Particle Based Weathering System Design

MSc CAVE
Animation Software Engineering
Bournemouth University
NCCA

Martin Davies

November 2014

Abstract

Image synthesising techniques tend to create very clean scenes. This is not reminiscent of reality where objects that are exposed to the environment change appearance and form over time. This is a process known as weathering. A desired outcome of computer graphics is to imitate what is seen in the real world. Therefore, creating weathered objects is a large but relatively unexplored research area. This project attempts to implement a well known solution to this problem using the notion of γ-tons. These are propagating particles that interact with the scene by depositing or transferring dirt and humidity attributes. The input of the system is an string, a file path to a .obj file. The output is a texture map that can be applied to the object. A preview window will also be available. In this document, the design of the system and the prototype code are presented.

Contents

Introduction

Objects exposed to the environment change in both appearance and form. This process is known as weathering. Weathering causes the properties of the object to alter. Some factors that can result in material alteration are: dirt, rusting (oxidation), water flow, lichen growth, fire and impacts etc. Photorealistic computer generated images attempt to fool audiences into believing that they are real. Weathering modelling is key to creating realistic scenes as we are exposed to weathered objects everyday. Therefore an audience will not be convinced if weathering is not added. Previously artists were subject to tedious texture creation[5]. This process was labour intensive and regularly resulted in textures breaking when applied to a model. An automated solution was required. Many researchers based their investigations around multilayering of textures and BRDFs [1][4]. Other research concentrated on automated solutions. Many used particle systems [2][5][6][13].

This project attempts to implement Chen et al's [2] solution for generating weathering effects. It will incorporated γ-tons to imitate the weather. γ-tons are very versatile, it is possible to imitate water droplets in the atmosphere, flowing water, scratches, impacts etc. by using them. Therefore, a system based on Chen et al's research would be very versatile but could also be very expensive. This is due to γ-ton processing, since a large amount of γ-tons will result in high computation time. A solution to this is parallelism using a GPU or, on a larger scale, a render farm[6].

The material being weathered has to be considered. For many materials, especially metals, the weathering process can be destructive [9]. Destructive corrosion exposes the base layer of metal. Therefore, corrosion is dependent on the amount of water that gains access. Other forms of corrosion exist, for instance; when copper is exposed to the weather it forms a patina [4]. Patina's are a protective layer of corroded metal; they do not allow water to access the base layer of metal. Therefore, the corrosion rate reduces after a patina is formed whereas corrosion rates increase in destructive forms. This project concentrates on the aesthetics of weathering, therefore accuracy to nature is not important but it is considered.

Introduction extracted from [3].

This document focuses on the design of a system similar to that of Chen et al [2]. A UML Class Diagram has been created to outline the classes that are required to construct the system. Proof of concept code has also been supplied with this document that demonstrates initial γ-ton behaviour. This is an initial design and therefore changes will be made as implementation occurs.

In [2] the authors refer to the particles that are used to produce a weathering effect as γ-tons. In this document they will be referred to as particles.

UML Class Diagram

2.1 Solver

Tracing particles is an iterative process [2][3][6] therefore we need a class to control this behaviour. Solver is the class that will manage this. It will have an operation that keeps track of the number of iterations the system has completed. The higher the amount of iterations the more severe the weathering will be, caused by larger amounts of particles in the simulation. Therefore, a higher amount of interaction between particles and the object occurs. Although it may be desirable to have severely weathered scenes the amount of iterations the user can perform should be restricted. By restricting, suitable effects will be achieved at acceptable computation times. It is necessary as each iteration will add particles and, therefore, will become more expensive to compute per iteration.

The compute time can be reduced by spreading particle processing over a number of cores. Open source systems, such as OpenMP[11] and OpenMPI[12], are available to integrate this functionality. This gives us the ability to expand computation over a number of cores in a CPU, GPU or even a high performance computer or render farm.

