Particle Driven Weathering System

Masters Thesis

João Montenegro Almeida

N.C.C.A Bournemouth University

September 10, 2007

This Masters Thesis is dedicated to

Solcito.

Acknowledgements

I would like to thank the following people:

All my lecturers at the NCCA, specially Jon Macey for his support during all the course.

All my colleagues at the NCCA, with a special thanks to Johannes Saam for sharing his valuable knowledge and friendship, Gerard Keating for his friendship and company during all those hours we spent in the labs working on our final projects, Michael Garrett for his enthusiasm on my work and for all the inspiring company during all the dinners of Chinese food during all the last terms, David Minor for his words of support, Ali Derweesh for his sense of humour that helped me to relax my mind from time to time in the labs and Ritchie Moore for his iconic Cave Troll model.

All my new friends from Bournemouth University, who were always present in all the great adventures during the year.

My best friends from Portugal, a endless source of inspiration.

My family for everything.

And to Sol, for her invaluable company (via internet) from the other side of the ocean during the last months.

Contents

List of Figures

Chapter 1

Introduction

Computer generated images are being used more every day to represent reality in photo-realistic and stylised ways. The mathematical models used in the process of synthesis of these images are usually simplifications of the phenomena in the complex reality. A common problem associated with this fact is that these images are usually too smooth, clean and simple, in other words: too perfect. Although the real world is more rough and ”less perfect” than the images synthesised in a computer, its ”imperfections” follow physical rules that originate patterns familiar to the human eye. Computer Graphics evolution tries to implement models capable of generating these imperfections.

Real objects are exposed to an environment that changes their shape and appearance through time. For example, buildings are exposed to rain carrying substances that stains their walls, rocks are exposed to rain and wind that erodes them, etc. All these weathering phenomena related with the natural interactions between objects and the environment are a source of visual imperfections that, when considered in the images synthesis, can produce more convincing results. Although the patterns generated by weathering phenomena don’t look totally random, they are usually fuzzy, which allows the use of non-physically based techniques to produce them in a convincing way.

A common approach to solve this problem is the use of the skills of texture-painting artists who paint these effects. When the scale of the models is high (for example, a full city) this approach may not be feasible and some automated processes can be used. An example of this is the film King Kong by Peter

Jackson, in which a particle driven weathering system was used to generate stains in the walls and snow in the rooftops of a procedurally generated New York of the 1930's [15]. Such a system must be able to generate weathering effects in a efficient way. A particle system is a good way of simulating the effects created by rain and wind hitting the surfaces of the objects.

In this Masters Project, a weathering system was developed based on a particle system that interacts with the model. In its development, the problem of the possible large scale of the models was taken into account, and a fast way of calculating the collision detection between the particles and the model was implemented. This collision detection is based on implicit representation of the geometry using depth maps rendered by cameras deployed in the scene. The time-efficiency of this approach doesn't depend on the amount of geometry, since the input data size and type doesn't change between different 3D models. The output of the system is a texture containing a mask of the weathering effects that can be camera-projected back into the geometry. This mask can be used in the change of appearance of the models, by changing the colour of the surfaces in the compositing stage (to produce staining, moss accumulation, rustiness, etc.) or as a displacement map, to deform the geometry (to produce erosion and sedimentation effects).

The developed system was designed to be plugged into any pipeline. For this reason a full particle system was implemented. As a proof of concept, it was also developed a Autodesk Maya's plugin that uses all the code of the system, integrating it in Maya's interface.

Chapter 2

Previous Work

2.1 Weathering Systems

Due to the specificity of the topic of automatically generated weathering effects, the amount of previous research and development is less than in other areas of Computer Graphics. Yet, some work from other researchers and developers was studied before the development of this project.

One of the main inspirations for the project was concentrated on one paragraph of a one page ACM Siggraph 2006 Sketch by Chris White (Wetta Digital) [15]. In this sketch the weathering system used by Wetta in the production of King Kong is shortly explained, although no details about the actual implementation is shown. It is based in depth maps and normal maps rendered by cameras positioned roughly close to the actual camera used in the shot. It uses Maya Particles to stain the walls of the buildings of New York and to deposit snow in the scenes. The particles are driven by a custom force field capable of calculating the collisions between the particles and the depth/normal maps. The collisions increase the brightness of a mask texture that could be camera-projected onto the buildings and composited afterwards.

Another important source of inspiration was the paper published by Yanyun Chen et al [3] in which an artist-friendly (rather than a physically correct) weathering system is specified. It was inspired by the photon tracing in Photon Mapping [8, 9, 4]. An entity similar to a photon (the gamma-ton, which is in this case a particle that carries a certain amount of dirt) is emitted by sources and collides with the surfaces. Each collision is stored in a gamma-ton map (similar to the photon map) that is used on a second pass to synthesise textures to be used as colour pass or displacement maps of the weathering effects. Since the emission of gamma-tons is an iterative process, the weathering effects produced can be animated. Upon a collision, a gamma-ton can either be reflected, bounced (same as being reflected, but being affected by gravity), flow on the surface or settle (or absorbed). The particle simulation is different from the classic approach of integrating the positions and velocities according to the forces, since it uses ray-tracing to explicitly calculate the impact positions of the gamma-tons without having to travell through the space between sources and surfaces. The surfaces have properties (similar in concept to shaders in photon mapping) that control the way particles react on each collision. This involves probabilities of mechanical behaviour of the particle (reflect, bounce, etc.) and also exchange of dirt values between the particles and the surfaces. The system is capable of generate convincing dirt maps, stain bleeding (transport of dirt, rust or moss collected by the particles upon a collision and consequent deposit on other surfaces) and effects like patina (chemical compounds formed in metallic surfaces) without using a physically correct process.

Physically correct models for weathering processes were researched before. Julie Dorsey et al. developed a model to weather stone [5] in which both the appearance and the shape of the stone is altered by interactions with the environment. This model considers phenomena like the transport of minerals and other substances by water that flows and evaporates inside the volume of the stone, crystallisation of minerals and erosion. A slabs system (a set of volumetric data-structures aligned with the surfaces) is used to delimit the space where the environment interacts with the objects. After the simulation of the physical and chemical processes, the rendering of the results uses some techniques like subsurface scattering. Julie Dorsey et al developed also a model of water flowing on surfaces [6], which is one of the most important phenomena related with surfaces weathering. In this model the water drops are modelled by a particle system that implements physically accurate movements of flowing drops as well as the calculation and chemical interactions with the surfaces. This are achieved by a set of surface parameters and a it uses a couple some differential equations to implement a physically based absorption and sedimentation processes.

Texture synthesis approaches were also researched and developed in order to generate textures with dirt, stains or other patterns with the objective of make the appearance of the surfaces less perfect and clean. One of the most famous approach is the noise model developed by Ken Perlin [12, 1, 7] that implements a $1/n^f$ noise (which occurs in natural processes) that can be used to synthesise noisy, organic textures. This textures can be used to create a less perfect and less smooth appearance of the objects. Witkin and Kass developed the Reaction-Diffusion approach [18] which generates weathering patterns on textures according to natural (specially biological) processes that generate, for example, the visual patterns on some animals (like the zebra stripes), patterns in sand ripples. The patterns can be used to generate weathering features (like moss generation) on the surfaces.

2.2 Particle Systems

The first particle system used in a real production was designed by William Reeves [13] for the movie "Star Treck - The Wrath of Khan". This particle system had all the basic characteristics of the modern ones: emitters, particle life duration, forces, etc.

Particles-surfaces collisions is an important feature that usually is the bottleneck of the computation of the particle system. A. Kolb et al [10] developed a particle system to be used in real-time in the GPU. In order to accelerate the collision detections the geometry was represented implicitly using depth maps. This an extremely efficient way of solving the problem (meant to be implemented in real-time simulations). Their paper shows how to calculate such a collision detection.

Chapter 3

Technical Background

This section presents some Computer Graphics principles, techniques and technologies that were used in the development of the project. The way these topics were actually used, as well as the reasons why they were used, will be presented in section 4 on page 22.

3.1 Particles Systems

Particles systems are widely used in Computer Graphics to produce different effects. The basic idea behind them is to have particles flying around the 3D scene. These can be rendered directly using different techniques (to create effects like debris of explosions being projected or fluid-like objects using metaballs, for example) or used in more indirect ways, making them interact with the 3D models (which is the case of the implemented Weathering System). Usually the user deploys particle emitters in the 3D space. The particles contain a position in space and velocity along with other optional values (acceleration, applied forces, life time, mass, amount of transported dirt, etc.). The user can also deploy force fields that change the movement of the particles by applying them some kind of forces. The particles can also collide with the objects in the scene, which can change their properties. The movement of the particles is usually driven by a physical model that calculates the positions and velocities on each step (or frame) of the simulation, taking into account forces being applied to the particles. This is achieved by using an integration of the accelerations into velocities and velocities into positions. There are several techniques to do this integration, one of the simplest one is the Euler Integration (which was used in this project). Other Techniques like the Verlet Integration [17] or Runge-Kutta [11] can also be used.

3.2 Euler Integration of Particles Movement

Newton's Second Law of Motion states that a force is directly proportional to the mass times the acceleration Since the mass of a given particle and the resulting force being applied are known, its acceleration on a certain moment can be calculated:

$$F = ma \Leftrightarrow a = \frac{F}{m}$$

(Newton's Second Law of Motion)

Knowing the acceleration of the particle, its velocity and position can also be calculated (which is the goal of each iteration of the simulation). Newton also defined the relationships between acceleration, velocity and position in an analytic way using calculus (actually, his research on this area lead to the co-invention of calculus itself by Newton). These are the relationships between acceleration, velocity and position between the interval of time [t0, t1]:

$$v(t) = \int_{t0}^{t1} a(t)dt$$

$$x(t) = \int_{t0}^{t1} v(t)dt$$

In a simulation these integrals must be calculated between t=0 and t=current frame's time, once per simulation step. The numerical methods usually applied in the computation of these integrals are iterative and use the calculations from previous frames to infer the next values. In this method the acceleration is calculated on time t+Δt (in which Δt is the duration of a frame o simulation step) using Newton's 2nd law then the velocity is calculated using the velocity on t (from the previous simulation step) and finally the position in a similar way [16]. These are the formulas to do this:

1. $a(t + \Delta t) = \frac{F(t)}{m}$;
2. $v(t + \Delta t) = v(t) + \Delta t\, a(t + \Delta t)$;
3. $x(t + \Delta t) = x(t) + \Delta t\, a(t + \Delta t)$;

For each simulation step and for each particle these formulas are applied to calculate the new position on the new step. Between steps, both the velocity and the position of the particle must be stored (in order to have the $v(t)$ and $x(t)$ of the 2nd and 3rd formulas).

3.3 Depth Map

A depth map is a rendered image of a 3D model containing in each pixel (or texel) the value of a distance to the corresponding rendered point in the surface of the model. This value can be stored on any channel of the image and can be a measurement of the a distance in any unit. In this project it was assumed that the channel that contains it is the RED channel of the image and the distance is measured in the unit used in the world space.

3.4 Normal Map

A normal map is a rendered image of a 3D model containing in each pixel (or texel) the normal vector of the corresponding rendered point in the surface of the model. This normal can be in any coordinate system (world space, object space, camera space, etc.). In this project it was assumed that all the normal maps represent the normals in world space.

3.5 Bresenham line drawing Algorithm

In order to draw correct particle flowing streaks it was necessary to connect consecutive particle positions in texture space using a line drawing algorithm. Bresenham Algorithm [2] was chosen due to its efficiency, since it uses only integer calculations. The basic algorithm draws a line in a raster space connecting two points P and Q, being the X and Y coordinates of Q higher than P and with a slope between 0 and 1 (for all the other cases, the same algorithm is used mirroring the coordinates or swapping the X and Y coordinates of the calculated points). Such a slope guarantees that there will be only one pixel of the line for each column of the image.

