Real-Time Rendering and Simulation of Trees and Snow

Daniel Reynolds

A thesis submitted for the degree of
Doctor of Philosophy
at the University of East Anglia
September 2014

Real-Time Rendering and Simulation of Trees and Snow

Daniel Reynolds

Abstract

As virtual environments get increasingly more realistic, demand grows for elements found in the natural world to be simulated within virtual environments at increasingly higher quality. Natural systems such as trees and snow pose consistently difficult problems due to their complexity and dynamic nature. While methods used to generate both virtual trees and snow covered environments are progressing, many limitations exist which require improvement. This thesis develops techniques and advancements in the generation of trees suitable for real-time dynamic applications and the simulation of snowfall accumulation on dynamic scenes in real-time.

Tree models created by an industry used package are exported and the structure extracted in order to procedurally regenerate the geometric mesh, addressing the limitations of the application's standard output. The structure, once extracted, is used to fully generate a high quality skeleton for the tree, individually representing each section in every branch to give the greatest achievable level of freedom of deformation and animation. Around the generated skeleton, a new geometric mesh is wrapped using a single, continuous surface resulting in the removal of intersection based render artefacts. Surface smoothing and enhanced detail is added to the model dynamically using the GPU enhanced tessellation engine.

A real-time snow accumulation system is developed to generate snow cover on a dynamic, animated scene. Occlusion techniques are used to project snow accumulating faces and map exposed areas to applied accumulation maps in the form of dynamic textures. Accumulation maps are fixed to applied surfaces, allowing moving objects to maintain accumulated snow cover. Mesh generation is performed dynamically during the rendering pass using surface offsetting and tessellation to enhance required detail.

Acknowledgements

I would like to thank my supervisory team first and foremost. Prof. Andy Day, whose advice and encouragement has been crucial in conducting my research and forging my academic career. Dr. Stephen Laycock, whose technical expertise and enthusiasm has helped me from several tight corners. Their combined experience and guidance enabled me to produce a piece of work I can take pride in, and inspired me to continue a career in academia. I would also like to thank my examiners, Dr. Kurt Debattista and Dr. Barry Theobald for their insight, effort and assistance in improving my work.

I would like to thank my parents, Amanda and Colin Reynolds, for their support and encouragement which has spanned many more years than this Ph.D. And also my girlfriend Clare, whose unending patience and encouragement has made this process possible to complete.

Publications

Several parts of this thesis have been published at the following venues:

- Daniel T. Reynolds, Stephen D. Laycock, Andy M. Day, "Remodelling of Botanical Trees for Real-Time Simulation", Theory and Practice of Computer Graphics, Eurographics Association, pages 1-8, 2011
- Daniel T. Reynolds, Stephen D. Laycock, Andy M. Day, "Real-Time Accumulation of Occlusion-Based Snow", The Visual Computer, Springer Berlin Heidelberg, pages 1-12, 2014

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations

Virtual environments have always been an important part of modern graphics applications and as hardware capabilities are improving, the expectation of realism in applications is getting higher and higher.

One area which has been particularly problematic in real-time simulation is the rendering of realistic natural environments and environmental effects. Simulating natural structures such as trees are difficult, due mainly to the sheer complexity of the elements. In addition to the difficulties in rendering highly complex natural scenes, the generation of the models used in a procedural and realistic way is an entirely separate area of research, focusing on creating virtual geometry which adheres to the underlying botanical structure whilst allowing enough randomisation to populate an area with multiple instances which avoid repetition. Trees are an important natural element to any outdoor scene, adding believability to an environment if implemented well and introducing unrealistic distractions if created poorly. Trees are an integral part of a variety of 3D applications from planning and architectural visualisation to the extremely demanding field of video games and often applications, especially games, require an environment to be believable, fully interactive and rendered at appropriate high speeds to achieve real time. Real time frame rates, accepted as

25 FPS (frames per second) for pre-recorded video and largely considered to be an ideal 60 FPS for interactive environments, pose difficult limitations and demands on rendering techniques to produce high quality results efficiently enough to reach the desired low computing time. While static trees are a difficult issue in virtual graphics, dynamic vegetation simulating the effects of weather and environment is substantially more complex. To achieve realism in a natural virtual scene, the elements within must reflect the environment to avoid looking out of place and distracting from the immersion of the area, as such trees must have the ability to move with wind flow, be disrupted by heavy rain fall and accumulate snow as the scene requires. While popular tools exist for the easy procedural creation of plants and trees, limitations in the produced models make them inappropriate for high quality simulation forcing bespoke solutions to be re-engineered for each individual application. The realistic modelling and rendering of branching structures is crucial for tree rendering, integral to virtual environment simulations such as video games, film, planning and design visualisations and virtual reality. In addition to the modelling of trees, branching structures are used in mathematical simulations such as fractal geometry and L-Systems as well as medical simulations visualising vein and capillary structures which are used in virtual training applications and demanding high quality in addition to real-time visualisation. The main challenges of producing a high quality tree rendering system which are not sufficiently solved by current techniques are the modelling of branching structures as continuous surfaces to remove visual artefacts at areas of high detail. These integrate skeletal structure to allow deformation and animation and level of detail rendering implementations to allow real time rendering without sacrificing visual quality.

In addition to the insufficiency of procedural tree models, research is unfinished

in the areas of weather simulations capable of realistically producing a fully interactive, dynamic environment effecting the vegetation included. The main area being focussed on is snow accumulation as it has the largest visual effect on a scene, not only changing an object's appearance but also its shape and size by introducing new surfaces. One of the most noticeable effects of natural snowfall is the occlusion of surfaces, allowing areas to be sheltered from accumulating cover. This effect has been widely achieved by current rendering techniques similar to those used in the process of projecting shadows. The main difficulty in implementing snow build-up is the storage of accumulation on a surface. Implementations which allow snow cover to be accumulated are most often implemented in a manner which limits their use to completely static, un-movable scenes and techniques which cannot be achieved at a speed allowing real-time simulation. Real-time implementations of snow simulations conversely are unable to store accumulation and as such, are also limited to static scenes in which animation would lead to unrealistic discontinuity of the simulation. Significant challenges of a real-time snow simulation also include computation of the stability of snow and the support of the surfaces beneath cover and the realistic rendering of snow surfaces in real time using level of detail approaches. These are challenges which are not met by current snow simulation techniques and are required for any real-time snow simulation usable with a dynamic scene. As well as the simulation of snow cover, an implementation of surface-based accumulation can have several applications including the accumulation of mass on a supporting surface, saturation of liquid in a rain simulation or energy transfer from direct lighting for example. This can be applied to many environmental simulations which require real-time visualisation such as video games, film or virtual reality solutions.

This thesis aims to determine whether advancements in rendering technology and non-traditional applications of the programmable rendering pipeline can be used to

improve upon the current methods of visualising natural environments. To that end, a system of modelling, simulating and rendering realistic trees in real-time environments under the influence of snowfall is proposed using current rendering hardware and the programmable rendering pipeline. To achieve this, a technique of remodelling tree structures based on a given hierarchy is developed to address the main challenges of continuous geometry, sufficient skeletal structure and level of detail visualisation. In addition to this, a simulation of real time snow accumulation is proposes to record snow fall on supporting surfaces using the programmable rendering pipeline which allows persistent snow cover on a dynamic scene, snow stability and real-time level of detail rendering.

1.2 Thesis Outline

This thesis proposes techniques and advancements in procedural tree generation and rendering for real-time applications in addition to efficient and effective weather simulation techniques and the modelling of their effect on real-time trees. Chapter 2 details a background of related work and relevant solutions and techniques influencing the field and its progression. A new technique for the remodelling of trees suitable for real-time, high quality simulation based on the output of industry recognised procedural generation tools is proposed in Chapter 3. Chapter 4 proposes a novel method for modelling and accumulating snowfall on a dynamic scene with occlusion, allowing snow to be accumulated onto complex objects while allowing free movement and animation of the scene. Conclusions and future work are discussed in Chapter 5.

Chapter 2

Background

2.1 Modelling and Rendering of Trees

Botanical trees are highly complex organisms made up of a hierarchical structure of smaller elements in differing stages of development from multiple levels of fully developed branches to countless smaller shoots and buds progressing into complicated flowers and thousands of individual leaves. Modelling the geometry of a tree manually, keeping enough variation between trees to allow more than one to be used in the same simulation and making the structure botanically accurate enough for the vegetation to be realistic is an unreasonable task. This has led to the in-depth research carried out in the area of algorithmically generating botanical geometry, for which several techniques have been developed as surveyed by Sen and Day, [SD05], and explored in detail within the current section of this review.

2.1.1 Procedural Tree Modelling

Although plant structures are typically too complex to manually model in a reasonable way, there are cases where a single plant or tree is a main point of focus in a given simulation and more control over its form is required than can be given by purely procedural modelling techniques. To this end, work has been done producing solutions which take a semi-procedural approach to vegetation modelling in that a

Figure 2.1: Stages of tree creation using user-friendly generation package (images from [LD99]).

structured rule-set it used to generate the hierarchy of elements, within a user-friendly environment for manual creation. One such system is put forward by Deussen and Lintermann, [LD98], where a hierarchy of user controlled structures is used to generate a complex tree model. The system functions by giving the user a selection of useful elements such as branching shapes and leaf nodes, and allowing them to be consecutively stacked to form the basis of a botanical model. To each element modifiers are available to be set as well, governing the shape produced and the distribution of child elements, which with the addition of available world constraints such as gravity and light allow for simple generation of high quality single models. The system was substantially improved by Lintermann and Deussen, [LD99], by adding much more control over the individual elements. The addition of graphical tools to manually adjust the shape of each substructure and user-friendly tools to allow free-form deformation of the model provide the ability to create highly realistic simulations as shown in Figure 2.1. Although this solution shows techniques which can provide the easy manipulation of tools to generate singular virtual trees, the process is still far too time consuming and manually operated to give a reasonable solution to virtual environments where several trees, or perhaps an entire forest are required. To achieve larger scale simulations, procedures are required which are more automatic and require less user input.

A manual solution which aims to be more useful to artists using virtual trees is put forward by Prusinkiewicz et al., [PMKL01]. Rather than beginning with the lowest level elements and working outward, the proposal provides an interface which can be used to start with the overall form of a tree and work backwards. This makes the manual process easier to control and faster to produce usable results taking the assumption that the more important points from an artistic perspective are the overall posture and distribution of tree parts, rather than the lower level details. The system uses an L-System procedure, a technique which will be explored within the next section of this review, to generate the structure from the imposed appearance of the plant, allowing a highly detailed tree structure to be created and manually edited if required. This gives artists a tool to have greater control of the visually important aspects of the desired vegetation without requiring as much manual input at a lower level. One drawback of the proposal is that the input interface is somewhat removed from the artistic notion of the tree, providing a method of controlling the growth using curves and functions instead of geometry. Although this reverse technique is more efficient, focussing input more onto the important aspects, larger scenes still require much more automation and procedural methods of generation.

The techniques of procedurally modelling the geometry of a tree can understandably be split into two areas, modelling the structure and form of a tree and modelling the actual geometry created around that structure. Given the pre-generated structure of a maple tree, Bloomenthal, [Blo85], details a method of generating high quality geometry to provide a visually realistic rendering of the tree. A method is shown which maps a three-dimensional circle of points at intervals around the line of a given tree branch, which are then used as the basis to create the polygonal mesh of the limb. At points where limbs join, complex procedures are required and implemented to join the meshes forming a *ramiform* at the junctions without intersecting or overlapping. The

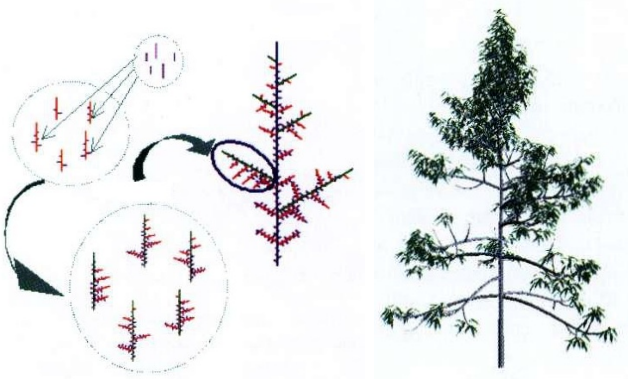
proposal models individual leaves as a simplification using only a few polygons which are then texture mapped to give the appearance of much higher detail, a technique which is also used on the barked areas of the plant with bump and normal mapping to give a rough, contoured look. The complications of procedurally modelling limb junctions is also tackled specifically by Lluch et al., [LVM04], where a pre-calculated structure using an L-System approach is taken as a skeleton upon which to generate the geometry in a single polygonal mesh. The idea of the research is to join different sections of the tree together using only one mesh without gaps. By using one continuous mesh, this ensures that there is no untidy overlapping or undefined intersections between primitives making up the tree. The proposal performs this by identifying the intersection points of elements and grafting the polygons together with a higher resolution triangle mesh, ensuring complete continuity.

Expanding on the previous examples, Weber and Penn, [WP95], detail a developed solution to automate the structure and geometry generation using a simple randomised approach. The main idea behind the approach is considering a tree as multiple recursive levels of branches terminated by one level of leaves and each element within a non-terminating level generates a collection of child elements created at randomly seeded points. The position of child branches can be restrained to generate tree models replicating a particular species and the system also allows branches to split into two of the same level instead of spawning the new branch as a child. Geometry is created using a standard primitive pattern and the thickness of all child branches is calculated as a function of the parent radius, with the exception of the main trunk, which is relative to the trees overall height. The simple approach is able to produce realistic models, as shown in Figure 2.2(a), however, the over simplification leads to complications when incorporating functions such as pruning. As the structure is randomly generated, an envelope cannot be made to influence the growth

and has to be used to trim any over-shooting branches and recalculate the children based on the new length, the drawback of which is that multiple passes are then required to modify the generation until no overlapping of the envelope is encountered. Kang et al., [KDRB+03], enhance a similar randomly based procedure by using a database of elements to greatly speed up the process of generating multiple trees. Each element of the tree, each branch, is treated as its own individual structure, and in addition to the main branch, each child element is a separate substructure. When a new substructure is created it is committed to a database of structures to be reused in future creation. When a database is available, each structure is then generated by stochastically sampling the database to produce a new random variation. Once the database grows in size, many different permutations are available to create a wide variety of tree forms whilst keeping within the set restrictions of the required tree species. Figure 2.2(b) shows a diagram of how the system is used to create a larger structure of smaller substructures and an example of a realistic model which can be produced using this method.

Pirk et al. propose a technique for the procedural modification of already generated trees given differing environmental conditions, [PSK+12]. The technique put forward takes a previously generated tree using common tree modelling software, assuming that the tree was grown in clear open space and recomputes the structure to simulate its shape if it were grown in proximity to an obstacle such as a solid wall. To produce this result, the input tree model is first analysed and reconstructed as a skeletal graph. By assessing the age and growth of the tree, the grown rate and age of individual branches can be estimated. Using the age of the branch, the system assesses the structure and light conditions for all stages of the growth, estimating the behaviour of the branch across its life. By assessing a branch's growth behaviour, the

(a) Weeping willow branch structure with and without pruning and fully generated model with leaves (images from [WP95]).



(b) Stochastic instance sampling and fully generated cherry tree (images from [KDRB+03]).

Figure 2.2: Procedurally generated and fully modelled trees.

influence of tropisms and light conditions on the growth of the branch can be estimated. Using these behavioural estimates, the proposed system can then remodel the tree in different scenes, modelling how the light distribution or proximity to obstacles would effect the final structure of the tree. The branching structure is transformed by its new environment, only transforming effected limbs making the process faster than Open L-Systems. Once a new branching structure is generated, it can be rendered in real time by creating the fine details such as twigs and leaves during render, using the GPU. While the proposed system demonstrates a solution to creating and altering tree models to fit around new scenes by interacting with obstacles and light conditions, it is limited to modelling light and tropisms and unable to model other influencing factors such as wind or varying nutrition. The system does not simulate the growth of new branches in differing conditions, it is limited to modelling the transformations necessary to branches already present in the skeletal structure of the input model. While the solution provides a simulated and procedural approach to adapting tree models to varied environments, it is limited to solitary tree models and still requires substantial user interaction in the form of inputting correct parameters for the growth simulation.

The other end of the spectrum within this area of research focusses solely on defining the structure of vegetation in a procedural manner without extensive consideration for the geometry to render it, a part of the research field which the remainder of this section will mainly be reviewing. The majority of investigations in this area approach defining the structure as a purely mathematical problem although biological aspects which govern how a plant grows must be considered for a truly accurate simulation. Several techniques of simulating branching are reviewed by Bell, [Bel86], with particular attention to the behaviour in modular organisms. Borchert and Honda, [BH84], detail a simulation of branching focussing largely on the number of branches created

in a given structure. The technique deals with symmetric and asymmetric branching and how development within a structure can be limited. In a different strategy to predicting the form of a plant at a given developmental point, Sims, [Sim91], looks at simulating the full growth by creating a system of artificial evolution. The solution can be used to generate not only three dimensional simulations of plant growth but also several types of image and texture information sets useful in many areas of computer graphics. The proposal uses genetic algorithms to simulate the development over time and gives the user a limited amount of control at a high level. Aside for individual unique approaches to tree generation as previously mentioned, there are three main areas of interest which constitute the majority of procedural vegetation research which are cellular automata, fractal geometry and widely considered the most appropriate simulation method, L-Systems.

L-Systems

L-Systems, or Lindenmayer Systems, were first put forward as a possible plant simulation method by Lindenmayer, [Lin68]. The original theory focused on the development of modules within a simulated structure and did not consider the geometry of a plant, as a development of Chomsky Grammars, [Cho56], which is a system of rewriting strings using a developed set of rules known as the productions rules. The main difference is that Chomsky Grammars work by applying the rules to each "word" in the string sequentially, whereas L-Systems are created to rewrite all of the strings in parallel. This makes it a better system of simulating the growth of organisms as each part of the system grows simultaneously in nature. Given the fractal nature of plants, such that often a branch of a tree may have the same structure of branching child limbs as its parent branch. Chomsky Grammars make a suitable simulation by the technique that a simple structure is built up by rewriting individual sections making them recursively more complex. A simple example of this technique performed

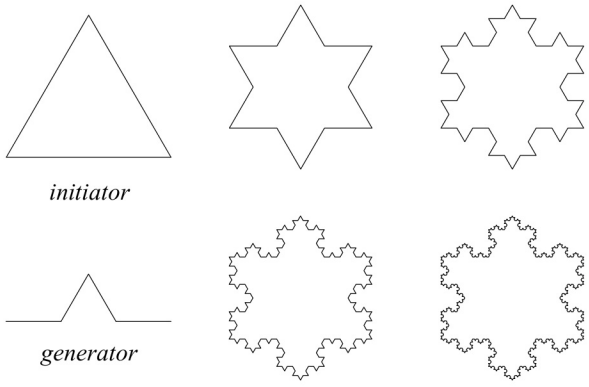


Figure 2.3: The *snowflake curve* (images from [PLH+90]).

graphically would be the *snowflake curve*, shown in Figure 2.3.

In order to generate effective graphical representations of branching structures in three-dimensions, the original L-System theory had to be expanded giving rise to several new variations. Prominent methods used in simulation include the most simple variety, DOL-Systems (deterministic context-free L-Systems), along with OL-Systems (context-free L-Systems) and parametric L-Systems. To generate a three-dimensional structure from these systems, turtle graphics can be used to interpret parts of the rewritten string as geometrical information. Much of this process and many examples of it are detailed by Prusinkiewicz and Lindenmayer, [PLH+90], and Prusinkiewicz, [Pru86], creating the structure of a plant around which geometry can be formed. The applications of plant modelling using L-Systems developed since the release of [PLH+90] are surveyed in depth by Prusinkiewicz et al., [PHHM97], with particular attention to the development of extensions of the L-System formalisation as well as new biological uses for the rewriting mechanism. Another survey of more recent developments is carried out by Prusinkiewicz, [Pru04], and several particular applications, relevant to the modelling of trees, will be discussed in the remainder of this section.

A method of describing and simulating plants using L-Systems is put forward by

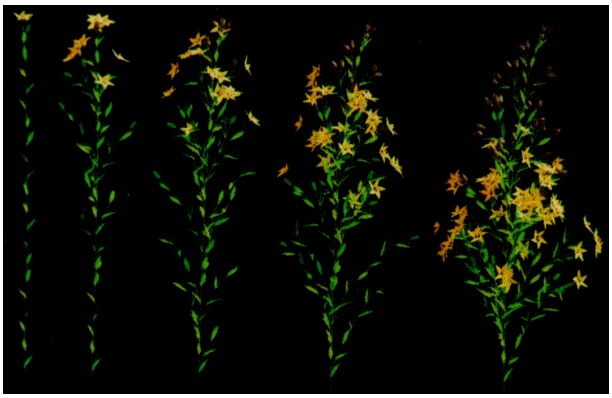


Figure 2.4: The development over time of a wall plant (image from [PLH88]).

Bell et al., [BRS79], which defines a plant by generating the accumulation and loss of individual elements over time such as individual buds and stems. The method strictly follows the botanically accurate function of plant sections rather than focussing on the aesthetic appearance of realism, although each element is also positioned in three-dimensions allowing a geometric representation to be generated along with the description of its development. Another approach which models the development of a plant over time in order to produce a realistic result, instead of generating an already aged plant by simply simulating its form, is considered by Prusinkiewicz et al., [PLH88]. L-Systems are used to map the development and growth of the organism and its botanical organs individually, incorporating the time relationship between parts in addition to their spatial positioning allowing several stages of development to be occurring at any one time, such as newly formed buds as well as flowers, as shown in Figure 2.4. One of the main issues arising from using L-Systems to simulate a continuous process in this way is that the formalisation is discrete, a constant simulation over time cannot be achieved, instead time-steps must be used to sample the development at set intervals.

Contrary to modelling plant structure with an emphasis on botanical accuracy, Costa and Landry, [DCL05], propose a technique to use L-Systems in conjunction

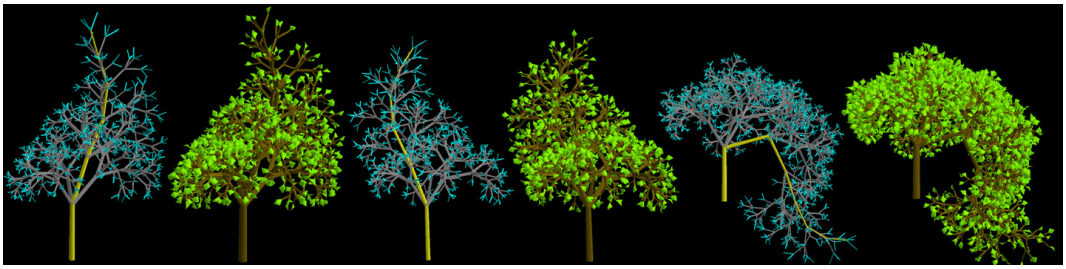
with genetic algorithms to generate structures which focus on geometric and visual accuracy. The paper details the method of using two-dimensional images as a reference to programmatically generate the growth model (L-System grammar) for any given species of plant or tree. A large selection of growth models are first produced by a deterministic generation algorithm and simulated to produce an extensive database of images for each resulting plant, from which each image is assessed algorithmically and evaluated using an appropriate fitness function. The growth models are sufficiently detailed, allowing for different methods of branching to occur throughout the tree structure and combine a formal definition of the plant as an L-System string as well as representing the organisms geometrically as a collection of specific shapes with an overall form. The fitness function is used to genetically evolve the algorithms to create a growth model which produces trees as similar as possible to the original input images. In tests evaluated on the procedure, the solution was found to always produce a model sufficiently similar to the original to be considered an appropriate simulation, however the simplistic grammars generated do not take into account several factors of the desired species such as the compactness or number of segments.

Although many examples show the power of L-Systems to procedurally model trees with great accuracy, they don't allow the artist much control over the form of the individual plants as can be said for proposals discussed in the previous section. User defined form is addressed to an extent by Prusinkiewicz et al., [PJM94], which proposes a method of allowing L-Systems to generate all plant structure but conforming to a geometrical envelope to simulate pruning the vegetation. This is based on the developed concept of environment sensitive L-Systems, a variation of parametric L-Systems where the parameters used are unknown at run-time, but calculated as needed using the position and surroundings of the turtle for example, in the case of turtle representation. Unlike simply clipping the branches of formed trees geometry

where intersected with its envelope, the system uses the clipping information to effect the further growth of the plant in an attempt to accurately model a plants response to pruning, which can be used to develop accurate simulations of reasonably complex environments as shown in Figure 2.5(a). Although an intuitive way to limit the growth of vegetation, Ijiri et al., [IOI06], detail a solution to allow the user, without knowledge of L-System procedure, to dictate the manner in which a tree grows at a low level. Once an L-System procedure for creating the vegetation growth has been detailed, which doesn't necessarily need to be created by the artist, a free-form stroke, can be drawn in three-dimensional space. The stroke is represented by a 3D poly-line and this dictates the overall growth pattern of the tree by using the specified axis as the central trunk around which the branches form, as shown in Figure 2.5(b). This gives the user a unique control over the growth of any particular plant, whilst keeping all the generation automatic to limit the required manual input to only the most important visual aspects.

Open L-Systems are used by Huang et al., [HJT+13], to procedurally generate bunches of grapes with a similar branching structure to trees. The proposed technique utilises Open L-Systems to simulate interactions within a bunch as the structure grows. Huang et al. have created formal rules governing the growth pattern of most types of grape bunch which accepts user input of shape using a polyhedron with 8 faces to define a required ideal shape. Once user input is taken, the proposal generates the parameters for an Open L-System which simulates branch interaction using the weight and volume of berries to pull the branching structure down while avoiding self-collision. While the process generates structures which fill all standard shapes of user input, simulating environmental interactions as well as grape thinning, the solution does not take into account factors such as sunlight or nutrition to simulate growth patterns.

(a) Plant response to user defined geometric pruning envelope (image from [PJM94]).



(b) Tree growth along user imputed axis (image from [IOI06]).

Figure 2.5: Using graphical user interaction to allow control of tree generation.

Although L-Systems have proven to be very successful in modelling static trees at distinct points throughout their development, the discrete nature of the procedure creates difficulties when attempting to animate a continuous process such as the growth of a plant. Prusinkiewicz et al., [PHM93], attempt to extend the L-System model to create dL-Systems in an effort to separate the discrete intervals required by the L-Systems from the growth model of the organism. Differential equations are used to model a continuous growth pattern, such as the growth of a limb, as opposed to sampling the growth at finite points, extending the standard L-System model to dL-Systems as a combination of continuous and discrete modelling. The benefit of this extension is that with a continuous time model, smooth animations can be produced at any speed required allowing for a wide range of applications. With the advantages of the approach, there are still many drawbacks of the proposal in the published state,

with dL-Systems unable to comprehend differential-algebraic equations and no built-in differential equation solver in the proposal, meaning that modelling cannot be done directly in terms of dL-Systems. There is no implementation of stochastic rules as with basic L-Systems and the solution produces somewhat unrealistic models with only the growth model implemented, not allowing for wilting or decay. As the individual elements are not environmentally-sensitive or aware of the development of surrounding limbs, there are cases of limbs geometrically intersecting each other. Noser et al., [NTT92], propose a system where L-Systems are used to animate the movement of a plant at a discrete time in its growth, so that the tree itself is generated in the usual manner and without introducing further growth, the structure's interaction with the environment is animated in the form of gravity and oscillating wind functions. Wind is the only elemental interaction so far considered and the approach does not simulate the effect of rain or snow for example, the animation itself is calculated using timed parametric context-free L-Systems (parametric tOL-Systems).

2.1.2 Tree Rendering

Although substantial research has been done on the generation and modelling of trees for virtual scenes, the rendering of tree models poses particular problems in real-time due to the complexity and detail of vegetation. Wang et al. proposed a method of realistic leaf rendering which focusses on the visual quality of global illumination effects on leaves, [WWDY06]. Leaf appearance in the model is described by a series of bidirectional reflectance distribution functions (BRDFs) and bidirectional transmittance distribution functions (BTDFs). The functions are obtained by examining the rough surface scattering and sub-surface scattering which gives leaves their visual appearance under global lighting conditions. Data from real leaf values are used to fit the model and create the BRDFs and BTDFs. The leaves themselves are modelled as

Figure 2.6: Rendering result of trees (images from [WWDY06]).

slabs with a rough surface and rendered using a two-pass algorithm which is derived from the Precomputed Radiance Transfer (PRT) approach. The model performs by assuming that the interior of a leaf does not need to be modelled in order to predict the appearance of lighting. During the first pass of the render, the indirect lighting component is computed along with environment lights using PRT. Once that is performed, the second pass uses the precomputed light visibility to calculate direct sunlight effecting the leafs appearance. The model, unlike PRT, is able to capture high frequency lighting effects such as soft shadows. LOD (Level of Detail) models are used for each leaf to speed up the rendering process and the BRDF and BTDF pairs are stored as RGBA textures for quick inclusion using the GPU. The system is capable of rendering leaves using environment maps or point lights as well as direct sunlight illumination and performs at frame-rates of 10 FPS when rendering a tree of over 500,000 vertices. While the model produces visually realistic results, as shown in Figure 2.6, it performs on only broad-leaf species and ignores small leaf details such as hairs.

Level of Detail Rendering

While the method put forward by Wang et al. produces visually realistic results, a frame-rate of 10 FPS while rendering a single tree is too slow for most real-time

applications. To deal with the natural complexity of models and rendering speeds necessary for interactive software, Level of Detail rendering is a commonly used technique. Level of Detail rendering works by substituting the full detail scene with simplified geometry when the detail is unnecessary due to viewer distance or occlusion. Gumbau et al. use a multiresolution scheme to produce camera dependant LOD tree rendering [GCRR11].Their method, which is designed for fast parallelisation on the GPU works by determining which areas of a tree are less visible dependant on camera position and renders those with simpler geometry. The technique consists of a pre-processing stage and a render stage, During the pre-process the mesh is divided spatially into oriented bounding boxes for easy grouping and visibility for each box is computed for a range of camera angles. Once the visibility for each cell has been calculated, the leaves within each cell are stochastically sorted. During the render, the visibility for each cell is determined using the camera position and the pre-computed visibility values, determining the LOD. Once the LOD is determined, a list of the sorted leaves within the cell is generated for rendering. The size and colour of the remaining geometry is then altered to preserve the general visual appearance of the model given that only a selection of the leaves are rendered. This LOD management system allows for the rendering of complex meshes at real-time frame-rates and is designed to be parallelisable using the GPU and CUDA to avoid costly data transfer between devices.

In the majority of cases, LOD rendering is based on simplifying complex geometry for faster rendering times without reducing the visual quality of the result, requiring that the model is only simplified when its visibility is impaired due to distance from viewer or occlusion. The previously described technique relies on selectively rendering only a portion of faces making up the whole model, while Livny et al. propose an LOD rendering technique which represents the tree canopy as textured planes, or “lobes”

which are used to reduce the complexity of tree models, [LPC+11]. The proposed solution takes a tree model as input and decodes the geometric information into a lobe representation for storage. The lobe representation comprises of branch and skeleton structure stored as spline curves stored with details required to regenerate the width of the limbs and lobes, which are groupings of areas of canopy which can be represented as single textures planes. This representation allows for compact storage and fast transmission of tree models by reproducing the detail of the tree at render. When the tree models are rendered, the splines denoting tree limbs are smoothed and tessellated dynamically, giving them three dimensional structure and textured surfaces. Tree foliage is synthesised by applying texture patches to the lobe geometry. Lobes are rendered as a collection of textures samples drawn from a species dependant library, allowing for the addition and removal of texture 'patches' to increase or decrease the LOD representation. Batches of leaf textures are arranged to fit the lobe geometry, allowing the tree canopy to be fully synthesised during render. This technique is used with tree data from laser-scanned models and has the advantage of fictionally synthesising foliage, making up for common gaps in scanned data. The technique does however rely heavily on the classification of trees and pre-generation of a library of species information and texture patches.

An example of the use of tree LOD models is put forward by Bao et al., [BLZD12], proposing a method of progressive client rendering of forest scenes. The proposal extracts LOD models from tree meshes using a new framework by representing leaf groups as textured quads, extracting the leaf model by defining a leaf vein quadratic interpolation function and using a leaf phyllotaxy based LOD model. The technique is used for rendering large forests on networked client applications, requiring that models be compressed as much as possible for transmission. The system first downloads a scene management file on the client application, followed by tree models ordered by

LOD, lowest first. Once the simplest LOD representation has been downloaded, rendering can begin on the client with higher resolution models replacing lower resolution models in the scene once they have been downloaded. Rendering is performed using multiple render passes, with the first render pass of the scene being used to determine required LOD representations and the second to render the tree models, instancing the LOD models and culling the unnecessary models entirely on the GPU. A LOD system is also used to generate shadow maps within the scene. LOD representations are essential in this type of application to allow for easy transmission of files across a network and allowing real-time rendering rates. The system achieves up to 33.9 FPS when rendering a scene of 7446 trees with over 1.5 million polygons within the view frustum.

One of the most useful recent advances in GPU capabilities is native dynamic tessellation, as used by Livny to render tree branches and detail. A new series of programmable shaders allow tessellation of a mesh and the addition of new geometry on the GPU during render, allowing view-dependant LOD rendering to be performed easily. The process of implementing dynamic tessellation using OpenGL 4.0 and above is detailed by Shreiner et al., [SSKLK13], and by Tatarchuk et al. for DirectX implementations, [TBB10]. The process of implementation within a LOD system requires the input of an over-simplified mesh which is dynamically detailed by the render pipeline, rather than beginning with a detailed mesh and simplifying it for lower LOD representations. Using OpenGL, the system adds two new shaders, the Tessellation Control shader and the Tessellation Evaluation shader. The Tessellation Control shader, processed directly after the Vertex shader, is used to determine the level of tessellation required by giving the user access to the data of all vertices within a given face. This data can be assessed and tessellation levels are provided individually for the subdivision of each edge and a level for the internal subdivision of the face.