2.2 Object Loader

The Object Loader class has two constructors. The constructor with no parameters will load in a default object, for example a cube or sphere. The constructor with a parameter takes a string. This is a file path to a wavefront .obj file. The NGL Obj Loader will be used to load the .obj into the system. The filename is stored, in case we need to reload the object for any reason, as a string. It will be suitable to store this as a std::string as the whole string will be needed to reload. It would be pointless to store it as an array as access to individual parts of the string will never be required. Also the vertices and faces of the object are stored in a std::vector. These vectors can be accessed using the getVertices() and getFaces() operations. The Collider class will need this information when it checks for collisions. Therefore, a way to quickly gain access and share the data is required. Operations in C++ can return whole std::vectors of information, therefore, they give the

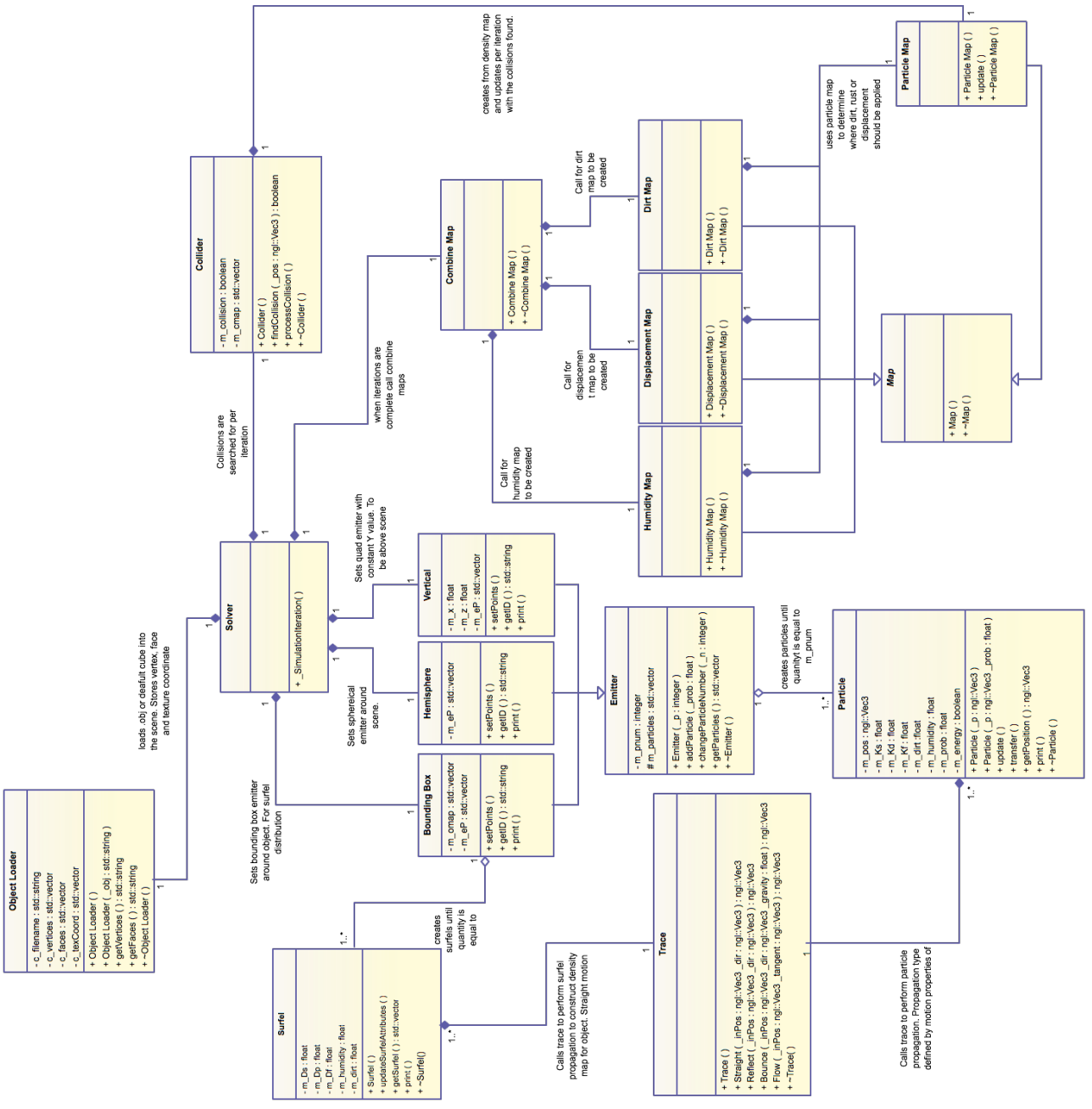

Figure 2.1: Class Diagram

functionality required. They were selected to store the vertex and face information for this reason.

2.3 Emitters

There are three particle sources described by Chen et al [2]. These are Point, Area and Environment. A point source exists on the object surface and slowly emits particles that travel along the surface. This creates trails of weathered effects, e.g. a trail of rust from a leaky pipe. Area sources are shapes, e.g. a rectangle, that has particles emitted from its area. These are used by Chen et al and in this system as a vertical emitter. Where particles are emitted downwards towards the object, similar to rainfall. Environment sources constrain particles within a volume. Chen et al uses a hemispherical environment source, as does this system. This system also implements a second environment source as the Bounding Box emitter, this is explained later.

There are three types of emitter in this system, Bounding Box, Hemisphere and Vertical. Each of them inherit from the Emitter class. The Emitter class constructor can take a value for the number of particles required, this is defaulted to 100, storing it in the parameter m_pnum. A pure emitter will never be required, but exactly one of each child emitter and a problem arises. A child emitter has a separate constructor and can not change the default 100 particles. To define the amount of particles the child emitter needs the operation, changeParticleNumber(), has been included to alter the amount.

