other than -1, 0 and 1 will linearly blend these values. For example, setting this parameter to 0.5 will blend the surface normal and Z axis one-to-one.

Exclude areas

This rollout allows the user to add all spline objects that should be used to modify the distribution as we discussed in Chapter 2. The spline modifiers are then listed in the box. When the user selects one spline modifier from the list, the rest of the controls becomes active and allows the user to set all input variables that we presented in Chapter 2 – the **Near** and **Far** region as well as the weight determining how much each modifier affects the scaling of the instances (spinner box **Scale**) and the density (spinner box **Density**). The last control available to the user are three radio buttons allowing the user to set the **projection axis** of each spline modifier.

Attachment 2 – CD contents

The compact disc attached to this thesis contains the following data:

- `full_thesis_hojdar_stepan.pdf`, the PDF version of this thesis
- Scatter plugin for 3ds Max in the `3dsPlugin\` folder. This is the compiled version of the Scatter Plugin for 3ds Max we developed in this thesis.
- OpenGL Viewer in the `OpenGLViewer\` folder. This is the compiled OpenGL Viewer we developed. The two important subfolders are:
  - `models\` this folder contains all scenes that can be loaded into the OpenGL Viewer.
  - `shaders\` this folder contains both the vertex and the fragment shader of the OpenGL Viewer.
- The folder `sources\` which contains source files of this thesis. Namely the Scatter Core and the OpenGL viewer.