Once Tessellation Control has been processed, the Tessellation Evaluation shader is processed on each new vertex individually and grants access to the vertex data and the vertex's position within the face, during this stage the final vertex position is specified incorporating any desired displacement or transformation. Using the same simplified vertex data, a mesh can be passed to the Fragment shader for rendering with much greater complexity and substantial added geometry, allowing greater detail to be rendered view-dependently.

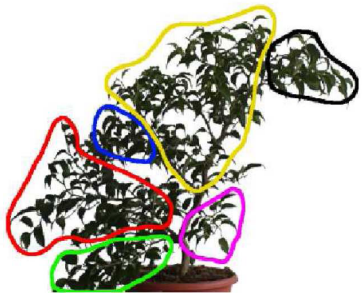
2.1.3 Tree Simulation

Wind Simulation

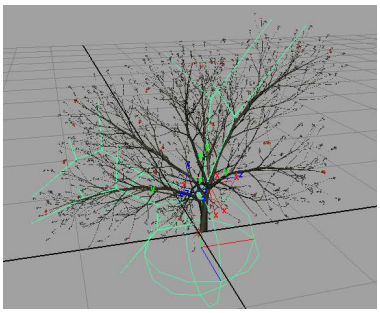
Whilst many techniques have been used to enable the high quality rendering of complex objects such as trees in real-time, to achieve the required realism in virtual simulations a technique of animating the vegetation is required as natural objects, such as trees are very rarely seen in a completely static state. The most noticeable natural movement of plants is their interaction with wind and airflow, which is the main focus of dynamically animating geometric representations and is an incredibly complex and complicated field of research. This section will cover current research in simulating trees interaction with the natural environment including the two major schools of study in the area: data-driven and full simulation.

The premise of the data-driven techniques is that the animation of elements is controlled using recorded, real-world data such as images or motion tracking. A popular form of this is video motion capture which has been applied to vegetation with varying degrees of success. Long et al., [LRBJ09], demonstrate the results of using traditional motion capture techniques to animate a small plant by placing around 100 reflective markers on the surface of the plants leaves and optically tracking their movement with a camera. The area taken by leaves and branches is calculated using

inverse volume rendering and all optical markers are grouped using a structure which replicates the hierarchy of the tree. Using the motion tracked by the camera, a system is developed to map the same motion to a virtual model of a similar plant, however the technique had serious limitations. Due to the manual input needed placing reflective markers and recording the outcome, the approach is not feasible for larger scale applications and allows for no variation in the results which could map more than one plant realistically without further capture. Another inherent drawback with optical motion capture is that movement is only detected in two dimensions, leaving the need to estimate any movement parallel to the camera view unless more than one camera is used. The need for manual input required in placing reflective markers is addressed by Diener et al., [DRF06], by using a film sequence of a plant moving in wind against a solid background in addition to film of the background alone and detecting the leaves from the film by comparing the individual pixels of each sequence and removing those deemed to be not showing the plant. Each leaf is detected and grouped with its surrounding leaves according to the branch hierarchy by using the movement and velocities of each leaf calculated by comparing consecutive frames of the video as shown in Figure 2.7(a). The branch structure can be defined from the groupings using the leaves as terminal nodes, allowing movement to be propagated down the hierarchy once it is recorded in the leaves. A degree of estimation is used to project the leaf nodes from their 2D recording into a 3D space, however once this is done the 3D information is automatically applied to all intermediate nodes in the structure as in Figure 2.7(b). The final step in the approach is to use the information as a virtual skeleton which can then be scaled to fit the model of a virtual tree and mapped to influence the position of relevant geometry, a process known as *skinning* and demonstrated in Figure 2.7(c). While this approach eliminates the need to manually place reflective materials, it is still unusable in an uncontrolled, outdoor



(a) Clustering of terminal nodes to represent underlying structure.



(b) Tree skeleton generated from detected terminal nodes in video capture.

(c) Virtual model (right) being controlled by a skeleton mapped to video movement (left).

Figure 2.7: (images from [DRF06]).

environment given its dependence on a constant background. Many of the other drawbacks still apply in that the information and motion recorded is only accurate in two dimensions and there is no facility to alter the direction or amplitude of the wind applied after the data has been recorded.

The main limitation of data-driven techniques for directly animating scenes is that without some manner of simulation, the forces and subjects cannot be altered or controlled dynamically without the addition of further data for the new scenario. Given that there would have to be a compromise between the two techniques, much research is carried out on simulating the process entirely using our knowledge of physics and mechanics to produce an approximation of the real effect. Much research in the field of biology is enhancing our knowledge of the behaviour and characteristics of plants in addition to data collections such as by Mayhead, [May73], which is directly

relevant to producing realistic simulations.

Weber proposed a system of simulating tree behaviour and deformation under wind and other external forces using a highly parallelized solution, [Web08]. The proposed technique models branch movement by dividing the simulation's dimensions and solving them separately in parallel. The system is modelled as a pair of 1 dimensional simulations in 2 dimensional space and by framing the axes of motion within the system, the axes are solvable separately. The technique is able to simulate force interactions in real time due to the parallelisation of the computations, allowing planar collision to be solved in two separate parts while kinematic calculations can be parallelised to a degree using the GPU. Hu et al. proposes a technique for modelling the movement of trees in wind using curved beam analysis, [HCH12]. The solution assesses individual branches and models three separate shapes of the branch under force, these shapes are blended between using a driven harmonic oscillator giving the vibrating motion of a branch under the influence of wind. The system can generate a motion similar to a Lissajous-curve vibration, incorporating natural oscillation frequencies and damping ratios, approximating forced vibration of the limbs. To enable real-time simulation the technique assumes that branches are isolated and that vibrations in the local coordinate system do not affect neighbouring limbs. The simulation is tested using wind force derived from a 3d $\frac{1}{f^\beta}$ noise and can produce simulations running at between 10 and 45 FPS without using pre-computation or GPU enhancements to calculation.

Some incredibly impressive results have been achieved by Habel et al., [HKW09], by using two-dimensional motion textures as a basis for tree movement. A large part of their technique is to calculate the force application by using the dynamics of a tapered beam, which more represents the form of actual branches than the assumption of a uniform beam which has been commonly used in the past. The

computation of the winds effect on the limbs is simplified by the assumption that adjoining branches do not influence each other as due to their size, the resonant frequencies of parent and child branches are completely different and much of the visual effect can be maintained by overlooking this transferral of force. Using recorded data, noise functions are generated which mimic the forces of the wind and these can be transformed into 2D motion textures, by giving all vertices an index into the texture, all animation and deformation can be performed on the vertex shader making the process highly parallelised. The motion textures are created one for each hierarchy of branches, as opposed for each branch and once the terminal nodes (leaves) have been animated, the motion can be propagated back down to the trunk. Although strong wind needs to show the effects of a visible direction, it was noted that due to the turbulence created by a tree, mild winds create an oscillation in the branches without a discernible direction. The benefits of this approach over motion capture is that all wind parameters can be altered dynamically and by using the vertex shader, the computational cost is minimal compared to the shading and rendering of the tree itself whilst providing a more realistic simulation than previous efforts by correctly accounting for the dynamics of tapering branches. Adversely, the transferral of forces between branches to model the full inertia present is implemented by Diener et al., [DRBR09]. In this solution a modal representation of the wind is used to break up the effect of the forces according to the frequencies and by assuming that the wind load across the whole tree is constant, the force simulation can be pre-computed independent of the complexity of the tree, although consistency is only adhered to for each tree and forces applied to separate trees are capable of variation. By removing time sensitive factors from the calculations it is possible to pre-compute a large amount of the interaction with simplifications made using the observation that low frequency modes effect the whole tree in the most visible manner whilst high

frequency oscillations only animate the smallest branches as shown in Figure 2.8. By removing the highest frequency modes of the wind forces much of the visual effect is retained allowing for effective level of detail control, however the animation remaining resembles less of the complexity of the natural effect it is visualising. With both of the discussed methods, strong winds as physically simulated have the adverse effect of stretching the branches unnaturally, which must then be corrected in a post-simulation step before rendering can be done. The results of the solution show that the controllable levels of detail in the wind function allows for efficient simulation of trees at any given distance from the viewer and the parallelization of node computation, which is independent, enables the animation of over 4000 trees at a minimum of 3FPS. The simulation does not take into account leaf drag due to its complexity as a mathematical problem, leaving the leaves to rigidly follow their parent branches. This is a disadvantage to the implementation and the unavoidable limitation of assuming that the wind load is constant across a while tree is that wind attenuation is not considered. This gives the unrealistic result of branches at the back of the tree, shielded from wind are applied the same force.

2.1.4 Summary

For the purposes of the development of real-time tree rendering solutions, within this thesis the requirements of an ideal tree model and rendering system are summarised in this section. An ideal tree model should have a full hierarchical structure reflecting the branching structure of natural trees and be modelled with continuous geometry to allow smooth, unbroken surfaces in areas of detail. The fine elements of a tree such as small branches should be modelled individually without the use of billboarding techniques to allow close and detailed viewing without visual artefacts. The model

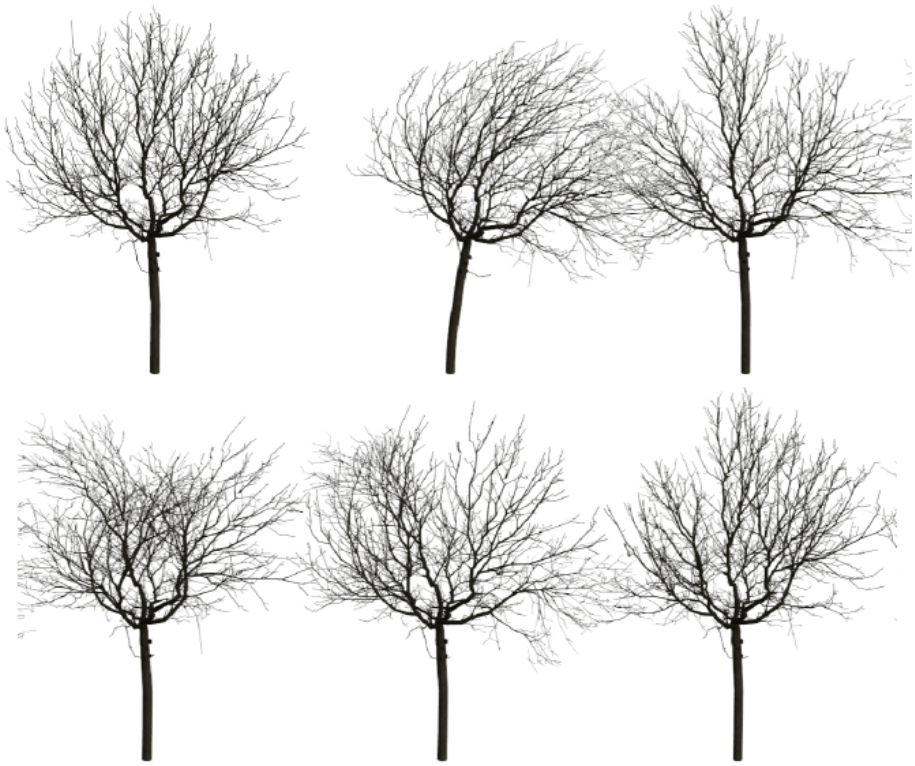


Figure 2.8: The effects of different modes of a modal wind representation on a tree structure (images from [DRBR09]).

should be deformable in some manner, allowing interaction with a dynamic environment through animation or simulation. The focus of this thesis is on real-time techniques, giving the requirement of interactive rendering speeds and a level of detail rendering approach to allow scalability within a scene. These requirements are summarised in Table 2.1.4, with the current techniques covered in this chapter which produce three-dimensional tree models or rendering systems.

	Hierarchical Structure	Continuous Geometry	Modelled Fine Elements	Level Of Detail Rendering	Real-Time Frame-rates	Deformable Model
Lintermann and Deussen (1999), [LD99]	✓		✓			
Prusinkiewicz et al. (2001), [PMKL01]	✓		✓			
Bloomenthal (1985), [Blo85]	✓	✓	✓			
Lluch et al. (2004), [LVM04]	✓	✓	✓			
Weber and Penn (1995), [WP95]	✓		✓	✓		
Kang et al. (2003), [KDRB+03]	✓		✓			
Pirk et al. (2012), [PSK+12]	✓					✓
L-Systems						
Prusinkiewicz et al. (1988), [PLH88]	✓		✓			
Prusinkiewicz et al. (1994), [PJM94]	✓		✓			
Ijiri et al. (2006), [IOI06]	✓		✓			
Huang et al. (2013), [HJT+13]	✓		✓			
Prusinkiewicz et al. (1993), [PHM93]	✓		✓			
Noser et al. (1992), [NTT92]	✓		✓			✓
Tree Rendering						
Wang et al. (2006), [WWDY06]	✓	✓	✓		✓	
Gumbau et al. (2011), [GCRR11]	✓			✓	✓	
Livny et al. (2011), [LPC+11]	✓			✓	✓	
Bao et al. (2012), [BLZD12]	✓			✓	✓	
Tree Simulation						
Long et al. (2009), [LRBJ09]	✓		✓			✓
Diener et al. (2006), [DRF06]	✓		✓			✓
Weber (2008), [Web08]	✓		✓			✓
Hu et al. (2012), [HCH12]	✓		✓		✓	✓
Habel et al. (2009), [HKW09]	✓		✓		✓	✓
Diener et al. (2009), [DRBR09]	✓		✓		✓	✓

Table 2.1: Summary of current techniques covered in Chapter 2.1 and their application of identified key requirements. Work is listed in the order it appears within the chapter.

2.2 Simulating and Rendering of Snow

Arguably one of the most visually striking and inspiring environmental conditions seen in natural scenes is one of heavy snow. Snow cover has the ability to turn a scene into a sweeping white landscape, drastically altering the form and behaviour of all structures beneath it. Thick, heavy snow accumulating on trees not only alters the vegetation's appearance, covering it in a complex surface of ice, but the weight of the build-up and connective forces exerted on separate elements joined by a shared snow surface completely alters the movement and behaviour of all branches and leaves. While several techniques for the accumulation of virtual snow have used static trees as an example for detailing the effectiveness of algorithms, little work has explored the simulation of vegetation or dynamic objects directly influenced in their behaviour or movement by the accumulated snow. This section outlines the current techniques and research for the rendering and simulation of snow cover on a virtual scene, as an essential pre-requisite to the effective simulation of snow covered trees and vegetation.

2.2.1 Snow Rendering

When dealing with the rendering and simulation of snow accumulation within a scene, the most fundamental aspect of the implementation is the rendering of the snow itself. In 1997 an early technique put forward for the rendering process of a snow surface by Nishita et. al. was a solution using metaballs, [NIDN97]. Rather than being represented as triangular geometry, the metaball technique describes a surface as a series of volumes with a density distribution, called isosurfaces. Rendering is carried out using ray tracing and casting rays from the view position into a metaball to determine the colour of the light received. Snow is layered onto each surface manually from a top down perspective, with the centre of each metaball lying on the surface and layered with more than one level of metaball if necessary. To achieve

a smooth surface appearance, metaballs are placed closely packed together, while a sparser placement creates a rougher snow cover. To render the metaball surface, snow space is divided into voxels for rays to be cast through, calculating energy and viewable colour. The sample space is prepared by collecting voxels which contribute to the current voxel. Scattered light calculations are made more efficient by using the sample space as a reference pattern. Each metaball contains within it sub-balls and prisms as well, to simulate snow crystals and give more realistic specular highlights and mirroring. However, while an early and effective solution to the snow rendering problem, the result of the technique is vastly less realistic and effective than more modern methods. Another significant drawback is the computation time of ray tracing rendering algorithms with a scene being rendered to an image 500 pixels wide, at time of the implementation's publication, taking 28 minutes to complete a rendering pass.

Yanyun et al. proposed a multi-mapping technique for the rendering of static snow covered scenes, [YSHW03]. The proposal divides the scene into components of a snow covered landscape and complex static objects, in this case, trees. To efficiently display the complex trees, ray casting is used to generate volumetric texture representations of the geometric models. Snow is then added to the volumetric model from above and covers the upward facing surfaces to give a densely snow covered tree. The landscape component of the scene is extruded to create snowy blocks covering the surface, each block being filled with an amount of snow determined by the probability of snow landing on the given point and the stability of the structure beneath the snow cover. The probability of snow landing on any given surface is determined by casting a ray from surface points upwards to determine occlusion. The snow cover in each block is stored in a displacement map applied to the bounding box. Once the scene had been pre-processed to apply all snow cover, it is rendered using ray tracing techniques. A simple shading model is applied when a ray intersects with a snowy

ground block defined by its stored displacement map and when rendering complex tree volumes the density of voxels are treated as scattering pixels. The rendering process is compared to rendering the scene as a polygonal representation and although very slow, over 33 minutes to render the scene once at 1488x918 resolution, the multi-mapping technique shows much more efficiency than polygonal rendering which took over 67 minutes in the implementation. Using a ray tracing technique with volumetric textures and displacement maps ensures a lower sampling where voxels are much denser, allowing for inherent level of detail representation and reduced sampling complexity on elements further from the camera, increasing the rendering efficiency dramatically. However, while only suitable for off-line rendering scenes given the high rendering time, the technique also required the scene to be completely static as any alterations or movements in the complex trees stored in volumetric textures would require recalculation of the pre-processing stages.

While several solutions have been explored for the rending of blanket snow cover on an object or scene, Langer et al. proposes an image based technique for the rendering of falling snow more effective than particle systems alone, [LZK04]. The solution uses a small number of particles combined with an image based spectral synthesis to generate the textural effect of dense, heavy snow. Snowfall is analysed in its effect on the final image and determined that while all snowflakes may be falling at a constant rate, in image space the flakes closer to the camera will move faster as well as appearing larger. To adjust for the differing rates of fall caused by the perspective, a number of motion planes are initialised bound to a set of frequencies to generate the appearance of movement. Within the simulation it is assumed that all snow falls at a constant 3D velocity. Using the set of motion planes to give a depth of varying motions, the solution uses spectral synthesis to generate a time varying opacity function. The opacity function can be used to create a moving snow

layer which can be overlaid on a image to give an animation of snow with the effect of considerable depth. For use in three-dimensional scenes the snow layer can be mapped to a plane rendered close to the camera with the best visual results achieved by using a combination of the opacity function and finite particles. By combining the two approaches, the particle system gives distinct individual snowflakes and the spectral synthesis gives the textural effect of dense falling snow. While the solution was developed for the overlaying on top of static 2D images or video, it has been adapted to work with 3D polygonal scenes with a moving camera. By keeping a low particle count and adding the opacity function, the visual effect is much more realistic than simply increasing the number of particles and while increasing the particle count greatly slowed rendering of the scene, the adding of the spectral synthesis technique to the 3D polygonal scene made no discernible difference to the rendering time achieved. The method is effective as well as flexible, being able to be extended to include rain, motion blur and a movable camera however the solution is still unable to achieve what would be determined as real-time frame-rates, reaching only 4-5 fps while rendering a static image at 512x512 resolution. In addition to the slow rendering time, the solution in its published state allowed for only a fixed rate of depth across the whole image, not accounting for any occlusion or differing depths as may be encountered in a dynamic 3D scene.

2.2.2 Snow Accumulation

Occlusion Based Accumulation

One of the major influences on the reality of any given snowy scene is occlusion. Snow should not settle and lay on a surface which is blocked by another surface above. Modern shadow rendering techniques tackle occlusion by rendering a scene first from the direction of a light source, and using the depth of all surfaces visible to

determine whether a surface rendered from the camera's view should be illuminated. Several snow techniques use a similar basis by projecting the areas to be covered with snow from above.

In 2004 Ohlsson and Seipel [OS04] proposed a technique to use deferred rendering, shadow mapping techniques to accumulate snow onto a complex scene. By rendering the scene initially from the point of view of the sky, occlusion is computed and the height of each point within the scene which is viewable from above is stored in a depth map. The depth information of each surface is then used to determine whether the face will receive snow cover. To smooth the edges of snow occlusion adding to the realism of the scene, several projections are taken from above at slightly different offset angles and the result is averaged. To smooth out artefacts caused by averaging several depth maps, the resultant snow accumulation is combined with a perlin noise stored in a three dimensional texture and the result saved to an alpha texture for rendering. The amount of snow received on a face is calculated using its visibility from the sky and the angle of the face, with greater amounts of snow accumulating on more horizontal surfaces. Once the scene has been rendered in its first pass to calculate the build-up, it is rendered again and the snow is visualised, using a noise texture bump map to improve realism. The main advantage of this solution is that the technique, using projection rendering, makes the snow calculation completely independent of the complexity of the scene and number of objects contained. However, there are several drawbacks to the implementation, with interactive speeds being achieved but not with high enough frame-rates to be considered real-time at this level, performing at 11 frames per second when rendering a 16,000 polygon scene at 900x900 resolution. Rendering the snow surface as an offset surface from the original object does not close the edges of the created displacement properly and leaves artefacts where snow can be seen to be floating above the surface it lies on. As snow calculation is performed

per pixel in the rendered vertical projection, the size of an accumulating surface is not taken into account allowing very small faces such as thin twigs to build up unnatural levels of snow. Finally, although the technique is considered to be interactive and dynamic scenes can be calculated, the snow is recalculated between frames causing the effect that if a snow covered surface were to move under the cover of another object, any accumulated snow would be lost upon recalculation.

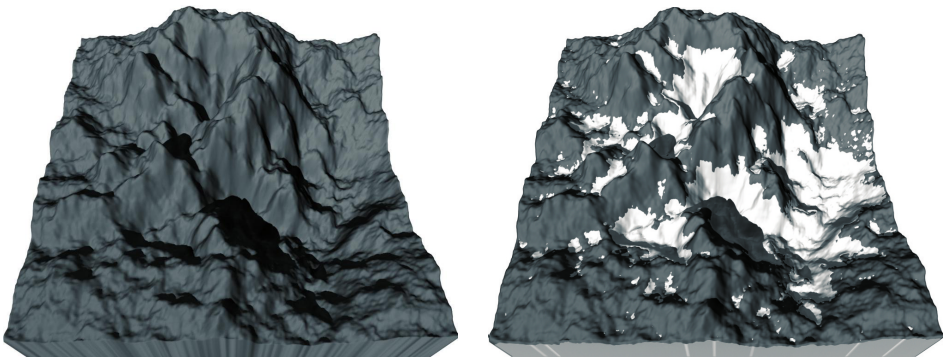
Tokoi proposed a similar method of snow accumulation, [Tok06]. As a pre-processing step, all upward facing surfaces in the scene are classified as snow coverage areas and, grouped in such a way that within each group there were no horizontally overlapping areas, i.e. no overlaps within a group when viewed from above. Each group then forms a snow coverage map, used to accumulate the snowfall. As with Ohlsson and Seipel, shadow mapping techniques are used to project the surfaces visible from above, using multiple projections from offset angles to simulate falloff and flake flutter. Once snow has been accumulated, each site is checked for the stability of the snow deposited. If any given snow site has accumulated more than the maximum amount of snow, the extra is deposited to a neighbouring cell and if the difference between two adjacent sites exceeds the angle of repose, snow is shifted to the lowest site. If snow is shifted off an edge, the excess is deposited on the highest snow site below the current surface. Once all stability tests have been completed, the snow surface is converted into a polygonal mesh for rendering. This snow accumulation technique has the benefit of being independent of the complexity of the scene, making it a good approach for complicated areas, however, it does have some significant drawbacks. Given the current implementation, it is very difficult to divide single objects within a scene into multiple snow coverage groups, making the system unable to process self occluding surfaces as groups must be non-overlapping. While processing time is not dependant on the polygon count of a scene, it does increase with larger numbers of

target groups and as groupings must not overlap, complex scenes can require a very high number of groups. Given the resolution of snow maps, fine small details such as individual leaves have difficulty defining a snow profile and, although the performance of the solution is technically interactive, a very low frame-rate of 3.38 fps is achieved at 300x300 rendering resolution, making the technique a substantial amount less efficient than real-time.

Foldes and Benes, [FB07], present a technique for rendering snowy large scenes from a great distance by using ambient occlusion and direct occlusion to determine snow-melt. Ambient occlusion is calculated on the large scene e.g. a mountain, to determine where snow would accumulate. If the ambient occlusion is too high it is deemed that snow would not penetrate to the area. Direct occlusion is calculated using projected rendering passes to determine which areas of snow would receive enough direct illumination to melt over the course of a day. Sunlight is averaged over the course of a full day, using the lights trajectory to determine melting. Once snow cover and snow-melt has been approximated, the scene can be rendered with realistic and high quality snow-cover as shown in Figure 2.9. The most expensive calculation in the technique is the computation of ambient occlusion which can be pre-computed as a pre-process. Low detail scenes can be rendered using the technique in a few seconds although several minutes are required for high detail models. The solution provides a very high quality result although rendering is not done in real-time and the scene cannot be dynamic without requiring a full re-computation of the snow cover.

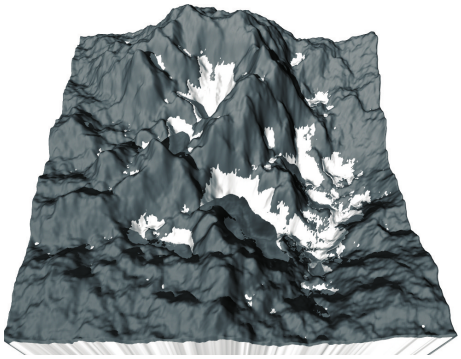
Geometry Based Accumulation

In the previous section, proposals were discussed which used rendering techniques to determine snow cover based on scene occlusion. There are several approaches to the problem which rely more on assessing the surrounding geometry creating the scene to determine snow accumulation. Fearing suggests an approach which uses particle



(a) Empty scene with no snow.

(b) Scene with snow accumulation generated using ambient occlusion only.



(c) Scene with snow accumulation generated using ambient occlusion and direct sunlight.

Figure 2.9: An example of the occlusion based technique put forward by Foldes and Benes, images from [FB07].

projection to calculate snow height for a given point, [Fea00]. Tackling the issue in the opposite manner than would be expected, rather than projecting from above to ascertain which surfaces would be hit by falling snow, Fearing proposes a method of projecting particles upwards from the faces to determine which are occluded and which eventually reach the sky. All snow supporting surfaces, surfaces which run horizontally enough to allow accumulation, are first divided up into launch sites. Each launch site emits a series of particles upwards and these particles are tested for collision with the surrounding scene. If no collisions occur and the particle reaches the sky-plane, a snow contribution is made to the launch site. This is performed so that each local surface has control of its contribution to the scene and can dynamically add or remove launch sites as needed. Adjacent launch sites which share the same snow accumulation can be merged into one and sites which differ radically can introduce more sites in-between them to refine the snow boundaries. Sites can be ordered by importance allowing greater focal points to spend more time processing and factoring in sites which have not had the chance to emit particles as the ordered list of sites is processed until a pre-defined limit of time is reached for each iteration of snow accumulation. Rather than calculating the result in one pass for rendering, the approach is gradual and the resulting scene is refined as the simulation continues. Particles are emitted upwards in a series of randomly offset vectors to simulate flake-flutter, a technique which can easily be combined with an overall direction vector to simulate wind force in a snow flake's travel. The sky plane is separated into a series of buckets and, when a particle reaches the sky, bucketing and filtering is performed to ensure that a small area of sky does not input too much snow onto the scene. After snow accumulation is calculated, stability tests are performed on each site by comparing its neighbours to calculate the angle between them, if the angle is greater than the pre-defined angle of repose, snow is shifted to a lower site or avalanched over

a surface edge if necessary. Once snow cover has been computed, launch sites are split into edge groups and Delaunay triangulation is used to convert the snow into a polygonal surface for rendering. Procedural noise textures and bump maps are used for snow dusting and to give a more realistic visual effect to the solid snow surfaces. The implementation proposed produces a very effective result with several advantages, allowing each local area to define its resolution allowing calculation of the most visually important sections at a much higher detail and realism. Simulation of flake flutter and wind travel are inbuilt into the solution as well as the ability to consider obstacles and blocking surfaces during edge transit of an avalanche. There are, however, limitations with the proposed method where objects that overlap themselves in the Z-axis causing great difficulty in obstacle calculations, often needing them to be split into more launch sites than necessary, sometimes as much as one per polygon. Stalagmite artefacts are caused by the over concentration of avalanched snow in the implementation and all snow accumulation is based on a static scene, not allowing the objects to be dynamic or animated in any way.

Haglund et. al. propose a simple method for rendering basic snow accumulation using a particle system, [HAH02]. Each surface is mapped to a two dimensional texture which stores the snow height at any given point. When a particle which is emitted from above collides with a surface, the height of snow cover at the corresponding pixel in the texture map is incremented. At surface boundaries a maximum snow height is used to limit shading artefacts at edges when rendered. For very light snow cover, an alpha blended texture is mapped onto the surface to give the visual effect of snow, whereas a new surface is generated from the stored accumulation where snow is thick enough. Using the regular grid of heights obtained from the texture, a new low resolution surface is procedurally generated using random triangulation patterns to minimise a noticeable repeating pattern of artefacts created by shading.

The solution proposes a simple and easy to implement method of accumulating snow cover on a scene, however, the results are very primitive and do not share the realism or accuracy of other methods mentioned in this survey.

Festenberg and Gumhold put forward a snow technique using height span maps of a scene, [FG09]. For the purposes of simulation, the technique focusses solely on the geometric properties of the scene to determine snow accumulation, and not aspects such as temperature, wetness or roughness. A height span map is created of the entire scene and depth peeling is performed to compute the height transitions of all points in the map, providing a representation of vertical occlusion. To ease the process of depth peeling, the scene is procedurally altered to add back facing polygons to all front facing surfaces which do not have any. Snow patches on each surface are grouped by picking a random starting point on each surface and flood filling the map until all connecting points are grouped. For each point in a group, outer and inner edge distances are also calculated. Starting from the highest point in the map, the maximum snowfall required by the scene is introduced and if the given point cannot support the maximum amount of snow, excess is shifted to neighbouring lower patches and avalanched onto lower surfaces if necessary. Once the span map has been completely traversed to assign snow cover to all points, the snow surface is triangulated for rendering. The process uses depth peeling to build a scene encompassing vertical occlusion, allowing for a desired amount of snow to be visualised in one pass rather than building up an accumulation over time, one major disadvantage of this is that the changing shape of a surface due to snow accumulation is not considered. The creation of an entire snow scene is computed in approximately 1.5 seconds allowing reasonably complex animations to be rendered off-line, but not achieving interactive or real-time speeds. Due to the depth peeling and height span map allowing the computation of snow cover from the highest point downwards in one

pass, snow stability does not need to be further calculated, increasing the techniques efficiency. One significant disadvantage to the implementation as presented is with finely detailed objects where a surface may only be able to contain a single row of snow patches, producing unwanted artefacts as shape cannot be computed from the limited cover.

Festenberg and Gumhold also put forward a method of snow cover generation based on a physical model of granular deposition, [FG11]. As with [FG09], a height span map is created of the scene with each pixel representing a possible snow site and depth peeling is used to generate a representation of the entire scene encompassing vertical occlusion with each site sorted by height. Each pixel of the snow map is then put through a discretized kernel, assessing the surrounding sites and determining snow cover using a diffusion equation for snow with strong adhesion and analysing upwards for visibility and downwards for surface support. As gaps smaller than the maximum possible bridge length are found within a kernels range, new snow sites are generated within the gaps producing a bridged surface, calculating its position and form using the distances to supporting edges, as shown in Figure 2.10. The accumulated data is then converted into a triangle mesh for rendering, adding an offset vector at surface boundaries to achieve overhangs. To increase realism, an alpha masked texture created by assessing real snow boundaries is applied to the edges of the generated surface, providing a high detail effect without adding to the complexity of rendered geometry. A marching technique is utilised to propagate edge texture coordinates across the mesh using a priority queue of vertices until coordinates have been assigned around the entire boundary reaching to one texture unit high. The solution put forward in this paper has the capabilities to generating high detail snow cover on a complex scene to any desired accumulation in one step, also giving the possibility of generating snow accumulation animation although, recalculating

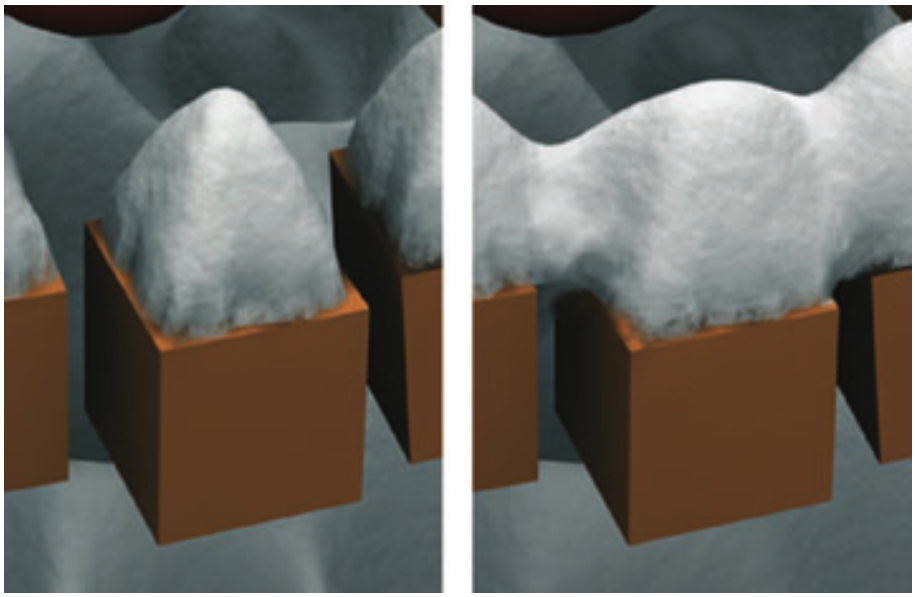


Figure 2.10: An example of generated snow bridging, images from [FG11].

the scene from scratch for every pass. While the results are much more realistic and visually convincing than those achieved in [FG09], as shown in Figure 2.11, the computation time required is much greater, with 95% of the time being the processing of each snow site using the kernel and the implementation taking an average of one minute per frame. Another serious disadvantage of the technique is that, like the majority of snow cover simulations, it is limited to static scenes as animation of the scene would not take into account snow accumulating on surfaces which were previously un-occluded due to a change in geometry.