In this basic case the algorithm starts at P iterates through all the columns between Px and Qx and on each step, instead of explicitly calculating the Y value of the pixel using the line equation, the algorithm decides if the current Y value should be decremented by 1 or not. The decision is made using an *error* variable (integer) that is initialized with half the distance in X between P and Q:

$$error = -\frac{P_x - Q_x}{2}$$

And on each iteration *error* is incremented with the Y distance between P and Q:

$$error = error + (Q_y - P_y)$$

Whenever *error* is larger than zero, then the current Y position is decremented by 1. Here is a pseudo-code version of the basic algorithm:

Algorithm 1 Bresenham's Basic Line Drawing Algorithm

```
BasicBresenham( P, Q, colour )
    deltaX = Qx - Px;
    deltaY = Qy - Py;
    error = - deltaX / 2;
    y = Py;
    for (x = Px to Qx) do
        error = error + deltaY;
        if( error > 0 ) then y = y + 1;
        drawPixel(x, y, colour);
    end for
end
```

The idea behind this algorithm is to use integer values instead of using the real (floating point) numbers calculated with the line equation. These integers can be seen as the corresponding floating point values scaled with deltaX.

3.6 Photon Mapping - 2nd Pass

In the development of the Weathering System a technique similar to the Photon Mapping's 2nd pass was used to gather the final dirt texture. The original Photon Mapping [8, 9, 4] is a Global Illumination rendering algorithm which is able to implement effects such as caustics and colour bleeding. This algorithm is divided into two passes.

In a first pass, photons (particles containing a certain amount of radiant power moving in a certain direction) are emitted from the light sources into the scene, bouncing on the surfaces on every contact (detected using raytracing). All these contacts are stored in a space partitioning data structure (usually a KD-Tree) with the positions, direction and amount of radiant power of the colliding photons.

In the second pass, a gathering technique is used to soften the light influence of each element stored previously on the surfaces of the scene. The final image's pixels are iterated and, for each, a standard raytracing calculation is done. On each ray-surface hit point, the density of indirect light is calculated searching the surrounding photons stored in the KD-Tree using a range search technique. The calculation of the light density is done with list of closest photons using the following formula:

$$L_r(x,w) \approx \textstyle\sum_{p=1}^{n} f_r(x,w_p,w)\frac{\Delta\phi_p(x,w_p)}{\pi\, r^2}$$

In which:

L_r = indirect radiance on that point;

f_r = BRDF on that surface;

$\Delta\phi_p(x,w_p)$ = radiant power of each photon;

r = maximum distance between the point and the photons in the list of closest photons;

A filtering might be used to attenuate the influence from photons farther away from the point. This filtering is done using a weight that is multiplied with the power of each photon, and is usually a function of the distance r. For example, to implement a cone filter, the following weight can be multiplied by each photon power:

$$weight = 1 - \frac{d_p}{k\,r}$$

In which k $>$ 1 is a value that characterizes the filter, dp the distance between the rendered point and the current photon and r is the distance to the farthest photon in the list.

3.7 Perlin noise / turbulence

In this project a turbulence force field was developed to add a more chaotic, yet organic, movement of the particles. The force is calculated using Perlin noise [12, 1, 7]. This noise was developed by Ken Perlin in order to create realistic noisy, organic textures. The calculated noise can be generated in spaces with any number of dimensions as well and can be used not only to produce textures, but to drive any value in space. The basic idea is to have a noise function in 4D space (X, Y, Z, time) with an extra variable, the frequency (that drives the size of the noise grain), that returns a noise value (usually between 0 and 1):

$$noise(x, y, t, freq) \;:\; \mathbb{R}^4 \rightarrow \mathbb{R}$$

This functions usually use a set of random values that can be used in different positions in space and then interpolated to fill all space with values. The density of these random points in space is proportional to the given frequency. In order to increase the quality of this noise, a turbulence function can be implemented. The idea is to calculate different noises with different frequencies and blend all the values so that the result becomes more irregular in terms of frequency, yet organic. Usually the frequencies are scaled by a factor of 2 between themselves, and because of this each noise is called octave (like the musical scales, in which the frequency of the notes of the next octave have twice the frequency of the current one). The blending between octaves uses an weighted sum that attenuates more the high frequencies.

Figure 3.1: Perlin noise: a) low frequency; b) high frequency; c) turbulence

3.8 Planar Texture Projection Mapping

A possible way of mapping a texture into a 3D model is to use a planar projection. This technique uses a projection matrix similar to the one used in a camera projection. In a camera, the projection matrix when multiplied by a vector or point in world space, transforms its coordinates into NDC space (Normalized Device Coordinates, a coordinate system in which the X and Y components align with the camera plane with values between 0 and 1 and the Z axis is perpendicular to them).

$$P_{NDC} = P_{world}\, M$$

in which:

P_{NDC}, P_{world}: the point being transformed (in NDC and World coordinates)

M: the projection Matrix

The planar texture projection uses a similar approach to apply a texture to an object during its rendering. If a camera is pointed towards an object, it can project a texture on it in render time by transforming the world space position of each rendered point on the surface into texture image space, returning the colour stored on the corresponding texel.

3.9 Maya API / Mel

Maya is a node-based 3D computer graphics software package extensively used in the Computer Animation and Visual Effects Industry. It presents a framework that allows software developers to add new features to the existing ones. Maya version 8.0 was used in the development of this project and presents two main ways of develop new features and plug-ins: the MEL scripting language (which is a high level simple scripting language that allows to manipulate virtually any node in Maya) and the Maya C++ API (which allows the develop new nodes or MEL commands).

3.9.1 Locator Nodes

A relatively simple way of developing a node that is able to show visual information on a OpenGL Maya 3D viewport is to use a Locator Node in a plug-in developed using the Maya C++ API. This is done extending the class *MpxLocatorNode*, a child of the standard node's class: *MpxNode*. The *MpxLocatorNode* provides the function draw(), which allows the developer to add OpenGL code that can be rendered in the Maya viewport. All the 3D visualisations of the Weathering System in Maya were developed using Locator Nodes.

3.9.2 Mel Utility and Interfaces Scripts

Mel is useful to do set-up work, such as creating new nodes, connecting existing nodes, etc. In this project it was extensively used to do all the weathering scene set-up work. Another useful feature provided by the Mel library is the user interface (UI) creation. This was also used in this project for all the interfaces related with the Weathering System in its Maya integration.

3.10 Open EXR image format and API

The only image format used by the Weathering System is the open source Open EXR format, developed by ILM. This format allows to store an arbitrary number of channels (not only the standard RGBA like in most of the formats) and may store the information in floating point values. ILM released not only the format, but also a C API that allows to integrate this format in any software. It presents two sets of functions to read and write images:

1. Simple: that assumes that the existing channels are RGBA and deals with the values in half-precision (16bit) floating point format. This set of functions is simpler to use due to these assumptions;
2. Advanced: which allows the developer to specify an arbitrary set of channels and other values formats (16 bit integer, 16 bit - half precision floating point and 32 bit -full precision floating point).

Due to the necessity of representing very precise Z-depth values, in this project the latest set of functions was used, in order to use the full precision floating point.

Chapter 4

Implementation

4.1 Main Objectives

The main idea of the developed Weathering System is to generate several weathering effects such as dirt/moss/patina/rust accumulation (surface appearance change), erosion and sedimentation (geometrical changes) on the surfaces of a model. This system was designed to allow to automatically generate these effects on any kind of model, no mater its scale. A common application of such a automation is in models of large procedurally generated cities, in which it would be impossible to paint by hand all these effects in a reasonable time frame. The main goal was not to create a 100% physically correct system, but to create convincing weather effects to be added both to photorealistic and non-photorealistic productions. The weathering effects are very fuzzy but follow some patterns, such as dirt and rust sliding in vertical planes, patina accumulation in more self occluded areas of the surfaces, etc. Another important objective was to develop a system that could be easily integrated on different 3D Software and Pipelines.

4.2 The Approach

The developed system can work as a standalone program that can also be integrated on different 3D packages. To demonstrate this, an integration on Maya was also implemented. The weathering effects are achieved by bombing the surfaces of the model with particles that change the properties of these surfaces.

A scalable particle system was developed from scratch, with different types of particles emitters that can be deployed on the scene and different force fields that can affect and drive the particles movements.

4.3 System Outputs

The output of the system is simply a texture with a mask that shows where a certain weathering effect occurs on the surfaces. This texture indicates the amount of interaction between the particles and the surfaces. For the effects that involve geometrical alterations of the surfaces (erosion and sedimentation) the resulting texture can be used as a displacement map. In the cases in which the effects affect only the surfaces appearance, the resulting mask can be used in compositing to reveal dirt, moss, rust, etc. The actual look of these types of effects must be rendered separately and premultiplied by the mask, for example. The output mask is camera-projected into the geometry by a camera previously specified by the user.

4.4 System Inputs

The inputs of the system are a description of the particle system's set-up and a set of maps containing information about the geometry and the surface properties. These maps are rendered from a set of orthogonal Input Cameras deployed on the scene by the user. Each camera renders the following maps:

- Depth Map: implicitly representing the positions of the surface points;
- Normal Map: with the normals on those points;
- Impact Map: with probabilities of different mechanical reactions of the particles upon the impact with those surface points;
- Dirt Map: with information of amount of accumulation of one weathering element (dirt, for example) and some depositing/removal rates of that element on the surface;

Each camera projection contains a 2.5D information of the surface (with the Depth Map), and several cameras can be added together to recreate the geometry of the model using only these projected maps. No polygon meshes or parametric surfaces are passed to the system. All the inputs information are described in an XML file plus the image files with the maps. The output mask is generated from the point of view of another Output Camera, which contains only the depth and normal maps.

4.5 System Process

The particles collisions are calculated using only the re-generated 3D model from the projected maps. Each particle transports a certain amount of a weathering element (for simplicity, lets call it just dirt) and upon a collision it interacts chemically with the surface according to the parameters specified in the Dirt Maps, depositing or collecting dirt from the surface. After this chemical interaction there is a mechanical reaction of the particle according to the value in the Impact Maps. There are 3 types of reactions:

1. Absorption: in which the particle dies after the collision;
2. Bounce: the particle is reflected on the surface and continues its movement;
3. Slide: the particle flows along the surface.

Both the bounce and the slide reactions are influenced by the particle's velocity, the forces applied to it and the surface normal (given by the Normal Maps).

After the simulation the information about the dirt in the different Dirt Maps are accumulated by the output camera (from its point of view) into the output map. At the end, a process similar to the Photon-Mapping's 2nd pass is used on the resulting texture using also the depth and normal maps rendered from the output camera to create a smoother filtered output taking the surface shape into account.

4.6 Justification of Relevant Decisions

a) Development of an independent Particle System instead an existing one (ex: Maya's Particles):

The development of a particle system from scratch was both an opportunity to gain some experience on the area of physical simulations and a way of allowing the inclusion of this system on any pipeline, detaching it from a specific Software.

b) Collisions against Depth/Normal maps and not the geometry itself:

As seen before, such a system is commonly used in large procedurally generated models. Particles collision detection is one of the most time-consuming processes in a particle simulation, this approach may not be 100% accurate, but is very fast and its execution time doesn't depend on the amount of geometric information. For example, it can take the same time to calculate collisions with a single house and a full city. The lack of some accuracy can be immensely reduced with a good input cameras placement.

c) Not totally implemented on an existing 3D environment (like Maya):

With this approach the design was initially concentrated on the computer graphics principles rather than on the technological challenges of developing for a specific package. Anyway, the integration on Maya allowed some work on this area. At the beginning of the project some tests were done with Maya particles driven by the API with a new Force Field Node, but it was soon realised that the problems of struggling with Maya API's would be bigger than the design of the system itself.

d) Use of Mental Ray as renderer of the projected maps in the Maya Integration:

Maya Software Renderer is fast, but Mental Ray has more nodes that can be easily used to generate the depth and normal maps, for example. Another reason is the lack of support by Maya Software Renderer for the Open EXR image format, which was chosen format due to its floating point data and its growing popularity in the Industry. Other renderers, like Renderman, were not chosen because some extra pipeline between it and Maya would be needed, which was out of the scope of this project.

e) Use of orthogonal planar projections:

Any kind of projection could be used by the system (any projection matrices can be supplied to the system), but Orthogonal cameras are preferable due to the simplicity of generation of their projection matrices and because perspective projections would create different resolutions of the depth and normal maps at different camera Z distances. Without a constant resolution through all the model, the exchanges of dirt between the particles with the surfaces would have to be compensated for different Z depths (in collisions farther from the camera the amount of deposited dirt on the Dirt Map would have to be less than in closer collisions).