The Emitter class could have been abstracted but as well as defining the location of the emitter the Emitter class also stores the particles assigned to the child emitter. The operations addParticles() and getParticles() have been included which allow the emitters to add an instance of the Particle class to a std::vector of Particles and to retrieve information about a particle respectively. As well as this control the Emitter class will also contain each of the particle information vectors, stored in a protected std::vector called m_particles.

The Bounding Box emitter will create a bounding cube around the object. It will then distribute surfels (explained later) over each inside face of the cube. This will be done by using the setPoints() operation. The setPoints() operation will use halton sequences[14] to gain an even distribution of surfels across each inside face. Halton sequences take a base prime and an range of numbers to split between. For a square in 3D three Halton Sequences are combined for the X,Y and Z coordinates of each point. The surfels will propagate towards the origin and stick to the object, when they collide. This creates a density map of the object. The surfels are able to do this by using the Trace class. Trace includes a number of interpolation techniques such as straight, reflection, bounce (reflection under gravity) and flowing (explained further later). These are defined by the motion probabilities of the particle [2][3]. When the density map has been created we can start to manipulate the

surfels by interacting them with particles.

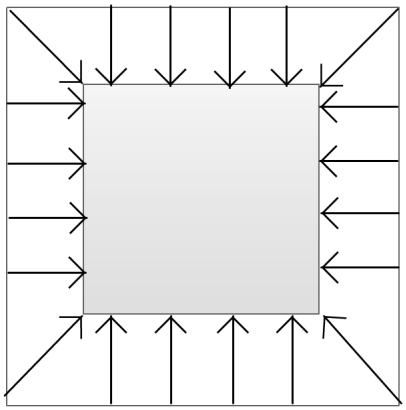

Figure 2.2: Bounding box emission.

The Vertical emitter creates a quad elevated in the Y axis. The Y is set to a default of 4 but this may need to be increased for certain objects. The parameters m_x and m_z define the width and depth of the quad. On the face we distribute the defined number of particles. Particles are distributed using the setPoints() operation. This sets the first four points as the vertices of the quad; the rest of the particles are distributed between the first four using halton sequences [14]. These particles travel vertically downwards until they either collide with an object or they fall out of scope. Particles that fall out of scope are killed and removed from the simulation. This is a process that is commonly used in systems that use particles [5][16][17][18]. A method coined by [16].