In 2013, Stomakhin et. al. described a snow simulation technique using a Material Point Method (MPM) to model realistic snow interactions with a complex, animated scene, [SSC+13]. MPM uses material points (particles) to track snow position, velocity and mass using a Cartesian grid. Particles are first rasterized to a grid where densities, volumes and forces are calculated and grid velocities are updated. The Cartesian grid allows self collision of the snow to be calculated. Once

Figure 2.11: An example of generated snow covering a pine tree, images from [FG11].

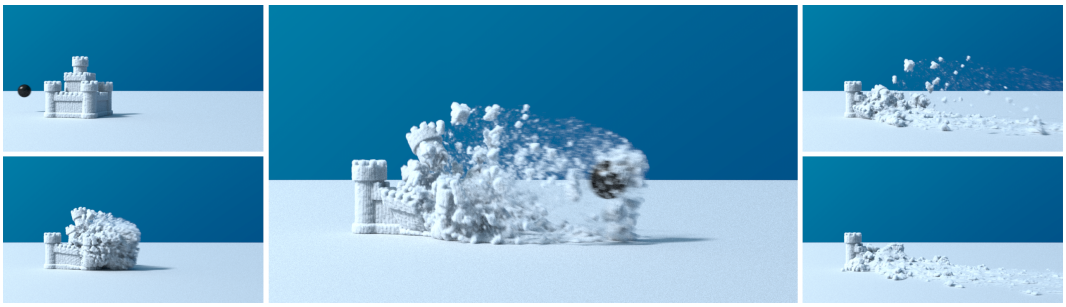


Figure 2.12: An example of the fracture of a modelled snow sculpture using an MPM method, images from [SSC+13].

self collision is performed, the linear system is solved and the particle velocities are updated. Particle collisions with the scene are calculated and finally particle positions are computed. Snow bodies are meshed to a geometric surface for rendering allowing the MPM system to only track unmeshed particles, which are rasterized to the simulation grid for render. The system allows for a user-controlled snow model with adjustable elastic properties and snow behaviour. The system provides an effective and realistic rendered result allowing user controlled snow behaviour. The MPM method is shown to give good results simulating stiffness, plasticity and fracturing of snow-based sculpted objects as shown in Figure 2.12. The system also interacts dynamically with an animated scene and complex animated characters. While the simulation is largely procedural, many parameters need to be tuned by hand, requiring substantial user control to produce the required results. Interaction with air is also not implemented in the system, which would be necessary for the simulation of powdered snow in situations such as avalanche simulation. The MPM method is an offline system, not capable of simulating interactions within real-time applications, scenes tested within the paper show a range between 2.1 minutes and 25.8 minutes per time-step, requiring a substantial amount of computing time to produce lengthy and complex simulation.

2.2.3 Environmental Simulation

While one of the most noticeable visual factors of snow accumulation is the occlusion of objects within a scene, the environmental state of the area plays a large part in where snow will land and if it will settle and build up or melt. Several studies have been made on the simulation of environmental factors such as wind or surface heat and how such aspects effect the generation of a realistic snow covered scene.

Muraoka and Chiba, [MC00], proposed a snowfall simulation which incorporated snowfall, snow cover and also the effect of snow-melt on the rendered scene. Wind forces are calculated, having an effect on snowflake movement, by combining vortex fields to generate a field of air currents. The currents are comprised of both simple currents (laminar flows) and turbulent flows of differing size to create a full air flow system. A particle system is used to simulate the individual snowflakes, which move through the scene acted on by both gravity and wind forces. The scene is discretized into a voxel field upon collision between a static object and a snowflake, the adhesive force is computed to determine whether the flake will stick to the surface. Once snow cover is generated, snow-melt is incorporated, calculated on the surface and underside of all snow accumulations. To perform the snow-melt calculations, all polygonal objects which are included in the scene are converted to voxel representations and included in the grid. Determined by an object's attributes, initial temperature is set in all voxels of the simulation and ambient temperature caused by scattering light into a bounding sphere, with temperature caused by direct sunlight incorporated only into the effected areas. For each voxel, heat exchange is calculated for its six neighbouring voxels and temperature increase by ambient and direct sunlight. After all heat transfers are computed, each voxel is inspected to determine any necessary state changes, such as from snow to air, with water being lost in the implementation. The radiation calculations are repeated multiple times with a changing sun position. To

render the scene once all simulation is completed, 3D textures are used for individual snowflakes whereas snow cover is rendered using a combination of a polygonal definition for its surface and a 3D noise texture for the rendering of the internal structure. The proposal produces a solution for the effective simulation of very complex natural phenomena, however the snow cover produced takes a largely simplified form which is visually unrealistic and the technique of decomposing the scene into a voxel representation allows for only static, unchanging areas to be simulated.

Feldman and O'Brien, [FB02], demonstrate an implementation of snow accumulation due to wind force. To achieve this, a given scene is discretized into a uniform grid and wind flow fields are calculated throughout the scene relative to any obstacles causing a change in air flow patterns. Once wind fields have been completely calculated for the scene, snow is introduced at a boundary and convected along the fields until resting on a snow site, stored using a height map. Periodically the wind fields are recalculated to allow for the changes in the scene caused by accumulating snow formations. The amount of snow deposited on each site is compared to neighbouring sites to ensure that the angle of repose is not exceeded and any excess snow is avalanched back into the scene to be deposited again by the wind fields. The stabilisation repeats until all snow sites are completely stable and the entire process continues until a pre-set desired amount of snow has been deposited throughout the scene. Once the snow cover had been computed, the height field is rendered as a catmull clark subdivision surface. Using flow fields, the approach allows realistic wind based snow accumulation to be created and simulated, allowing for new effects in winter scenes, however, the process is very slow and computing a single scene using Matlab can take several hours limiting the procedure's use to off-line rendering applications only.

Wang et. al. uses the Boltzmann equation to create a three dimensional mind

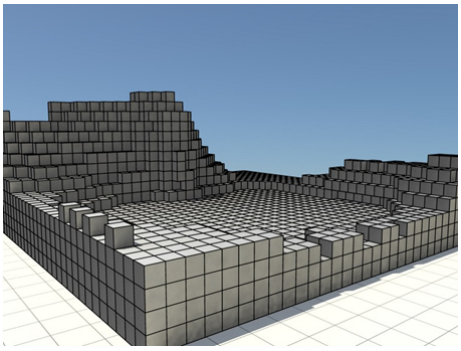
model in order to simulate snow accumulation by airflow, [WWXP06]. As a pre-processing step, the scene is divided into a regular discrete grid to allow the wind model to be sampled at each grid node. According to a probability function, snow is introduced to the model procedurally across the upper and left boundary, and its motion is modelled through the wind field, ignoring snowflake collision which is deemed to be infrequent enough to be negligible. Each snow supporting surface is mapped to an accumulation texture, upon which a snowflake counter is recorded using alpha values and when enough snowflakes have settled on any given site, its height is increased. The rate of snow erosion is computed as a function of wind speed, snow concentration, saturation concentration and efficiency of travel. This solution provides a very visually effective result giving realistic snow build-up around obstacles which create a vortex in the wind model. Rendering the implementation at a 600x400 resolution gave a real-time application achieving 26 frames per second on very low powered computing hardware.

Maréchal et al. [MGG$^+$10], detail an implementation of an environmental model for winter scenes which incorporates air and dew point temperatures, precipitations, day and night cycles, cloud cover and heat transfer between materials. A thermal volume simulation is used to calculate conductive, convective and radiative heat transfers through a scene. Simulation of phase changes are also included allowing the modelling of both freezing and melting processes. Once simulation is complete the solution generates polygonal models of snow and ice layers throughout the defined area. The first step of readying the scene for simulation is to generate a voxel representation of the terrain using a coarse voxel grid, a finite volume technique is used to calculate heat transfers through elements in the grid. In the first stages of the simulation, a weather model is used to calculate the precise parameters for the environment at any given stage and snowfall and accumulation is simulated throughout the scene. Once the

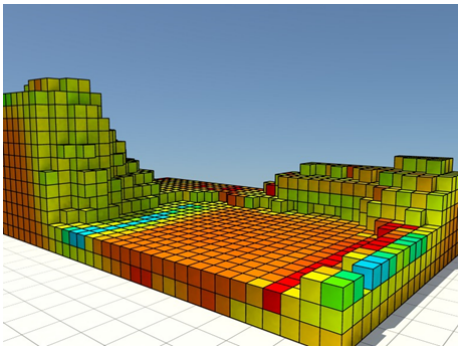
weather effects have been computed, heat transfer between each voxel and its neighbouring voxels is calculated and all state changes computed for materials defined by voxels. Phase changes also have an effect on neighbouring voxels, for example, melting snow will introduce water to soil blocks below it, altering their thermal attributes. Each voxel contains identifiers for its contained material, attributes, temperature and energy which is assessed by the simulation loop. Although the simulation takes into account a variety of heat flows, solar energy and radiative transfers with air are discovered to have the most noticeable effect on the scene. Once simulation has completed, the voxel representation is used to generate three polygonal meshes for rendering, one for water, ice and snow. The snow mesh is generated as a height field used to displace underlying surfaces where as a general water/ice mesh is generated before decomposing it into its relative parts. Smooth transitions between meshes whilst rendering are achieved using texture blending. A visual representation of the proposed process is shown in Figure 2.13. The solution provides a very realistic and effective approximation of complex natural phenomena, generating very high quality off-line rendered results for static scenes.

2.2.4 Summary

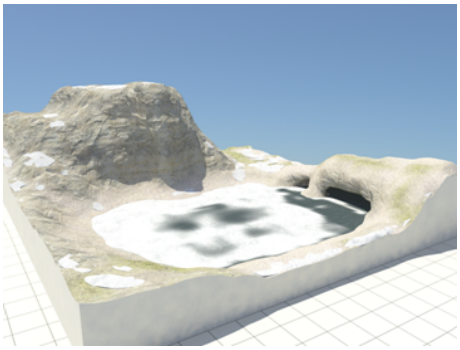
To develop an effective snow simulation technique, the requirements of an ideal technique are set, based on the important factors of current work. To allow a snow system to be used in a dynamic environment, it must have persistent accumulation which remains accurate in respect to the occlusion of the scene. Snow stability must be computed to allow a dynamic scene to alter the snow cover through movement. To achieve visual realism, surface offset must be incorporated to allow snow build-up to generate new surfaces throughout the scene and fine details of snow cover must be



(a) Voxel representation of scene.



(b) Heat transfer simulation.



(c) Simulated scene rendered as geometry with ice and snow.

Figure 2.13: Images from [MGG+10].

incorporated to give a realistic effect when viewed at close range, such as surface tessellation or per-pixel lighting using normal mapping techniques. To allow interaction with the scene, real-time frame-rates are required as well. These requirements and the applications of them in current techniques are summarised in Table 2.2.4.

	Persistent Accumulation	Scene Occlusion	Snow Stability	Surface Offset	Real-Time Frame-rates	Fine Detail
Nishita et. al. (1997), [NIDN97]				✓		✓
Yanyun et al. (2003), [YSHW03]	✓			✓		
Langer et al. (2004), [LZK04]					✓	✓
Occlusion Based Accumulation						
Ohlsson and Seipel (2004), [OS04]		✓	✓	✓		✓
Tokoi (2006), [Tok06]	✓	✓	✓	✓		
Foldes and Benes (2007), [FB07]		✓				✓
Geometry Based Accumulation						
Fearing (2000), [Fea00]	✓	✓	✓	✓		✓
Haglund et. al. (2002), [HAH02]	✓			✓	✓	
Festenberg and Gumhold (2009), [FG09]	✓	✓	✓	✓		✓
Festenberg and Gumhold (2011), [FG11]	✓	✓	✓	✓		✓
Stomakhin et. al. (2013), [SSC$^+$13]	✓		✓	✓		✓
Environmental Simulation						
Muraoka and Chiba (2000), [MC00]	✓	✓	✓	✓		
Feldman and O'Brien (2002), [FB02]	✓	✓	✓	✓		✓
Wang et. al. (2006), [WWXP06]		✓	✓	✓	✓	
Maréchal et al. (2010), [MGG$^+$10]	✓	✓		✓		

Table 2.2: Summary of current techniques covered in Chapter 2.2 and their application of identified key requirements. Work is listed in the order it appears within the chapter.

Chapter 3

Remodelling of Botanical Trees for Real-Time Simulation

This chapter describes a technique to model detailed, high quality tree models using pre-existing branching structures. The physical geometry is regenerated using the created structure to produce a virtual tree suitable for high quality, real-time simulation and rendering. Many tree modelling solutions currently available create highly realistic tree structures, however, the geometry exportable is often unsuitable for very high quality visualisations and important structural information such as a usable tree skeleton often over simplified, making it difficult to create or control tree movement. In addition to inadequate skeletal information, many procedurally generated trees have the disadvantage of being composed from multiple, distinct surfaces rather than one continuous mesh, resulting in crudely shaped junctions between branches and unrealistic artefacts. The techniques described in this chapter derive a full, high resolution skeleton from an existing branching structure, organised appropriately to describe the inherent hierarchy of the tree. New geometry is generated around the skeleton as a single, continuous polygonal mesh introducing higher quality modelling of branch joins and incorporating full bone weighting of the mesh, giving a system able to be animated in a similar way to applications of character skinning. The model is optimised for cutting edge rendering techniques utilising recent advancements in

graphics hardware, most prominently the tessellation engine which is used to both add considerable detail procedurally and remove unnecessary complexity providing efficient, dynamic level of detail representation. Automatic bone weighting is employed to transform the mesh and add further detail, as described fully in this chapter.

3.1 Related Work

As discussed in more detail in Section 2.1, manually modelling complex trees can be unfeasible for most applications, and as such, substantial work has been done to procedurally generate realistic tree structures and geometry. One system of modelling a trees branching structure is Xfrog, put forward by Lintermann and Deussen, [LD98]. Xfrog uses a hierarchy of user controlled structures employed to generate a complex tree model by giving the user a selection of useful elements such as branching shapes and leaf nodes, and allowing them to be consecutively stacked to form the basis of a botanical model. Modifiers governing the shape produced and the distribution of child elements in addition to world constraints such as gravity and light allow for simple generation of high quality single models. Deussen and Lintermann improved the system by adding much more control over the individual elements, [LD99]. One major advantage of Xfrog over its competitors is that it models every individual element including each leaf in a simple manner without using a technique known as billboarding where several elements will be grouped together into one image which is used to texture a plane as a simplified representation. By avoiding billboarding and creating every element, the produced trees can be used with highly detailed simulation techniques such as those put forward by Ota et al., [OTF+04]. While the proposal can be used to create excellent tree structures which appear to be very realistic, as shown in Figure 3.1(a) and Figure 3.1(b), there are two major drawbacks to the geometry which is generated. One large limitation is that only the mesh

(a) Generated Xfrog Japanese Maple model with leaves.

(b) Generated Xfrog Japanese Maple model without leaves.

(c) Artefacts caused by modelling branch joins as a non-continuous mesh.

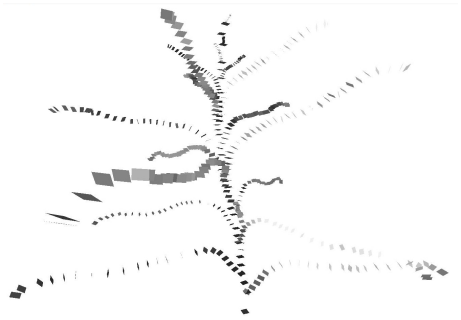
Figure 3.1: A young Japanese Maple tree, generated using Xfrog.

of the tree itself is accessible when exported for inclusion outside of the software. Without access to the structural skeleton of the tree and how this relates to the physical geometry, it becomes impossible to use the model in any sort of dynamic scene. With enhancements in the quality of graphical applications, the expected realism in simulations often requires animation of elements in accordance with weather conditions and physical interaction. The lack of structure to properly simulate this movement is one of the major issues tackled in the work put forward in this thesis. The second drawback to the approach of the software, is that models are created using a separate, unconnected mesh to represent each branch. The effect of this approach is that if the root of a child branch does not have a width matching that of its parent at point of connection, unrealistic artefacts are caused as shown in Figure 3.1(c). At close range, this inaccuracy and over-simplification of the branch junctions becomes very noticeable, and drastically reduces the visual quality once independent branch motion is introduced. The software provides an effective and easy to use tool to create the form and structure of a virtual tree. However, the geometry produced is inappropriate for animated scenes where the trees may become the visual focus, such as in recent simulations performed by Habel et al. [HKW09].

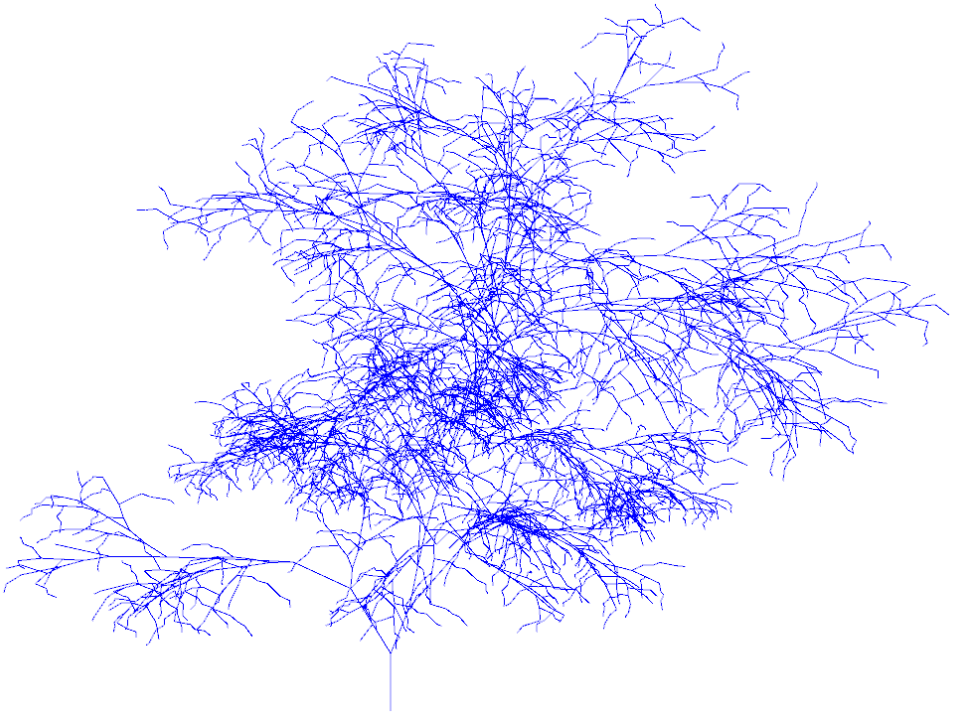
The techniques of procedurally modelling the geometry of a tree can understandably be split into two areas, modelling the structure and form of a tree and modelling the actual geometry created around that structure. Given the pre-generated structure of a maple tree, Bloomenthal details a method of generating high quality geometry to provide a visually realistic rendering of the tree, [Blo85]. A method is shown which maps a three-dimensional circle of points at intervals around the line of a given tree branch, which are then used as the basis to create the polygonal mesh of the limb. At points where limbs join, complex procedures are required and implemented to join the meshes forming a *ramiform* at the junctions without intersecting or overlapping.

Although the geometry created by this proposal is of a very high visual quality and is very realistic, it is inappropriate for use in real-time or animated applications. The costly computation of smooth mesh curves at branch junctions is performed as a pre-process to rendering and is too inefficient to compute in real-time. This disadvantage makes the model completely static as any change in the shape of the tree, especially where a child branch connects to its parent limb, would require re-calculation of the mesh. In addition to this problem, the technique creates tree structures of a very high complexity which may be appropriate if the tree itself is the main focal point of the scene, but makes it infeasible if groups of trees or alternative points of focus are required without drastically reducing the mesh resolution. The complications of procedurally modelling limb junctions is also tackled specifically by Lluch et al., [LVM04], where a pre-calculated structure using an L-System approach is taken as a skeleton upon which to generate the geometry in a single polygonal mesh. The idea of the research is to join different sections of the tree together using only one continuous mesh. The proposal performs this by identifying the intersection points of elements and grafting the polygons together with a higher resolution triangle mesh, ensuring complete continuity. While the approach rectifies the issues of unconnected surfaces whilst maintaining the form of an original input tree, it shares the same limitations as the previous proposal in that the models produced are created using a very high number of polygons making it ineffective in many real-time simulations. The remodelling of branch connections is based upon a static tree. Should motion and animation be introduced to the tree structure, connected limb junctions would need to be recalculated and the geometry regenerated, making the proposal infeasible for dynamic scenes.

(a) An exported Xfrog branch representation in full.



(b) As 3.2(a), showing only the first two branch levels for clarity.



(c) The skeleton extracted from an Xfrog representation, shown in blue.

Figure 3.2: Example of skeleton generation from an Xfrog tree.

3.2 Generation of the Skeleton

The techniques described in this chapter will be applied using branching structures exported from the tree modelling software Xfrog, but are not limited to a single tool. The Xfrog program is chosen due to both its high usability and ability to decide how branches of a generated structure are represented. The technique will be demonstrated using a stock tree model packaged with the software depicting a Young Japanese Maple with the choice of primitive representing the branches being the only customisation required before exporting the model using Wavefront OBJ format. The primitive selection process available in Xfrog simplifies the bone extraction task, by choosing a square representation, each branch of the tree is created using square cross-sections positioned at the point of segmentation along each branch length. This creates an array of small planes forming the line of the limb as shown in Figure 3.2(a) and Figure 3.2(b), both in full and only showing the first two levels of branching for clarity. By giving a name to each level of the structure within Xfrog, the exported geometry is grouped within the resulting OBJ by individual branch, each labelled with a sequential number pre-fixed with the name given to that level of branch. This grouping and naming convention is then used in the processing of the data to separate the polygons representing each branch and group the collections by the level they belong to. A polyline of bones making up the branch is then generated by connecting the midpoint of each square to the midpoint of the next. Already knowing which level of the structure an individual branch belongs to, the distance between the root of the branch skeleton and the points of the next lowest level can be checked to find the closest point and, as such, the parent branch and point along that branch which the current limb connects to. Using this information, the collection of branches is re-ordered and combined into one hierarchy of elements, storing this main trunk which contains a list of child branches. In turn each possesses a list of child

branches until the entire tree is defined. Using this approach, not only is the skeletal information present for bone weighting and animation, but the inherent hierarchy of the tree is fully captured giving an important and useful simulation aid. In terms of basic animation purposes it provides an efficient system to use during the real-time generation of stacked transformations across the structure. For physical simulation the network of branch connections is readily available for the calculation of force transferral among other interactions present between parent and child branch.

As one of the aims of the project is to implement a system of adding detail dynamically based on the visibility of a section of tree, it is important that the base geometry be as simple as possible. To this end, further simplification can be made to the skeleton itself. As curvature and smoothing of a limb can be added dynamically, described in Section 3.4, the level of segmentation and number of bones in a branch imported directly is largely unnecessary. By reducing the number of bones present, the number of polygons in the final mesh is significantly lowered and to achieve this, all branches of the lowest level of the hierarchy are left as they are to maintain the uneven crookedness which is desired. All branches of higher levels are redefined by their root points plus the roots of all child limbs. As a result of this procedure, all segmentation and curvature of a branch between the roots of its child branches is removed to be added dynamically, connecting the child roots with single straight bones, as shown in Figure 3.2(c). The overshooting of parent branches past the root of their last child branch, which occurs in the original model, is also removed as upon inspection of real vegetation patterns, the tendency of natural branches is to terminate at the beginning of a smaller branch shoot or fork into smaller branches rather than continuing on. This process gives a much less complex base structure to be used as the lowest level of detail, with smooth curve detail occurring in real-time as a result of tessellation of the final mesh in combination with bone weighting.

3.3 Creation of the Polygonal Mesh

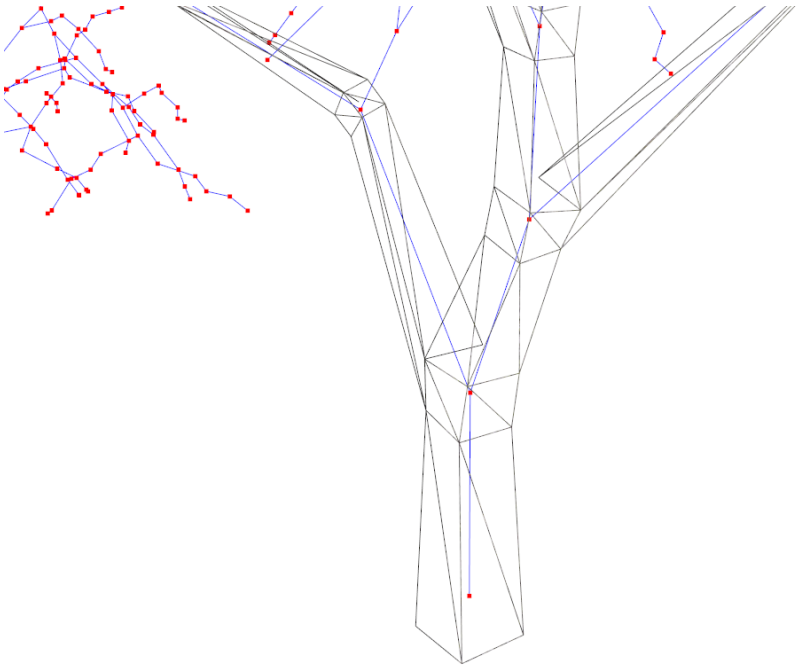
To solve the problem of branch roots not connecting to their parent branches to form a continuous mesh, rather than editing the existing model to form elegant joins as detailed in [LVM04], the geometry of the tree was completely regenerated around the extracted skeleton. This process allows a tree structure to be created specifically at a low level of detail base which can be easily tessellated to create the desired complexity. Vertices are created using the position of the skeleton points and a function to describe the width of a branch at any given point. This can be the original width of the exported Xfrog tree, or optionally a more realistic curve to add to the general form of the tree. In the case of the example shown in this chapter, the width of a branch is calculated to simulate the curve of $\frac{1}{x}$ for $1 \leq x \geq 4$ with the starting width at the root being the width of the parent branch at that point, as shown in Equation 3.3.1,

$$B_w = \frac{1/(3 * B_l * L_b + 1) - 0.25}{0.75} * R_w \tag{3.3.1}$$

where B_w is the branch width, B_l is the branch length, L_b is the length along the branch and R_w is the root width. The curve of $\frac{1}{x}$ is chosen due to the rapid flare it gives to the root of the branch. In reality, branch widths can be seen to be more linear in shape, however the curve of $\frac{1}{x}$ gives a more linear appearance as the value of x increases. The effect of using this curve is that a sudden widening is introduced at the root of the branch as it merges with its parent limb. As the square sides of the parent limb are expanded during tessellation at render, rounding the surface, the curved base of the child branch incorporates this rounding in its structure giving a smoother transition.

The procedure to wrap the tree in geometry starts at the root of a branch and iterates down the length creating vertices around each bone joint and is then called recursively on each of the current branch's child shoots. At the lowest level of detail,

the tree is comprised of four sided branches throughout to give a very low polygon count but also to facilitate clean simple joins. To generate a ring of four vertices around each bone joint, first the tangent of the joint between the two connecting bones is found at the joint and used to align the new points. As a more efficient way of calculating the four corners, the closest of the three global axes to the tangent is calculated and the line is projected into two dimensions down both of the remaining axes. By finding the lines perpendicular to the projected tangent, the two remaining local axes around the tangent are found. To generate the four vertices the calculated local axes are made into vectors with a length of half the branch width at that point and added to the bone joint position in all four combinations of the two vectors and their inverse. Once these are calculated they are joined with the points around the previous bone joint by a ring of triangular faces and the algorithm proceeds along the limb to the next segmentation. In the situation where a child limb joins the current section of geometry, two rings of points are created parallel to each other separated by the width of the branch, forming a knuckle on the linear extrusion and giving the child element a simple, clean join. The side of the branch closest to the direction its sub-branch shoots from is left unconnected by faces forming a hole in the mesh, the four vertices surrounding the gap being passed to the algorithm wrapping the child branch to be used as the initial ring of points, as shown in Figure 3.3(a). There are several advantages to modelling a tree in the form of a continuous mesh such as this, the most evident being the vastly improved visual quality. Branch joints appear to be more natural and consistent whilst removing problematic and unrealistic artefacts which occur at the junction of two unconnected surfaces. In addition to the aesthetic improvements, representing connections as a consistent surface allows for the proper simulation of force transferral between the elements and the effect applied force and movement has on the junction itself.



(a) Generation of a new model around the calculated skeleton.

(b) Visual representation of a single bone's influence on the geometry.

Figure 3.3: Procedural recreation of the tree's mesh around the skeleton.

One of the common problems encountered when 'skinning' polygonal meshes (applying a weighting to each vertex to calculate how much influence a given bone has on the point), is the miss-classification of bone weighting due to complex structures of elements being very close to each other and often closer to an incorrect bone than the desired one. Skinning a mesh is a very complex problem, however, as the geometry has been generated entirely relative to the skeleton, it is implicitly known which elements should influence any given section. When wrapping a ring of new vertices around a bone joint, they are each given an influence from the two connected bones of 0.5, with the exception of branch end points which only have influence from one bone at 1.0. As the geometry is modelled as one continuous mesh, where one branch joins another the vertices are shared allowing the influence from both limbs to be accumulated at the points and scaled down to total 1.0 as an additional process after all weighting has been assigned. Once all weighting per vertex has been calculated, an algorithm iterates through the mesh to calculate all bone influences on a per face basis. By examining each face and compiling a list of all skeletal elements that affect any of the three points included, a comprehensive list of all the bone weights for any given face can be generated. In the case where a bone influences some points of a triangle and not others it is assigned a weighting of 0.0 to the uninfluenced vertices, resulting in a constant gradient of influence across the face of the triangle. With traditional 'skinning' techniques it is usually necessary to generate the polygonal mesh in the desired form separately from the skeleton, which is then applied to the model using bone weighting. Depending on the form and complexity of the model it is often impossible to procedurally assign bone influence using a simple algorithm without misclassification of vertices due to distance or shape being closer to the range of a nearby but incorrect bone. As the generated tree is created to directly replicate the

previously developed skeleton and during formation of the individual points, the relevant bones are instantly accessible, vertex grouping can be performed during the process ensuring complete accuracy as demonstrated in Figure 3.3(b). In addition to allowing the mesh to be properly deformed by skeletal animation, the bone weighting assigned provides a blending definition which can be used to refine the geometry itself, as described below.

3.3.1 Adaptations for Differing Species

While the technique works well for the Japanese Maple tree which is used to demonstrate the results, adaptations must be made to the mesh generation to make the solution more robust and accurate when dealing with different species. The procedure of skeletal extraction and mesh generation performs equally well regardless of tree species, however the main difference which must be addressed is the width of child branches relative to their parent limb. For broad-leaf, deciduous trees such as maple, oak, and many other examples, a realistic form is created by tapering child branches from the width of their parents, however for many evergreen species such as pine, child limbs are much thinner and straighter from their root. To model this difference, at time of generation the equation used to determine branch width ($\frac{1}{x}$ for $1 \leq x \geq 4$ as discussed in Section 3.3) is modified by increasing the upper bounds. This makes the initial heavy taper of the branch appear much sharper and earlier along the limbs length, merging with the circumference of the parent limb and giving a much straighter branch rather than giving a more gradual taper to the root. This alteration is indicated at the beginning of structure extraction and only needs to be made to the form of the first level of child branches, being the only level which typically exhibits such a drastic difference in limb width.

3.4 GPU Enhancements

The implementation of the tree rendering after processing the data is created using OpenGL 4.1 on an NVIDIA GeForce GTX 460. One of the major advancements of graphics developments in recent times is the introduction of the programmable rendering pipeline, allowing developers to control how vertex and face calculation is performed at the rendering stage. However, the most important and newest breakthrough the implementation makes great use of is the tessellation engine included in OpenGL 4.0 and above. Tessellation works by taking an input face in the form of a simple triangle and subdividing it into multiple faces before adjusting the position of newly created vertices according to a displacement map or function. As this is performed within the rendering pipeline, no new data needs to be copied across to the GPU per frame allowing for an efficient method of incorporating dynamic level of detail representations without changing the initial low polygon mesh or introducing impostors such as billboards. The engine includes two new shaders into the rendering procedure, the tessellation control shader which is executed on every face of the initial model and determines the level of subdivision to be performed, and the tessellation evaluation shader which executes on every vertex of the newly subdivided mesh to calculate appropriate point positions. Starting with the simplified, low polygon model created by wrapping the skeleton, extra detail is added by tessellating the surface based on several factors and smoothing the resulting mesh using bone weighting. These factors are considered to determine the correct complexity depending on how much of the mesh can be seen. The main factor is the distance of the tree from the viewer and a linear progression is used to increase the level of subdivision as the viewer approaches the model. The implementation allows for customisation of how effective the mesh refining is by allowing the user to set the maximum desired level of tessellation and the distance from the tree at which tessellation should begin. In

addition to distance, this is combined with the area of the face being processed, with a larger face needing higher levels of subdivision due to its greater visibility, as shown in Equation 3.4.1,

$$T_l = (1 - \frac{1}{M_{td}} * F_d) * (\frac{1}{M_{fa}} * F_a) * M_{tl} \tag{3.4.1}$$

where T_l is the tessellation level, M_{tl} is the maximum tessellation level, F_d is the face distance, M_{td} is the maximum tessellation distance, F_a is the face area and M_{fa} is the maximum face area. Triangle area and other static per face attributes of the base, low polygon mesh are calculated as a preprocessing stage to avoid costly computation at render time. In addition to triangle area, the rendering pipeline includes back face culling, where any triangles facing away from the camera and as such, unseen, are removed from processing to save costly calculations being performed where unnecessary. The final factor contributing to the calculation is what level of branch the triangle belongs to, allowing further detail to be added only to the areas of the tree which have the highest visual impact such as the trunk or larger limbs. This dynamic range of tessellation which is calculated on a per face basis gives a smooth transition between representations, avoiding the issue of 'popping' occurring when switching the rendered object with one of a different level of detail by recalculating the mesh gradually.