4.7 The Usage Workflow

4.7.1 Generic Workflow

The system contains a set of core classes that can be compiled into a standalone program. In a simple and generic workflow for the Weathering System this standalone application can be used. To operate the system, the user must provide:

- a set of input projection maps with the corresponding projection matrix;
- one output projection (with a depth and normal maps and the file name of the output texture file which will contain the dirt mask);
- the information about the particles system set-up (positions and parameters of the emitters and the force fields).

After running the particles simulation (which can be seen in a OpenGl viewer in the standalone version of the system) the output is an image file with the dirt mask to be camera-projected back into the geometry using the output camera's projection matrix. Any renderer or 3D software could be used in this pipeline, since the input format is very generic (maps using the Open EXR floating point format).

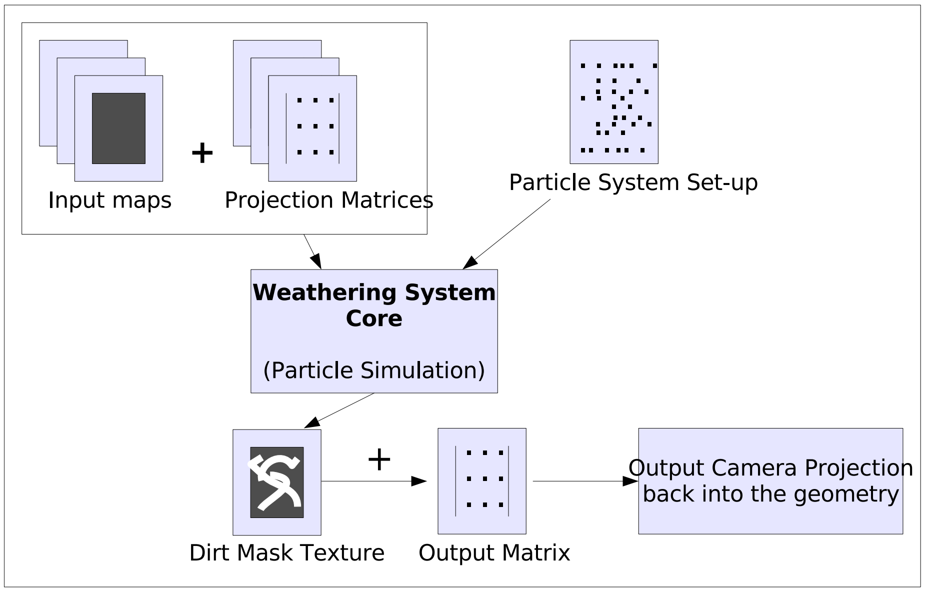


Figure 4.1: Generic Weathering System Usage Workflow

4.7.2 Maya Integration specific Workflow

A Weathering Plug-In containing several scripts and nodes was developed in this Maya integration. A main Weathering Simulation node contains all the core classes contained also in the standalone version of the system, which means that no re-writing of the system was necessary in this integration. The user just have to have a clean polygon mesh version of the geometry (other types of surfaces can be easily converted into polygons) and then follow these steps:

1. Set-up the Scene (which creates all the shaders, render layers and renderer settings);
2. Create the input and output cameras;

3. Render the input maps;
4. Create and place the Particle Emitters (which are different from the ones provided by Maya);
5. Export the scene into the XML format;
6. Create a Weathering Simulator node and specify the XML scene file;
7. Run the Simulation;
8. Generate the output mask using the Weathering Simulation Node;
9. Camera Project the mask into the geometry using wither a Surface Shader or a Displacement Shader (depending on the type of weathering effect);
10. Render the scene from any camera (which can output the dirt maps pass which can be composited with the actual render of the scene);

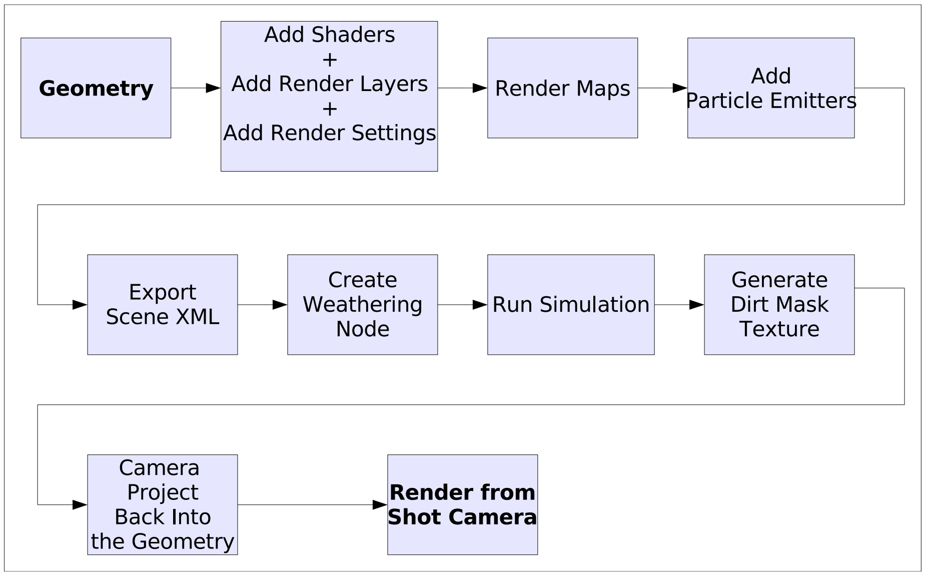

Geometry
Add Shaders + Add Render Layers + Add Render Settings
Render Maps
Add Particle Emitters
Export Scene XML
Create Weathering Node
Run Simulation
Generate Dirt Mask Texture
Camera Project Back Into the Geometry
Render from Shot Camera

Figure 4.2: Maya Integration Usage Wokflow

Although the list of steps in this process is relatively extensive, most of them are automated by the scripts of the Weathering Plug-in, which means that the user has only to click one button from most of these steps, as will be shown later in this document. Mapping this workflow with the generic one presented before:

- Steps 1 to 5: correspond to the Inputs preparation;
- Steps 6 to 8: correspond to the actual running of the simulation by the core;
- Steps 9 to 10: corresponds to the usage of the output mask back into the scene.

4.8 System's Core Implementation

In this section the most relevant C++ classes that implement the core of the Weathering System will be described. The implementation of each class can be found in the *.h* and *.cpp* files with the same name of the classes. The class diagrams presented in this section contain only the most relevant classes, member variables and methods. Along with this Thesis, the code and the corresponding documentation generated with *Doxygen* is handed in, where more detailed information about the classes can be found.

4.8.1 High Level Design

From a high level point of view, the code can be seen as black box containing the set of classes that implement the particle system, which includes also the weathering processes happening between the particles and the surfaces. This set of classes has connections with two external classes:

- `WeatheringScene`: responsible for the parsing of the XML file with the scene description;
- `OpenGLViewer`: responsible for the rendering (on a OpenGL window) of the particles simulation, to be used by the standalone implementation.

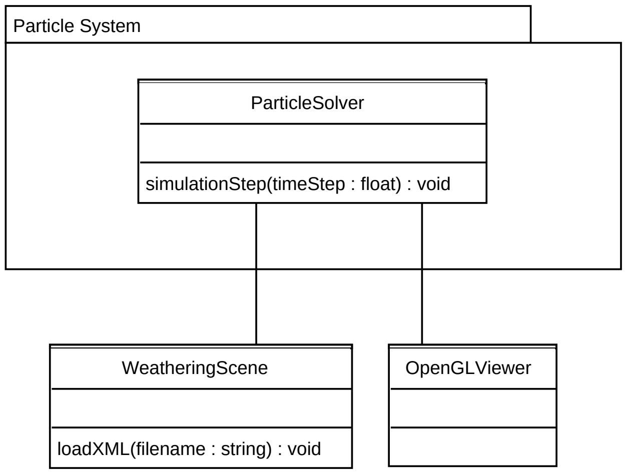

Particle System
ParticleSolver
simulationStep(timeStep : float) : void
WeatheringScene
loadXML(filename : string) : void
OpenGLViewer

Figure 4.3: High Level Class Diagram

4.8.2 Particle System

The core of the particle system is relatively detached from the rest of the system, which allows its use on other applications as well. New particle emitters, force fields and particle-surface colliders can be easily integrated extending the abstract classes *`ParticleEmitter`*, *`ParticleForceField`* and *`ParticleCollider`* respectively. Figure 4.4 on the next page shows the class diagram for this part of the code.

`ParticleEmitter` class and its childs are responsible for the generation of new particles in the simulation. The *`emitParticles(float currentTime)`* method checks the amount of particles that must be emitted since the last simulation step, given by the emission rate multiplied by the *`deltaT`* of the simulation step and generates those particles. Two implementations were provided:

- *`PointParticleEmitter`*: emitts particles from a point in space in all the directions randomly selected in a uniform way each time a new particle is created.
- *`HemisphereParticleEmitter`*: emitts particles from an hemisphere with a specified radius and center. The hemisphere is uniformly sampled generating a random point in a cube (randomizing the X, Y and Z coordinates between -1 and 1), if the point is outside the unit sphere (calculating the

distance of the point to its center) then this point is discarded and a new one is sampled, until a valid point is found. If the resulting point is in the negative Y direction, then it is mirrored to have a positive value in this axis. Finally the vector defined by the point is normalized, scale by the radius of the emitter and translated according to the hemisphere center.

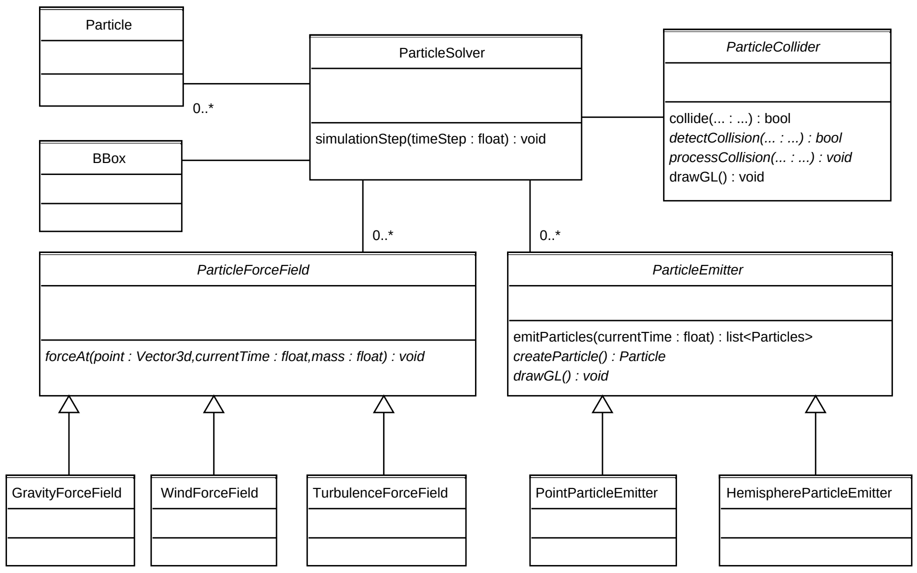

Particle
0..*
ParticleSolver
simulationStep(timeStep : float) : void
ParticleCollider
collide(... : ...) : bool
detectCollision(... : ...) : bool
processCollision(... : ...) : void
drawGL() : void
BBox
0..*
0..*
ParticleForceField
forceAt(point : Vector3d,currentTime : float,mass : float) : void
ParticleEmitter
emitParticles(currentTime : float) : list<Particles>
createParticle() : Particle
drawGL() : void
GravityForceField
WindForceField
TurbulenceForceField
PointParticleEmitter
HemisphereParticleEmitter

Figure 4.4: Particle System Class Diagram

ParticleForceField and its child are responsible for the generation a force on each particle during the simulation. The force is function of the particle's position and the current time of the simulation. Three force fields implementations are provided:

- *GravityForceField*: generates a vertical force with a constant acceleration magnitude provided by the user and with a direction along the negative Y;
- *WindForceField*: generates a force with a magnitude and direction provided by the user;
- *TurbulenceForceField*: generates a force using a Perlin Turbulence calculation (as shown in section 3.7 on page 18) using the X, Y, Z of the parti-

cle and the current time. The user can specify the frequency and the number of octaves of the turbulence. The function `latticeNoise(...)` returns the random numbers lattice value for the given position, `noise3d(...)` generates a Perlin noise octave value for each point in space and time and `turbulence(...)` mixes the values between octaves with the weighted sum, as explained before.