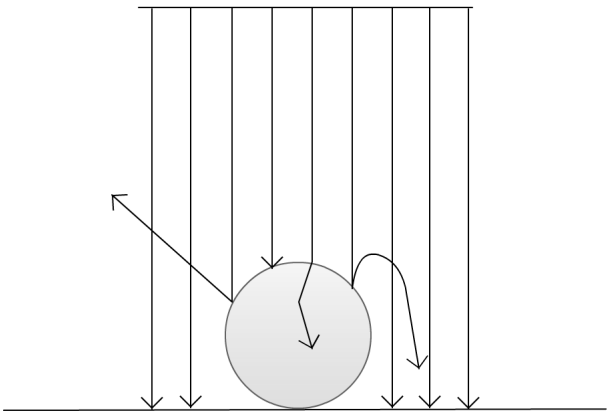

Figure 2.3: Vertical Emission.

The Hemisphere emitter creates a hemisphere of particles around the object. The hemi-

sphere is defined by using a constant radius, which is defaulted to 2. Again this may have to increase if the object exceeds the size of the hemisphere radius. Similarly to the Vertical emitter the particles are distributed using the setPoints() operation but behaves in a different way specific for a hemisphere. When the simulation begins the particles will travel from their origins, where the setPoints() operation set them, towards the origin. All of these particles will be expected to collide with the object therefore we will not need to kill any of them. After they interact with the object they may become troublesome. Therefore, a situation may arise where a particle, with a hemisphere origin, needs to be killed. This could be if it leaves the hemisphere, bounces underneath our scene etc. This can be resolved by either killing the particle and resetting it or by reflecting it back into the scene.

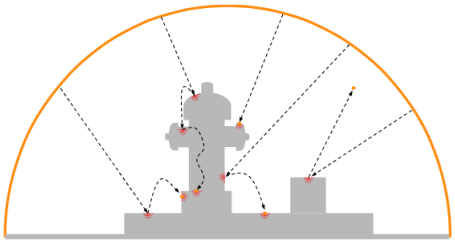

Figure 2.4: Hemispherical emission. Image is from [2]

The previous emitters have been selected because a hemisphere provides an easy way to contain all objects in the scene and the particles. A bounding box, using halton sequences[14], provides accurate way to create a density point cloud/map of the object. The vertical emitter is not really required but adds good diversity to the particle behaviours. A point source is not included. With time permitting one will be implemented. If this is not possible the code will be left in such a way that a point source can easily be added.

2.4 Surfels

A surfel stores all of the information about the surface properties of an object, these are represented by a primitive. Chen et al defined that a Surfel has two components [2]. These are the γ-reflectance and material properties. γ-reflectance are parameters that are used to deteriorate the motion probabilities of a particle. These are represented by m_Ds, m_Dp, m_Df in this system and correlate to the straight, parabolic (bounce) and flowing motions mentioned previously. The material properties refer to the material properties of the object at the point where the surfel exists. In this system these are represented m_humidity and m_dirt, which represent how much humidity and dirt is in the surface. The amount of humidity and dirt will alter when the surfel interacts with the particles this is known as γ-transport[2]. When collision occurs the updateSurfelAttributes() operation is performed. The surfel will also affect the motion. The values stored in m_Ds, m_Dp, m_Df can be used

to change the motion probabilities of a particle, this is explained later.

Surfels represent the object surface materials and density. They are extensively used when processing particle actions. When a collision occurs the nearest surfel needs to be found. To have the best search the surfels will be organised into a kd-tree[10].