An additional enhancement that can be made using the tessellation control shader is the dynamic culling of faces. By setting a tessellation level of 0.0 the face is removed for the rendering set and travels no further through the pipeline, allowing individual triangles to be turned off when unnecessary. There are two ways in which the implementation utilises this ability to increase rendering efficiency, the first being to remove small branches at such a distance where their rendering is unnecessary for the visual appearance of the tree. Given that the level of each branch face is already known along with its distance from the viewer, the face can be compared to

the customisable range of detail and the triangle removed if it is too far away. This is demonstrated in Figure 3.4, which shows four different level of detail representations. The effect of this procedure is that at key distances the highest visible level of branches will be completely removed from rendering, happening per face to give a smooth gradual transition between representations which is not casually noticed with the inclusion of rendering leaves. The effect of the level of detail is shown with relative scaling in Figure 3.5(a) without leaves and in Figure 3.5(b) with leaves, as it would appear within application. As with the modern programmable rendering pipeline individual faces can be removed altogether. Per face view frustum culling is also introduced to increase efficiency in a way that was previously not possible when rendering a static mesh in one pass. GLSL code used for the tessellation shaders is included in Appendix A. This shows the procedure for calculating the level of tessellation to be applied to an individual face, Appendix A.1, and the calculation of the positions for the newly created vertices using bone weighting, Appendix A.2.

One of the most costly procedures during rendering is the copying of data from memory onto the graphics hardware, making it vital that as much as possible is transferred as a pre-processing step and remains unchanged. To enable as much data to be pre-processed as possible, the implementation compiles the information into a continuous array and loads it onto the GPU memory in the form of a texture. This allows the tessellation shaders to look up the relevant information required at render time using their own inbuilt ID as an index. There are two major collections of data being generated prior to rendering, the first being per face information accessed from the tessellation control shader. This includes the area of each face, the level of branch it belongs to and a list of all skeleton bones having influence on the face along with their weighting at each individual vertex. The list of bones included serves as an index into the second collection of data which is information pertaining to the bones

Figure 3.4: Example of varying dynamic levels of detail using the tessellation engine.

themselves such as their length, their position along the branch they belong to, the branch's minimum and maximum width and the endpoints defining the individual bone. These data are used within the tessellation evaluation shader to generate the appropriate position of all new vertices and saves recomputing costly calculations which would severely impede rendering speed.

Shaping and adding detail to the branches is performed on the mesh after subdivision to create an accurate form for any given level of detail. Face information is accessible using vertex attributes for all three points of the triangle, with newly generated vertices being defined by their barycentric coordinates within the face. For each bone having influence on the face the closest point on the bone to a newly generated vertex is found, the vector between these points giving an offset direction to ensure all new points are positioned uniformly around the skeletal structure. By using bone information such as the position of the bone along the branch and the size of the limb,

(a) Scaled level of detail representations without leaves.

(b) Scaled level of detail representations without leaves.

Figure 3.5: Example of varying dynamic levels of detail, relatively scaled.

the same branch width calculation as used for the generation of the initial mesh is used, as shown in Equation 3.3.1. This ensures a perfectly consistent rounding of the new limb model when the vertex is offset to this width along the previously calculated vector. This new position is calculated relative to each necessary bone and the final position is averaged across all calculated ones in accordance to the bone's weighting at the particular point, as shown in Equation 3.4.2,

$$\vec{V_p} = \sum_{i=0}^{N_b} \vec{R_i} * W_i \tag{3.4.2}$$

where $\vec{V_p}$ is the final vertex position, N_b is the number of bones, $\vec{R}$ is the vertex position relative to the bone and W is the bone weight. This merging provides not only a smooth realistic rounding of each limb, but a continuous, gradual, connection at branch junctions and a steady curvature along the length of the branch. Using the weights pre-calculated at each original vertex and barycentric coordinates of the new points, bone weighting is linearly interpolated across the surface of a triangle. However, the fall-off rate for weighting which is present at one side of a triangle can be altered. In the case of branch junctions, a more gradual interpolation is used for the influences of bones belonging to a lower level of branch, causing the large branches and trunk to have a greater effect on the form of the connections than the smaller limbs diverging. Other than shaping the form of the tree, the main advantage of dynamic tessellation is to add finer details when required which this approach allows quite easily. A displacement map in the form of a single channel texture is included to introduce variety in the mesh surface, which when accessed using texture coordinates calculated at the stage of initial geometry production, gives a displacement value which is simply added to the vertex offset distance from the bone. Multiple levels of displacement can be combined into one texture allowing the physical modelling of large detail elements such as knots and splits in the wood along with fine detail such as the unevenness of the bark without introducing further computational cost. With

Viewer Distance (m)	Frame Rate (FPS)		Polygon Count
	Mean	Std. Dev.	
10	279.70	0.46	195353
20	527.40	0.58	26782
40	696.80	0.51	6776
60	791.00	1.34	1624
80	791.25	0.70	1624

Table 3.1: Rendering frame rates of a remodelled, tessellated tree from various camera distances. Distances are shown in metres relative to tree height, taken as the mean height for an adult Japanese Maple.

tessellation being variable, at great distances the displacement mapping has no effect on the form of the mesh. However, if the viewer moves very close to the tree, a wide range of complex distortions of the surface is modelled in high detail, providing much greater realism than other techniques used to only simulate the effect such as parallax mapping and parallax occlusion. The culmination of these techniques to provide a highly realistic tree model is shown in Figure 3.6.

Figure 3.7 shows a system diagram of the remodelling technique described in this chapter. This shows the workflow in addition to defining tasks which are done on the CPU as a pre-process and tasks on the GPU which are used for rendering.

3.5 Results

Table 3.1 shows approximate frame rates achieved when rendering the generated tree model at different levels of detail. When compared to that which was achieved rendering the static model obtained directly from the Xfrog software, approximately 1660 fps consistently, the static model is displayed dramatically faster than the virtual tree produced using the technique put forward in this chapter, however, rendering time alone is not a fair comparison between the two. The proposed tree is calculating considerably more at time of render. Displaying the tree using the techniques described is

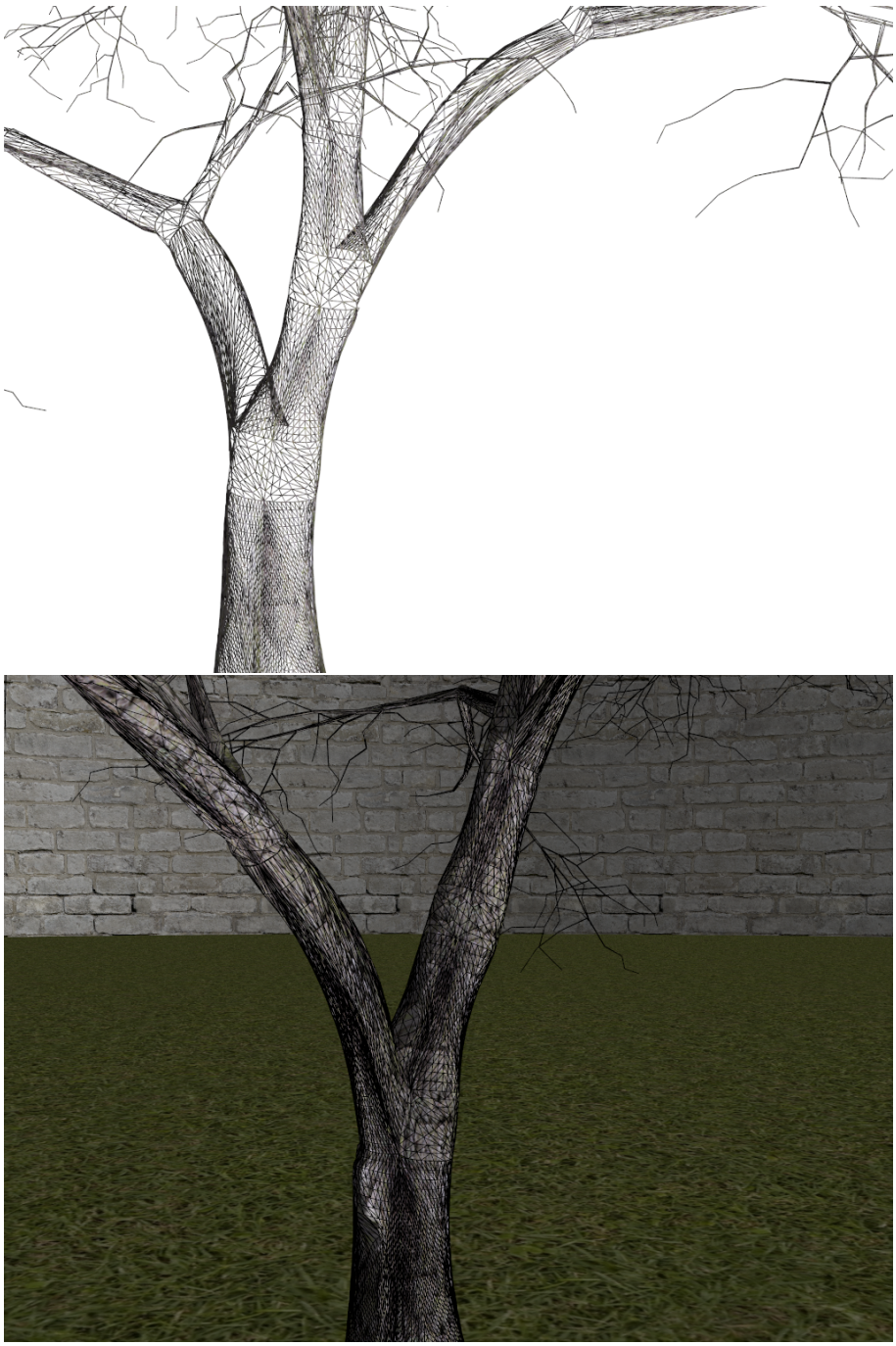


Figure 3.6: Added detail and mesh smoothing applied to a high polygon representation at close range. Shown in wireframe for clarity (above) and full colour with wireframe (below).

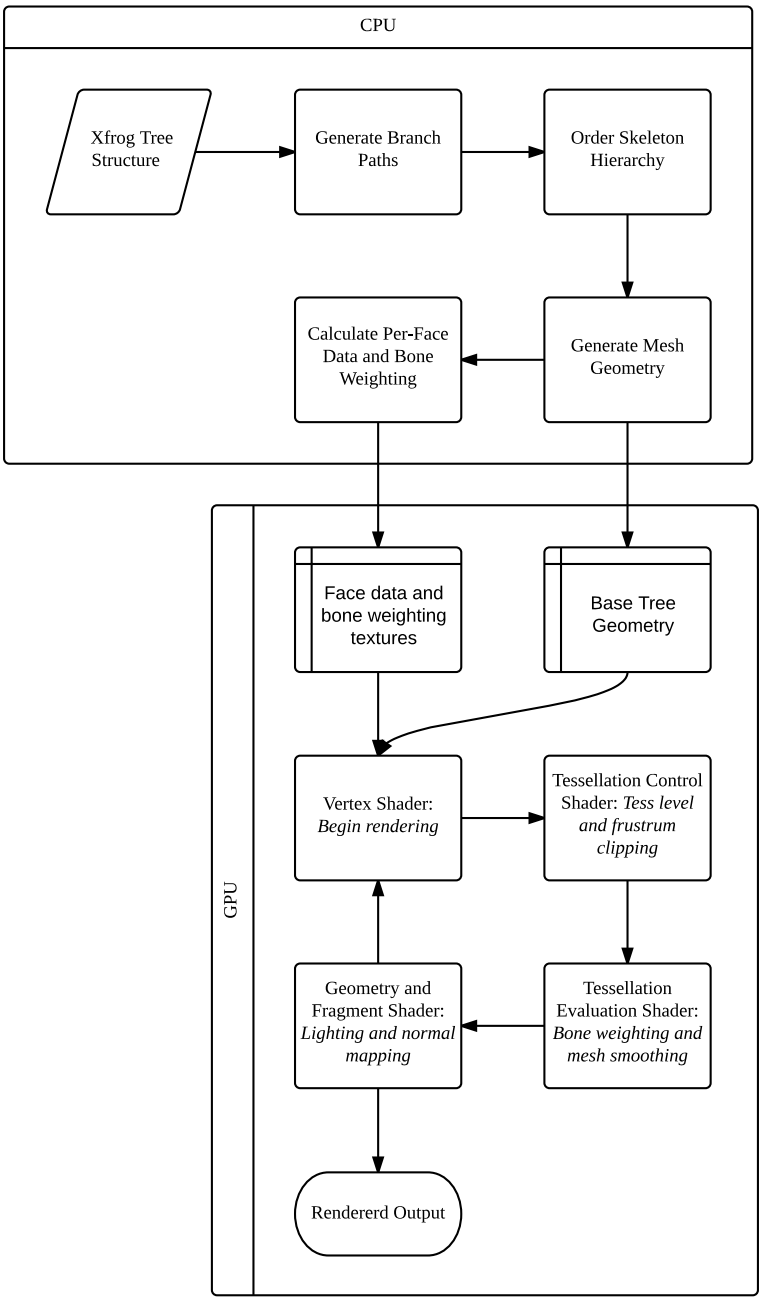


Figure 3.7: System diagram showing the process of tree remodelling technique, describing the computations performed on the CPU and the GPU.

inherently including full bone weighting of the mesh and skeletal deformation at every frame, which is necessary for a dynamic, movable scene element. As the limitations of the static mesh were not ones of efficiency, but rather of missing structural information and poor visual quality, comparison must be qualitative rather than quantitative. Although the results do show that the dynamic level of detail approach used without the introduction of impostors does greatly improve rendering times and shows that the implementation can be effective for non-static scenes where the virtual trees are not necessarily the constant visual focus.

One of the major drawbacks of the Xfrog model is visual artefacts introduced by disconnected branch geometry and Figure 3.8 compares the original branch connections formed with remodelled geometry as described, demonstrating that generating a continuous mesh around the structure removes the unrealistic effects caused by separated surfaces and increases visual quality. Additional examples are included in Appendix B. As newly tessellated points are procedurally positioned as a function of skeletal influence, Figure 3.9 shows that the automatically generated bone weighting, calculated at mesh creation, provides an effective solution without manual interference. This is demonstrated by introducing smooth blending between branch segments without artefacts as well as gradual curvature along limbs and across joins.

While the stock Japanese Maple has been used as an example to detail the procedure and results, the system is robust enough to give the same high standard of model with a variety of tree species exported from Xfrog in the same manner, as shown in Figures 3.10-3.13.

3.6 Integration into Industry Tools

During the creation of end-user applications such as simulations and games, many developers will use external graphics engines to render and simulate environments.

(a) Artefacts apparent in the original static mesh, highlighted in red.

(b) Generated model shown without connection artefacts.

Figure 3.8: Comparison of branch junctions in the original model and the newly generated model.

Figure 3.9: Fully tessellated tree showing curvature and shape being a function of skeletal influence.

Figure 3.10: Black Pine generated and rendered using the proposed system.

Figure 3.11: Colorado Spruce generated and rendered using the proposed system.

Figure 3.12: Horse Chestnut generated and rendered using the proposed system.

Figure 3.13: Weeping Willow generated and rendered using the proposed system.

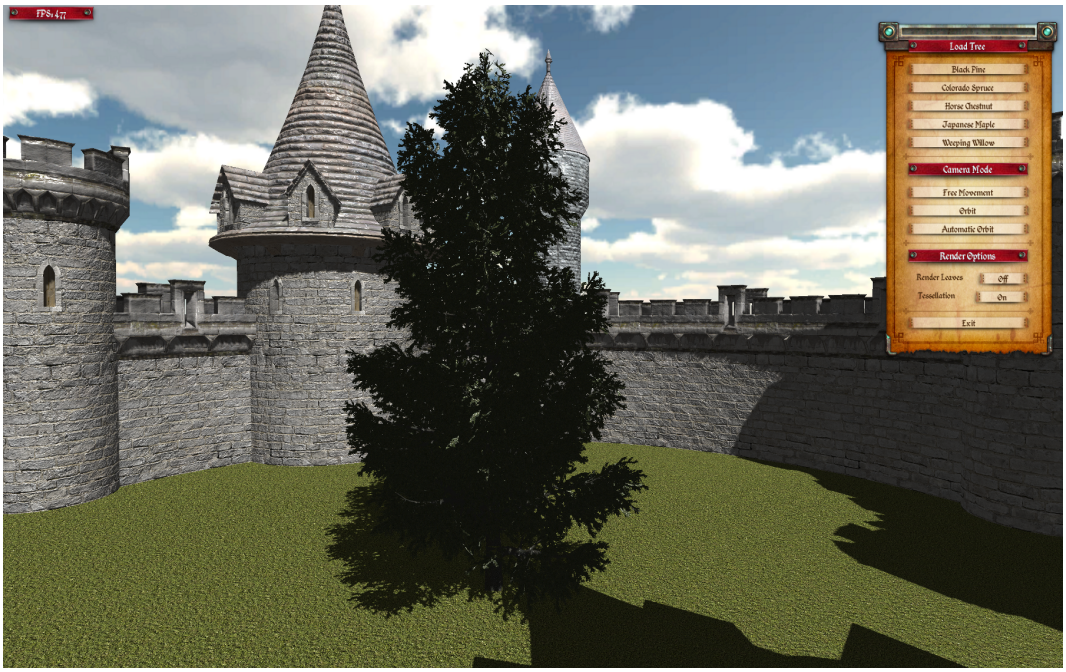


Figure 3.14: Colorado Spruce model in the Unity3D engine, Full view.

These graphics or game engines often include highly optimised rendering pipelines and physics engines capable of performing complex simulations without manual development of anything other than the applications unique processes. There are a variety of industry used game engines employed by developers around the world including examples such as Unity3D (Bladeslinger, Ghost of a Tale, Deus Ex: The Fall), Unreal Engine (Bioshock, Mass Effect, Gears of War), CryEngine (Crysis, Far Cry, Ryse: Son of Rome), RAGE Engine (GTA IV, Red Dead Redemption, L.A. Noire). To demonstrate the flexibility of the technique described in Chapter 3 and its implementation and simulation uses, this chapter focuses on integrating the remodelled trees into Unity and rendering them using their industry recognised graphics engine.

3.6.1 Model Integration

The importing of trees generated by this technique into Unity was an incredibly simple task. A short script was written to read the files output by the procedural generation

Figure 3.15: Colorado Spruce model in the Unity3D engine, close up view.

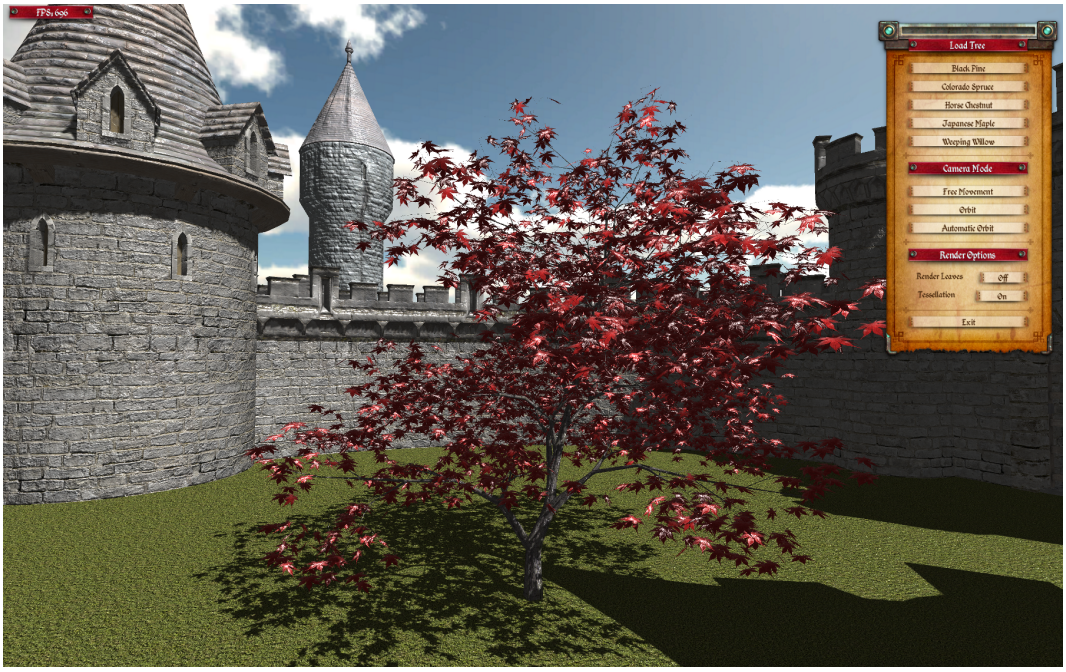


Figure 3.16: Japanese Maple model in the Unity3D engine.

Figure 3.17: Japanese Maple model in the Unity3D engine.

Figure 3.18: Horse Chestnut model in the Unity3D engine, without leaves.

stage and translate this vertex information into Unity's own mesh data structure. Once this was complete, the base model of untessellated trees is available for rendering within the Unity engine with all the previously described benefits of a model created using a single connected mesh. The process shows that with a simple step, which could easily be packaged into a user friendly plugin for Unity, the procedurally modelled trees are available to simply insert into industry used tools. Once imported, the trees can be rendered using any graphical effects expected from a commercial engine such as shadow mapping and dynamic lighting effects, bump mapping and ambient occlusion. Trees modelled using the technique described in this chapter are shown as rendered using the high quality Unity engine in Figures 3.14, 3.15, 3.16, 3.17 and 3.18.

One of the key aspects of the technique is dynamically added detail to compensate for the simple base model. In the original rendering engine and with Unity, this is performed using dynamic tessellation. Figure 3.19 and Figure 3.20 show a close-up view of the tessellated model in Unity. The built in Unity tessellation shader supports both uniform levels of tessellation and levels based on the projected size of triangles, performing the size calculations described in Section 3.4 and previously computed manually. To achieve a similar result as gained from the manual pipeline, all that must be added to the Unity tessellation shader is level of detail rendering based on the branches level in the tree hierarchy, adding more detail to the larger branches and trunk while reducing detail or discarding the smaller branches based on camera distance. This is done by importing the level of each branch from the original output and finding the lowest level branch that has weighting influence on any given vertex. This branch level is used to compute a tessellation modifier within the shader, which the tessellation level is multiplied by. The final tessellation level is computed using this modifier as shown in Equation 3.6.1,

$$T_L = T_{Lp} * \left(1 - \left(\frac{B_L}{T_{BL}}\right)\right) \tag{3.6.1}$$

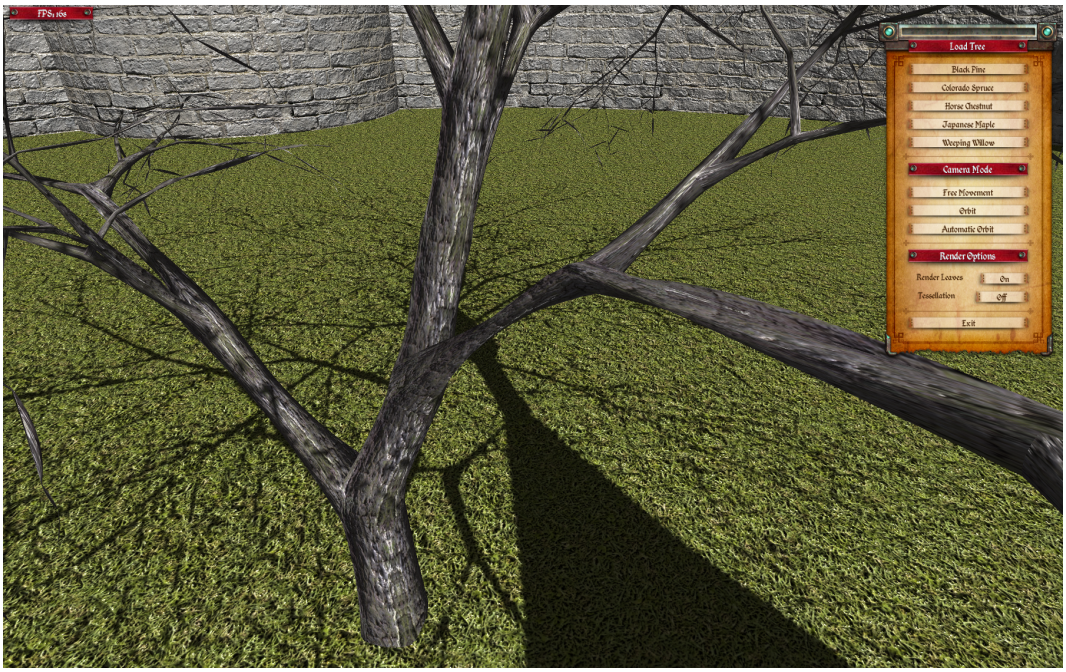


Figure 3.19: Japanese Maple model in the Unity3D engine, tessellated without leaves.

where T_L is the tessellation level, T_{Lp} is the previous tessellation level (as calculated by the standard unity shader), B_L is the branch level of the vertex (with 0 being the trunk and higher values assigned to levels of branch hierarchy branching from the trunk) and T_{BL} is the total number of branch levels within the tree.

3.6.2 Performance

Table 3.2 shows the results of framerate tests using a variety of generated trees at set viewer distance intervals. The results are taken for tree models which are both tessellated and rendered using the standard rendering pipeline without tessellation. The results show a considerable improvement using a commercial graphics engine over the test engine, showing that performance is high enough for the tree models to be used within high quality interactive applications. The results recorded for trees undergoing dynamic tessellation show a substantial improvement in framerate between the viewer distances of 20 and 40 metres, bringing the performance to a

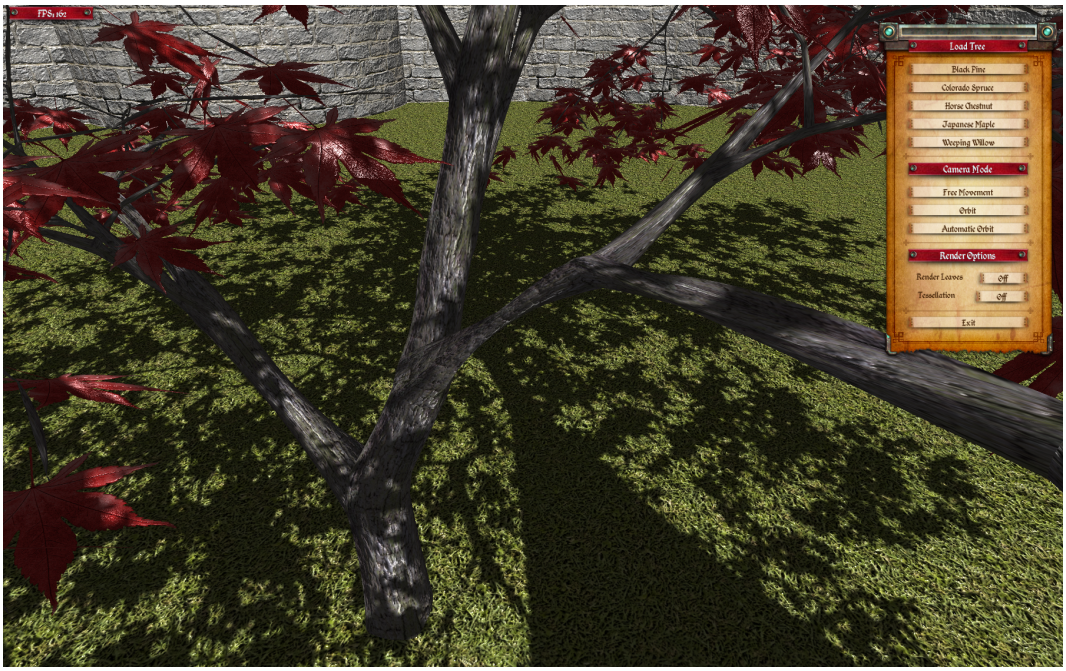


Figure 3.20: Japanese Maple model in the Unity3D engine, tessellated with leaves.

Viewer Distance (m)	Frame Rate (FPS)					
	Japanese Maple Tessellated		Horse Chestnut Tessellated		Colorado Spruce Tessellated	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
10	601.10	1.18	467.70	2.03	533.15	1.42
20	665.20	1.21	507.95	4.99	644.10	2.14
40	992.80	2.29	945.45	6.25	947.50	2.06
60	992.60	1.98	933.30	11.39	946.85	2.01
80	991.95	1.96	945.50	6.35	947.20	2.16
	Japanese Maple Untessellated		Horse Chestnut Untessellated		Colorado Spruce Untessellated	
10	1011.05	2.80	956.35	6.75	997.40	3.22
20	1020.50	1.50	986.85	7.20	1009.40	2.99
40	1021.50	1.88	989.20	7.49	1010.55	2.44
60	1051.65	1.74	990.45	6.97	1010.30	2.03
80	1020.95	3.04	990.60	7.17	1009.90	2.74

Table 3.2: Table showing performance of Unity3D engine tree rendering with varying viewer distance and tree models. Results are in FPS, distances are shown in metres relative to tree height, taken as the mean height for an adult tree.

similar level as the untessellated models. This shows that the dynamic tessellation is adding considerable detail to the model at close range, but detail is being removed as distance increases and the need for fine detail is lessened. It should also be noted that the Unity3D implementation contains further rendering techniques not included in the test engine such as shadow mapping for example, resulting in a more visually realistic output.

3.7 Conclusion

There are software packages available such as Xfrog, which provide an intuitive, highly interactive tool for the generation of virtual plants which require more artistic manipulation than purely procedural methods can allow. However the resulting geometry can be inappropriate for high quality rendering and lack vital structures for the animation and control of movement. This chapter presents a technique for extracting key data from exported trees to generate a modifiable skeletal structure and remodel the physical geometry around the structure to allow high quality visualisation and dynamic refinement using recent GPU techniques. New geometry is created to correct inaccuracies caused by unconnected surfaces and to allow consistent modelling of limbs and branch connections when motion and deformation is applied. By modelling the tree in a simplified form around the bone system, dynamic recalculation of the base mesh based on skeletal motion can be performed at a much lower computational cost. High resolution modelling of complex sections such as limb connections is performed in real-time using the tessellation engine and bone weighting is incorporated in the smoothing of the mesh and generation of new geometry in addition to its conventional use of the transformation of existing vertices.

The proposed procedure allows developers to create vegetation using an existing, industry accepted tool and to reformat its output. This allows inclusion in dynamic,

animated scenes and simulation for which it was not suitable, without any further manual interaction. The procedure uses novel techniques to base the geometry solely on the extracted skeleton, creating a simple, but highly refinable continuous mesh with automatic bone weighting avoiding many common limitations of procedural surface skinning. The resulting mesh, while able to be included with additional features in a purpose built application, can also be simply imported and used in commercial, industry standard graphics engines such as Unity3d for use in real-time, interactive applications.

Chapter 4

Real-Time Accumulation of Occlusion-Based Snow

Chapter 3 details a technique for remodelling tree models suitable for high quality simulation in normal conditions. One of the main elements that can drastically change the appearance and behaviour of a tree is weather conditions. This chapter deals with the effect of accumulating snow on complex structures such as trees as unlike many weather conditions, snow can have a lasting, persistent visual and behavioural effect as the addition is cumulative opposed to the momentary effects of wind and rain.

The effect of snow cover on outdoor scenes is both very visually striking and important for any system that simulates natural environments. Depending on context, winter scenery can be of vital importance to any number of virtual scene applications from academic simulation to video-game implementations. Snowfall and the simulation of snow accumulation poses many distinct complex problems to the rendering of natural areas, not only changing the surface properties of materials as rain would, but altering the form drastically by adding thick layers of complex snow cover to any non-occluded surface which can support it. In addition to visual build-up, snow considerably alters the behaviour and movement of structures under its influence. Considerable weight is accumulated on snow supporting surfaces and bridging of snow cover between two separate objects introduces new connective forces between

them. Natural complex structures such as trees and vegetation are heavily influenced in environments of snowfall and effective simulation of such scenes can have striking effects.

While much work has been carried out in the field of snow simulation and visualisation, as detailed in Section 4.1, there are serious limitations to the approaches currently used. One of the most noticeable effects of natural snowfall is the occlusion of surfaces, allowing areas to be sheltered from accumulating cover. This effect has been widely achieved by both geometric analysis of the surrounding area, and occlusion-based rendering techniques similar to those used in the process of projecting shadows. The main difficulty in implementing snow build-up is the storage of accumulation on a surface, due to the expensive process of mapping build-up between data structures and the copying of data between the CPU and GPU. Implementations which allow snow cover to be accumulated are most often implemented in a manner which limits their use to completely static, un-movable scenes and techniques which cannot be achieved at a speed allowing real-time simulation. Real-time implementations of snow simulations conversely are unable to store accumulation and as such, are also limited to static scenes in which animation would lead to unrealistic discontinuity of the simulation.

The novel contributions presented in this chapter are: 1) the mapping of occlusion to dynamic accumulation maps without transfer of data from the GPU, allowing the task to be performed in real-time; 2) The use of the new technique to implement a simulation of occlusion-based snow cover accumulation in real-time. Section 4.1 details the related published work forming the current techniques used in graphical snow visualisation. The technique proposed in this chapter is explained in Section 4.2 with Section 4.2.1 detailing the accumulation of snow on a dynamic scene and Section 4.2.2 covering the addition of detail providing a visually realistic environment.

The results of the implementation are discussed in Section 4.4. Section 4.5 contains the conclusions of the proposal and areas of future work.

