THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Interactive Video Game Content Authoring using Procedural Methods

Dacre Denny

This thesis is submitted in partial fulfillment of the requirements for the Degree of Master of Science at the University of Waikato.

June 2010

Abstract

This thesis explores avenues for improving the quality and detail of game graphics, in the context of constraints that are common to most game development studios. The research begins by identifying two dominant constraints; limitations in the capacity of target gaming hardware/platforms, and processes that hinder the productivity of game art/content creation. From these constraints, themes were derived which directed the research's focus. These include the use of algorithmic or 'procedural' methods in the creation of graphics content for games, and the use of an 'interactive' content creation strategy, to better facilitate artist production workflow.

Interactive workflow represents an emerging paradigm shift in content creation processes used by the industry, which directly integrates game rendering technology into the content authoring process. The primary motivation for this is to provide 'high frequency' visual feedback that enables artists to see games content in context, during the authoring process.

By merging these themes, this research develops a production strategy that takes advantage of 'high frequency feedback' in an interactive workflow, to directly expose procedural methods to artists', for use in the content creation process. Procedural methods have a characteristically small 'memory footprint' and are capable of generating massive volumes of data. Their small 'size to data volume' ratio makes them particularly well suited for use in game rendering situations, where capacity constraints are an issue. In addition, an interactive authoring environment is well suited to the task of setting parameters for procedural methods, reducing a major barrier to their acceptance by artists.

An interactive content authoring environment was developed during this research. Two algorithms were designed and implemented. These algorithms provide artists' with abstract mechanisms which accelerate common game content development processes; namely object placement in game environments, and the delivery of variation between similar game objects. In keeping with the theme of this research, the core functionality of these algorithms is delivered via procedural methods. Through this, production overhead that is associated with these content development processes is essentially offloaded from artists onto the processing capability of modern gaming hardware.

This research shows how procedurally based content authoring algorithms not only harmonize with the issues of hardware capacity constraints, but also make the authoring of larger and more detailed volumes of games content more feasible in the game production process. Algorithms and ideas developed during this research demonstrate the use of procedurally based, interactive content creation, towards improving detail and complexity in the graphics of games.

Acknowledgments

This research and thesis would not have been possible without the help and support of a number of people and institutions.

I would like to thank The University of Waikato for providing me with an opportunity to carry out this Masters research. In particular, I would like to thank Bill Rogers of the Computer Science department, for supervising this research and providing ongoing support and assistance.

In addition, I would also like to thank staff at the game development studio Sidhe, who provided me with an opportunity to meet and discuss the themes and objectives of this research. Tyrone McAuley, Stewart Middleton, Jeremy Burgess and Keir Rice were particularly instrumental in this discussion, setting aside time in their schedules to talk with me in person.

Finally, I would like to offer immense and special thanks to my family who provided ongoing support, throughout the course of this research.

Contents

Abstract i
Acknowledgments iii
Contents v
List of figures vii
List of tables xi
Chapter 1: Introduction 1
Chapter 2: Literature review 9
Procedural methods 9
Elements of procedural methods 9
Practical usage 11
Advantages of procedural methods (explained via textures) 12
Common procedural functions 14
Procedural methods in games 19
Motivations for procedural methods 26
Content creation pipelines 27
Summary 31
Chapter 3: Project design 33
Development constraints 34
Industry consultation 35
Automated object placement 37
Automated object variation 41
Chapter 4: Implementation 47
Interactive tool chain 47
Overview 48
Real time content encoder (RTCE) 52
Game rendering context (GRC) 72
Instancing algorithm 100
Structural overview 100
Implementation of first pass 110
Implementation of second pass 128
Object variation algorithm 130
Objectives and overview 132
Non-uniform deformation 133

First tessellation approach 142
Second tessellation approach 151
Final shader implementation 155
Chapter 5: Demonstrations 161
Tool chain interaction and material composition 162
Real-time generative instancing 170
NPD algorithm demonstrations 180
Chapter 6: Conclusion 187
Future work 195
Material composition system 195
Real-time generative instancing (RTGI) 195
Non-uniform procedural deformation (NPD) 197
References 199
Appendix A 211
Appendix B 213
Appendix C 215

List of figures

Figure 1 Depicts the ‘functional nature’ of PM’s. Legible’ procedural data generation often relies on consistent/contiguous input data 10

Figure 2 Illustrates the different types of data flow through procedural functions 11

Figure 3 Correlation between numerical result and visual representation 12

Figure 4 Quality comparison between procedural and conventional reproduction 13

Figure 5 A procedural tree structure composed in ‘MapZone’ 14

Figure 6 Grittiness and grime achieved via procedural noise 14

Figure 7 Visual representation of Perlin noise. 15

Figure 8 Illustrates continuous nature of Perlin noise (1D) when evaluated against contiguous parameters 15