2.5 Particles

Particles are required to enable the weathering process to take place. The vectors that contain particles are stored within the Emitter class, made up of instances of the Particle class. The Particle class is invoked when an emitter calls the addParticles() operation in the Emitter class. This operation creates a particle, setting the initial position based on the emitter type, and adds it to the vector. This process is repeated until we have added the same amount of particles as defined in the total particles required for the emitter.

Chen et al[2] defined their γ-tons similarly to their surfels. Their particles were made up of two components, carrier attributes and motion probabilities. Carrier attributes define the amount of weathering elements the particle carriers. There is dirt and humidity in this system. These are represented by the parameters m_dirt and m_humidity. Motion probabilities are probabilistic weights that define how a particle should propagate in the scene. These are defined as m_Ks, m_Kp, m_Kf. Again, these correlate to the straight, parabolic(bounce) and flowing motions implemented in the Trace class. In this system the particles are slightly more complex. Their attributes are:

- m_pos = The position of the particle.

- m_Ks = Straight motion of the particle.

- m_Kp = Parabolic motion of the particle.

- m_Kf = Flowing motion of the particle.

- m_dirt = Amount of dirt carried by the particle.

- m_humidity = Amount of humidity carried by the particle.

- m_prob = Probability that the particle will settle.

- m_energy = Lifetime of the particle. If energy = 0 the particle is killed.

When a particle collides with an object γ-reflectance occurs. By using Monte Carlo Russian Roulette [15] and the m_prob parameter a decision can be made on whether the particle settles or not. If the particle does not settle we can stochastically select a behaviour for the particle, between reflection, bouncing or flowing along the object surface. As well as behaviour we will need to have a method that performs γ-transport. This can be done by calling the transfer() operation when a collision is found by the Collider class.

A method to update particles as they interact with the scene is required. A number of things have to be checked, for instance where have they moved to?, have they collided?, what are their new properties?, what is their new direction of travel?, have they settled?, have they run out of energy or scope? The update() operation is used to check these. It allows us to update or remove particles from the simulation.

2.6 Trace

Each particle will perform different motion. They can all invoke the Trace class to interpolate them in the scene based on their motion probabilities. If the particle has straight motion the Straight() operation can be performed, usually performed at particle birth. This uses parametric lines to interpolate the particle using a parameter, *t*. This is performed using:

$$NewPosition = (CurrentPosition * (1 - t) + origin * t) \tag{2.1}$$

When a particle reflects straight motion is required but now there is no end point. When the Reflect() operation is called the following equation is performed:

$$NewMotion = CurrentPosition + (t * NewDirection) \tag{2.2}$$

This shoots the particle from its current position in the direction of "New Direction".

Parabolic motion (bouncing) is straight motion under the influence of gravity. Gravity will initially be a constant. The final system will allow the user to define gravity as $0 \leq$gravity≤ 1. Therefore, to perform parabolic motion straight motion can be used, with explicit manipulation of the Y value:

$$NewYPosition = CurrentYPosition * GravityStrength \tag{2.3}$$

$$CurrentPosition = [XPosition, NewYPosition, ZPosition] \tag{2.4}$$

$$NewPosition = (CurrentPosition * (1 - t) + origin * t) \tag{2.5}$$

Flowing motion is performed by using straight motion along the tangent of the surface at the point where the particle intersects.

2.7 Collider

Particles and surfels will interact with one another to allow the transfer of material properties. Therefore, a method which checks whether a particle has collided with the object is required. The Collider class is in place to check this. If a collision is found then the operation findCollision() will return true. This tells the Bounding Box emitter which particle has collided so it can work out which surfel is closest. When the closest surfel is found the properties of the particle and surfel are updated and we prepare for the next iteration. If more than one surfel is found the extra surfel(s) can affect the particle or ignore the particle.

The processCollision() class is used to keep the particle map up to date. It makes an update to the particle map at the end of every iteration.