The core of the simulation is the class `ParticleSolver`, responsible for the Euler Integration and the coordination of the particle emitters, force fields and colliders. This class has the `simulationStep(float timeStep)` method that calculates one simulation step using the parameter `timestep` as the `deltaT`. One simulation step consists in the following set of actions:

1. For each Particle:
2. Accumulate the total force applied to the particle in that position by summing the forces produced by the force fields;
3. Invoke the `collide(...)` method from the `ParticleCollider` (which checks for collision between the particle and the surfaces adjusting its velocity/position if necessary);
4. For each Particle Emitter, invoke its `emitParticles(...)` method and appends the newly emitted particles to the particles list.

The Particle Solver stores the list of particles under its control in the `particles` list. An STL linked list was chosen to allow fast removals of particles that die during the simulation.

4.8.3 Particle Data

A single particle, implemented by the class `Particle`, contains the following data:

- Position in world-space;
- Position in world-space in the last simulation step;
- Velocity vector;

- Mass: used to calculate the acceleration created by the resulting force applied to it;
- Alive flag: boolean that specifies if the particle is dead or alive;
- Collisions Count: used to kill the particle after a certain number of collisions;
- Maximum Collisions: the maximum number of collisions until the particle gets killed;
- Last Impact Type: specifies if the particle bounced or slide in the last collision;
- Dirt: amount of dirt being transported by the particle (a floating point value);

4.8.4 Particle Collider and Output Dirt Mask Generation

The particle-surface collision detection is implemented by extending the *ParticleCollider* abstract class. The particle collider has the followings responsibilities:

- Check for collision events between particles and surfaces;
- Processing of the mechanical reaction of the particles, implemented in the *collide(...)* method (this method is not virtual because the mechanical reactions don't depend on the ;
- Processing of the chemical interaction between the particle and the surface calculating the dirt transferences in the collision point;
- Gathering of the generated dirt in the surface and generation of the output maps.

ParticleCollider is abstract, which allows several implementations of this important part of the system. One implementation was developed, the *ProjectedMapsCollider*, providing the projected maps (depth map, normal map, etc.) collision approach.

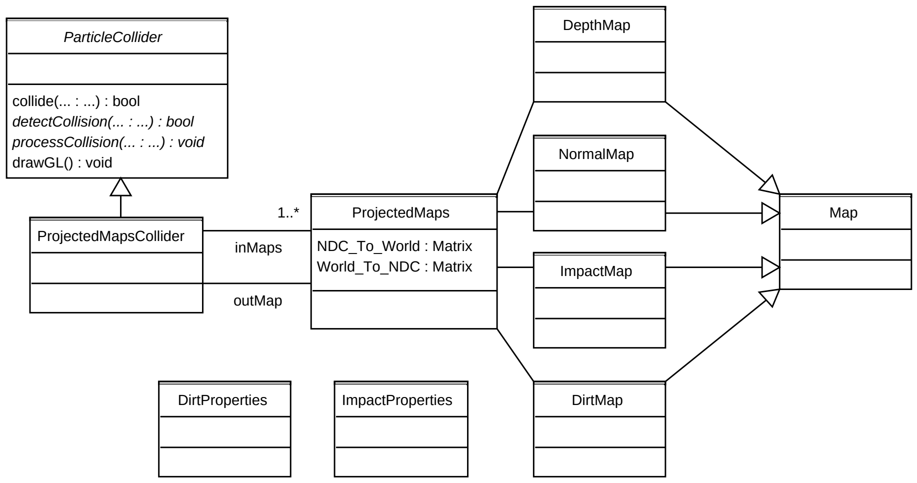


Figure 4.5: Particle Collider Class Diagram

4.8.4.1 Collision Detection

The collision detection is done by extending the virtual method *detectCollision(...)* in *ParticleCollider*. This method returns true when a collision is detected and also returns (using the arguments passed by reference) the position, surface normal, impact probabilities (used in the decision of which mechanical reaction will the particle have) and the dirt properties (used to calculate the dirt chemical transferences between the particle and the surface) of that point in the surface where the collision occurred. In the provided implementation (*ProjectedMapsCollider*) these values are read from the depth maps, normal maps, impact maps and dirt maps respectively. This type of collider uses several of these sets of maps (implemented by the class *ProjectedMaps*), one per orthogonal camera deployed in the scene by the user. This technique was inspired by the depth-map technique presented in [10].

One *ProjectedMaps* instance exists per input and output cameras. This class contains the four maps (depth, normal, impact and dirt), a world to NDC projection matrix, a NDC to world projection matrix and the file name and path for the output map (used only if this is used in an output camera). The matrices allow to project the positions of the particles into NDC coordinates in map space and vice-versa. A position in NDC space contains also the Z component, which corresponds to the depth value measured as a distance in world space from the camera to the point being projected. The collision detection between a particle and the *ProjectedMaps* is done using the following algorithm:

1. Project the Particle's position into NDC space (plus the Z depth);
2. Get the Depth Map's value in the resulting (X, Y) coordinate in NDC space;
3. Compare the point's Z depth with the Depth Map value. If the later is shorter(closer to the camera):
4. Return Collision True
5. Else:
6. Return Collision False;

This process verifies if the particle is behind the surface. If it is, then a collision is identified.

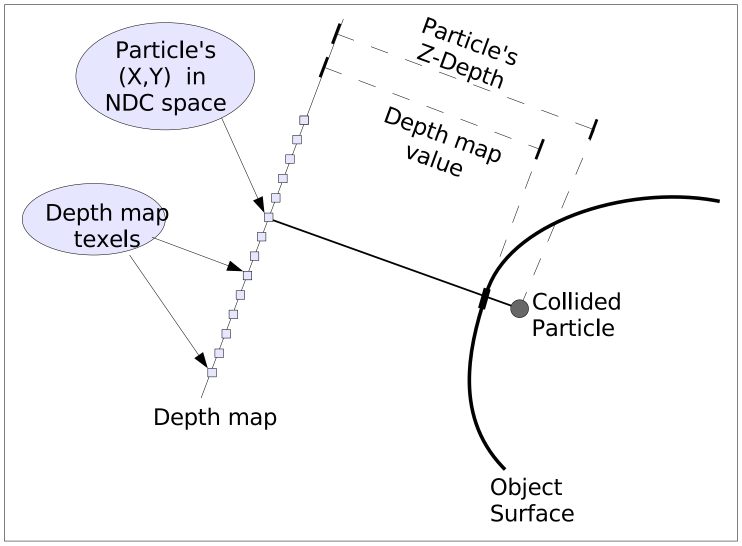


Figure 4.6: Collision Detection With One Depth Map

Using more than one input camera (i.e. *ProjectedMaps* instance) at the same time allows the user to cover a bigger area of the model's surface, but the collision detection algorithm must be adjusted:

1. For each camera:
2. Check if the particle collides with it. If there is at least one in which no collision occurs then:
3. Return Collision False;
4. Else:
5. Return Collision False in the camera with a smallest distance between the depth map value and the particle collision.

In some cases, a particle may be behind the surface seen from a certain camera, but there is no collision (the particle can be behind the all object, for example). In these cases, an extra point of view from another camera can detect that the particle is outside the boundaries of the object, this justifies why one camera detection of a non-collision is enough to infer a non collision at all.

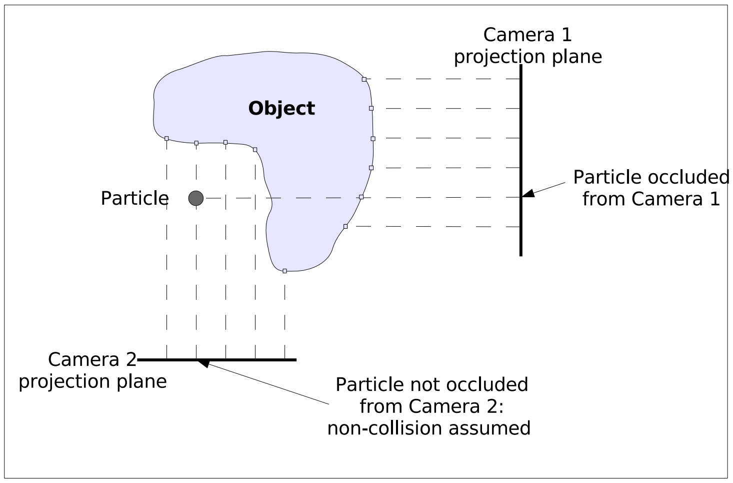


Figure 4.7: Collision Detection With Several Depth Maps

Choosing the camera with a smaller distance between the particle's position and the surface point (given by the depth map value) makes sure the known surface point closest to the particle is chosen as a hit point (Figure 4.8 on the next page).

Upon every collision an new entry of the *hitPoints* list (member of the collider that simply contains points in space) is created to allow the visualisation of all the occurred collisions.

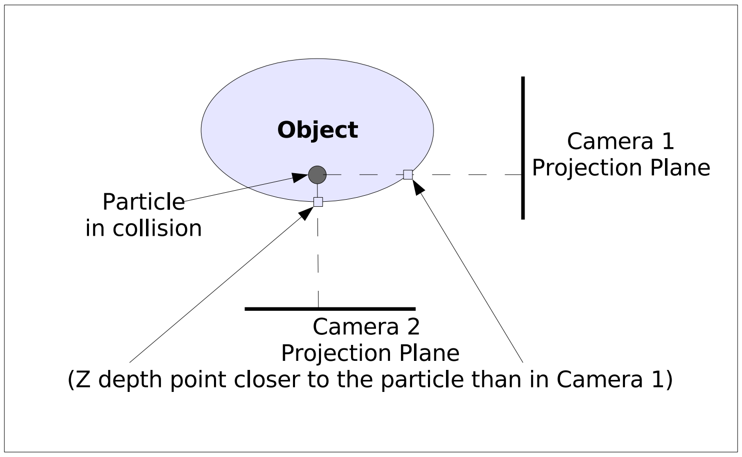


Figure 4.8: Collision Detection - Choosing The Closest Depth Map

4.8.4.2 Mechanical Reaction

After a collision, a particle will react in one of three possible ways: bounce, slide or be absorbed. This mechanical reaction is processed in the *ParticleCollider* abstract class in the *processCollision(...)* method . This is not a virtual method, like in the rest of the collision event implementation, because no matter how the collider works internally, the particle movement after the collision is always the same, depending only on on the normal and the impact probabilities on the collision point, which is provided by the *detectCollision*.

The impact reaction probabilities are implemented in the class *ImpactProperties*. This class contains the probabilities of the particle being absorbed, bounce or slide. These probabilities are stored in two floating point values:

- *absorption*: the probability of absorption by the surface (between 0 and 1);
- *bounceOrSlide*: if the particle is not absorbed then it will either bounce or slide, this value contains the probability of the bouncing event.

The *ProjectedMapsCollider* reads these values directly from the Impact Maps (red and green channels respectively). After getting the correspondent value for the detected collision, a Russian Roulette method is used to decide which event will occur:

Algorithm 2 Impact Type Russian Roulette

```
a = random number between 0 and 1;
if ( a <= absorption) then
   return Absorption;
endif
b = random number between 0 and 1;
if ( b <= bounceOrSlide ) then
   return Bounce;
else
   return Slide;
endif;
```

After calculating the event the particle will either be marked as dead (and consequently removed from the list of particles by *ParticleSolver*) if an absorption occurs or its velocity will change if it bounces or slides along the surface. The *processCollision(...)* method decomposes the velocity of the particle into normal and tangent in relation to the surface using two dot products. The normal component of the velocity is inverted, which corresponds to the mechanical reaction on a collision.