4.1 Related Work

4.1.1 Rendering

As discussed in greater detail in Section 2.2, the rendering of snow itself is integral to the simulation of snowy scenes. An early technique put forward for the rendering process of a snow surface by Nishita et al. was a solution using metaballs, [NIDN97] whereas Yanyun et al. proposed a multi-mapping technique for the rendering of static snow covered scenes, [YSHW03]. While several solutions have been explored for the rendering of blanket snow cover, Langer et al. proposes an image based technique for the rendering of falling snow, [LZK04]. Impressive work has been performed on the simulation of ice crystal development allowing a realistic level of detail to be used on the scale of individual snowflakes. Kim and Lin, [KL03], and Kim et al., [KHL04], propose high quality methods of simulating the structure of ice formation across surfaces.

4.1.2 Accumulation

Occlusion-Based Accumulation

Occlusion is a major influence on the visual plausibility of a snowy scene. Snow should not accumulate on a surface which is blocked by another object. Shadow mapping techniques deal with occlusion by rendering a scene first from the direction of a light source, and using the depth of all surfaces visible to determine whether a surface should be rendered in shadow. Several snow techniques use a similar basis by projecting the areas to be covered with snow from the direction of fall.

Ohlsson and Seipel, [OS04], detail a technique to use deferred rendering, shadow

mapping techniques to accumulate snow onto a complex scene. The scene is first rendered from above, following the direction of snowfall, and the height of all un-occluded points is stored in a depth map. While the technique is capable of producing results at an interactive speed using a dynamic scene, the snow is recalculated between frames causing the effect that if a snow covered surface were to move under the cover of another object, any accumulated snow would be lost.

A similar method is proposed by Tokoi, [Tok06]. All upward facing surfaces are classified as snow coverage areas during a pre-processing stage and grouped into sets where no surfaces horizontally overlap. Shadow mapping techniques are then used to compute all surfaces which are not occluded and visible from the sky, using multiple projections to smooth the boundaries, simulating falloff and flake flutter. A stability test is performed on the surfaces as a final stage before rendering. Fine details of the scene cause difficulty in defining a snow profile due to the resolution of the snow maps. While the solution is capable of interactive frame rates at 3.38fps achieved at a rendering resolution of 300x300, the proposal is considerably too slow for real-time application. Foldes and Benes, [FB07], present a technique for rendering snowy large scenes from a great distance by using ambient occlusion and direct occlusion to determine snow-melt. The solution provides a very high quality result although rendering is not done in real-time and the scene cannot be dynamic without requiring a full re-computation of the snow cover.

Geometry Based Accumulation

There are multiple techniques which focus on assessing the surrounding geometry to determine snow accumulation. A technique proposed by Fearing calculates snow accumulation for any given point using particle projection, [Fea00]. The method projects particles upwards from defined snow sites across each surface, recording which particles reach the sky without collision. The technique is, however, limited by the

requirement of a completely static scene with snow not being able to be accumulated on dynamic or animated objects. Another particle system approach is proposed by Haglund et al., however despite being simple and easy to implement, the results are very primitive and do not share the realism or accuracy of other methods, [HAH02]. Festenberg and Gumhold propose an accumulation technique using height span maps of a scene, [FG09]. Starting from the highest point in the map, the maximum snowfall required by the scene is introduced and shifted to neighbouring lower patches or avalanched onto lower surfaces if necessary. Festenberg and Gumhold also put forward a method of snow cover generation based on a physical model of granular deposition, [FG11]. The results are much more realistic and convincing than those achieved in [FG09], however the computation speed is much slower, requiring an average of one minute per frame. Similarly to the majority of snow accumulation simulations, the technique requires static scenes as animation of the scene would not take into account snow accumulating on surfaces which were previously un-occluded due to a change in geometry.

Unlike the majority of occlusion-based methods explored, which render the scene's geometry using a height-map to differentiate snow covered surfaces, the proposals in this section use height-maps to triangulate new mesh geometry denoting the surface of snow cover. By leaving the snow cover as a height-map and applying it to the existing geometry at render-time, the occlusion-based proposals achieve a much higher simulation rate approaching real-time and also make it possible to incorporate further effects of snow and scene interactions, such as the disturbance of laying snow cover proposed by Sumner et al., [SOH99].

4.2 Real-Time Accumulation

The technique proposed in this chapter uses traditional GPU based shadow mapping approaches to determine occlusion within the scene. The environment is rendered from the direction of snowfall to compute which faces are accessible and will accumulate snow. Each object within the scene is mapped to a 2D texture which is used to store snow heights at any given point and a new approach is used for mapping from our occlusion render to the appropriate areas of accumulation maps. This allows the animation of scene objects during real-time simulation which was previously not addressed by occlusion rendering based techniques able to achieve real-time results. Snow stability on a surface is considered in the accumulation and random noise is used in addition to Gaussian blurring of the height-maps to achieve a visually realistic result. Once snow has been accumulated, normal maps for the new surfaces are computed and dynamic tessellation is used where necessary to add a high level of detail to the produced result. This technique focusses on the simulation of snow accumulation on surfaces within a dynamic scene in real-time, physical simulation of settled snow is not addressed.

4.2.1 Snow Cover Generation

Render-Based Occlusion

The initial stage of accumulating snow is determining which surfaces are visible from the direction of snowfall and un-occluded, and therefore able to accept snow cover. A technique traditionally used during real-time shadow mapping of scenes is to render the environment from the position of the light source, allowing the rendering process to test the faces based on depth and produce a fully occluded picture. This technique has been used in snow simulation to great success, [OS04] [Tok06] [FB07], incrementing a global value of snow height and determining at render time whether a

Figure 4.1: Occlusion render, highlighting surfaces directly visible from above (in red).

point should be covered by snow, as shown in Figure 4.1. With the inability to store this information in surface bound accumulation data structures, this causes the snow occlusion to be recalculated at every frame. Given this limitation, should a dynamic scene be required with animated occluding surfaces, the areas behind the surfaces which were unable to accumulate snow would move with the occluding objects, creating an unacceptable and unusable visual effect. In order to make this usable with a dynamic scene, snowfall must be mapped from the occlusion render to surface bound accumulation maps in real-time.

To store snow accumulation, 2D buffers are mapped to the surface of each object in the scene, unwrapped so that no two faces overlap and any given point in the buffer

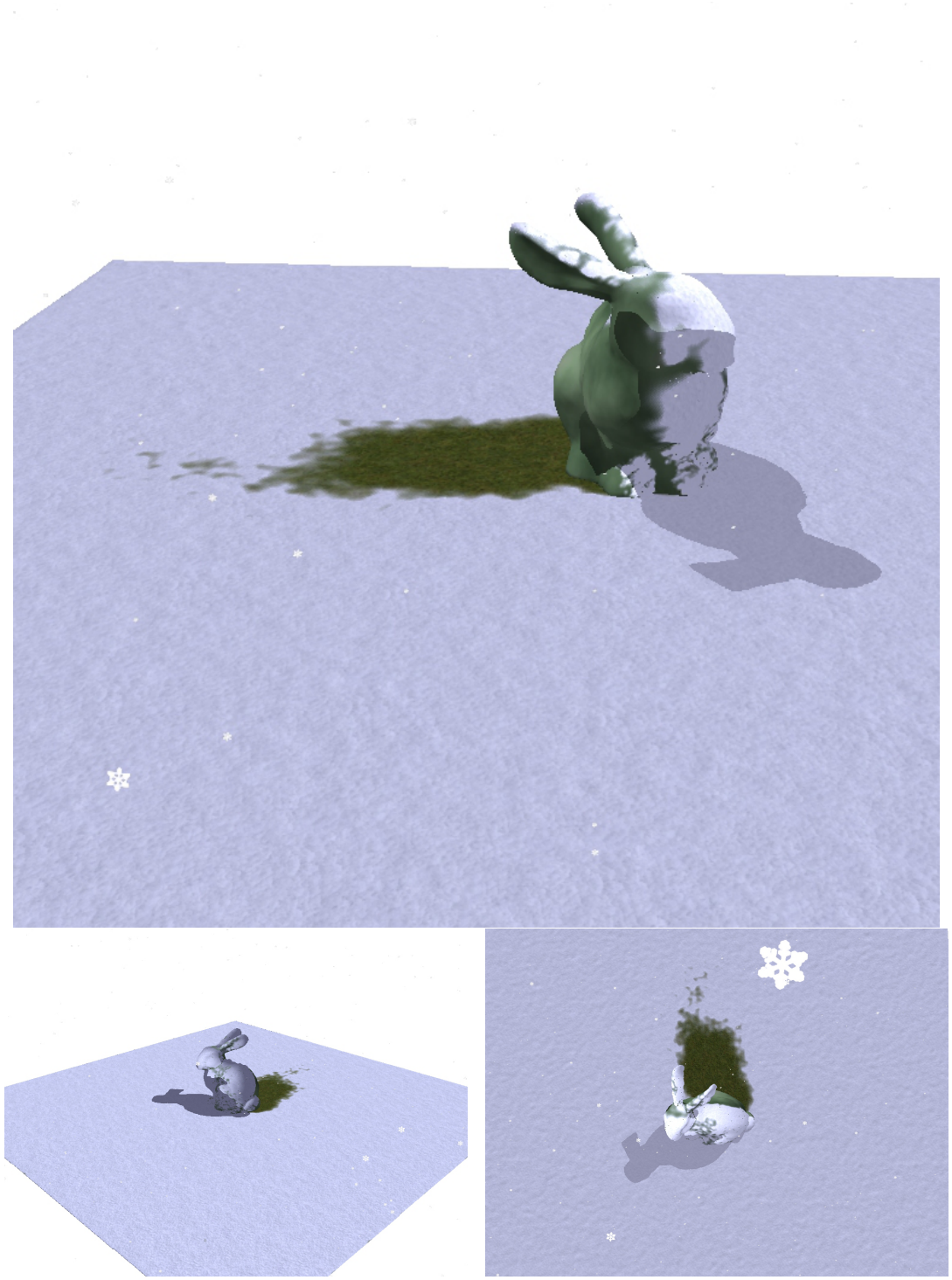


Figure 4.2: Snow occlusion, shown by the dark area of grass behind the model, projected at a sideways angle due to the inclusion of uni-directional wind forces.

corresponds to only one point on the surface. The generation of these unique UV co-ordinates is not a trivial problem, however it has been solved and used in processes such as generating object light maps and texture atlases as shown by Lévy et al., [LPRM02]. This unwrapping can be performed as a pre-processing technique on the object in the scene and as such has little impact on the real-time process. Binding the accumulation buffers to the objects allow the surfaces to be moved and animated, moving the accumulation maps with them. Texture resolution is determined at compile time to the necessary level of detail, with a higher map resolution giving a greater level of detail in accumulated snow. By altering the direction of the occlusion render, a differing snowfall direction can be introduced at no extra computational cost to incorporate the effects of a varying, uni-directional wind force if required, as shown in Figure 4.2. In addition to altering the direction of occlusion render to incorporate strong wind forces, a random offset of a small amount is introduced at each frame to simulate the behaviour of "flake flutter", where a snowflake will move throughout the air in a non straight path potentially landing in a range of snow sites. Instead of outputting colour information as would be done in a normal render pass, or depth information as would be used in shadow mapping or currently used snow occlusion techniques, the render outputs all the information necessary to map the visible area to its corresponding accumulation buffer. In order to update the accumulation map, the information required consists of a unique ID to identify which map is being referenced, and the texture co-ordinates of the surface at the visible point to determine where within the accumulation buffer snow needs to be added. In addition, the minimum and maximum texture co-ordinates contained within the rendered texel are output, which allows calculation of the texture density. Texture density is variable throughout a scene due to differing accumulation map resolutions and object orientation, storing the density at each point allows for equal snow distribution across the scene,

Channel	Texture 1	Texture 2
Red	Unique Texture ID	Min U Co-ord
Green	Noise	Min V Co-ord
Blue	Accumulation Amount	Max U Co-ord
Alpha	-	Max V Co-ord

Table 4.1: Data output from the occlusion render, required to map a texel within the image to the corresponding accumulation texture visible at that point.

independent of individual objects. Two output textures are used to store the range of texture coordinates covered by the pixel, the unique identifier of the accumulation texture visible and a noise value which will be explained in Section 4.2.2. Table 4.1 shows the output of this data.

Accumulation Mapping

The main new technique proposed in this chapter is the mapping of accumulation to surface bound buffers. Each texel in the occlusion texture represents a square area within one of the accumulation maps which could be at any arbitrary position and of any size. GPU based parallel processing platforms such as OpenCL or CUDA make the iteration through large sets of data substantially faster, but there are still substantial overheads involved. The arbitrary nature of referenced areas, maps and the number of maps make the task non-trivial by any GPU based method and the technique described here focuses on exploring a render-based approach. To enable the accumulation to be feasible, the standard render pipeline must be used to update an arbitrary area within one of multiple textures for each texel of the occlusion render. In order to perform the mapping, a render pass is performed containing a single quad for each pixel in the occlusion render. While this is a large rendering task, only simple transformations need to be performed for each, no matrix calculations or shading calculations are involved and it is possible to perform this once per frame without jeopardising the real-time requirements. A geometry shader is written for the

necessary transformation with each quad having pre-determined texture co-ordinates indexing a separate texel within the occlusion outputs. Both occlusion textures, as detailed in Table 4.1, are bound as texture inputs and the geometry shader looks up the data stored in the texel corresponding to the current quad. The points of each quad which are initially set to arbitrary values are transformed by the geometry shader to result in a quad covering the referenced area of the accumulation map as projected onto screen-space co-ordinates (ranging from -1 to 1 along both axes). If no snow supporting surface is visible at the given point, for example along boundaries around the edge of the scene, the quad is discarded entirely and not sent through the rest of the render pipeline. This stage is detailed using pseudo code in Algorithm 1 with all necessary shader inputs and outputs at each stage.

Multiple render targets are used to bind all relevant accumulation textures as output of the pipeline. Once each quad has been transformed by the geometry shader, the fragment shader compares all of the render targets with the unique identifier of the desired accumulation map which is stored in the occlusion pass. Once the correct accumulation map has been discovered the area is shaded by the fragment shader with the level of snow to be added to the point. The added level of snow is determined by Equation 4.2.1,

$$s = \begin{cases} \frac{r_n * (\vec{m} \cdot \vec{y})}{(o_2 - o_0) * (o_3 - o_1)}, & \text{if } \vec{m} \cdot \vec{y} \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.2.1}$$

where s is the amount of snow stored in the texels determined by the occlusion map, r_n is the random noise value, $\vec{m}$ is the normalised surface normal, $\vec{y}$ is the unit vector along the positive y axis and o is the column vector produced by the second output of the occlusion render, containing texture co-ordinate mapping as defined in Table 4.1. The result of this pass is a collection of textures with the same surface mapping of each accumulation buffer, each storing the areas of snow accumulation for the

Algorithm 1 Accumulation mapping shader pseudo code.

```
                                                          ▷ Vertex Shader
in: position, texcoord
out: position, texcoord
function VertexShader
    out:position ← in:position;
    out texcoord ← in:texcoord;
end function

                                                          ▷ Geometry Shader
in: position[3], texcoord[3]
out: accumulation, texID
uniform: occlusion_render1, occlusion_render2
function GeometryShader
    tex1 ← occlusion_render1(in:texcoord[0]);
    tex2 ← occlusion_render2(in:texcoord[0]);
    vec2 halfsize ← (vec2(tex2[2],tex2[3]) - vec2(tex2[0],tex2[1])) * 0.5f;
    vec2 midpoint ← vec2(tex2[0], tex2[1]) + halfsize;
    for  i = 0; i < 3 do
        glPosition[i] ← midpoint + in:position[i] * halfsize;
        i++;
    end for
    out:accumulation ← (tex1[1] * tex1[2]) / ((halfsize[x] * halfsize[y]) * 4 );
    out:texID ← tex1[0];
end function

                                                          ▷ Fragment Shader
in: accumulation, texID
out: accumulation
uniform: accumulation_buffer[]
function FragmentShader
    accumulation_buffer[in:texID] ← in:accumulation;
end function
```

current frame with full scene occlusion. While using the technique of multiple render targets does not introduce significant performance overheads above that of binding a texture, as the multiple textures are only used for output, an issue occurs when reaching the maximum number of render targets available on the GPU. Once the target maximum is reached either texture atlasing or multiple render passes must be performed. Additional objects can be included by performing the mapping pass on a new set of render targets or combining many object's accumulation buffers into a larger texture atlas, performing accumulation on several objects per render target.

This method is developed over the standard shadow mapping technique for several advantages. With shadow mapping, once occlusion has been rendered the depth of each un-occluded fragment is stored in the render, allowing each surface to be rendered finally, testing the depth at each point to determine whether it is in shadow. The persistent accumulation of snow would require that this technique renders each surface in an additional pass to update the values in each accumulation buffer. By mapping directly from the occlusion render to the buffers rather than the other way around, the addition of multiple objects in the scene requires only an additional render target to this mapping and not an entire separate render pass. By mapping only the snow accumulating pixels in the occlusion render, the number of mappings to be processed are purely dependent on the resolution of the render output and completely independent of scene complexity, allowing for greater scalability.

Snow Stability

When developing techniques for use with real-time simulation, the requirement of low computation times often forces the focus of simulation towards visual realism rather than physical realism. The stability of complex surfaces of binding intricate materials is beyond the scope of this chapter. However snow stability must be considered in order to produce an implementation with visual believability. Due to the changing

nature of snow's tension, with dry fine snow behaving almost like moist sand, and wet snow clinging like a solid structure, the angle of repose which would usually be used to calculate stability is incredibly variable. Tall layers of snow cover with overhangs and bridging between gaps in the supporting surfaces show that snow in the real world can support an edge angle of more than 90 degrees given the right conditions as noted by Doumani, [Dou67]. While steep inclines around the borders of snow cover can be allowed without adversely effecting the visual realism of the simulation, to allow the scene to be fully dynamic a system must be in place to ensure that the rotation and orientation of objects effects the amount of snow they accumulate and that snow is lost when a surface is rotated beyond vertical. In order to do this, during the updating of the accumulation map, as the current frame's snowfall is being added to the persistent snow cover, the surface's orientation is included in the computation.

As an object's accumulation map is being processed, its points and texture co-ordinates are included to give access of geometrical data for each point corresponding to the position of the map. This process is detailed in Equation 4.2.2,

$$s_r = \begin{cases} a - (a - (\vec{n} \cdot \vec{y}) * u) * f, & \text{if } (a - (\vec{n} \cdot \vec{y}) * u) \geq 0 \\ a, & \text{otherwise} \end{cases} \tag{4.2.2}$$

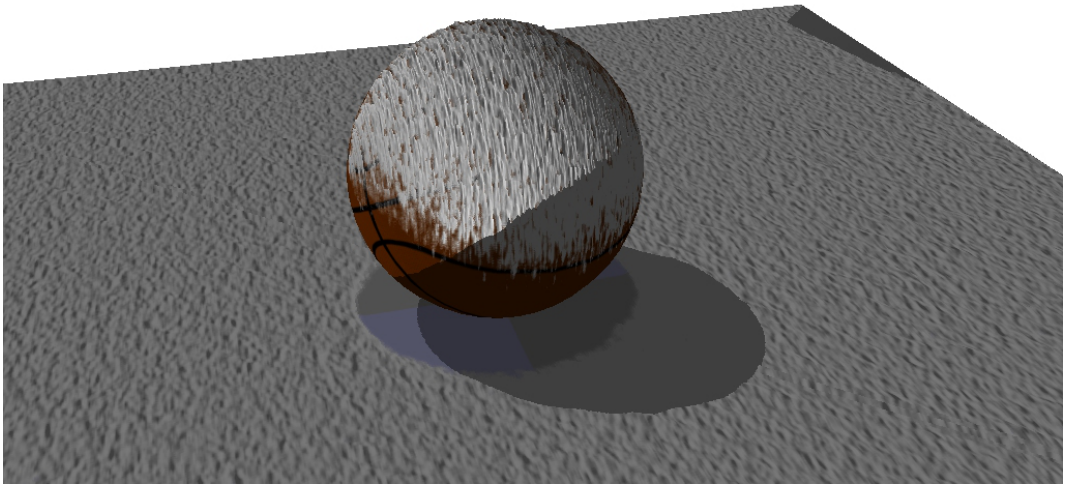
where s_r is the amount of snow retained per texel, a is the current accumulation in the texel, u is the maximum height of snow supportable in metres, f is the preset snow falloff rate such that $1 \geq f \geq 0$, $\vec{n}$ is the fragment's unit vector normal and $\vec{y}$ is the unit vector along the positive y axis. Texture co-ordinate density is also used to determine accumulation by ensuring that if a surface in the world scene has a higher texture resolution or greater density due to it being at a steep angle when viewed from above, less snow is introduced per point to allow individual snow sites to deposit equal amounts of snow throughout the scene. During the individual passes of each buffer detailed in Section 4.2.2 to allow dynamic detail, each surface is assessed to determine

whether rotation has caused the surface to point downwards, and in such cases the deposited snow is removed. Removing snow is done over the course of several frames by subtracting in increments, giving the effect of snow sticking to the surface slightly in areas of high build-up. In the current stage of the work, avalanched snow is simply removed from the scene as the technique focuses mainly on the previously unsolved problem of persistent global accumulation, however this snow would be re-introduced to the scene to accumulate on lower surfaces in a more general implementation of snow simulation. As the range of maximum supporting angles is dependent on varying properties of the snow such as moisture content and temperature, the range of stable angles and rate of falloff is configurable at runtime to allow a more defined simulation. Snow stability is demonstrated in Figure 4.3, showing a basketball accumulating snow and then gradually losing it from the underside when rotated.

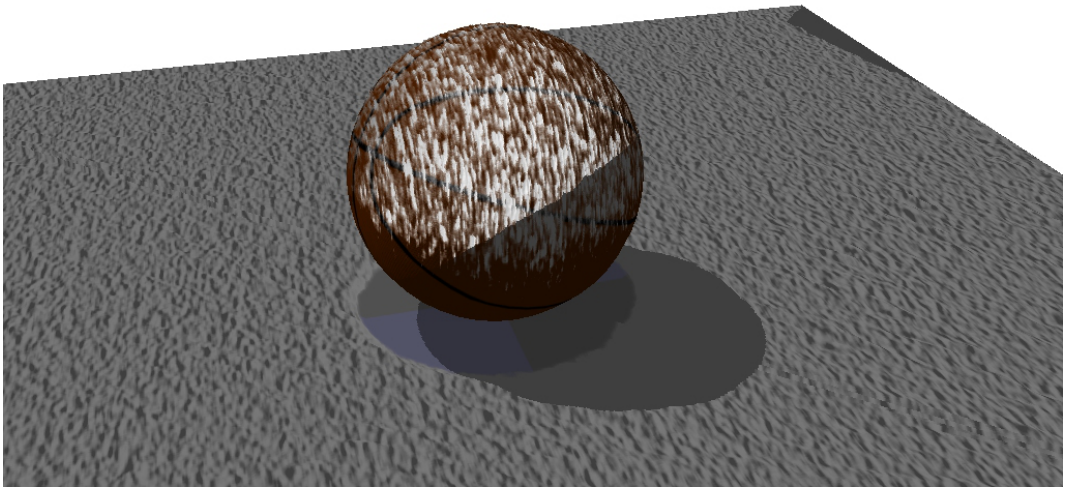
4.2.2 Dynamically Adding Detail

Noise, Blurring and Normal Mapping

Implementation of the technique described in Section 4.2.1 would create a scene with even snow cover accumulating uniformly on all un-occluded surfaces. In the real world, snowfall is not a uniform modifier but falls randomly collecting on surfaces as the combination of uncountable single snowflakes. To achieve the visual effect of the randomness of snow, a simple noise texture is introduced at the occlusion stage of the process. This uniform noise texture is mapped across the viewable area to span the texture produced by the occlusion render pass and sampled at each pixel, the noise value (a floating point number between 0 and 1) is stored in the output data as shown in Figure 4.1 and displayed on the scene in Figure 4.4. The noise texture is generated once at runtime by filling a texture buffer with normally distributed random values to a given density and then reused every frame with an offset into the



(a) Snow accumulating on a rotating round object.



(b) Snow being removed when the surfaces rotate, resulting in snow falling from surfaces due to gravity.

Figure 4.3: Example of snow stability on a curved, rotating object.

texture coordinates generated randomly each frame to give a non uniform variation in snow placement. The density of the noise can be used to alter the global amount of snow introduced into the scene each frame, allowing the accumulation rate to be customised and to give a regular simulation which is independent of frame-rates. By introducing the random noise during the occlusion stage, a uniform projection of the noise is applied throughout the scene. Applying noise to individual surfaces within the scene would lead to issues in mapping the effects evenly with differing sizes and texture densities throughout the environment. The noise value stored is used during the mapping of each pixel of the occlusion render to the accumulation maps, adding a level of optimization as pixels which contain a zero noise value (a considerable number of them) can be discarded entirely and not mapped to their corresponding height buffers.

By accumulating snow using a projected 2D grid as created by the occlusion render, hard and defined edges occur around the boundaries of deposited snow made more noticeable by the addition of random noise. In real world scenarios, deposited snow forms a smooth, un-angled surface over its supporting structure. Individual deposits of snow will dome and gradually join the surface beneath without stiff boundaries. Once snow has been accumulated onto a surface, smoothing must be performed across the generated height-map. One possible approach is to consider snow stability between sample sites with a maximum angle (angle of repose) between them, however with a possible angle of repose of greater than 90 degrees given the right condition, snow site stability would not produce a realistic smoothing in all situations. To smooth the surface of the snow cover, a Gaussian blur is performed on the accumulated height-maps. This gives a smooth edge and gradual change to levels of snow approximating the curved falloff which occurs when a collection of snow settles onto a supporting surface. Figure 4.5 shows the effect of blurring the snow height, comparing

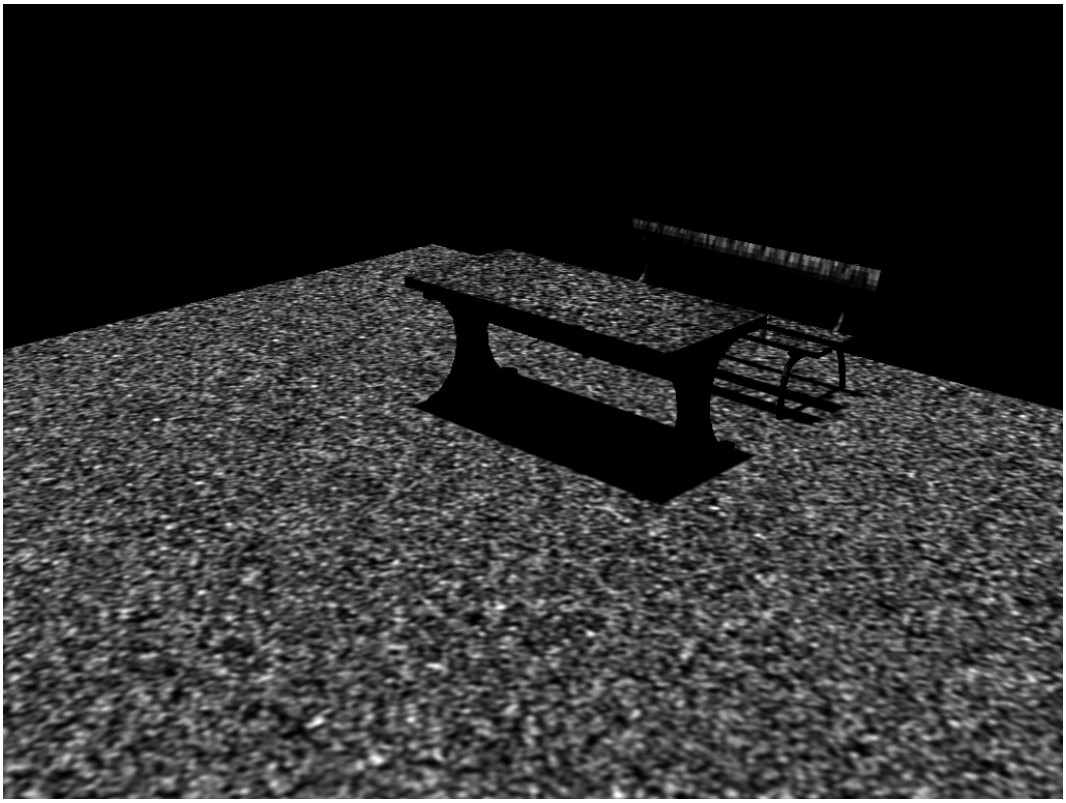


Figure 4.4: Random noise projected onto the scene from above to denote snowfall.

a scene with and without the blur pass.

While blanket snow cover can appear to be smooth and uniform at first glance, the uneven accumulation and random variation causes a noisy and disturbed surface. Given that the snow accumulation is stored as a single value height-map, the buffer can be sent through a render pipeline similar to the method used in blurring to generate a normal map from the data. The normal map shader calculates a new normal using Equation 4.2.3,

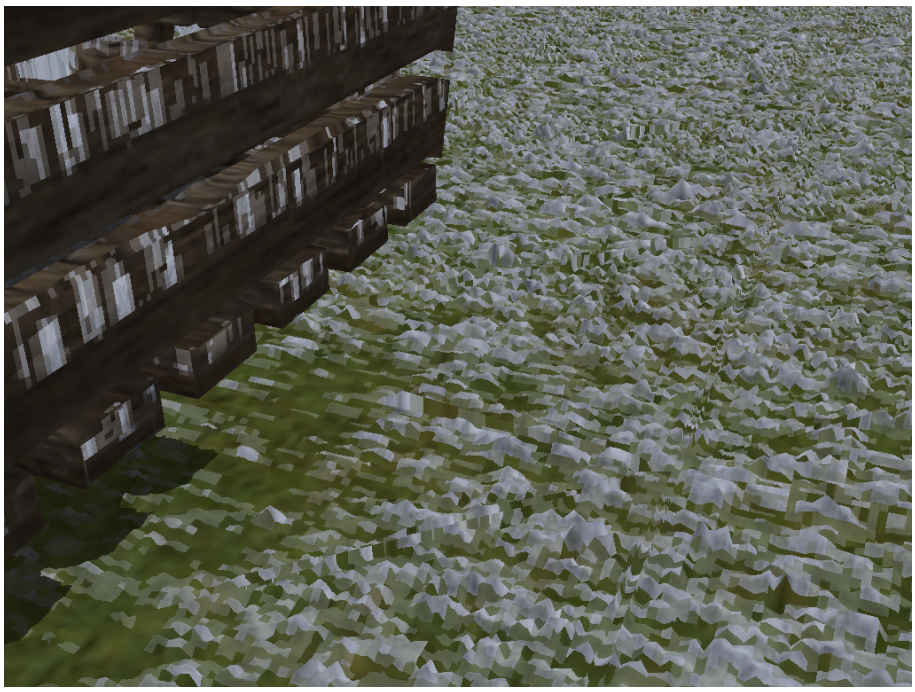
$$\begin{aligned} \vec{c}_x &= \vec{d}_y = \vec{v}_x, \quad \vec{c}_y = \vec{d}_x = \vec{v}_y \\ \vec{c}_z &= a_{x+1,y} - a_{x,y} \quad \vec{d}_z = a_{x,y+1} - a_{x,y-1} \end{aligned} \tag{4.2.3}$$

$$\vec{n_n} = \frac{\vec{c}}{||\vec{c}||} \times \frac{\vec{d}}{||\vec{d}||}$$

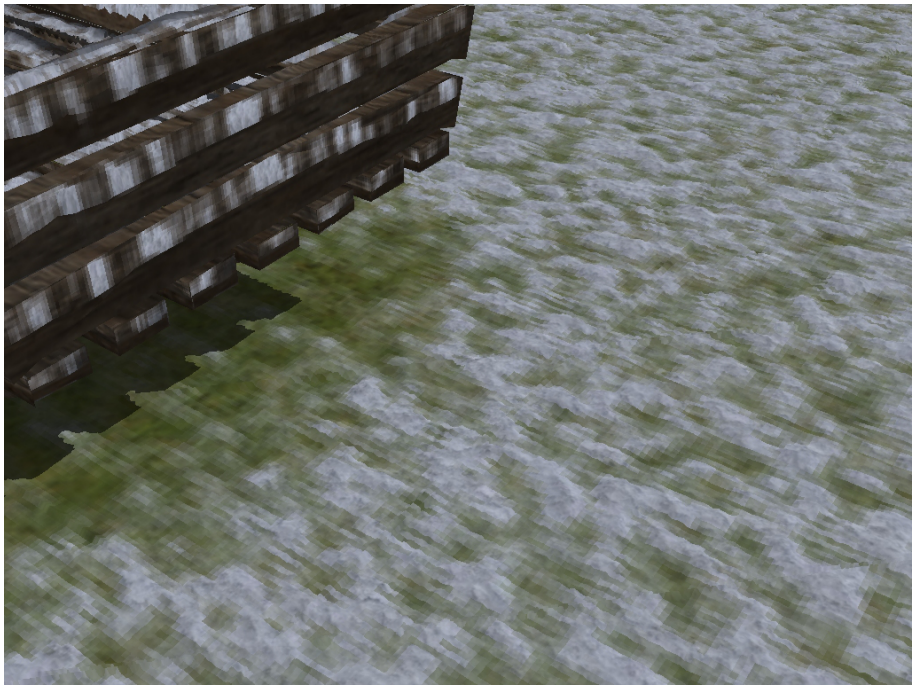
where $\vec{n_n}$ is the new normal vector, a is the object's accumulation buffer, $\vec{v}$ is a two-dimensional vector determining the measure of normal variation and $\vec{c}$ and $\vec{d}$ are three dimensional vectors defined in Equation 4.2.3. By generating high detail normal maps, per pixel lighting can be used to create very fine detail on smooth snow cover without the need for any additional geometry, as shown in Figure 4.6. This procedure allows the production of views of a higher visual quality by re-texturing the original object mesh without introducing any new procedurally generated surfaces, allowing for efficient, real-time rendering.