2.8 Maps

The purpose of this system is to synthesise texture maps that when applied to an object produce a weathered effect. Therefore, a process of producing these maps is required. An abstract class Map that describes how a map is created is implemented. From this the classes Displacement Map, Humidity Map, Dirt Map and Particle Map all inherit. The Displacement Map, Humidity Map and Dirt Map are created when the Combine Maps class is constructed. The Particle Map is created from the density map created by surfels. This is a dynamic map as it will alter after each iteration. The Combine Maps class is constructed once the Solver class has completed all of its iterations. Once each of the maps have been created they will be combined and be applied to the object.

The Displacement Map class creates a displacement map from the particle map. The particles that have caused dents or destructive corrosion are found and copied to the displacement map. The Humidity Map performs the same process. Instead it looks for surfels and particles with high levels of humidity. These are copied from the particle map into the humidity map. The dirt map performs in exactly the same way. Instead of humidity it searchs for high dirt levels. Once the maps are created and combine they are applied to the object and viewed in an OpenGL preview.

Proof of Concept

Supplied with this document is prototype code. The prototype classes that have been developed are : Emitter, Hemisphere, Vertical, Particle, Iteration Control (Solver) and a template for the Object Loader and Bounding Box class. Some of the functionality that will be contained in single classes has been implemented in others. For example, the straight motion operation in the Trace class has been included in Emitter. This will be extracted and implemented into its own class in the next iteration of code development.

The Solver class is implemented. It contains a two loops. These control iterations for each emitter. Later a single instance of the Bounding Box emitter will be performed before the rest. Currently the prototype is set to perform a single iteration, for testing purposes. This will be increased in the final system.

The proof of concept creates a hemispherical and vertical emitter. Currently, they only have operations to return an ID in the form of a string. The functionality in the prototype is implemented in the Emitter class and the child classes, Hemisphere and Vertical, inherit all of their functionality from this. This is performed by calling the operation they want to perform whilst providing an ID. It is not a good design and the functionality will be extracted and implemented into individual classes at a later stage. When the emitters are created a loop is initialised that fills a std::vector with instances of the Particle class. The loop is engaged until the control variable is equal to the member variable m_pnum, which stores the amount of particles required. Each particle is placed at the vertices, if a hemispherical emitter is created. In the prototype they all propagate towards the origin, therefore the same path is used for each particle at a certain vertex. In the final system each particle will have different motion probabilities [2][3]. Therefore, it does not matter that some particles have the same origin. When the vertical emitter is constructed the same process of adding particles to a vector occurs. When the particles are assigned starting positions the first four are assigned to the vertices of the rectangle. The remaining particles are randomly distributed across the face of the rectangle by keeping their Y value constant and using a random number generator for their X and Z coordinates. The prototype uses ngl::Random to generated these numbers but does not give us even distribution. In the final system even distribution is required. To achieve this halton sequences will be used [14].

The Particle class is called by the Emitter class when an emitter calls the setPoints() operation. This creates an instance of a Particle where its start position is defined by the type of emitter and the rest of its attributes are defaulted to 0 or 1. The particles in the prototype are forced to travel in straight lines. Therefore, motion probabilities do not have an effect yet.

To draw the particles I have included shaders. These shaders were presented in a lecture on the OpenGL Shading Language (GLSL)[8]. The base code for NGLScene, OpenGLWindow and main classes were extracted from an NGL demo, BlankNGL[7], and modified to fit the scene.

Conclusion

Presented in this document is the design for a particle based weathering system. This system is inspired by [2]. Further investigation into related work and implementation methods for the final system can be found in [3].