$v_t = v.T;$

$v_n = -v.N;$

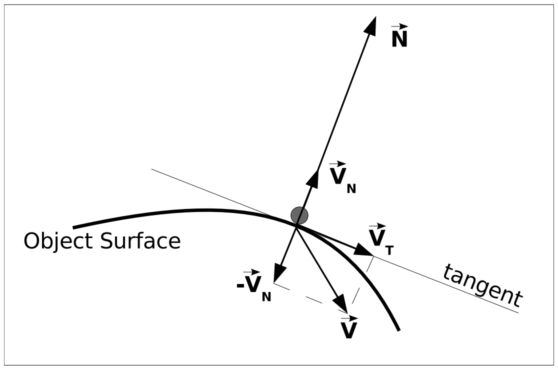


Figure 4.9: Decomposition of the Velocity Vector into Normal and Tangent Velocities

The normal vector is known and the tangent one is calculated using a double cross product:

$T = N * (v * N)$

in which:

N: the tangent vector;

T: the normal vector;

Then the resulting velocity is calculated by multiplying a resilience factor with the normal component and a friction factor to the tangent one. These two values are members of the ***ParticleCollider*** class. In the case of a bounce, the friction factor is overridden and forced to be 0 (eliminating the tangent component of the velocity). In the sliding the resilience is overridden and set to 0 (eliminating the normal component of the velocity). The resilience and friction factors specify the amount of absorbed energy after the collision in he normal and tangent components:

$v = v_n R + v_t F$

in which:

R: resilience factor

F: friction factor

An extra adherent force is applied to the particle if a slide occurs to keep the particles in contact with the surface, even if an external force pulls it away. This force is applied along the negative normal direction (pushing the particle against the surface) and its magnitude calculation uses an adherence factor (also member of the ***ParticleCollider*** class). In this case, an Euler Integration is also used to calculate the resulting velocity for this force:

$v = v - \frac{A\, N\, timestep}{mass_{particle}}$

in which:

A: adherence factor;

N: the normal;

4.8.4.3 Chemical Interaction

The implemented model of the chemical interaction between particle and surface upon a collision is not meant to be chemically nor physically correct. It simply allows the user to configure the amount of dirt being transmitted from the particle into the surface and vice-versa, according to the values present in the Dirt Map. In this map the RGB channels are used to store the following data:

- Red Channel: the amount of dirt in the surface;
- Green Channel: the percentage of dirt in the particle that must be transferred to the surface point where the collision occurred (deposit value);
- Blue Channel: the percentage of the dirt in the surface that must be transferred to the particle (erosion value).

After some tests, it was noticed that the particles flowing in the surface may not follow a completely continuous path due to the nature of their motion integration (Euler Integration) which, above a certain velocity, makes the particle jump several pixels. To avoid this the type and position of the last impact type of the particle is stored in the particle itself. If both the current and the previous impact type are of the type *Flow*, then the system assumes that the particle travelled on the surface linearly between the two points. I order to keep a certain visual continuity, Bresenham Line Drawing algorithm is used to draw the correspondent Dirt Map texels. This is implemented in the class ***DirtMap***, method ***addLine(...)*** (Figure 4.10 on the following page);

After a collision, the calculations of amount of dirt being transferred between the particle and the surface are calculated in the class ***DirtProperties***, method ***calculateDirt(...)***. The resulting particle dirt is written in the particle and the surface dirt in the red channel of the dirt map in the corresponding texel. The following formulas are used for this transfer during a collision:

$$dirt_{particle} = dirt_{particle}\,[1 - (D\,deposit_{surface})\,(1 + (E\,erosion_{surface})];$$

$$dirt_{surface} = dirt_{surface}\,[1 + (D\,deposit_{surface})\,(1 - (E\,erosion_{surface})];$$

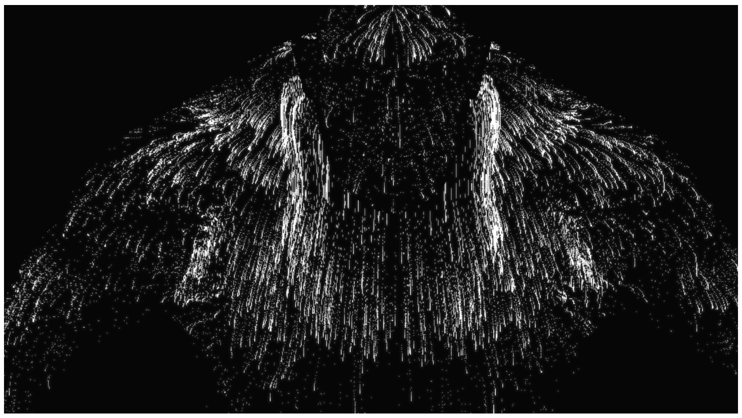
in which:

$dirt_{particle}$: dirt transported in the particle;

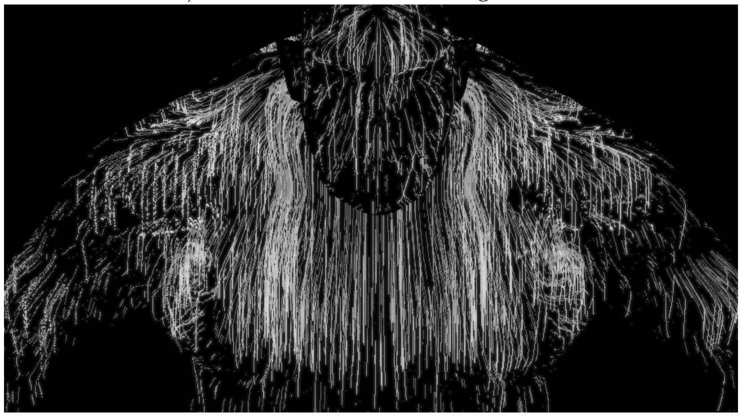
$dirt_{surface}$: dirt in the surface, stored in the Dirt Map;

D: a global Deposit multiplier defined in the collider class;

E: a global Erosion multiplier defined in the collider class;



a) Without Bresenham Algorithm



b) With Bresenham Algorithm

Figure 4.10: Cave Troll Dirt Map without and with Bresenham Line Algorithm (In the left image some particle paths are drawn with a sequence of dots, in the right one when a particle flows along a surface, a line connects two consecutive positions.)

$deposit_{surface}$: the deposit value read from the Dirt Map;

$erosion_{surface}$: the erosion value read from the Dirt Map;

This bi-directional dirt transfer allows not only the deposit of dirt in the surface, but also allows the particle to collect dirt in certain areas and transport it into others. The global erosion and deposit variables stored in the *`ParticleCollider`* class are used to scale all the erosion and deposit values from the Dirt Maps.

4.8.4.4 Output Textures Generation

The generation of the output texture with the dirt mask from the point of view of the output camera is implemented in the method *`saveDirt(...)`* of the class *`ProjectedMapsCollider`* and is done in two steps:

1. For each texel of the output texture, gather the correspondent dirt values from the Dirt Maps of the input cameras visible from the output camera;

2. Use an approach similar to the Photon Mapping's second pass to get a more blurred result. This is done because a particle collision occurs only on one texel of one of the input Dirt Maps. This results in a very high frequency result with isolated pixels, instead of a more scattered and blurred dirt. Using the information in the normal and depth map a blur along the surfaces can be achieved, which is better than simply do a 2D blur on the resulting texture.

In the first step, for each texel of the Output Map the corresponding depth value from the output camera Depth Map is get. The texel coordinates and this depth value are converted into world space and then converted into NDC space for each input dirt map, then the dirt values of all these input maps is summed and the value is written in the output map's current texel. With this operations, the dirt present on the input Dirt Maps is collected for each point on the surface visible through the output camera.

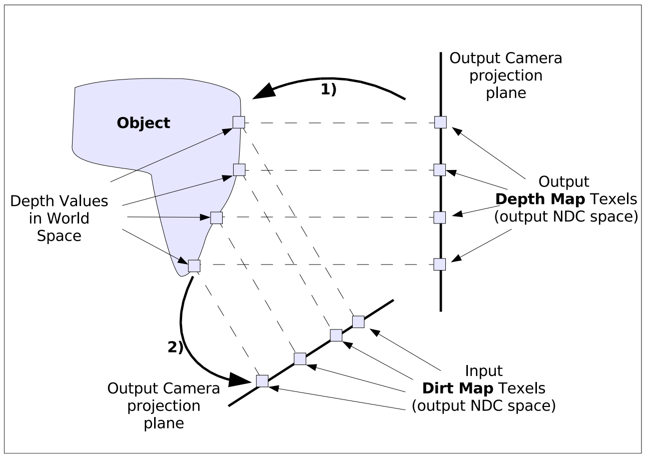


Figure 4.11: Gather of Dirt Values of All Input Cameras From The Ouput Camera Point of View
The ouput depth values are used to transform them into world space (1), and then from world space into the input dirt map space (2).

The result usually has a very high contrast and with high frequency noise, since each collision was recorded in the Dirt Maps in only one texel at a time (or with lines with a width of 1 texel).

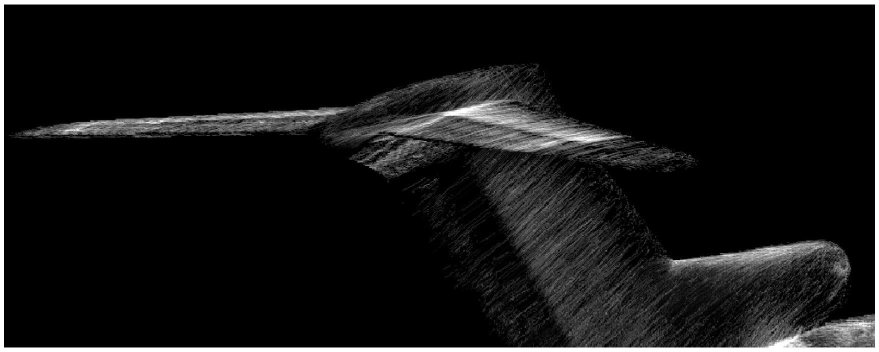


Figure 4.12: Output Dirt Mask Texture Without the Second Pass (Noisy)
(Airplane Model provided by Michael Garrett)

In order to solve this, an approach similar to the second pass of the Photon Mapping is performed. In Weathering System, this algorithm is adapted to its data structures and data values:

- Dirt values on the surfaces seen from the output camera are gathered instead of radiance.
- Instead of using a space search acceleration structure, like a KD-Tree, the search is done querying the output map's surrounding texels.

The texels of the output map are iterated another time. For each one of them, the dirt of the surrounding texels is summed. Using the directional and positional information on the output normal and depth maps, a gathering formula similar to the one of Photon Mapping can be used:

$$dirt_q = \sum_{p=1}^{n} \frac{dirt_p(N_q N_p)(1-r_{pq})}{r}$$

in which:

p: each texel surrounding the current texel with a distance $< r$;

q: the current texel;

$dirt_p$: the dirt amount in the texel p;

$dirt_q$: the dirt amount in the texel q;

N_p: the normal in the texel q;

N_q: the normal in the texel p;

r_{pq}: the distance between p and q;

r: the maximum distance allowed between p and q;

Comparin with the Photon Mapping formula, the $dirt_p$ is similar to the photon radiant power (ϕ) the dot product between the normals is similar to a lambertian BRDF in the and the $\frac{(1-r_{pq})}{r}$ is similar to a cone filter being applied to the result.

The results are much smoother, taking the directional and positional information into account, to avoid blurring certain points into areas where these points shouldn't have influence (Figures 4.12 on the previous page and 4.13 on the following page).