Dynamic Tessellation

A common technique in the generation of snow is the procedural generation of a new mesh to visualize the accumulation. Snow accumulation must be performed in world-space, i.e. snow must accumulate on surfaces, even if they are hidden from view as with a dynamic scene, areas which are obscured from the rendering position may come into view with camera or scene movement. While accumulation must be performed



(a) Generated snow cover shown without blur filter.



(b) Generated snow cover shown with blur filter.

Figure 4.5: Comparison of scene with and without blur filter.

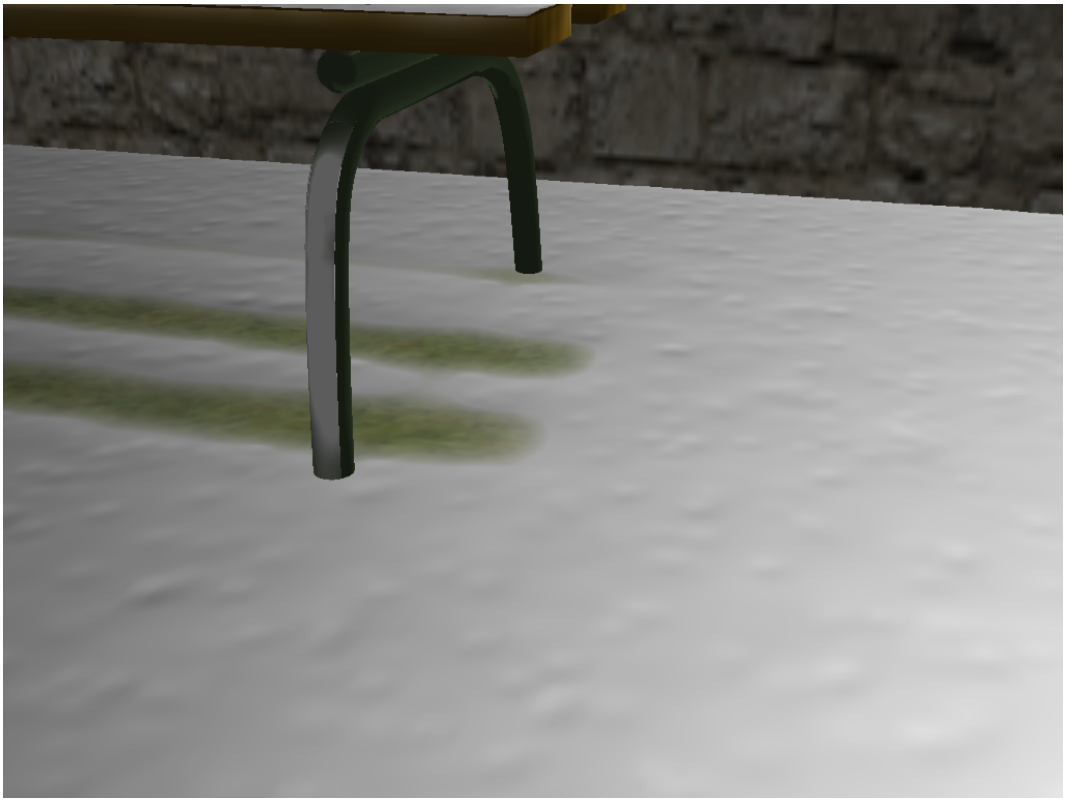


Figure 4.6: Per-pixel lighting giving a high level of detail using procedurally generated normal maps.

across the whole scene, generation and visualization of the snow surface need only be done in areas which are directly visible, allowing for screen-space optimization. Procedurally generating a snow surface which can be altered with each frame is a computationally expensive task which would not be possible to achieve effectively in real-time. As stated in the previous section, a vast amount of detail can be added to continuous snow cover with the inclusion of procedural normal map generation, this can be applied directly to the geometry of the base scene.

For areas where added geometric detail is required to produce a realistic result, dynamic tessellation of the surface is performed. While further detail across the surface of non snow supporting geometry is not required, geometric snow detail can be added view-dependently as part of the final rendering. By performing this view-dependently, unnecessary geometry in areas unseen or providing little visual benefit can be avoided. Around the edges of snow cover where the underlying surface can be seen is the area most noticeable in changing shape and the main area which requires more detailed geometric remodelling. All surfaces are displaced when rendering by the snow levels stored in the height-map, however tessellation provides needed definition along boundaries.

The main attribute used in determining tessellation level is the amount of snow the face can support. Horizontal, upward facing triangles can support the most snow and as such, require higher detail. In addition to the snow levels, the relative orientation and size of the supporting face on screen is used as the smaller a polygon becomes when viewed, the less visible detail is required when added using subdivision.

Tessellation levels are calculated using Equation 4.2.4,

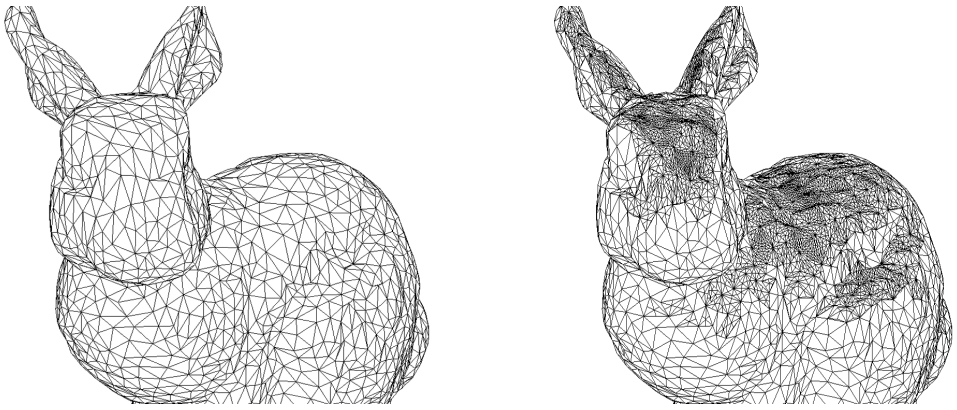
$$s_p = \begin{cases} \vec{n} \cdot \vec{y}, & \text{if } \vec{n} \cdot \vec{y} \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

$$t_f = s_p * m * \frac{|| \, (p\vec{o}s_1 - p\vec{o}s_0) \times (p\vec{o}s_2 - p\vec{o}s_0) \, ||}{2 * s} \tag{4.2.4}$$

where t_f is the final tessellation factor, m is the maximum desired tessellation level, $\vec{n}$ is the unit vector normal to the face, $p\vec{o}s_0$, $p\vec{o}s_1$ and $p\vec{o}s_2$ are the face's vertex positions in screen space, s is the screen area in normalised device co-ordinates, $\vec{y}$ is the unit vector along the positive y axis and s_p is a scalar defined in Equation 4.2.4. The tessellated mesh is then offset by the values stored in the height-map to geometrically show rising levels in snow accumulation in addition to the visually striking colour and material difference. The visual result of tessellating the surfaces is shown in wire-frame in Figure 4.7. This example is over tessellated to illustrate the result of the technique due to the geometric complexity of the base model. The maximum levels of tessellation can be controlled dynamically by applying the tessellation rate based on a minimum viewable triangle size, or as a constant limitation allowing for finer subdivision on already complex and detailed models. Figure 4.8 shows a clear offset of the tessellated surface, forming natural peaks in the snow cover over the "Stanford Bunny" test model.

4.3 Implementation

The implementation of the technique described in this chapter and the access and the reading from and writing to accumulation buffers at multiple points throughout the workflow is described in Figure 4.9.



(a) Snow cover un-tessellated, wire-frame render.

(b) Snow cover tessellated, wire-frame render.

Figure 4.7: Comparison of "Stanford Bunny" model with and without dynamic tessellation.

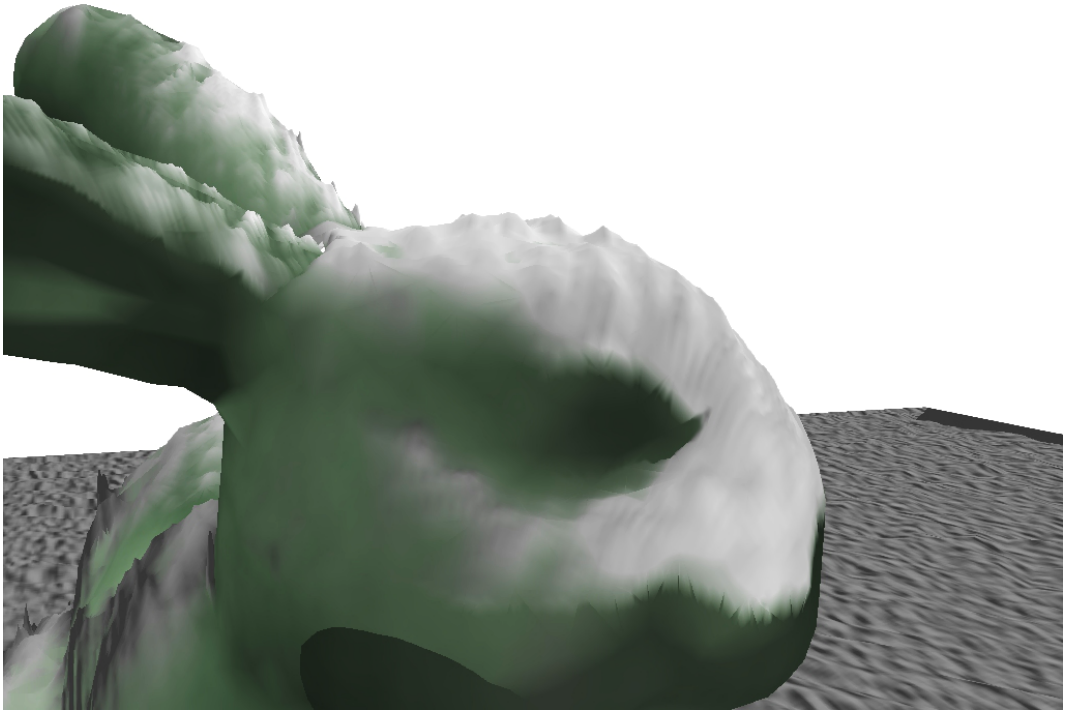


Figure 4.8: Snow height rendered as offset tessellation, forming peaks.

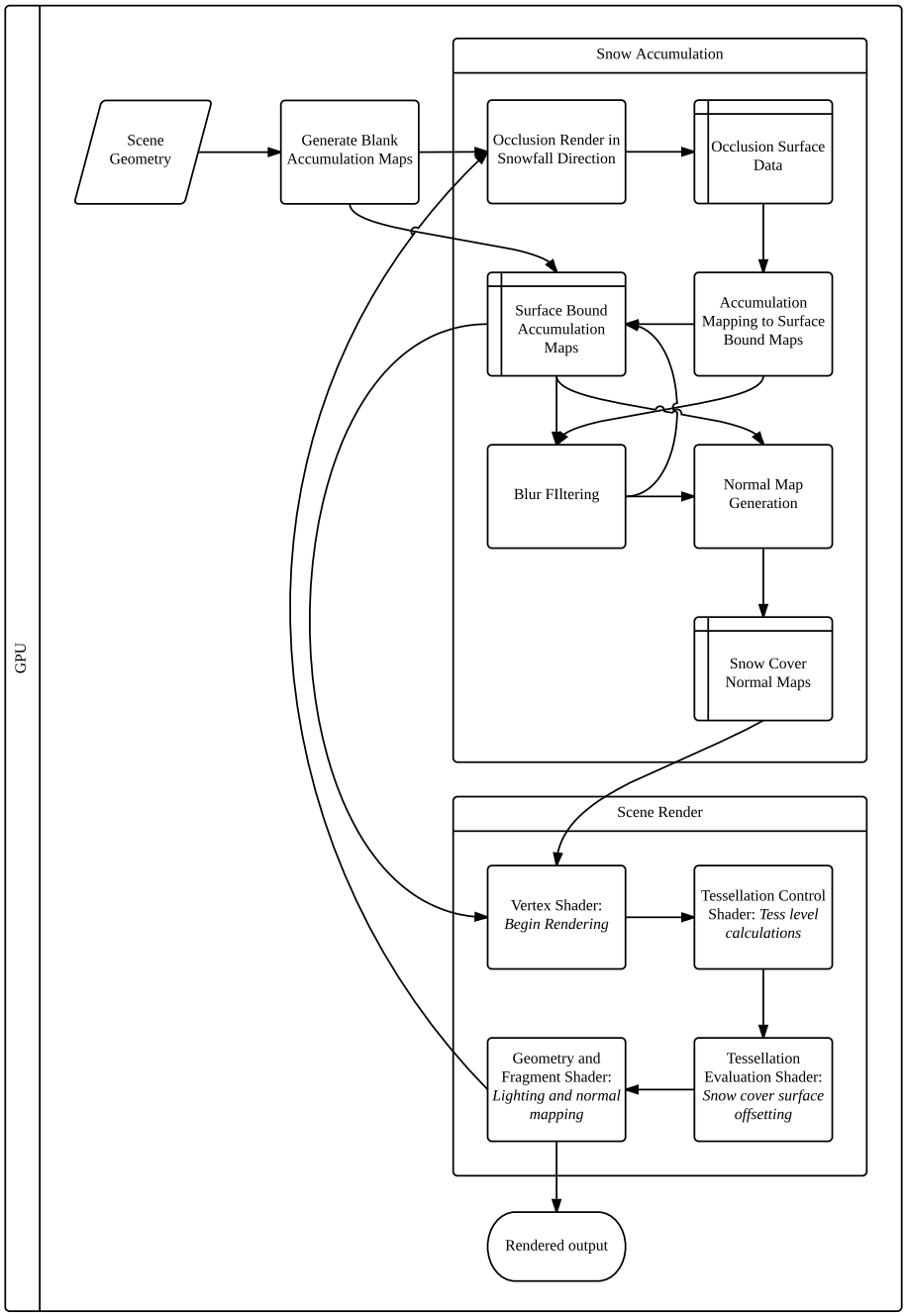


Figure 4.9: System diagram showing the process of snow accumulation and access of stored data buffers throughout the workflow.

4.3.1 Accumulation Mapping

The technique was implemented using the C++, OpenGL and GLSL languages to make use of modern graphics programming features allowing for the possibility of real-time application. Frame Buffer Objects (FBOs) were used extensively to allow rendering tasks to be outputted to a generated texture for analysis and computation at later stages of the rendering path. By using FBO render textures, the implementation performs the accumulation using multiple render passes to accumulate all data required for the final scene render. Multiple render targets are used with the FBOs when mapping occlusion to accumulation maps, allowing multiple accumulation buffers to be updated in a single pass. Rather than altering vertex data at run time, rendered quads during the accumulation mapping are transformed using a programmable geometry shader which is computed on all rendered faces after vertex calculation. This allows bespoke mappings to be done during the render stage using only a single texture query per triangle as opposed to with each vertex.

4.3.2 Dynamic Detail

The random noise used to denote snowfall patterns is generated per pixel, assigning each with a uniformly distributed random value between 0 and 1. This gives a random distribution of snow accumulation and allows density variation to be performed at runtime by altering a threshold value determining whether snow is present at each pixel. By lowering the threshold value, a larger number of randomly distributed pixels in the occlusion render contribute snow to the scene with each frame.

Once snow has been accumulated by a surface, a Gaussian blur is applied to each texture to simulate the effect of smooth snow height transitions. The blur is performed with two separate passes, first horizontally using Equation 4.3.1 and then

vertically using Equation 4.3.2,

$$p_{(x,y)} = \sum_{i=0}^{5} a_{(x+o_i,y)} * w_i \tag{4.3.1}$$

$$b_{(x,y)} = \sum_{i=0}^{5} p_{(x,y+o_i)} * w_i \tag{4.3.2}$$

where p is the result of pass 1, b is the final resulting accumulation in a single texel, a is the two-dimensional accumulation buffer, o is the set of offsets defined as {-3.2307, -1.3846, 0.0, 1.3846, 3.2307} and w is the set of weights defined as {0.0702, 0.3162, 0.2270, 0.3162, 0.0702}. By splitting the blur procedure into two passes, the same effect is produced as doing it in one but only requiring 10 texture samples per pixel as opposed to 25 for a 5x5 blur.

Rendering of a height-map can be performed in several different ways. While such techniques as ray-tracing and standard tessellation are usable, the implementation uses a dynamic tessellation using the GLSL Tessellation Shader for a more flexible and faster approach when dealing with a constantly changing surface. The required tessellation level is calculated as described in Section 4.2.2 and performed using the Tessellation Control Shader. Once new geometry has been generated at the required complexity, surface vertices are offset from their original position by the Tessellation Evaluation Shader to geometrically simulate the height increase in addition to the texture application to produce snow.

4.3.3 Texture Resolution

When working with a pixel based application such as the snow accumulation buffers, texture resolution is vital to producing a usable but effective solution. Each accumulation map is bound using a unique mapping found as a pre-processing step, the implementation uses a constant texture resolution. Resolution is variable between occlusion render and accumulation maps, and also between individual maps but is

Scene	Frame Rates (FPS)		Polygon Count	
	Mean	Std. Dev.	Final	Base
Maple	112.15	0.36	18,742	6,834
Suzanne	108.60	0.49	21,086	3,938
Bunny	108.05	0.22	16,960	4,970
Ash	106.05	0.22	64,740	42,918
Basketball	105.10	0.54	24,152	1,106
Garden	103.02	2.67	347,454	4,594

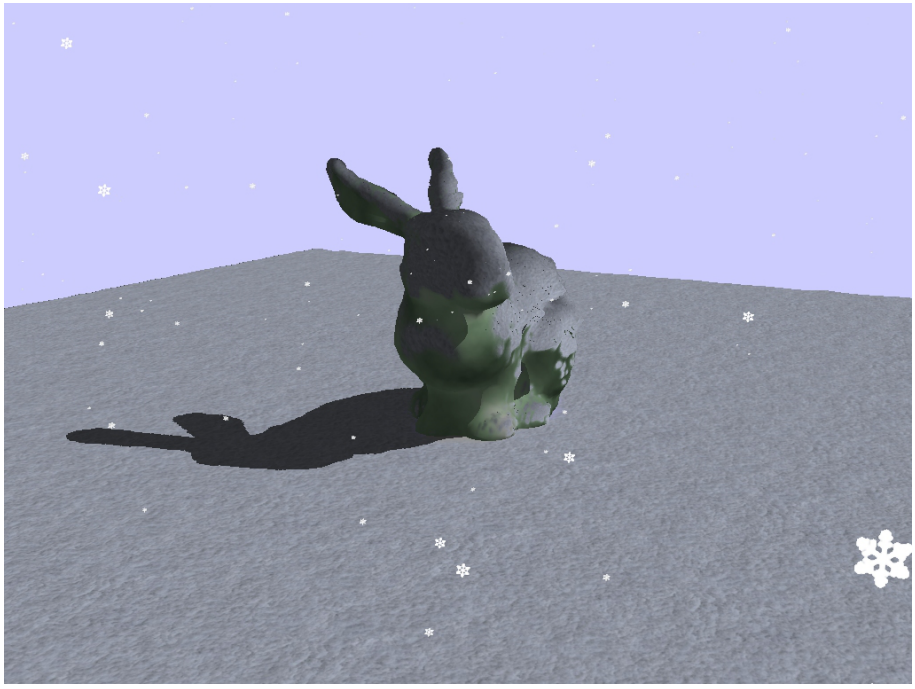
Table 4.2: Experimental frame-rates achieved with varying scenes and polygon counts. Results are in FPS, polygon counts are given for both the base model and the final tessellated render.

not varied at runtime. While using a technique of overlaid maps of varying detail such as with cascading shadow maps is possible, Section 4.4 details the experimentation of different constant resolutions and the effects they have on both computation time and visual result.

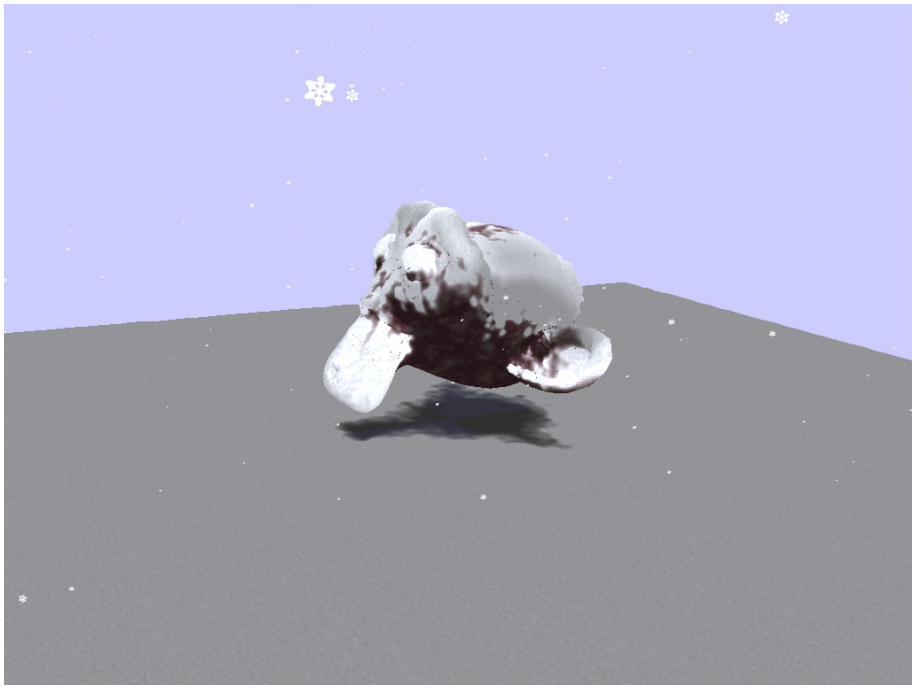
4.4 Results

Figure 4.12(a) shows the "Garden Bench" test scene used in development as it shows the effects of large, uniform areas of occlusion as well as finer detail such as the thin gaps between bench panels. The final results show that occlusion accumulation performs well as expected, the effect is a realistic, visually striking winter scene. Procedural normal mapping as shown in Figure 4.6 gives fine detail to the surface which enhances realism and the overall visual quality.

Table 4.2 shows benchmark tests using scenes of differing complexity. As the number of polygons in each scene is increased, the rendering speed of the technique varies only slightly which can be accounted for by the standard rendering of more complex models. These tests show the performance of snow accumulation to be largely independent of scene complexity and are each rendered using 1024x1024 resolution occlusion and accumulation maps. Figure 4.13 shows the details of how rendering time

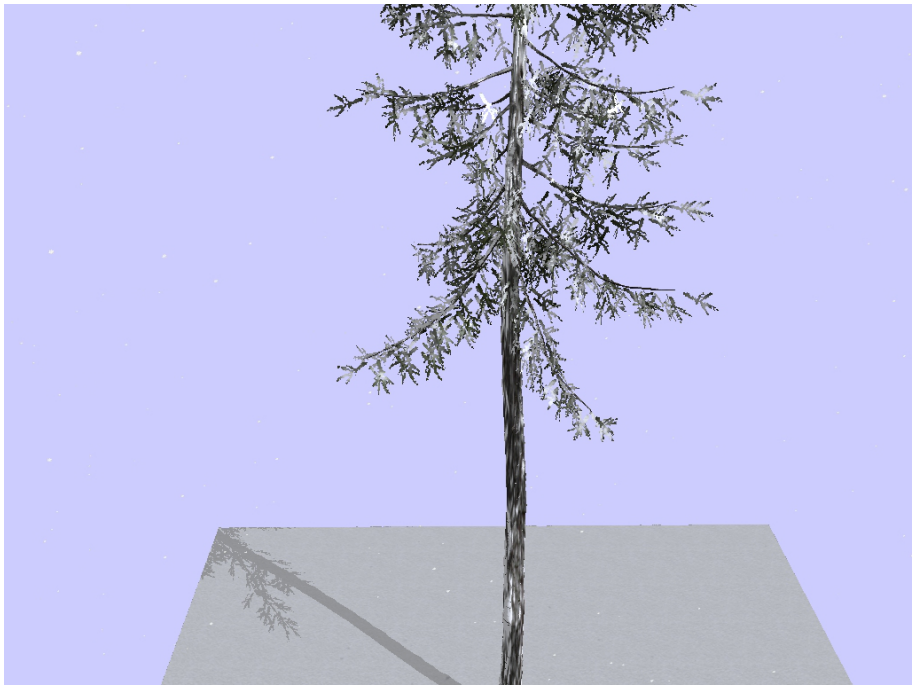


(a) “Stanford Bunny” test model.



(b) “Suzanne” Blender Foundation test model.

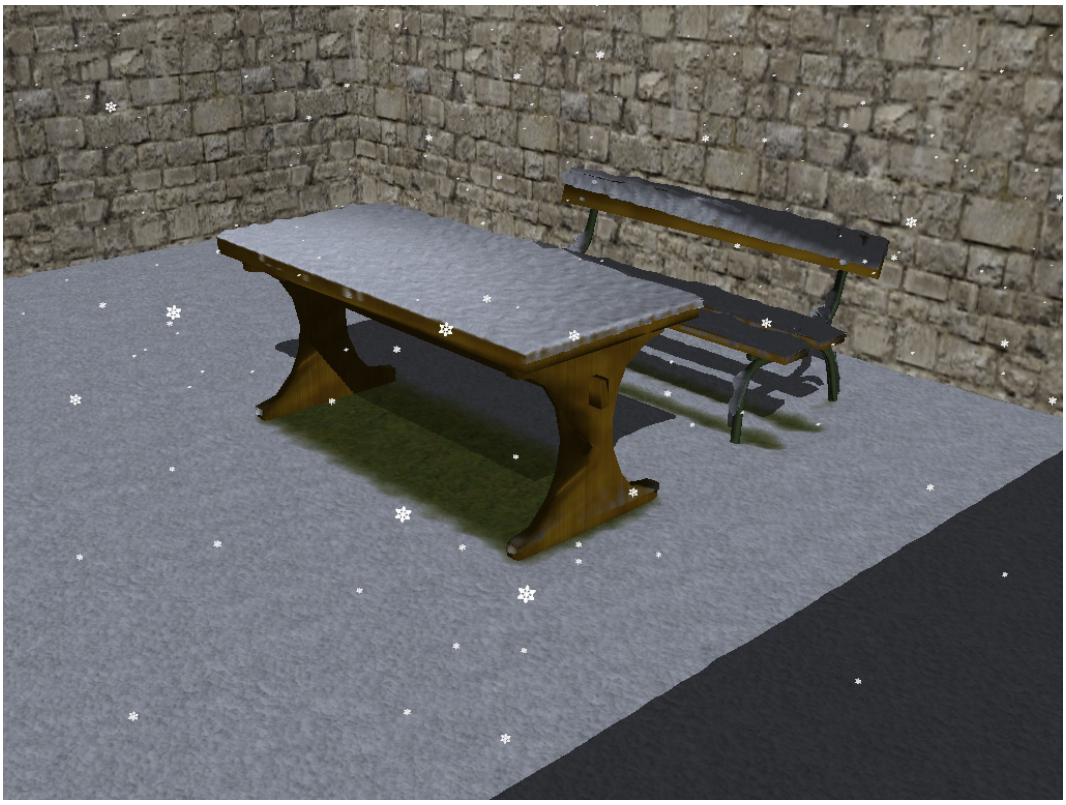
Figure 4.10: Accumulation of snow on varied scenes.



(a) Norwegian Spruce.

(b) European Mountain Ash.

Figure 4.11: Accumulation of snow on varied scenes.



(a) “Garden Bench” test scene accumulating persistent snow cover, with 1024x1024 resolution occlusion and accumulation buffers.

(b) Test scene with 512x512 resolution occlusion and accumulation buffers.

(c) Test scene with 256x256 resolution occlusion and accumulation buffers.

Figure 4.12: Rendering of the test scene at differing resolution of accumulation and occlusion maps.

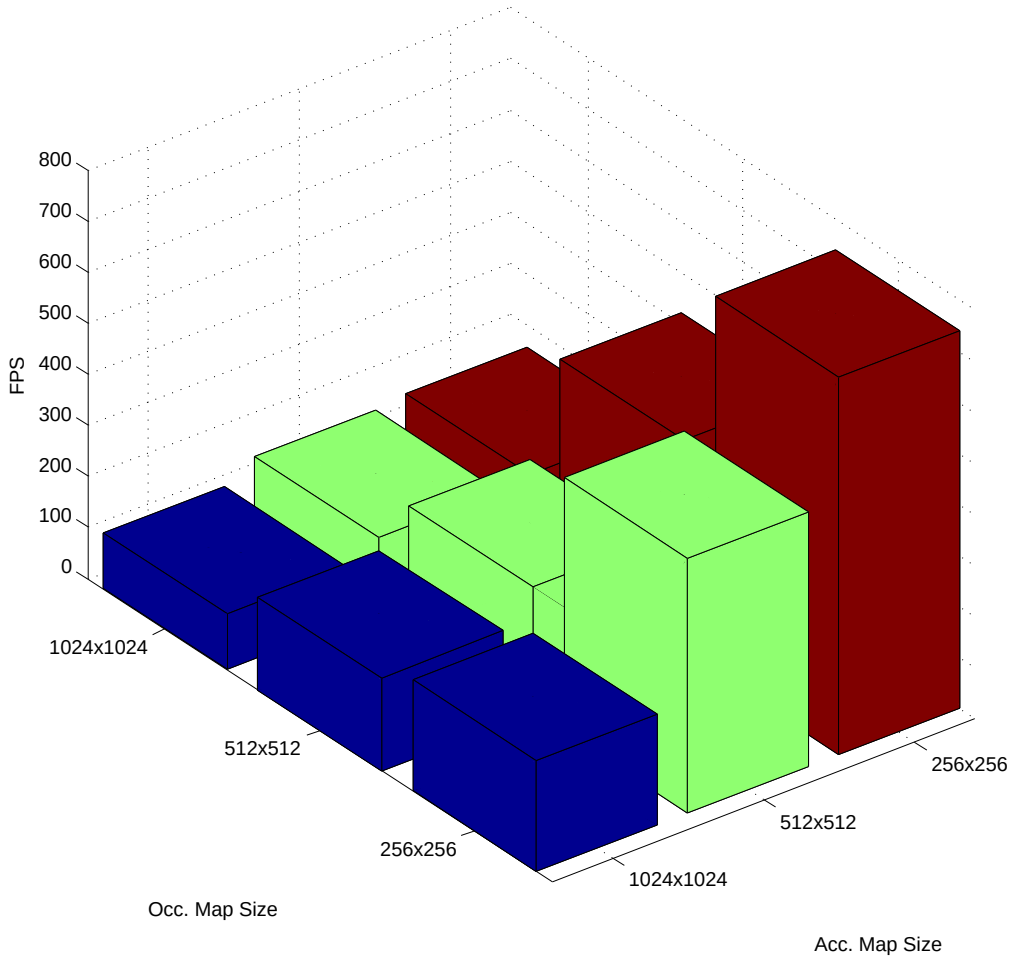


Figure 4.13: The frame-rates (FPS) with varying accumulation and occlusion map resolution.

differs with map resolution. Both the occlusion map and accumulation maps are varied using the "Stanford Bunny" scene and the plotted results clearly show the effects of each variation. Lowering the resolution of the accumulation maps has a substantial effect of frame-rates whereas lowering the resolution of the occlusion map has more of an effect the lower the other map's quality is. This suggests that the performance of the technique is more dependent on accumulation map procedures such as the blur and stability calculations than the occlusion projection stages. The visual difference between map resolution in the simulation is shown in Figures 4.12(a), 4.12(b) and 4.12(c). The similarity between Figures 4.12(a) and 4.12(b) show that buffer resolution can be reduced to some degree with minimal visual difference, losing only finer detail. Figure 4.12(c) shows that the higher the reduction in resolution, the more fine detail is lost which can be seen evidently around the edges of snow accumulation. Reduction of buffer resolution is an option when memory requirements are too high for the end system, whether using a texture atlasing technique to minimise rendering calls as discussed in Section 4.2.1 or storing separate buffers for each object, the technique adds the equivalent of two textures per object for accumulation buffers and computed normal maps. Given this usage, video memory requirement increases linearly with the number of objects in a scene as buffers can only be used by multiple objects if they are guaranteed to be influenced by the same amount of snow, i.e. not occluded, static objects. The only additional memory overheads of the technique are two textures for the occlusion render and a single VBO used for quad rendering as described in Section 4.2.1.