This project has gathered a lot of inspiration from the automated system developed by Chen et al[2]. It attempts to implement these ideas and extend them. By using γ-ton tracing the user can produce aesthetically pleasing, although not accurate, weathering effects for their scene. Chen et al's [2] system has been groundbreaking by making the generation of effects such as "stain bleeding" trivial. The original system did not allow for interactive visual feedback from the simulation. This implementation will have a preview of the weathering effect developed using OpenGL. The preview will allow users to set some parameters, such as the amount of γ-tons to emit from each emitter, the strength of gravity, the levels of humidity and dirt per γ-ton etc. It will also attempt to gain a speed up in γ-ton tracing by using parallelism techniques. Lastly a point emitter will not be implemented but the source code will be easily extensible so it can be added in the future. Paragraph extracted from [3]

The design has been a success and all areas of the software have been considered. Construction of a UML class diagram and proof of concept code has provided the foundations of how a particle based weathering system could be implemented. The next phase is implementation of the full system. This phase will follow the UML class diagram as a blueprint of the software. The end result should be a working weathering system. Some issues might arise in the future which will require the class diagram to be revisited. An area that may be likely to encounter a situation like this is the map creation section. Since this area has not been implemented in the proof of concept code, there is less of an understanding about how it would work. Therefore, ideas displayed in the design may not be suitable for implementation and may require further research.

Bibliography

[1] Carles Bosch et al, "A Physically-Based Model for Rendering Realistic Scratches", *Computer Graphics Forum*, Volume 23 2004, 361-370.

[2] Yanyun Chen et al, "Visual Simulation of Weathering By γ-ton Tracing", *ACM Transactions on Graphics*, SIGGRAPH 2005, 1127-1133.

[3] Martin Davies, "Particle Based Weathering System", unpublished.

[4] Julie Dorsey and Pat Hanrahan, "Modeling and Rendering of Metallic Patinas", *In Proceedings SIGGRAPH '96*, SIGGRAPH 1996, 387-396.

[5] Julie Dorsey, Hans Køhlig Pedersen and Pat Hanrahan, "Flow and Changes in Appearance", *In Proceedings SIGGRAPH '96*, SIGGRAPH 1996, 411-420.

[6] Tobias Günther, Kai Rohmer and Thorsten Gorsch, "GPU-accelerated Interactive Material Aging", *In Proceedings VMV*, Vision, Modelling & Visualization 2012, 63-70.

[7] Jon Macey. (2014). *BlankNGL* [Online]. Available:https://github.com/NCCA/BlankNGL

[8] Jon Macey, "GLSL Lecture", By Communication. 2014.

[9] Stephane Merillou, Jean-Michel Dischler and Djamchid Ghazanfarpour, "Corrosion: Simulating and Rendering", *In Proceedings Graphics Interface '01*, GI 2001, 167-174.

[10] Andrew W. Moore. (1991). *An introductory tutorial on kd-trees* [Online]. Available : http://tinyurl.com/cja9o9

[11] OpenMP Architecture Review Board. *The OpenMP API Specification for Parallel Programming* [Online]. Available : http://openmp.org/wp/

[12] The OpenMPI Project (2004). *OpenMPI : Open Source High Performance Computing* [Online]. Available : http://www.open-mpi.org

[13] Eric Paquette, Pierre Poulin and George Drettakis, "Surface Aging by Impacts", *In Proceedings Graphics Interface*, GI 2001, 175-182.

[14] Dan Phaneuf. (2012). *Halton Sequences* [Online]. Available : http://tinyurl.com/njj46ns

[15] Magdi Ragheb. (2013, 17 March). *Russian Roulette and Particle Splitting* [Online]. Available:http://tinyurl.com/qde6t7r

[16] William T. Reeves, "Particle Systems - a Technique for Modeling a Class of Fuzzy Objects", *ACM Transactions on Graphics*, SIGGRAPH 1983, 91-108.

[17] Andrew Selle, Nick Rasmussen, Ronald Fediw, "A vortex particle method for smoke, water and explosions", *ACM Transactions on Graphics*, SIGGRAPH 2005, 910-914.

[18] Alexey Stomakhin et al, "A material point method for snow simulation", *ACM Transactions on Graphics*, SIGGRAPH 2013, Article 102, 1-10.

Appendix

A standalone class diagram image has been provided with this document.

To access the proof of concept code, initial design, full scale UML Class Diagram (PNG or PDF) and the CGI Tech document: https://github.com/NCCA/md349-ASECGITech