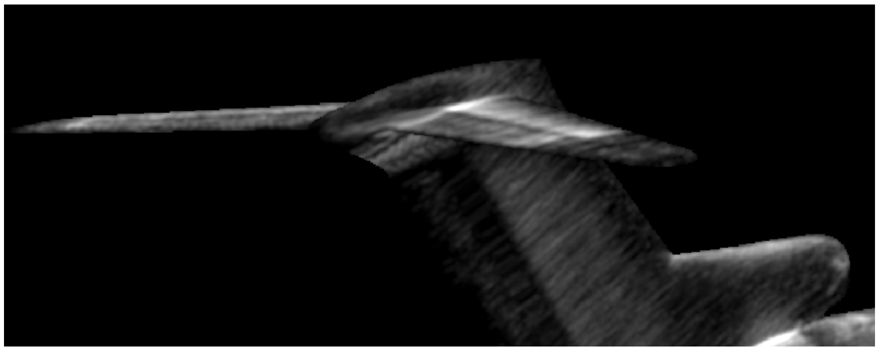


Figure 4.13: Same Output Texture, but with the Second Pass
(Airplane Model provided by Michael Garrett)

4.8.5 Particle System OpenGL visualisation

All the colliders and emitters have a *drawGL()* method in which some OpenGL code is used to show these objects in the viewport. In the only implemented collider, *ProjectedMapsCollider*, the input and output cameras are presented as rectangles in space, representing where the maps are projected from, and also some samples of the depth maps are shown in world space, which shows the shape of the objects of the scene. The point emitter renders a box with a large point inside and the hemisphere emitter uses two curved lines to represent the hemisphere itself. An example is shown in Figure 4.14 on the next page.

The particle system itself is also drawn using points for each particle. The hit-points list in the collider is also iterated and each point is drawn, which shows all the positions in space where collisions happened. This gives an idea of how the final output texture will look (Figure 4.15 on the following page).

This OpenGL visualisation code is used in both the standalone program (in a glut window) and in the Maya viewport (through the *draw(...)* method of the Locator Node class).

4.8.6 XML Scene Description Language Parsing

The XML scene description language was parsed using the open source C++ XML parser *TinyXML*, a simple XML parser developed by Lee Thomanon [14]. The language specification will be described later in this document.

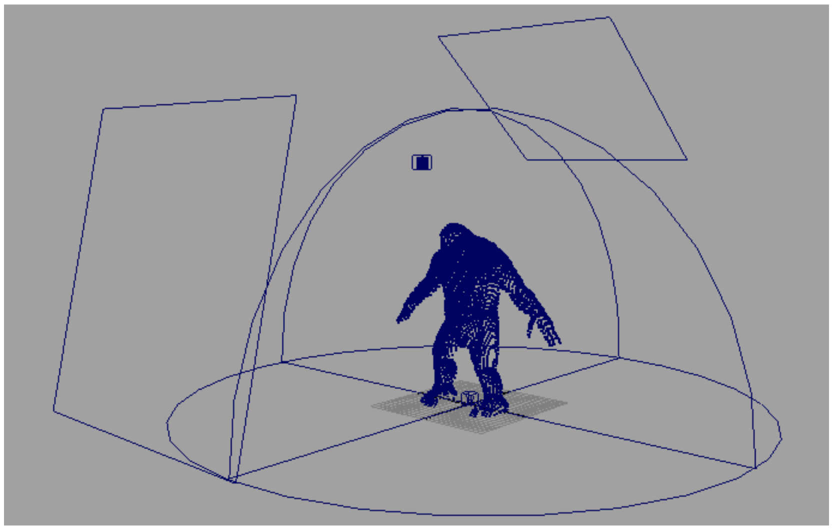


Figure 4.14: Collider and Emitters being drawn with OpenGL in Maya viewport In this scene, the collider draws the depth map samples of the Cave Troll and two cameras. A point emitter and an hemispherical emitter are also being drawn.

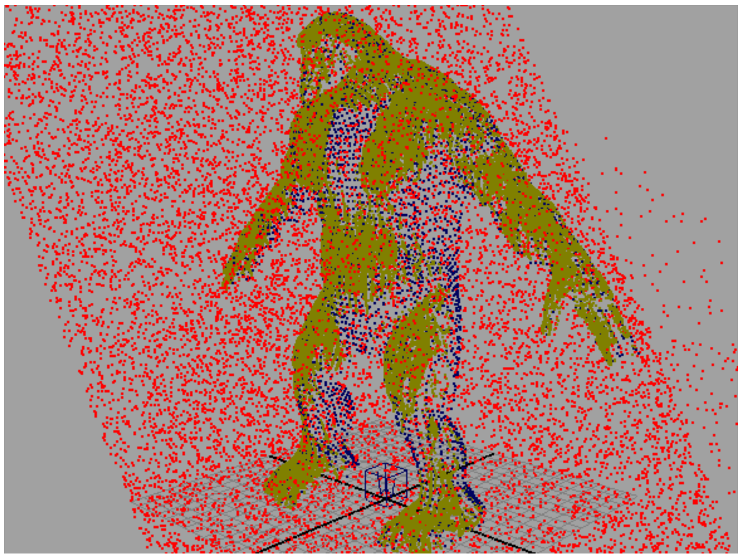


Figure 4.15: Particles and hit-points being drawn with OpenGL in Maya viewport

4.9 Maya Integration Implementation

4.9.1 Overview

The C++ classes described before in this chapter were totally reused in the Maya Integration. This integration consisted only in the implementation using Maya's API of a new node to contain these classes, plus one Locator Node for each particle emitter (to allow the user to place the emitter in the scene) and a set of Mel scripts that help the automation of the set-up process and to provide a user graphical interface integrated in Maya.

The Mel code is divided into three files:

- *`WeatheringUI.mel`*: contains all the Graphical User Interface code;
- *`WeatheringSceneSetup.mel`*: contains all the code that configures the Maya scene (shaders set-up; render layers set-up; main Weathering and particle emitters nodes creation and render properties set-up);
- *`WeatheringXMLScene.mel`*: contains the code responsible for the creation of XML Scene Description file with the information about the scene setup.

4.9.2 Input Maps Rendering in Mental Ray

The depth, normal, impact and dirt maps are rendered using Mental Ray. The Mel procedure *`weatheringSetupScene()`* sets up the scene for this render by performing the following tasks:

- Create the four shaders;
- Set the Renderable flag to true for the input and output orthogonal cameras;
- Create four render layers (to allow the rendering of the four maps per camera);
- Sets some Mental Ray properties (like switching the output format to 32bit precision floating point).

4.9.2.1 Depth Shader

This shader uses Maya's *SamplerInfoNode* to get the Z value in camera space and outputs it as the RGB of a Surface Shader

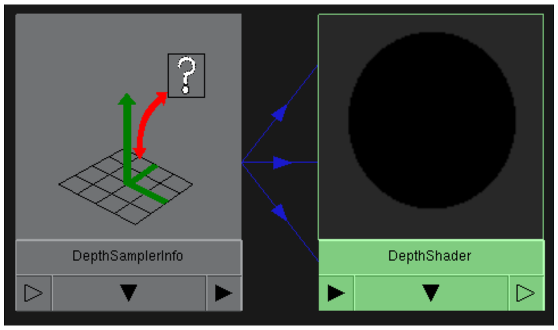


Figure 4.16: Depth Shader Setup

4.9.2.2 Normal Shader

This shader uses a small trick to render the normals in word space, it uses MentalRay's *Mib_amb_occlusion* (Ambient Occlusion) texture node with 0 samples and output mode 2 (render bent normals). Since there are no samples, the bent normals are the same as the original normals. Output 2 returns the normals in world space.

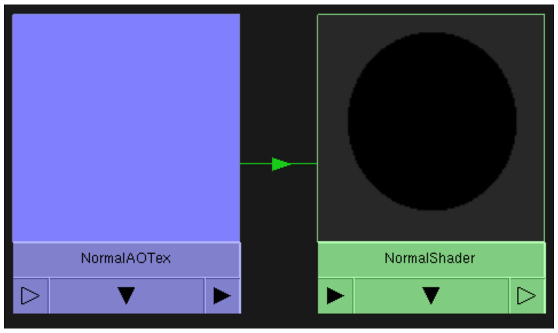


Figure 4.17: Normal Shader Setup

4.9.2.3 Impact and Dirt Maps

These shaders are similar. Both use MentalRay's *mentalrayVertexColor* node to output the vertex colors painted previously by the user. The node must be connected to each individual geometry object.

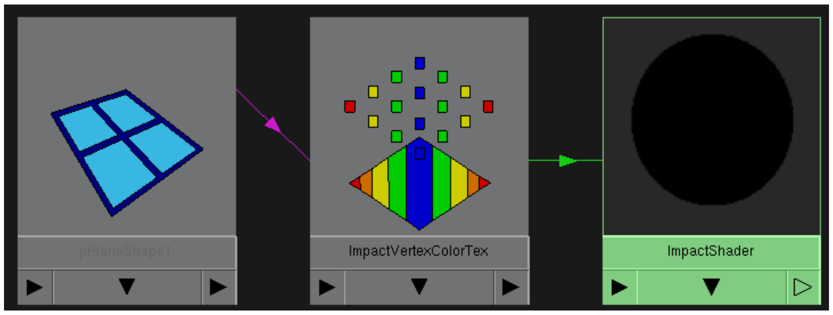


Figure 4.18: Impact Map Shader Setup (similar to the Dirt Map Shader)

4.9.3 WeatheringNode - Main Locator Node

The C++ classes that implement the Weathering System's core are packaged into the node called `WeatheringNode`, implemented in the class `MayaWeatheringNode`. This is an locator node, which provides the ability of drawing in Maya's OpenGL viewport. Internally it contains a reference to an instance of `ParticleSolver`, which gives it access to the particle system.

This node provides the following input attributes:

- Time: connected to the scene's time node. Whenever it changes (for example, when the play animation is pressed), the node asks internally to the Particle Solver to calculate one simulation step;
- SceneFile: a string with the XML scene file to be loaded;
- Reload: when set with 1 it loads the XML scene file and the input maps (depth, normal, impact and dirt); when set with 2 reloads the XML scene file, without reloading the input maps (useful to change only the particle emitter possitions, for example);
- Save: when set with 1 it generates the output dirt mask texture and save it in the file path specified in the XML scene file;

- Several values of the Force Fields, to be written in the XML file.

4.9.4 Particle Emitter Nodes

Two simple locator nodes were developed in order to allow the user to place the particle emitters in the scene:

- *MayaPointEmitterLocator*
- *MayaHemisphereEmitterLocator*

The user can translate them in the scene and set the emission parameters in the Attribute Editor.

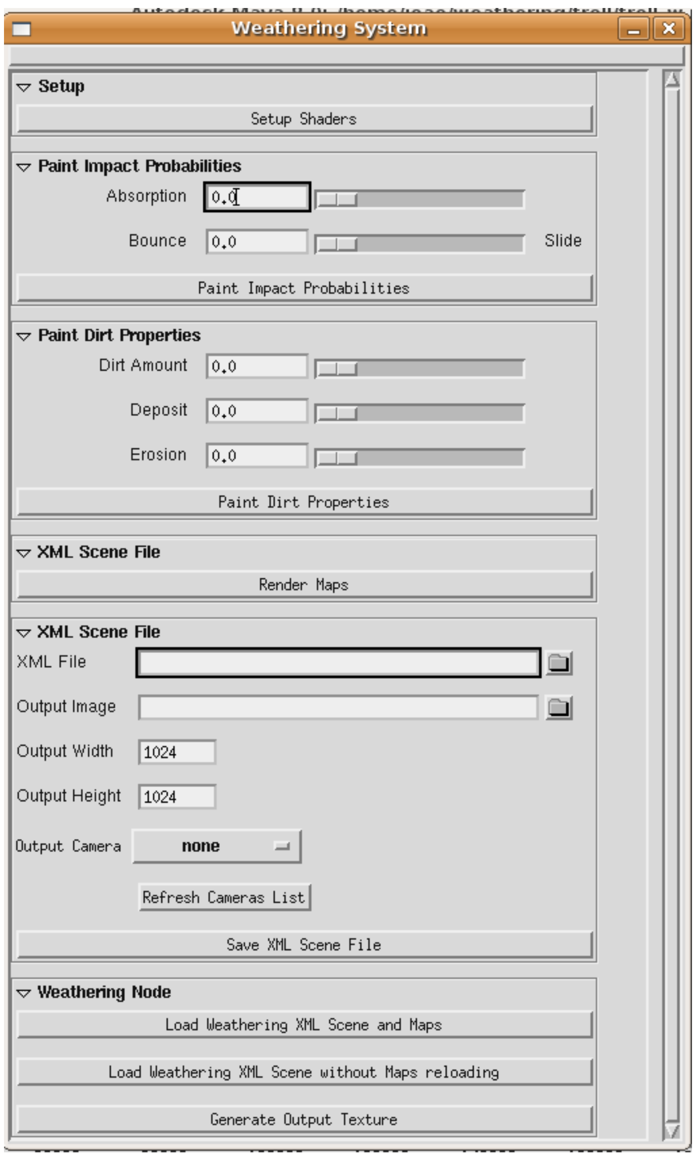
4.9.5 XML Scene Export

The procedure *weatheringCreateXML(...)* iterates through the selected nodes in the scene, searching for input and output cameras (being the output camera the last one being selected) and particle emitters to generate the XML scene description file and save it.