All tests were performed on an Intel i7 PC with an nVidia GTX 580 GPU. A particle system showing falling snowflakes is shown in Figure 4.10, Figure 4.11 as an indicator of snowfall and visual aid to snow direction, however does not form part of the described technique which focusses on snow accumulation. While the highest

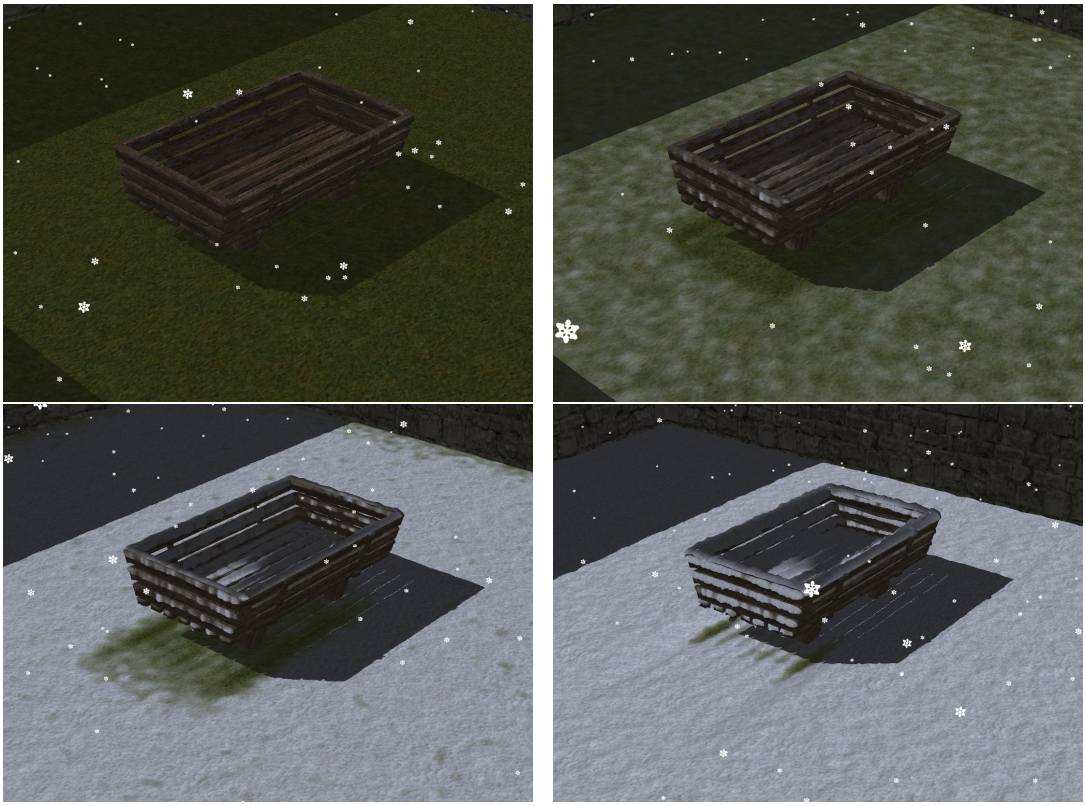


Figure 4.14: Accumulation of snow on the ground beneath a moving cart, showing areas of grass gradually revealed to the snowfall.

detail simulation performed well at real-time rates, substantial speed increases are seen when lowering the resolution of the buffers, due to limiting the work performed by the accumulation mapping. As shown in Figures 4.12(b) and 4.12(c), lowering the resolution of the buffers produces a similarly realistic accumulation of snow across blanket areas, however fine detail is simplified and lost under the bench. The main contribution of this solution is the ability to accumulate snow on a moving scene which was previously not possible in real-time. Due to the problem not being addressed in previous work, it is difficult to compare results with standard techniques. The closest example evaluated of real-time high quality snow accumulation is that of Ohlsson and Seipel, [OS04]. In 2004 they described a technique which achieved snow accumulation with a frame rate of 11 FPS when rendering to an output resolution of 900x900. Advancing hardware capabilities would suggest that higher frame-rates are achievable with a new technique but the key difference is the continuity of snow accumulation on a dynamic scene which was not approached. In previous real-time techniques, moving an object throughout a scene would compromise snow cover by recalculating and losing previous occlusions. More realistically behaving simulations used adjacent geometry to assess snow accumulation such as work done by Festenberg and Gumhold, [FG11], however these techniques were not possible in real-time situations and as such have differing applications which cannot be compared as equals. Figure 4.14 shows the effect of gradual build-up on the ground under a moving cart as may be seen in real-time applications such as video games. The gaps between wooden planks in the cart's base show how build-up varies between areas un-occluded at all times and surrounding areas which are gradually revealed to the open sky.

In simulations of extreme wind, the angle of snow fall approaches horizontal causing occlusion to be projected in a different axis and glancing contact between snow volumes and horizontal supporting surfaces. Figure 4.15 shows tests of "Stanford

Bunny" and monkey head with angle of snowfall of 80° from vertical. Areas of ground occluded by the object are projected behind and snow cover builds up on the front of the object. The images show substantial build up along the top edge of the object where occlusion begins giving an overhang supported adequately by the horizontal surface. Due to the gradual falloff rate of snow on unstable surfaces, cover is being accumulated on the front of the object faster than it can fall off causing a solid snow surface encompassing the front side of the object. This effect is optional, as the falloff rate used by stability calculations is customisable according to the requirements of the simulation. Figure 4.16 shows snow accumulation projected at an 89° angle on a similar scene. With an extreme angle of fall and high wind forces, the effect on the ground given by the accumulation and normal mapping is streaked heavily in the direction of travel. This gives the visual effect of high wind forming the surface of snow cover. At an extreme angle, an artefact is introduced by accumulation immediately behind the occluding object. This is caused by the resolution of the occlusion render and it's mapping to the accumulation maps of the surfaces. At such an angle, a larger area of the accumulation map is covered by each pixel of the occlusion render, while occluded surfaces are only calculated at the centre of the pixel. This causes snow sites to span the entire area covered by the occlusion render pixel regardless of occlusion occurring elsewhere in the area. In situations where large surfaces are projected at a very high angle of snowfall, this issue can be limited by increasing the resolution of the occlusion render where required.

One of the key limitations of the technique is aliasing issues in the generated snow cover caused by buffer resolution. As shown in Figure 4.12, the quality of snow surface generation is limited by the resolution of accumulation maps and the resolution of the occlusion render. Smaller textures used to store these elements lead to less detailed generation and a grid-like appearance to generated surfaces. In addition to rendering

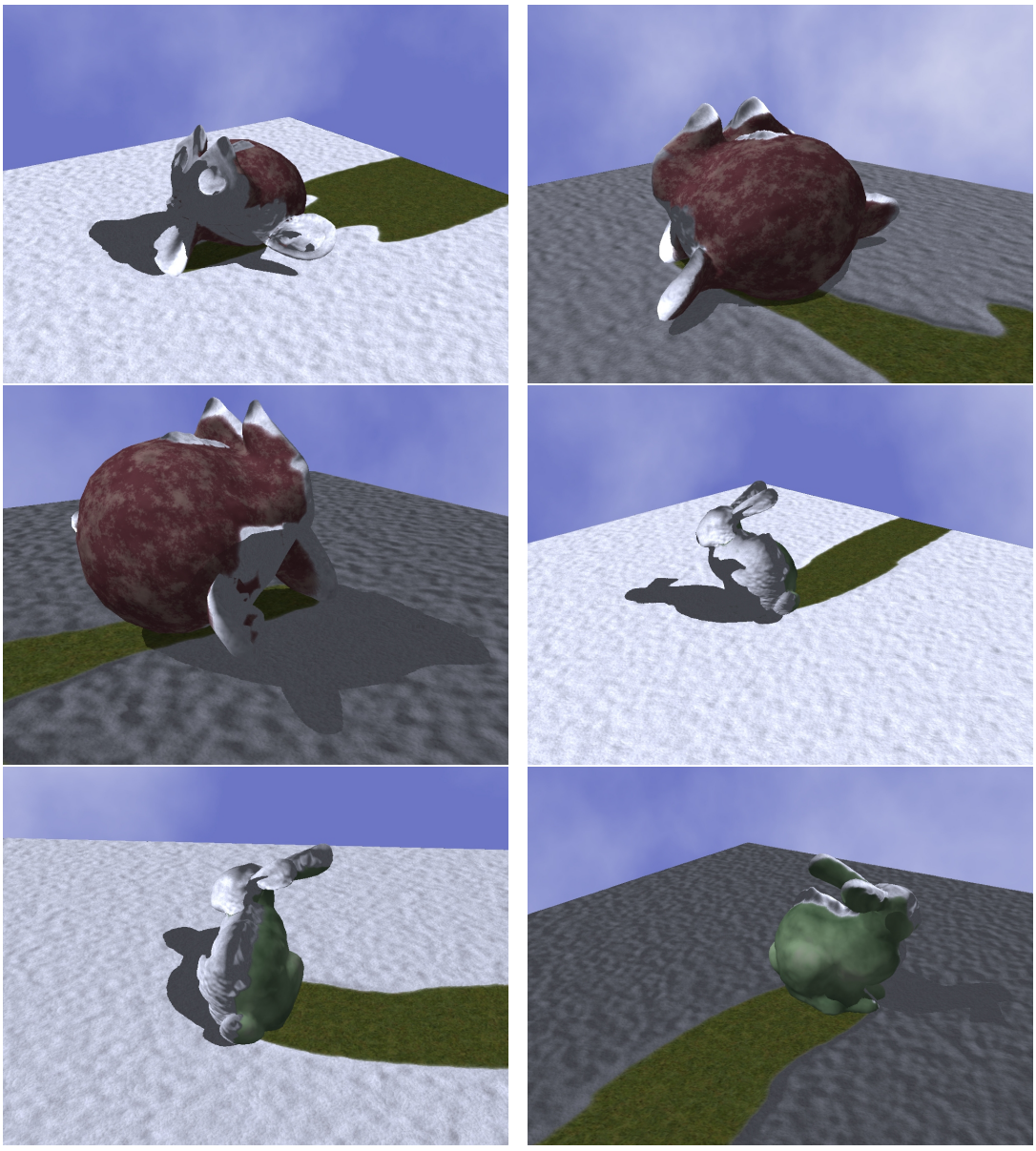


Figure 4.15: Screenshots of snow accumulation falling at a 80° angle due to wind forces.

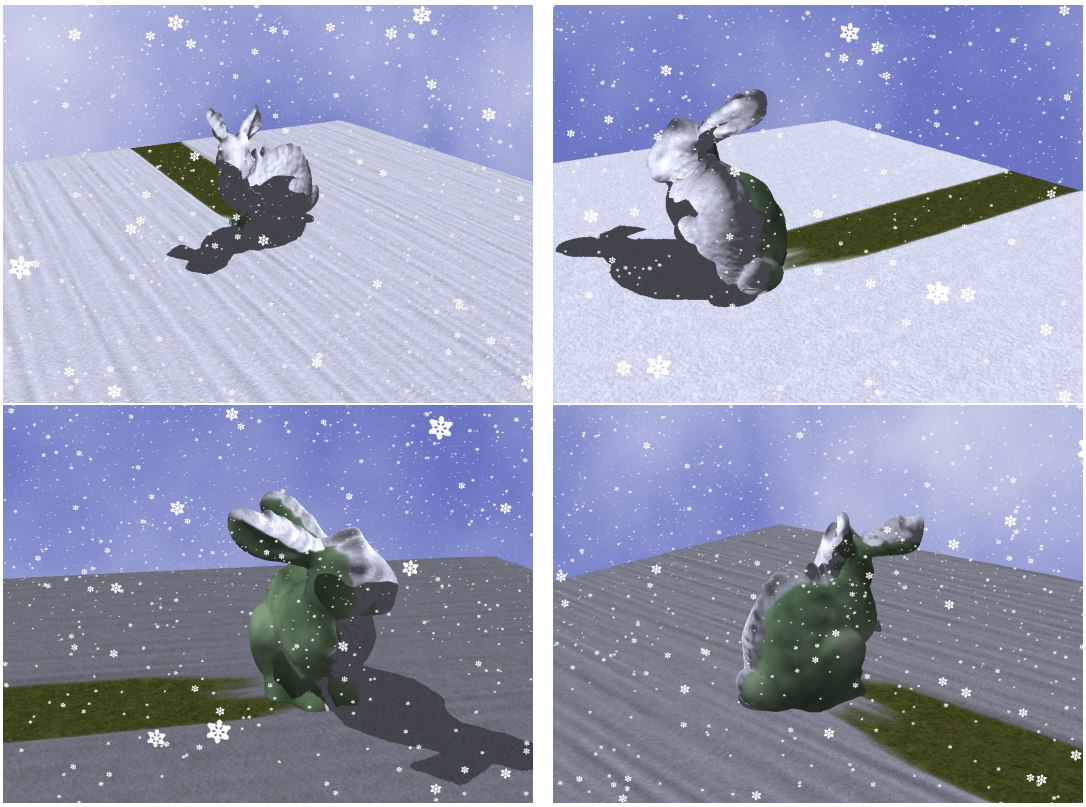


Figure 4.16: Screenshots of snow accumulation falling at an 89° angle due to wind forces.

speed, increased texture resolution imposes a memory requirement which may limit the techniques application in some situations. During instances where the projected occlusion render and the accumulation buffer are vastly different resolutions, such as when projecting snow onto a surface at an extreme angle or onto a small, complex object, the density of texels in the accumulation buffer as projected onto the texels of the occlusion render can differ greatly. This difference in texel density can cause aliasing artefacts in the projected snow cover and lead to unrealistic surfaces. The aliasing effect in the sampling of accumulation buffers is improved considerably by the inclusion of blur filtering, as shown in Figure 4.5, however in extreme cases due to a surface's size or angle, artefacts can still be present. Another limitation caused by aliasing during projection is the inclusion of extremely fine detail. Very small objects or those with very narrow surfaces relative to the rest of the scene can cause difficulty being detected and sampled properly by the occlusion render. Traditional anti-aliasing techniques such as super sampling or stochastic sampling could be employed to improve the performance of the technique in applications where thin, fine details cause such an issue. Due to limitations on graphics hardware and the memory requirements of large texture based maps, it becomes increasingly difficult to store accumulation maps and the occlusion render as a scene increases greatly in size, forcing a much higher resolution to create the same detail in snow cover relative to scene size. To address this issue, a system of accumulation map and occlusion render tiling can be implemented, separating the scene into distinct areas to be processed independently, without compromising the dynamic capabilities of the scene. The memory requirements of the surface-bound accumulation maps cause further issues with the scaling of scene complexity as with each object introduced, a separate accumulation map and normal map must be stored to allow snow to be accumulated on the surfaces of the object. As multiple render targets are used for the accumulation mapping

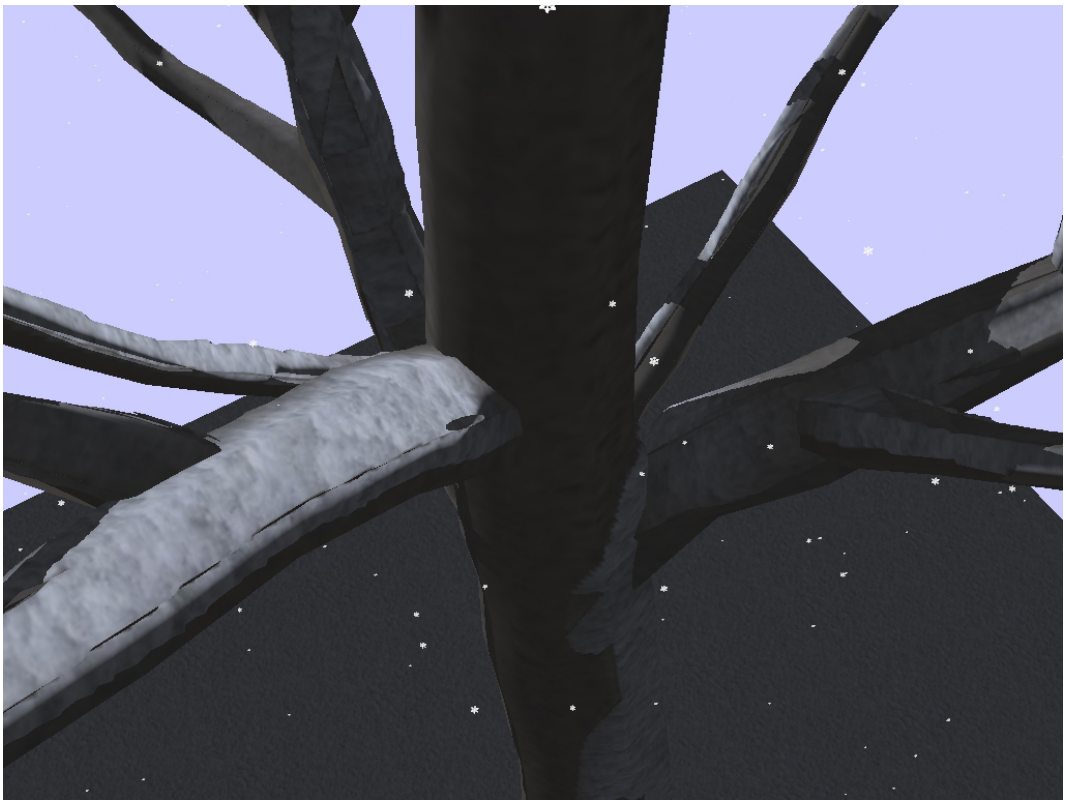


Figure 4.17: Close-up view of snow accumulation on a primitively modelled Ash.

process, graphics hardware limits the number of maps which can be accumulated to simultaneously to the maximum number of render targets. Should the number of accumulation maps surpass the maximum number of render targets, the mapping stage must be divided into multiple passes to process all supporting surfaces. The memory requirements and multiple mapping pass requirements of more complex scenes can be limited by texture atlasing, a technique of mapping multiple surfaces of multiple objects to a single unique texture, reducing the required number of textures to display all surfaces.

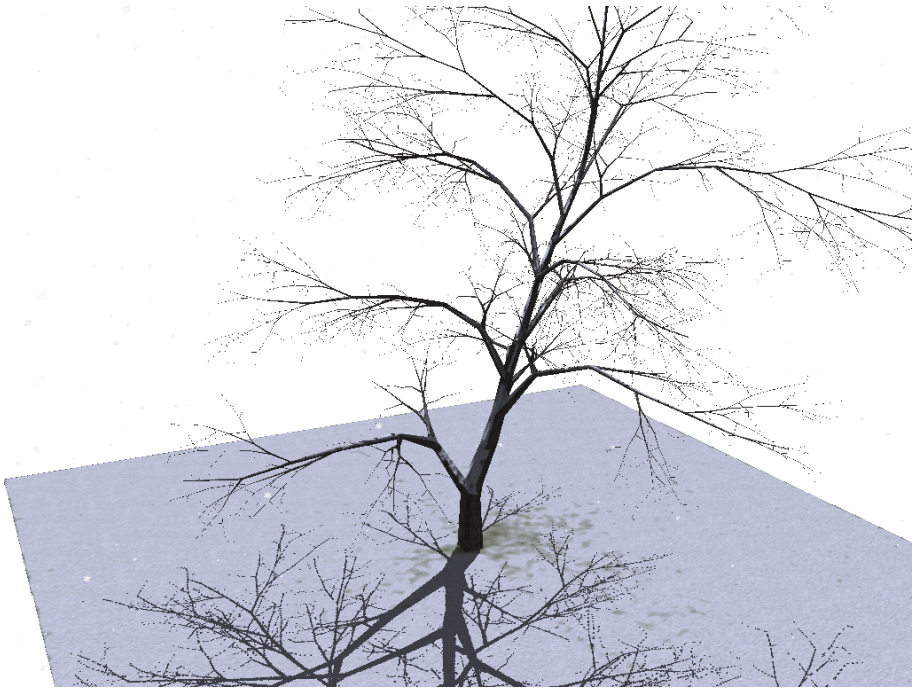
4.4.1 Snow Accumulation on Remodelled Trees

Figure 4.17 shows the technique applied to a primitively modelled European Mountain Ash tree. The tree, modelled using the Xfrog software package, is comprised of many

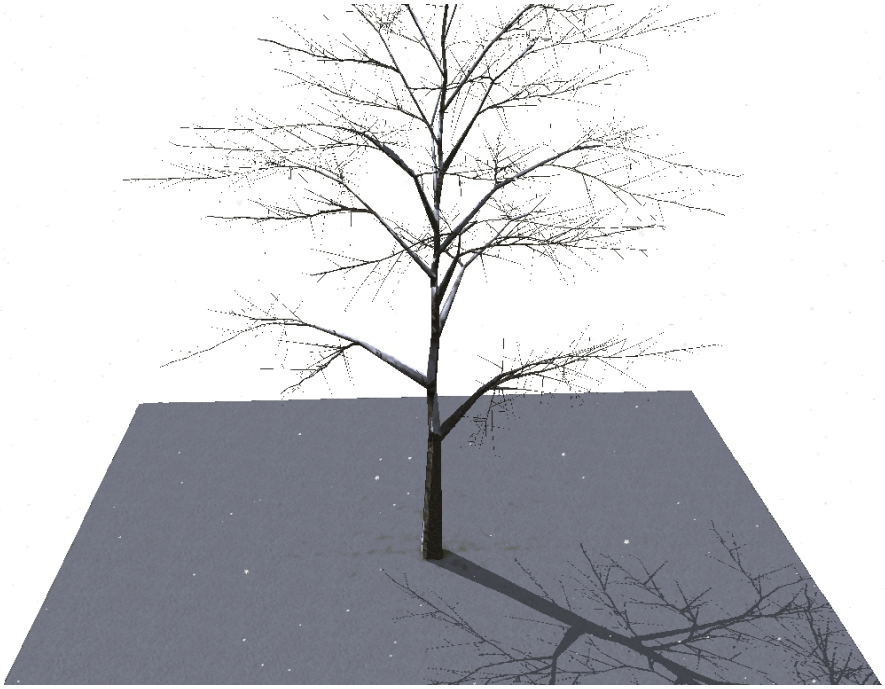
pieces of distinct geometry intersecting each other rather than a continuous mesh, with leaves displayed as billboarded images. This discontinuity of surfaces causes an unavoidable disjoint in map coordinates. The sudden change in texture mapping causes the accumulation technique to produce hard lined edges which accentuate the discrepancies. With the connection of normal, continuously meshed objects such as those in Chapter 3, the effects are less noticeable, but artefacts become quite visible in models which feature many intersecting surfaces in close proximity. This limitation of the described technique can be removed by the use of models featuring continuous, solid surfaces.

Figure 4.18 shows dynamic snow build-up on remodelled Japanese Maple and Horse Chestnut trees developed in Chapter 3. By modelling the complex structures as a continuous mesh, the unwrapping of the texture coordinates is not limited to geometric boundaries between elements and textures can be mapped across junctions and joints without segmentation. By mapping an unbroken accumulation buffer across a complex area of the model, this allows smooth and unbroken snow cover across the section without the artefacts caused by seams in the texture mapping. Figure 4.19 shows the result of consistent snow accumulation over branch junctions in both the Japanese Maple and horse Chestnut models. When compared to the results of a primitively modelled tree as shown in Figure 4.11 and Figure 4.17, it clearly shows the limitations of accumulation on disjointed geometry and how it can be improved substantially by modelling with continuous surfaces.

Another limitation is that image based techniques are dictated by image resolution and, while larger solid objects perform well, occlusion sampling can miss tiny details such as thin, alpha blended leaves as shown in Figure 4.11(a), causing dynamic scenes of this fine detail to require a higher than normal occlusion map resolution. However, Figure 4.20 shows the result of detailed snow build-up on geometrically thin branches



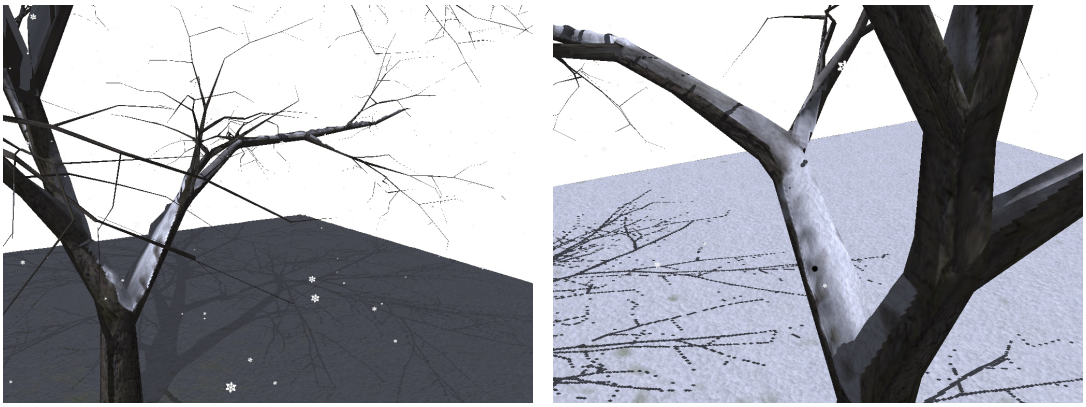
(a) Snow accumulation on a remodelled Japanese Maple.



(b) Snow accumulation on a remodelled Horse Chestnut.

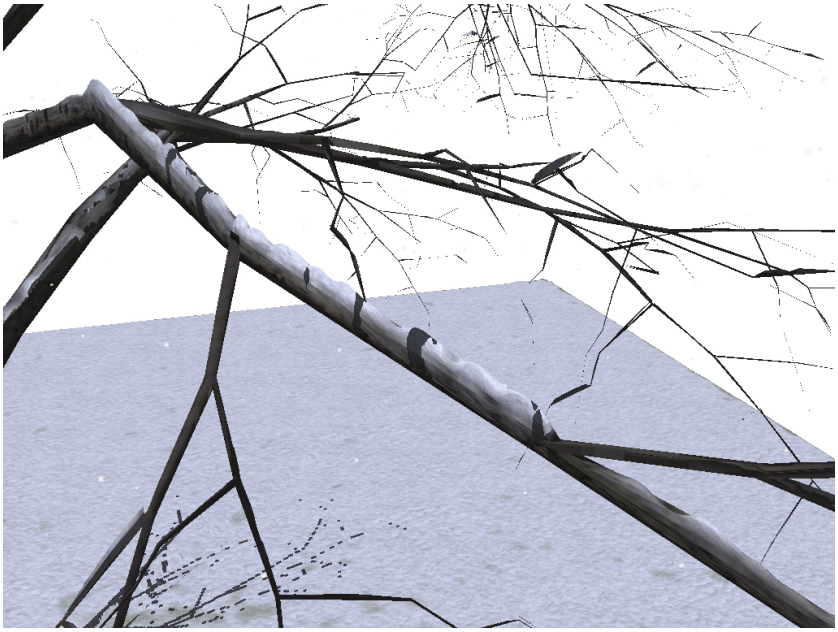
Figure 4.18: Snow accumulation on remodelled trees.

(a) Snow accumulation on a Horse Chestnut branch junction.

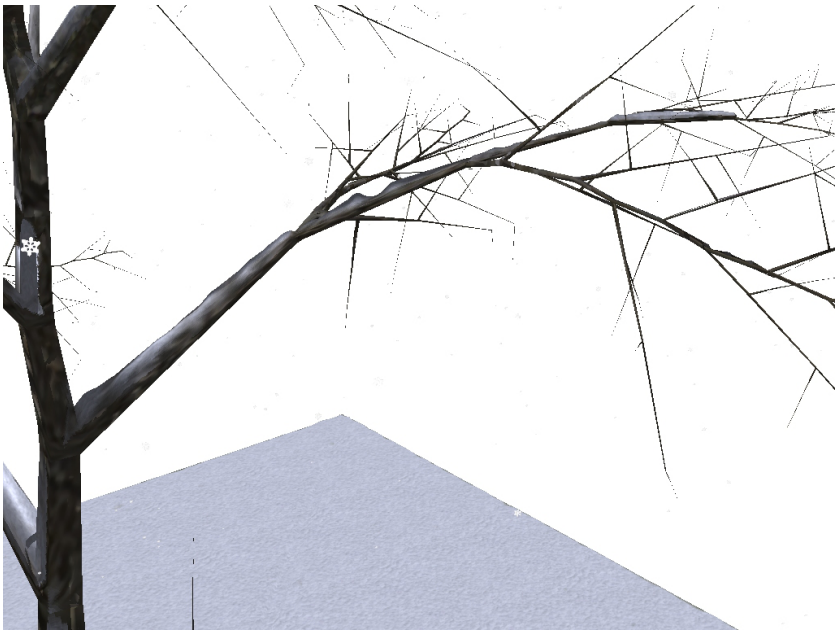


(b) Snow accumulation on a Japanese Maple branch junctions.

Figure 4.19: Screenshot showing unbroken snow cover across the branch junction of remodelled trees.



(a) Screenshot to show snow build-up on the thin branch of a remodelled Japanese Maple.



(b) Screenshot to show snow build-up on the thin branch of a remodelled Horse Chestnut.

Figure 4.20: Screenshots to show the snow build-up on the thin branchs of remodelled trees.

with accumulation maps of only a few texels wide, showing that the limitation is encountered only with extremely fine detail. This limitation can be dealt with using higher resolution for both occlusion and accumulation maps should the application require it.

4.5 Conclusion

Section 4.4 shows that using a high quality simulation of 1024x1024 buffers gives both a visually realistic result and a processing speed usable in real-time applications. Processing speed is increased by reducing buffer resolution at the cost of removing fine detail. There is no inherent limitation to the use of multiple occlusion renders and a system of differing density textures where required similar to the approach of cascaded shadow mapping would produce high quality simulation around areas of great detail while maintaining the high speed of lower detail maps where usable without reducing the quality. By basing snow accumulation solely on occlusion renders and mapping height buffers directly to object surfaces, scenes can be dynamically animated in any way without any additional computational cost to the snow simulation. Current techniques for simulating snow accumulation share the similar limitations of either not being achievable in real-time due to the computational cost of copying data across hardware devices, or unable to support persistent accumulation of snow making the approach unusable in dynamic scenes. The mapping of occlusion data to accumulation maps without the transfer from GPU, as described in Section 4.2.1, allows the updating of persistent accumulation buffers based on scene occlusion without this computationally expensive limitation. This new technique is the first to allow the simulation to be carried out on a dynamic scene of fully animatable objects, while maintaining persistent and realistic accumulation in real-time.

4.5.1 Future Work

The proposed solution allows the real-time accumulation of snow on dynamic scenes, while the build-up of snow has a striking visual effect, large amounts of snow also exert a substantial force on the surface supporting them. The ideal next stage of this research is the realistic modelling of snow movement throughout the scene in response to a moving environment and a method of surface feedback previously uninvestigated to allow accumulating snow to inflict weight on physically simulated supporting structures. By utilising similar methods of mapping accumulation to texture-bound buffers, it is planned to explore the possibility of simulating the reaction of soft-body and rigid-body surfaces to the weight of the constantly accumulating snow. In addition to the effect snow has on its surrounding scenery, the improvement of snow travel within the simulation would increase realism as currently only uni-directional wind forces can be incorporated and not turbulent flows.

Chapter 5

Conclusion

5.1 Discussion

Chapter 3 details a technique for remodelling botanical trees to give a higher quality result in real-time applications. One of the main limitations found with current tree modelling techniques is the combination of multiple mesh surfaces with disjointed junctions between them, for example, branches created using a separate mesh unconnected to the parent branch. This approach causes unwanted artefacts and errors which become noticeable when viewing from a close distance or animating the structure causing movement at the joint. The technique described in Chapter 3 addresses this issue by remodelling the trees generated by an industry used tool to create the entire branching structure from a single, continuous mesh. This technique results in clean, joined branch junctions which deform effectively and are more appropriate for use in any dynamic scene which require animation of the structure. This is possible due to the high resolution skeleton and rigging produced by the procedural method. The decision to work with a single continuous mesh came from analysis of natural trees and the surfaces they create; it was concluded that to effectively mimic the natural structure, the mesh should be based on the structure's natural state as much as possible. Single continuous meshes are optimum for simulating continuous real-world surfaces and as a tree's surface is largely unbroken, a continuous mesh would be the

best approximation. In basing the model's underlying structure on the real equivalent as well as its shape, a model is produced which is not only realistic in form, but can be smoothed, deformed and reshaped without the introduction of unrealistic artefacts.

The technique described in Chapter 3 began with extracting structure information and generating a complete skeleton before producing an element of the 3D model itself. This ensured that branching joints were based on the movable sections of the tree and any deformations were clean and without artefact. A common method in procedural tree modelling is to begin with a structure but to model around the hierarchy disjointedly, only focussing on individual branches at a time. This issue creates more problems than the previously discussed separation of meshes at junctions. A full structure enforces the natural hierarchy and branching of a tree, allowing elements to effect all others sharing a connection. The technique described in this thesis models every branch with distinct knuckle sections at the point of child growth allowing the natural point of torsion and force transferral to be deformed without distortion to the shape. This encourages modelling which adheres to the natural shape of a branch allowing both forks and changes in limb direction without unrealistic stretching of the faces and compression of the points around a curve. Considering these key points, it can be concluded that when procedurally modelling complex natural structures, using trees as a specific example, more appropriate results can be produced by adhering to the natural structure at every point of the process. Rather than basing a model purely on the visual aspect of what is being mimicked, assessment of the underlying structure, both its hierarchy and the structure of its surface, can produce a model which avoids many common artefacts and inaccuracies as well as allowing a more robust model with the capability of being used in a wider variety of applications.

Chapter 4 details a method of persistent snow accumulation on a dynamic scene. One of the main limitations of current snow accumulation techniques is that they

are unsuitable for dynamic scenes as occlusion-based snow does not accumulate persistently. In a real-world scenario, if an occluding object were to move, the area occluded would lack snow until it built up due to new accumulation, where as the newly occluded area would receive no more snowfall but retain its current level of accumulation. The technique described in Chapter 4 is the first to allow persistent accumulation with dynamic occluders in real-time. One of the key aspects of a weather condition such as snowfall is that the effects are cumulative over a period of time, rather than momentary. While simulating a cumulative effect, fidelity of the scene over time must be considered and incorporated. Scene changes occur in a wide variety of ways including a change in snowfall direction due to variable wind without any direct movement of occluding surfaces. If a snow simulation does not incorporate persistent accumulation, any dynamic scene elements would produce unrealistic and unwanted visual effects which reduce the quality of the application.

Optimising procedures for GPU parallelisation has been a popular topic for years as the number of independent cores in GPUs have risen to numbers unreachable by any CPU. Much focus is being put on systems such as CUDA or OpenCL which open the GPU to non graphical, parallel tasks and, while these approaches are fast and efficient for parallel tasks, they require an additional development stage incorporating a separate environment with its own computational overheads. Chapter 4 shows that parallel computation tasks which do not implicitly involve rendering can be performed using the rendering pipeline using innovative shader techniques. The benefit of using the rendering pipeline for geometric tasks is that all geometric data which may influence the calculations is already accessible without extra steps taken to share data, and organised by the pipeline to perform per-vertex or per-face calculations inherently. It can be concluded that while much focus is placed on alternative GPU parallelisation environments, there are still novel approaches and uses available

within in the programmable render pipeline.

The technique detailed in Chapter 4 is used to update separate accumulation maps for each object in the scene with snow height at given points. While it performs this task as required, it opens many possibilities for the techniques use with other tasks. The re-mapping within the render pipeline can be used to update any simulation which accumulates values over any given surface. The values stored in the accumulation map are used as a height-map for snow rendering, whereas they could also be used, for example, as saturation within a rainfall simulation. By mapping skeletal bone structures to texels within the map, a simulation can use this technique to calculate cumulative forces within a simulation, transferring variable wind forces which act differently in different areas of the projection, or the added weight on a structure due to mass accumulation. This could be achieved by using one of the vacant channels in the accumulation map textures to store a permanent bone id which corresponds to the weighting of the mapped face. It can be concluded from this that while the technique detailed performs adequately in the simulation of snowfall accumulation, its applications are not limited to this solution and can be applied to a number of simulations.