4.9.6 GUI Elements

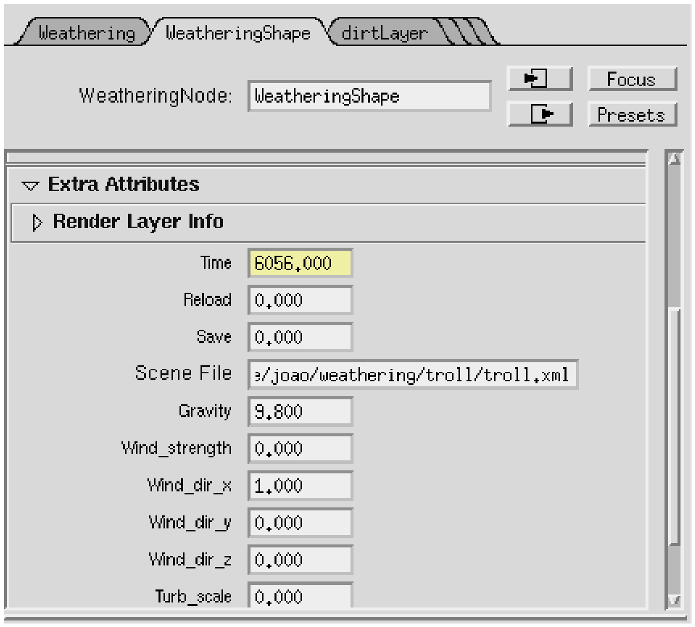
The main GUI interface is a window implemented in *WeatheringUI.mel*. This window allows the user to set-up the scene and generate the output dirt mask texture. Most functionalities present in the workflow (section 4.7.2 on page 27) can be accessed from this window and are organized vertically from top to bottom (Figure 4.19 on the following page).

Each Locator Node's attributes can be accessed in the Attribute Editor in the Extra Attributes tab (Figure 4.20 on page 52).



Absorption 0.0
Bounce 0.0
Dirt Amount 0.0
Deposit 0.0
Erosion 0.0
XML File
Output Image
Output Width 1024
Output Height 1024
Output Camera none

Figure 4.19: The main Maya Plugin Window



WeatheringNode: WeatheringShape

Time	6056.000
Reload	0.000
Save	0.000
Scene File	e/joao/weathering/troll/troll.xml
Gravity	9.800
Wind_strength	0.000
Wind_dir_x	1.000
Wind_dir_y	0.000
Wind_dir_z	0.000
Turb_scale	0.000

Figure 4.20: An example of an Attribute Editor (WeatheringNode's attributes)

4.10 XML Scene description Language Specification

The XML Scene description Language represents all the settings of the particle system as well as the information about the input cameras and their correspondent maps, output camera and output dirt texture. The XML code is delimited by the tag:

```
<Weathering>
...
</Weathering>
```

Inside this tag any of the sub-tags can appear once in any order.

4.10.1 Solver Tag

This tag configures the particle system solver's basic parameters (only implemented one is actually the bounding box of the system). Example:

```
<Solver>
<BoundingBox size="100" centerX="0" centerY="0" centerZ="0"/>
</Solver>
```

4.10.2 Emitters Tag

Inside this tag any number of any kind of particle emitter tag can appear. Example:

```
<Emitters>
  <Point start="0" end="10" rate="100" mass="1"
                dirt="0.3" initialVelocity="7"
                x="0" y="0" z="0"/>
  <Hemisphere start="0" end="10" rate="100" mass="1"
                dirt="0.3" initialVelocity="7"
                centerX="0" centerY="0" centerZ="0"
                radius="1"/>
</Emitters>
```

4.10.3 Collider Tag

This tag contains inside another tag with the information about the actual collider type. The only implemented collider is the Projected Maps Collider, hence the only available tag is ***ProjectedMaps***. Inside this last tag, any number (> 1) of cameras may exist. The only type of camera projection implemented is the orthogonal one, so the only available camera tag is: ***Ortho***. This tag contains sub tags for the world-to-NDC and NDC-to-world projection matrices and for the depth, normal, impact and dirt maps. If the camera is an output one, then the ***out="true"*** attribute must be used. In this case the ***OutputMap*** sub-tag must be present, containing the path of the output dirt texture and its dimensions in pixels. Example:

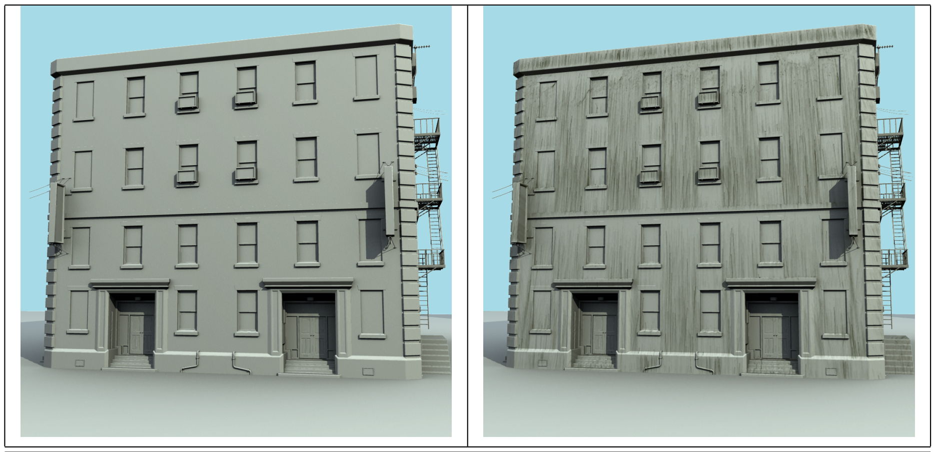
```
<ProjectedMaps>
  <Ortho cameraName="inCamera1">
    <CTWMatrix matrix="1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 "/>
    <WTCMatrix matrix="1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 "/>
    <DepthMap file="depth.exr"/>
    <NormalMap file="normal.exr"/>
    <ImpactMap file="impact.exr"/>
    <DirtMap file="dirt.exr"/>
  </Ortho>
  <Ortho cameraName="outCamera" out="true">
    <DepthMap file="depthOut.exr"/>
    <NormalMap file="normalOut.exr"/>
    <OutMap file="depthOut.exr" width="1024" height="1024"/>
  </Ortho>
</ProjectedMaps>
```

Chapter 5

Results And Future Work

5.1 Rendered Examples

5.1.1 Dirty Building: Weathering in Appearance



a) Without weathering | b) With weathering

Figure 5.1: Dirty Building: Weathering applied only to the colour

For this render, a point emitter was used with light turbulence and some wind pushing the particles against the front wall of the building. Two input cameras (one in the front and another in the back of the building) were used to capture the geometry. The output texture was camera-projected back into the model using a Maya planar projection with the same transformation of the output camera that generated the output texture. The mask was multiplied by a green colour and composited on top of the clean building image. All the input and output maps had a size of 2K (2048 x 2048 pixels).

5.1.2 Eroded and Rusty Cave Troll: Weathering in Displacement

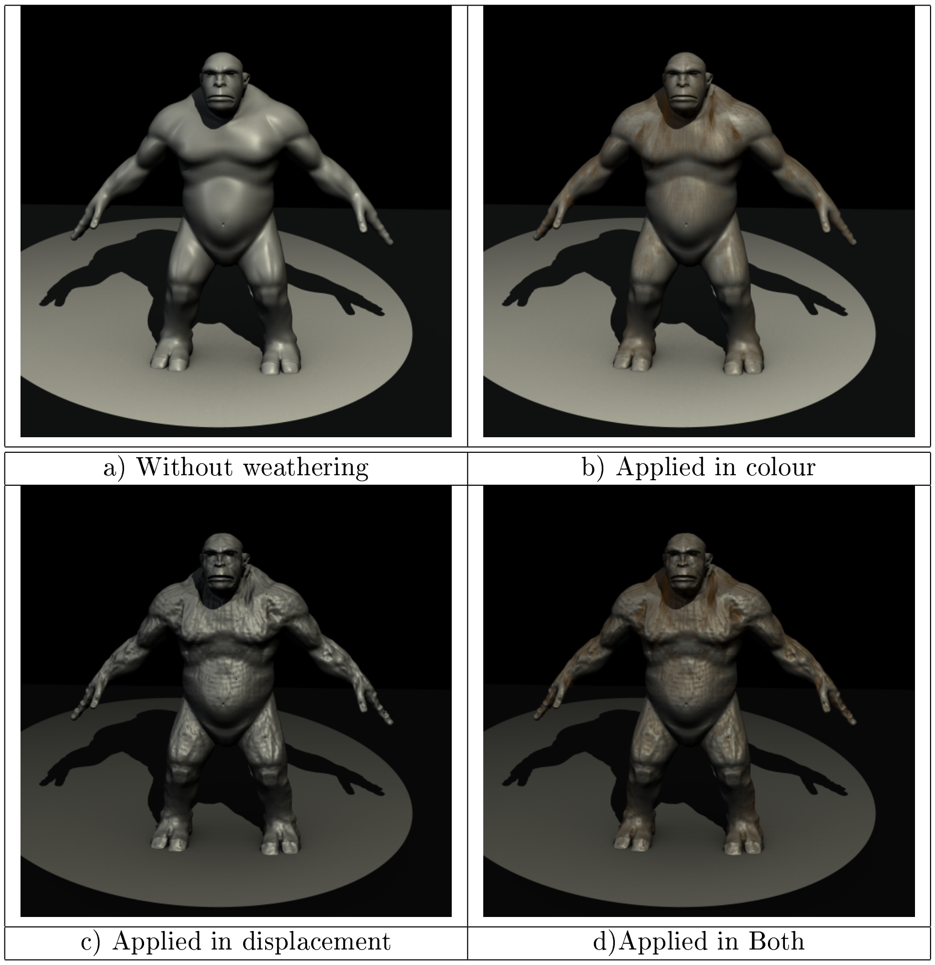

a) Without weathering
b) Applied in colour
c) Applied in displacement
d)Applied in Both

Figure 5.2: Eroded and Rusty Cave Troll: Weathering applied in displacement and colour

The set-up for this rendering was very similar to the previous one: one point emitter and two input cameras registering the model's geometry. The output texture was projected into the model as a displacement value and was also composited against the model's original colour. All the input and output maps had a size of 2K (2048 x 2048 pixels). The circular banding in troll's belly is due to the use of depth maps. These don't have a totally smooth representation of the geometry (its texels may generate some discontinuity in the surface).

5.2 Set-up Time

Each model was weathered in around 30 minutes. The rendering of all the input passes for all the cameras (during which the artist might have to wait without doing anything else) took around 8 minutes. Larger models (with more triangles) may take more time to render the normal and depth passes, but all the rest of the set-up process doesn't get much affected by the number of triangles, unless if more particle emitters must be deployed.