From both Chapter 3 and Chapter 4, it can be concluded that dynamic tessellation can play a large part in the simplification of graphics problems. Tree branches and junctions were modelled in a primitive manner in Section 3.3, consisting of four-sided tubes joined at a cube shaped junction to ensure correct mesh construction. It is possible to model high quality structures using over-simplified shapes by adding geometric detail during render, allowing the tessellation engine to subdivide necessary surfaces and smooth all transitions. In Section 4.2.2, generating a separate mesh to model snowfall is unnecessary as the scene's own geometry is altered by dynamically

tessellating the surfaces to the required complexity and offsetting its height at render, allowing for real-time simulation by removing the need to generate world-space geometry as part of the simulation. While screen-space tessellation can be used to add high levels of geometric detail only when needed, improving the visual quality of simulations, inclusion of this practice in the design of solutions allows research problems to be over-simplified in their geometric requirements without reducing the quality and usability of the output.

5.2 Future Work

Given the opportunity to continue the research further than the scope of this thesis, there are several aspects of the work which could provide interesting research. One of the key elements of Chapter 3 is inclusion and generation of a high quality skeleton within tree models. This is used to form the mesh of the tree and animate the branches, however the skeleton is of very high resolution and too complex for many applications. A system of reducing the complexity of the skeleton programmatically and dynamically at run time would allow finely detailed animation using a dynamic level of detail approach. A dynamic approach would allow the application to use the full detail of the skeleton when needed but reduce the overheads not using the entire structure. This could be performed by clamping bones to their parents and removing any simulation of them individually or offsetting the mesh by the transforms of larger parent bones rather then the particular bone the model was generated around.

Chapter 4 uses accumulation maps to store the height of accumulated snow in individual texels, future enhancements of this technique could use the accumulated snow height as a measure of mass for the snow stored at each point, enabling simulation of forces created by the weight of snow cover. By applying the additional force to

physically modelled surfaces at the point of accumulation an effective physical simulation of surface feedback could be achieved. Reducing the resolution of accumulation maps and mapping larger areas of the surface to single texels would be sufficient to simulate the accumulated weight as the force would exert as a combined force from many areas of snow. When applied to the trees created in Chapter 3, each face in the tree mesh contains information about its weighting to the bones of the tree skeleton, allowing feedback from the surface bound maps having access to both the amount of accumulated snow and its position, as well as the skeletal influences. This can be used to apply the accumulated snow weight as a force to the underlying bone structure, simulating the behaviour of the tree under the additional weight of snowfall.

In addition to modelling snowfall, the techniques described in Chapter 4 could be amended to simulate other accumulating material and natural elements such as sand or water. The technique currently simulates the behaviour of material which bonds to itself, forming a single surface, which could be expanded to simulate more granular material by adjusting and changing the stability calculations detailed in Section 4.2.1. By making the created structure considerably more unstable, an effective simulation of drier materials such as sand accumulation could be achieved in a similar manner. While the rendering of this system uses the accumulation maps as a height-map in order to render snow offset, other uses could include moisture or saturation in a water simulation. During rainfall, the water will run off uneven surfaces as well as largely accumulating in the form of soaking into permeable materials. This changes their appearance greatly and could be modelled effectively using the system put forward by accumulating moisture at any given site in the same way snow height is currently stored. The rendering of the simulation would use the information to darken wetter areas to give the effect of moisture and introduce sharp specular highlighting in addition to increasing the weight of surfaces for simulation.

Appendix A

Dynamic Tree Tessellation Shaders

A.1 Tessellation Control Shader - GLSL

```
#version 410
layout(vertices = 3) out;

in vec3 vPosition[];
in vec2 vTexCoord[];
in vec3 vNormal[];
in vec3 vTangent[];
in vec3 vBitangent[];
in vec4 vBoneWeights[];
in vec4 vBoneIndex[];

out vec2 tcTexCoord[];
out vec3 tcPosition[];
out vec3 tcNormal[];
out vec3 tcTangent[];
out vec3 tcBitangent[];
out vec4 tcBoneWeights[];
out vec4 tcBoneIndex[];

patch out int numberOfBones;
patch out int bones[5];
patch out vec3 boneweights[5];
patch out float tcTessLev;

uniform sampler2D WeightingTexture;
uniform float WeightingTextureWidth;
uniform mat4 ModelViewMatrix;
uniform float MaxLevels;
uniform vec3 FrustumPoints[6];
```

```
uniform vec3 FrustumNormals[6];

const float maxTessDistance = 20.0;
const int maxTessLevel = 5;

#define ID gl_InvocationID

void main()
{

	//Transferring data to the Tessellation Evaluation Shader
	tcPosition[ID] = vPosition[ID];
	tcTexCoord[ID] = vTexCoord[ID];
	tcNormal[ID] = vNormal[ID];
	tcTangent[ID] = vTangent[ID];
	tcBitangent[ID] = vBitangent[ID];
	tcBoneWeights[ID] = vBoneWeights[ID];
	tcBoneIndex[ID] = vBoneIndex[ID];

	int viewed = 0;

	//Performing View Frustrum Culling
	if(ID == 0)
	{
		for(int j = 0; j < 3 && viewed == 0; j++)
		{
			vec3 point, norm;
			int frusttest = 1;

			for(int i = 0; i < 6 && frusttest > 0; i++)
			{
				point = vec3(FrustumPoints[i].x,
				    FrustumPoints[i].y, FrustumPoints[i].z);
				norm = vec3(FrustumNormals[i].x,
				    FrustumNormals[i].y, FrustumNormals[i].z);

				vec3 test = normalize(vPosition[j] - point);

				if(dot(test, norm) < 0)
				{
					frusttest = 0;
				}
			}

		viewed = frusttest;
		}
```

```
}

//Calculating Tessellation Level
if(ID == 0)
{
        int texelindex = gl_PrimitiveID * 6;

        ivec2 index;
        index.y = texelindex / int(WeightingTextureWidth);
        index.x = texelindex - (index.y *
            int(WeightingTextureWidth));

        vec4 facedata = texelFetch(WeightingTexture, index, 0);

        float tess = length(vec3((ModelViewMatrix *
            vec4(vPosition[ID], 1.0))));
        float tessLevel = 0.0;

        if((facedata[2] <= 1.0 || tess < (maxTessDistance*MaxLevels)
            * ((MaxLevels-facedata[2]) / MaxLevels)) && (viewed > 0))
        {
                tess = ((1.0/maxTessDistance)*tess);
                tess = clamp(tess, 0.0, 1.0);
                tess = 1.0-tess;
                tessLevel = 1.0 + ( (maxTessLevel-1) * tess );
                tessLevel = tessLevel *
                    ((MaxLevels-facedata[2])/MaxLevels);
        }

        gl_TessLevelInner[0] = tessLevel;
        gl_TessLevelOuter[0] = tessLevel;
        gl_TessLevelOuter[1] = tessLevel;
        gl_TessLevelOuter[2] = tessLevel;

        tcTessLev = tessLevel;

        //Determining how many bones effect the face
        //Transferring their IDs and weightings to the Tessellation
            Evaluation Shader
        if(tessLevel > 0)
        {
                numberOfBones = int(facedata[0]);

                for(int i = 0; (i < 5) && (i < numberOfBones); i++)
                {
                        texelindex++;
```

```
                        index.y = texelindex /
                            int(WeightingTextureWidth);
                        index.x = texelindex - (index.y *
                            int(WeightingTextureWidth));

                        vec4 weightdata =
                            texelFetch(WeightingTexture, index, 0);

                        bones[i] = int(weightdata[0]);
                        boneweights[i] = vec3(weightdata[1],
                            weightdata[2], weightdata[3]);
                    }
                }
            }
}
```

A.2 Tessellation Evaluation Shader - GLSL

```
#version 410 core

layout(triangles, equal_spacing, ccw) in;

in vec2 tcTexCoord[];
in vec3 tcNormal[];
in vec3 tcPosition[];
in vec3 tcTangent[];
in vec3 tcBitangent[];
in vec4 tcBoneWeights[];
in vec4 tcBoneIndex[];

out vec2 teTexCoord;
out vec3 tePositionEye;
out vec3 teNormal;
out vec3 teLightPos;
out vec3 tePositionEyeUniform;
out float teTessLev;

uniform mat4 ModelViewMatrix;
uniform mat4 ProjectionMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ViewMatrix;
uniform vec4 light_direction;
uniform sampler2D DisplacementMap;
uniform sampler2D BoneTexture;
```

```
uniform sampler2D WeightingTexture;
uniform float BoneTextureWidth;
uniform float WeightingTextureWidth;
uniform float MaxLevels;

patch in int numberOfBones;
patch in int bones[5];
patch in vec3 boneweights[5];
patch in float tcTessLev;

//Function to determine the closes point on a line segment to an arbitrary
    point
vec3 closestPoint(vec3 p0, vec3 p1, vec3 p)
{
        vec3 closest;
        vec3 d = p1-p0;

        if(d.x == 0.0 && d.y == 0.0 && d.z == 0.0)
        {
                closest = p0;
        } else {
                float t = (((p.x-p0.x) * d.x) + ((p.y-p0.y) * d.y) +
                    ((p.z-p0.z) * d.z)) / ((d.x*d.x) + (d.y*d.y) +
                    (d.z*d.z));
                closest = p0+(d*t);
        }

        return closest;
}

void main()
{
        //Interpolating point data to the newly created vertex
        teTessLev = tcTessLev;
        teTexCoord = (gl_TessCoord.x * tcTexCoord[0] + gl_TessCoord.y *
            tcTexCoord[1] + gl_TessCoord.z * tcTexCoord[2]);

        vec3 norm;
        norm = (tcNormal[0] * gl_TessCoord.x) + (tcNormal[1] *
            gl_TessCoord.y) + (tcNormal[2] * gl_TessCoord.z);
        norm = normalize(NormalMatrix*norm);
        teNormal = norm;

        vec3 pos;
        pos = (gl_TessCoord.x * tcPosition[0] + gl_TessCoord.y *
            tcPosition[1] + gl_TessCoord.z * tcPosition[2]);
```

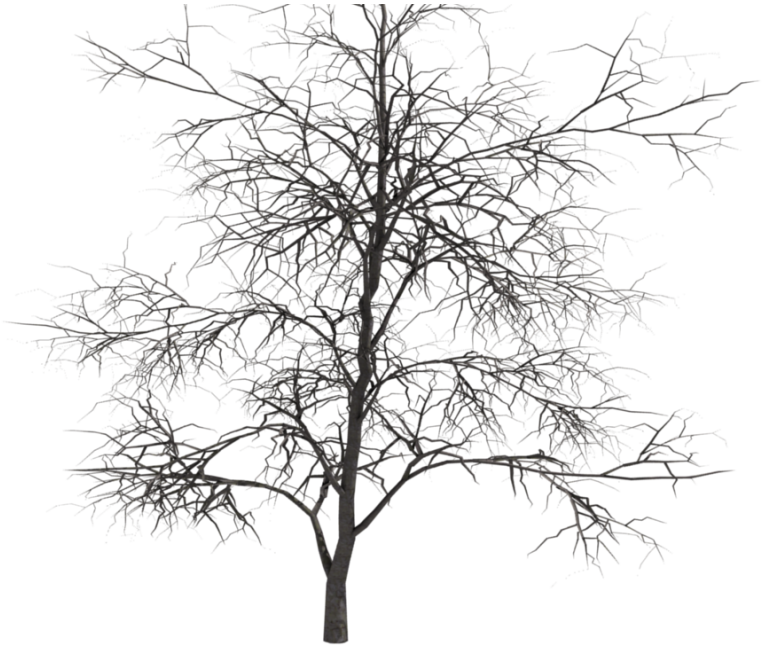
```
//Reading bone weighting data from texture and interpolating bone
    weighting from original vertices using a weighted mean
int texelindex = gl_PrimitiveID * 6;

ivec2 index;
index.y = texelindex / int(WeightingTextureWidth);
index.x = texelindex - (index.y * int(WeightingTextureWidth));

vec4 facedata = texelFetch(WeightingTexture, index, 0);

float weights[5];

float total = 0.0;
for(int i = 0; i < numberOfBones; i++)
{
        weights[i] = clamp((boneweights[i][0] * gl_TessCoord[0]) +
            (boneweights[i][1] * gl_TessCoord[1]) +
            (boneweights[i][2] * gl_TessCoord[2]),0.0,1.0);
        total += weights[i];
}

float weightmod = 1.0 / total;

vec3 newpos = vec3(0.0);
vec3 newskeletonpoint = vec3(0.0);

//For all influencing bones, calculating new position relative to
    that bone, and averaging positions by their bone weighting
for(int i = 0; i < numberOfBones; i++)
{
        ivec2 index1, index2, index3;

        index1.y = bones[i] / int(BoneTextureWidth);
        index1.x = bones[i] - (index1.y * int(BoneTextureWidth));
        index2.y = (bones[i]+1) / int(BoneTextureWidth);
        index2.x = (bones[i]+1) - (index2.y * int(BoneTextureWidth));
        index3.y = (bones[i]+2) / int(BoneTextureWidth);
        index3.x = (bones[i]+2) - (index3.y * int(BoneTextureWidth));

        vec4 bonedata, p0, p1;
        bonedata = texelFetch(BoneTexture, index1, 0);
        p0 = texelFetch(BoneTexture, index2, 0);
        p1 = texelFetch(BoneTexture, index3, 0);

        vec3 closest = closestPoint(p0.xyz, p1.xyz, pos);
```

```
        vec3 offsetvec = pos-closest;

        offsetvec = normalize(offsetvec);

        float radius;

        float L, W, a, b, x;
        L = bonedata[2];
        W = bonedata[1]*0.5;
        a = p1[3];
        b = 5;
        x = bonedata[3] + distance(closest, p0.xyz);

        x = x - W;
        L = L - W;

        radius = (a*W) / (((x * (b - a)) / L) + a);

        radius = clamp(radius,0.0,W);
        newpos = newpos + ((closest + (radius *
            offsetvec))*(weights[i]*weightmod));
        newskeletonpoint = newskeletonpoint * (weights[i]*weightmod);
}

pos = newpos;

gl_Position = ProjectionMatrix * ModelViewMatrix * vec4(pos, 1.0);

vec4 PositionEye = (ModelViewMatrix * vec4(pos, 1.0));
tePositionEyeUniform = PositionEye.xyz;


//Constructing TBN matrix using new data (for normal mapping)
vec3 n, t, b;

n = (gl_TessCoord.x * tcNormal[0] + gl_TessCoord.y * tcNormal[1] +
    gl_TessCoord.z * tcNormal[2]);
t = (gl_TessCoord.x * tcTangent[0] + gl_TessCoord.y * tcTangent[1]
    + gl_TessCoord.z * tcTangent[2]);
b = (gl_TessCoord.x * tcBitangent[0] + gl_TessCoord.y *
    tcBitangent[1] + gl_TessCoord.z * tcBitangent[2]);

n = normalize(NormalMatrix*n);
t = normalize(NormalMatrix*t);
b = normalize(NormalMatrix*b);
```

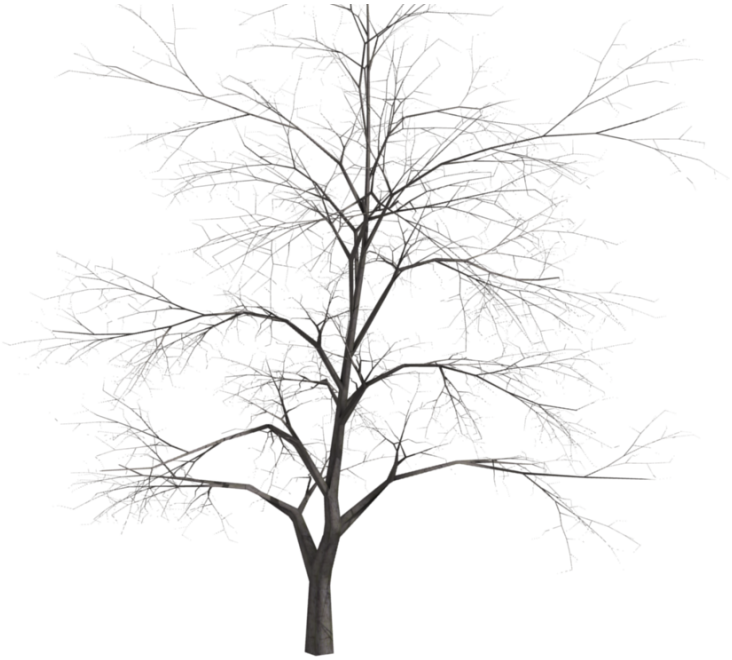
```
        mat3 tbnMatrix = mat3( t.x, b.x, n.x,
                               t.y, b.y, n.y,
                               t.z, b.z, n.z);

        //Transferring data to Fragment Shader
        tePositionEye = PositionEye.xyz / PositionEye.w;
        vec3 L = normalize(light_direction.xyz-tePositionEye);
        tePositionEye=-tePositionEye;
        tePositionEye = tbnMatrix*tePositionEye;
        teLightPos = tbnMatrix*L;
}
```

Appendix B

Tree Remodelling Comparison

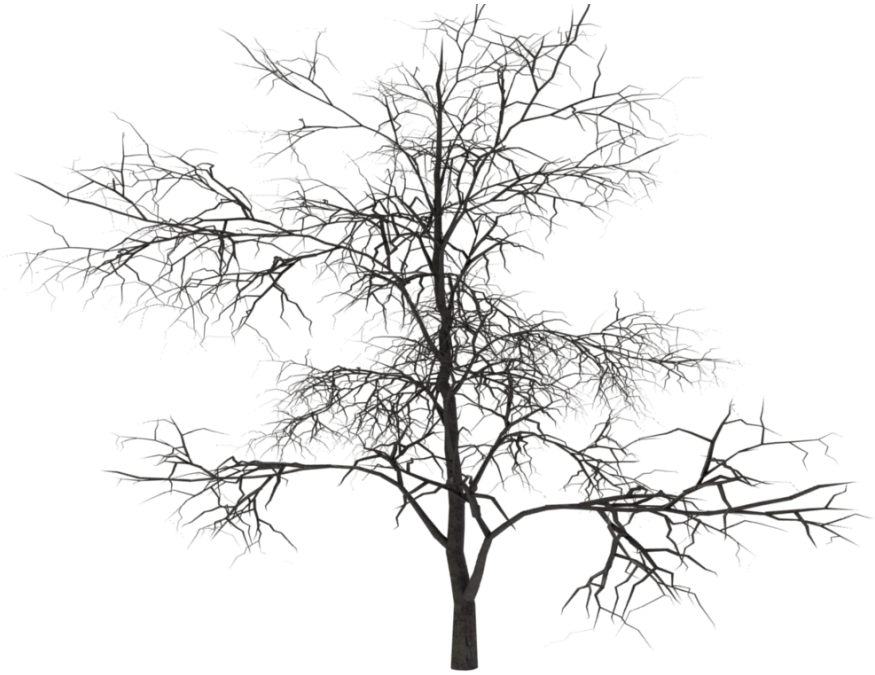


(a) The original static mesh.

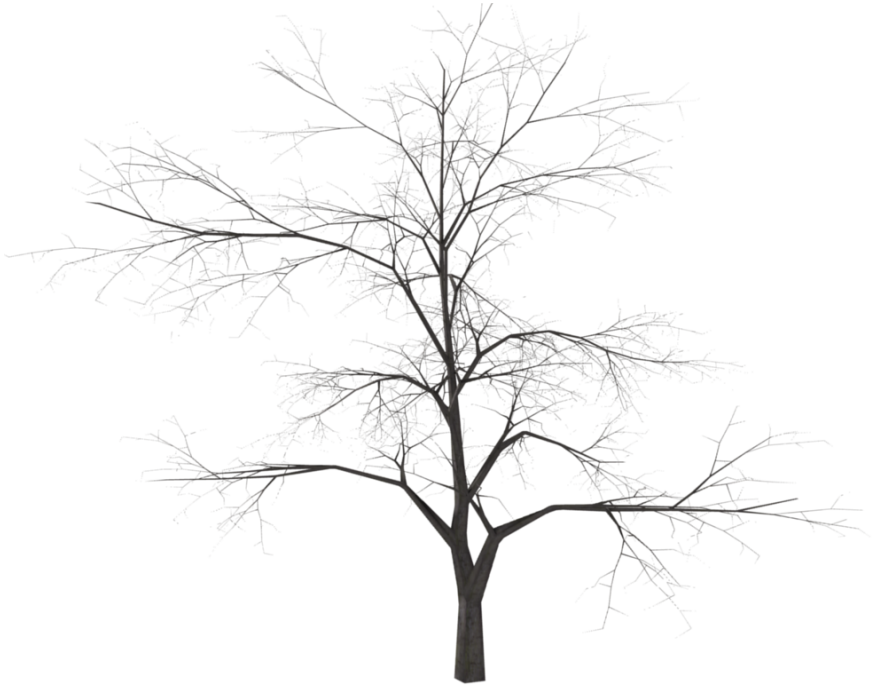


(b) The remodelled mesh.

Figure B.1: Comparison of original Xfrog tree models and remodelled geometry around the same generated structure.



(a) The original static mesh.



(b) The remodelled mesh.

Figure B.2: Comparison of original Xfrog tree models and remodelled geometry around the same generated structure.

(a) The original static mesh with connection artefacts highlighted in red.

(b) The remodelled mesh.

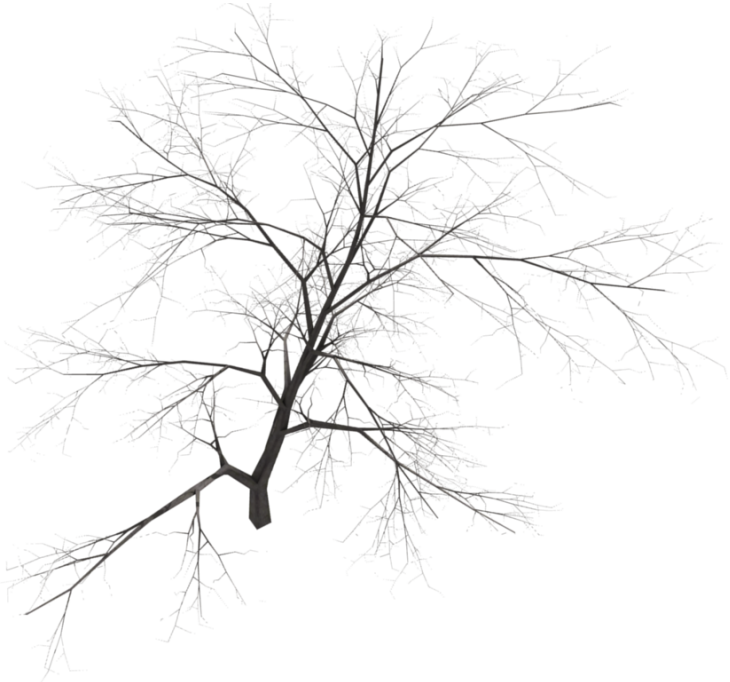
Figure B.3: Comparison of original Xfrog tree models and remodelled geometry around the same generated structure.

(a) The original static mesh with connection artefacts highlighted in red.

(b) The remodelled mesh.

Figure B.4: Comparison of original Xfrog tree models and remodelled geometry around the same generated structure.

(a) The original static mesh.



(b) The remodelled mesh.

Figure B.5: Comparison of original Xfrog tree models and remodelled geometry around the same generated structure.

(a) The original static mesh with connection artefacts highlighted in red.

(b) The remodelled mesh.

Figure B.6: Comparison of original Xfrog tree models and remodelled geometry around the same generated structure.

Bibliography

[Bel86] A D Bell. The Simulation of Branching Patters in Modular Organisms. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 313(1159):143–159, August 1986.

[BH84] R Borchert and H Honda. Control of development in the bifurcating branch system of Tabebuia rosea: a computer simulation. *Botanical Gazette*, pages 184–195, 1984.

[Blo85] J Bloomenthal. Modeling the mighty maple. *Proceedings of the 12th annual conference on Computer graphics and interactive techniques - SIGGRAPH '85*, pages 305–311, 1985.

[BLZD12] G Bao, H Li, X Zhang, and W Dong. Large-scale forest rendering: Real-time, realistic, and progressive. *Computers & Graphics*, 36(3):140–151, May 2012.

[BRS79] A D Bell, D Roberts, and A Smith. Branching patterns: the simulation of plant architecture. *Journal of theoretical biology*, 81(2):351–75, November 1979.

[Cho56] N Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[DCL05] L E Da Costa and J Landry. Generating grammatical plant models with genetic algorithms. In *Adaptive and Natural Computing Algorithms*, pages 230–234, 2005.

[Dou67] G A Doumani. Surface structures in snow. *Physics of Snow and Ice: proceedings*, 1(2):1119–1136, 1967.

[DRBR09] J Diener, M Rodriguez, L Baboud, and L Reveret. Wind projection basis for real-time animation of trees. *Computer Graphics Forum*, 28(2):533–540, April 2009.

[DRF06] J Diener, L Reveret, and E Fiume. Hierarchical retargetting of 2D motion fields to the animation of 3D plant models. *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 187–195, 2006.

[FB02] B E Feldman and J F O Brien. Modeling the accumulation of wind-driven snow. In *ACM SIGGRAPH 2002 conference abstracts and applications*, pages 218–218. ACM, 2002.

[FB07] D Foldes and B Beneš. Occlusion-based snow accumulation simulation. *Reality Interactions and Physical Simulation*, 2007.

[Fea00] P Fearing. Computer modelling of fallen snow. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*, pages 37–46, 2000.

[FG09] N Festenberg and S Gumhold. A Geometric Algorithm for Snow Distribution in Virtual Scenes. In *Eurographics Workshop on Natural Phenomena*, pages 15–25. The Eurographics Association, 2009.

[FG11] N V Festenberg and S Gumhold. Diffusion-Based Snow Cover Generation. *Computer Graphics Forum*, 30(6):1837–1849, September 2011.

[GCRR11] J Gumbau, M Chover, I Remolar, and C Rebollo. View-dependent pruning for real-time rendering of trees. *Computers & Graphics*, 35(2):364–374, April 2011.

[HAH02] H Haglund, M Andersson, and A Hast. Snow Accumulation in Real-Time. In *Proceedings of SIGRAD*, pages 11–15, 2002.

[HCH12] S Hu, N Chiba, and D He. Realistic animation of interactive trees. *The Visual Computer*, 28(6-8):859–868, April 2012.

[HJT+13] C Huang, W Jheng, W Tai, C Chang, and D Way. Procedural grape bunch modeling. *Computers & Graphics*, 37(4):225–237, June 2013.

[HKW09] R Habel, A Kusternig, and M Wimmer. Physically Guided Animation of Trees. *Computer Graphics Forum*, 28(2):523–532, April 2009.

[IOI06] T Ijiri, S Owada, and T Igarashi. The sketch l-system: Global control of tree modeling using free-form strokes. *Smart Graphics*, 1, 2006.

[KDRB+03] MZ Kang, P De Reffye, J Barczi, B G Hu, and F Houllier. Stochastic 3D tree simulation using substructure instancing. In *Plant Growth Modeling and Applications*, pages 154–168, 2003.

[KHL04] T Kim, M Henson, and M C Lin. A hybrid algorithm for modeling ice formation. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation - SCA '04*, page 305, New York, New York, USA, August 2004. ACM Press.

[KL03] T Kim and M C Lin. Visual Simulation of Ice Crystal Growth. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 86–97, 2003.

[LD98] B Lintermann and O Deussen. A Modelling Method and User Interface for Creating Plants. *Computer Graphics Forum*, 17(1):73–82, March 1998.

[LD99] B Lintermann and O Deussen. Interactive modeling of plants. *Computer Graphics and Applications, IEEE*, 19(1):56–65, 1999.

[Lin68] A Lindemayer. Mathematical models for cellular interaction in development I and II. *Journal of Theoretical Biology*, 1968.

[LPC+11] Y Livny, S Pirk, Z Cheng, F Yan, O Deussen, D Cohen-Or, and B Chen. Texture-lobes for tree modelling. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 53:1–53:10, 2011.

[LPRM02] B Lévy, S Petitjean, N Ray, and J Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics*, 3:362–371, 2002.

[LRBJ09] J Long, C Reimschussel, O Britton, and M Jones. Motion capture for natural tree animation. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, pages 77:1–77:1, 2009.

[LVM04] J Lluch, R Vivó, and C Monserrat. Modelling tree structures using a single polygonal mesh. *Graphical Models*, 66(2):89–101, March 2004.

[LZK04] MS Langer, L Zhang, and AW Klein. A spectral-particle hybrid method for rendering falling snow. *Rendering techniques*, 4:217–226, 2004.

[May73] G J Mayhead. Some drag coefficients for British forest trees derived from wind tunnel studies. *Agricultural Meteorology*, 12:123–130, 1973.

[MC00] K Muraoka and N Chiba. Visual simulation of snowfall, snow cover and snowmelt. In *Proceedings Seventh International Conference on Parallel and Distributed Systems: Workshops*, number 4, pages 187–194. IEEE Comput. Soc, 2000.

[MGG+10] N Maréchal, E Guérin, E Galin, S Mérillou, and N Mérillou. Heat Transfer Simulation for Modeling Realistic Winter Sceneries. *Computer Graphics Forum*, 29(2):449–458, June 2010.

[NIDN97] T Nishita, H Iwasaki, Y Dobashi, and E Nakamae. A Modeling and Rendering Method for Snow by Using Metaballs. *Computer Graphics Forum*, 16(3):C357–C364, September 1997.

[NTT92] H Noser, D Thalmann, and R Turner. Animation based on the Interaction of L-systems with Vector Force Fields. *Proc. Computer Graphics International*, 1992.

[OS04] P Ohlsson and S Seipel. Real-time Rendering of Accumulated Snow. In *Sigrad Conference*, pages 25–32, 2004.

[OTF+04] S Ota, M Tamura, T Fujimoto, K Muraoka, and N Chiba. A hybrid method for real-time animation of trees swaying in wind fields. *The Visual Computer*, 20(10):613–623, November 2004.

[PHHM97] P Prusinkiewicz, M Hammel, J Hanan, and R Měch. Visual models of plant development. In *Handbook of formal languages*, pages 535–597. Springer, 1997.

[PHM93] P Prusinkiewicz, M S Hammel, and E Mjolsness. Animation of Plant Development. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, volume 93, pages 351–360. ACM, 1993.

[PJM94] P Prusinkiewicz, M James, and R Měch. Synthetic Topiary. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, volume 94, pages 351–358. ACM, 1994.

[PLH88] P Prusinkiewicz, A Lindenmayer, and J Hanan. Development models of herbaceous plants for computer imagery purposes. In *ACM SIGGRAPH Computer Graphics*, pages 141–150, 1988.

[PLH+90] P Prusinkiewicz, A Lindenmayer, J S Hanan, F D Fracchia, D R Fowler, M J M de Boer, and L Mercer. *The algorithmic beauty of plants*. Springer-Verlag New York, 2 edition, 1990.

[PMKL01] P Prusinkiewicz, L Mündermann, R Karwowski, and B Lane. The use of positional information in the modeling of plants. In *Proceedings of the*

28th annual conference on Computer graphics and interactive techniques, volume 2001, pages 289–300. ACM, 2001.

[Pru86] P Prusinkiewicz. Graphical applications of L-systems. *Proceedings of graphics interface*, pages 247–253, 1986.

[Pru04] P Prusinkiewicz. Modeling plant growth and development. *Current Opinion in Plant Biology*, 7(1):79–83, February 2004.

[PSK+12] Sn Pirk, O Stava, J Kratt, M A M Said, B Neubert, R Měch, B Benes, and O Deussen. Plastic trees. *ACM Transactions on Graphics*, 31(4):1–10, July 2012.

[SD05] S I Sen and A M Day. Modelling trees and their interaction with the environment: A survey. *Computers & Graphics*, 29(5):805–817, October 2005.

[Sim91] K Sims. Artificial Evolution for Computer Graphics. *Computer Graphics*, 25(4), 1991.

[SOH99] R W Sumner, J F O'Brien, and J K Hodgins. Animating Sand, Mud, and Snow. *Computer Graphics Forum*, 18(1):17–26, March 1999.

[SSC+13] A Stomakhin, C Schroeder, L Chai, J Teran, and A Selle. A material point method for snow simulation. *ACM Transactions on Graphics*, 32(4):1, July 2013.

[SSKLK13] D Shreiner, G Sellers, J M Kessenich, and B M Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3.* 2013.

[TBB10] N Tatarchuk, J Barczak, and B Bilodeau. Programming for Real-Time Tessellation on GPU. *AMD whitepaper*, 5(3), 2010.

[Tok06] K Tokoi. A Shadow Buffer Technique for Simulating Snow-Covered Shapes. *International Conference on Computer Graphics, Imaging and Visualisation (CGIV'06)*, pages 310–316, 2006.

[Web08] J P Weber. Fast Simulation of Realistic Trees. *IEEE Computer Graphics and Applications*, 28(3):67–75, May 2008.

[WP95] J Weber and J Penn. Creation and rendering of realistic trees. *Proceedings of the 22nd annual conference on Computer Graphics and Interactive Techniques*, pages 119–128, 1995.

[WWDY06] L Wang, W Wang, J Dorsey, and X Yang. Real-time rendering of plant leaves. *ACM SIGGRAPH 2006*, pages 712–719, 2006.

[WWXP06] C Wang, Z Wang, T Xia, and Q Peng. Real-time snowing simulation. *The Visual Computer*, 22(5):315–323, April 2006.

[YSHW03] C Yanyun, H Sun, L Hui, and E Wu. Modelling and rendering of snowy natural scenery using multi-mapping techniques. *The Journal of Visualization and Computer Animation*, 14(1):21–30, February 2003.