5.3 Critical Analysis on the Results

5.3.1 Visual Quality

Although it is difficult to imagine how the output masks look in a totally photo-realistic production, the results were interesting enough. Unfortunately it was not possible to have the the help from an experienced artist who could composite the output dirt masks in a more professional way. The expected effects, both in terms of appearance and geometry shape were achieved with satisfactory results, although some improvements could be done, as shown later in this chapter. The flow of particles on the surfaces seem convincing in most of the cases (for example, the lack of dirt bellow the windows due to the accumulation of dirt on the window geometry), but it also looks too streaky. The flow paths are very random instead of merging into bigger streams. This could be implemented using a more accurate flowing model, using capillarity of the water, that combines water drops together when they get close to each others. The erosion effect is interesting and can be used to generate rougher geometry.

5.3.2 Maya Plugin Usability

Although the user can have full control on every step, since most of the features use standard Maya nodes, setting up the scene can become a tedious task. Each time any change is done, the XML file must be saved and reloaded. For example, after changing the position of a point emitter, the XML file must be saved and reloaded in order to reflect that in the weathering node.

Another problem is when the input cameras change the maps must be re-rendered, so the camera positioning must be done with some care to avoid this.

The painting of the mechanical and chemical probabilities in the surfaces can be a tedious process. This can be overridden using textures applied to the model with the correct values in the RGB components (overriding the provided vertex colour shader). Another problem is that the painting colours can be visually difficult to distinguish when the probabilities of two close areas in the surfaces are not very different, which can be visually confusing for the user.

5.4 Possible Improvements

- In order to solve the XML reloading usage-overhead, this can be either automated (writing the XML file whenever a relevant change in the scene is done) or overridden (using more Maya Nodes Plugs to communicate faster between nodes, avoiding the need of using an XML file);
- The painting of probabilities interface can be improved using some other way of specifying the surfaces properties using a similar approach to [3], in which this properties are set using entities similar to shaders that can be applied to the geometry;
- The chemical model could be more chemically-based in order to create different results for patinas, moss, rust, etc. For this approaches similar to [6, 5] could be used;
- The sliding model could be improved, it has some imperfections due to the nature of the geometry used (depth maps) and to the simple integration used (Euler Integration);

- Depth Map filtering when reading the depth map values to represent smoother surfaces. This would avoid some banding problems, like the ones in the Cave Troll renders. This can also be solved using a higher resolution depth map;

- Camera projecting the result can have some problems. For example, some areas more perpendicular to the camera plane will be more stretched, and in models with lots of detailed geometry, there may exist the problem of self occlusion from the camera point of view. To workaround this, the used should always use camera projection as close as possible to the shot real camera, in which the occluded and perpendicular areas will be roughly the same. Another possible solution would be to use the UV layout of the models to write directly in the UV texture.

Chapter 6

Conclusions

Although some features could be improved (as seen in section 5 on page 55), the Weathering System produced satisfying results that can potentially be used by experienced compositors to produced interesting images. Some interesting weathering effects like humidity sliding down the walls and staining them, geometry changes by erosion, and rust/patina accumulation in eroded areas of the models are produced by the system.

The development of this Weathering System approached several areas of Computer Graphics (different geometry representations, particle systems, physical simulations, collision detection, rendering, texture synthesis, etc.) which allowed its author to consolidate a vast array of Computer Graphics Principles. The choice of implementing a non pipeline-dependent system core proved to be an interesting approach, since it allowed its development to be concentrated only on the referred principles on a first stage. The Maya integration, done in a second stage of the development, isolated the technology-specific problems from the first stage. It also proved to be a good proof of concept and an opportunity to learn and be comfortable with this specific software broadly used in the Industry.

Bibliography

[1] Paul Bourke. Perlin noise and turbulence - website, January 2000.

[2] J. E. Bresenham. Algorithm for computer control of a digital plotter. pages 1–6, 1998.

[3] Yanyun Chen, Lin Xia, Tien-Tsin Wong, Xin Tong, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Visual simulation of weathering by gammaton tracing. *ACM Trans. Graph.*, 24(3):1127–1133, 2005.

[4] Peter Comninos. *Mathematical and Computer Programming Techniques for Computer Graphics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[5] Julie Dorsey, Alan Edelman, Henrik Wann Jensen, Justin Legakis, and Hans Kohling Pedersen. Modeling and rendering of weathered stone. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 4, New York, NY, USA, 2006. ACM Press.

[6] Julie Dorsey, Hans Kohling Pederseny, and Pat Hanrahan. Flow and changes in appearance. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 3, New York, NY, USA, 2006. ACM Press.

[7] Francis J. Hill. *Computer Graphics Using OpenGL*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[8] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[9] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, July 2001.

[10] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131, New York, NY, USA, 2004. ACM Press.

[11] Wolfram MathWorld. Runge-kutta method.

[12] Ken Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press.

[13] W. T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[14] Lee Thomason. Tinyxml website, January 2000.

[15] Chris White. King kong: the building of 1933 new york city. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 96, New York, NY, USA, 2006. ACM Press.

[16] Wikipedia. Euler method.

[17] Wikipedia. Verlet integration.

[18] Andrew Witkin and Michael Kass. Reaction-diffusion textures. *SIGGRAPH Comput. Graph.*, 25(4):299–308, 1991.

Appendix A

Full Size Colour Rendered Images

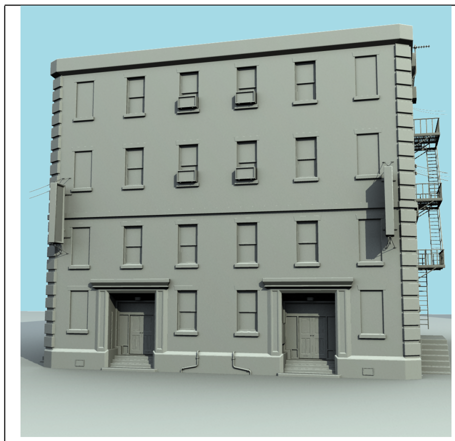
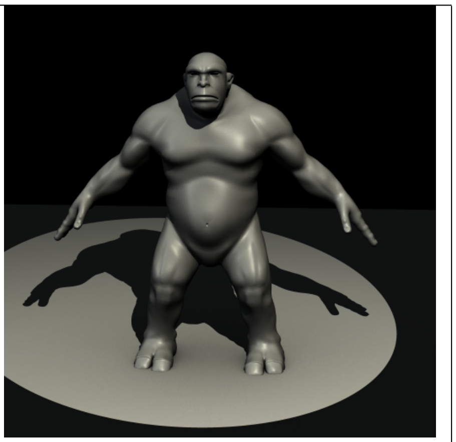
	
Building	Cave Troll

Figure A.1: Original Images

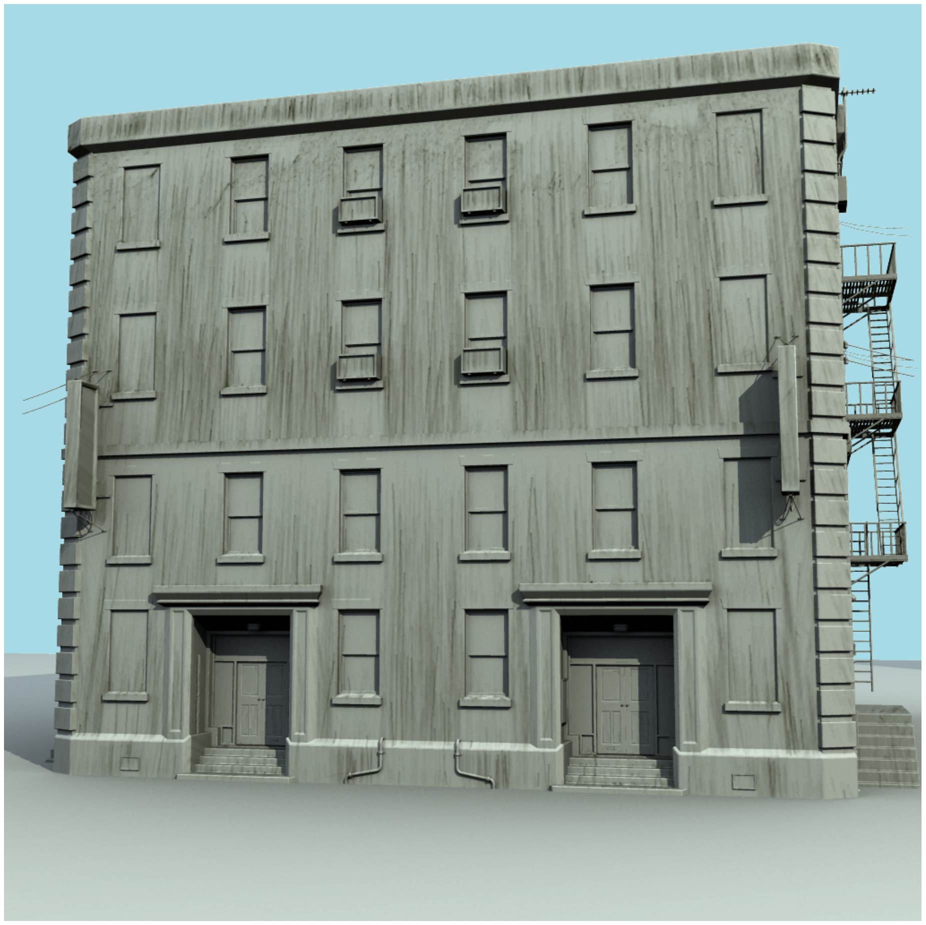


Figure A.2: Dirty Building

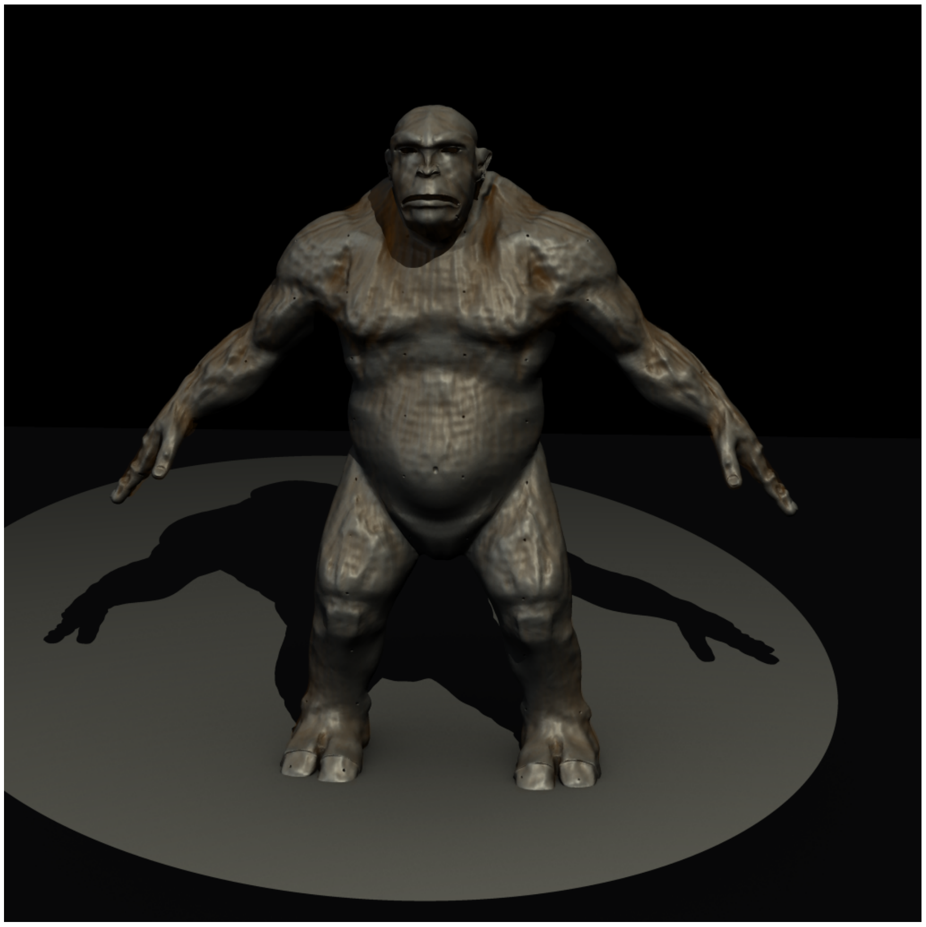


Figure A.3: Rusty and Eroded Cave Troll