Figure 9 Illustrates a demonstration program that was developed. The ‘uniform-grid’ and ‘auxiliary vectors’ which underlie Perlin noise are depicted. 16

Figure 10 Composition of the Gabor kernel 17

Figure 11 The noise result achieved by ‘splatting’ Gabor kernels 18

Figure 12 Achieving isotropic noise via random distribution of Gabor kernel orientations 18

Figure 13 Illustrates a range of results that can be achieved via GRC noise 19

Figure 14 The Sentinel uses procedural methods to generate a large set of playable levels 19

Figure 15 Quake 3 Arena applied textures via improved capabilities of consumer graphics hardware 20

Figure 16 Illustrates terrain, procedurally generated by Age of Empire’s level editor 21

Figure 17 Illustrates the result of procedurally based enemy placement and control in Left 4 Dead. 22

Figure 18 Illustrates the use of procedurally generated textures in Roboblitz 24

Figure 19 The interactive shader/material creation offered in Unreal 3’s tool suite 30

Figure 20 Image of Id Software’s current game project ‘Rage’ 33

Figure 21 Image of a forest scene in Fable 2 37

Figure 22 Visualization of the algorithmic instancing concept, showing the relationship between geometric elements and procedural functionality 38

Figure 23 Visualization of other channels of procedural data in procedural instancing 38

Figure 24 Abstract illustration showing a clear correspondence between a checker procedural and the instancing outcome 39

Figure 25 Illustrates 'local rotation axis' for instances. These axes are equivalent to corresponding surface normals 40
Figure 26 Modern games achieve increased realism by populating environments with many props and objects 42
Figure 27 Compares different parameterization schemes for a deformation procedural function 44
Figure 28 Shows how increased levels of 'geometric tessellation' yield more legible deformation results 45
Figure 29 Visualization of the 'painting' metaphor when assigning deformation to geometry 46
Figure 30 Screenshot of Maya, highlighting the software's 'viewport' interface element 48
Figure 31 Schematic diagram, illustrating work flow configurations via the 'connection model' 52
Figure 32 Example of scene representation in Maya 2008 using abstract graph structure 54
Figure 33 Illustrates how improved lighting can be achieved via computations that use per-vertex normal vectors 55
Figure 34 RTCE's material hierarchy for showing custom composition of procedural functions for a material channel 61
Figure 35 Illustrates an enhanced surface material via composition of numerous procedural elements in the RTCE plug-in 62
Figure 36 Illustration of DAG structures in Maya 63
Figure 37 Schematic of the GRC's shader system that supports custom shader functionality in conjunction with dynamic vertex formats 86
Figure 38 Illustrates the adaptive mechanism which handles the automatic substitution of parameters for custom shader functions when required 89
Figure 39 Illustrates the GRC's shader based material composition system 92
Figure 40 Shows how procedural functions are included/excluded in a procedural composition during shader compilation 93
Figure 41 Illustrates the GRC's material structure 96
Figure 42 Illustrates integration of 'material module' and objects of the 'scene module' 97
Figure 43 Illustrates motion blur as a post-processing effect in Motostorm 100
Figure 44 Illustrates adaptive terrain, where tessellation is a function of view position 101
Figure 45 Sophisticated fluid flow achieved by applying geometry shading to a particle system 102
Figure 46 Illustrates poor utilization of stream via first strategy 109
Figure 47 Schematic diagram illustrates general flow control in the instancing shader 110

Figure 48 Illustrates the ‘coverage’ strategy which is used during instance generation 111
Figure 49 Illustrates what information is known about a triangle at the start of the instance generation process 112
Figure 50 Illustration of sample clipping during iterative sampling of sub triangle 113
Figure 51 Illustrates visual artefacts that resulted in the first implementation of the instance generation process 114
Figure 52 Illustrates the ‘virtual rectangle’ which encloses triangles processed by the instance generation phase 115
Figure 53 Illustration of tangent space on arbitrary manifold. Note the orthogonal nature of this space. 116
Figure 54 Illustrates the process of ‘virtual rectangle’ computation via tangent space and orthogonal projection 117
Figure 55 Illustrates how texture coordinates ranges influence marching ‘density’ 118
Figure 56 Illustrates limitations of instance generation in the second instancing approach 119
Figure 57 Shows how a texture coordinate based ‘virtual rectangle’ can be used for instance generation 120
Figure 58 Illustrates the sample transformation process for the final triangle coverage algorithm 122
Figure 59 Schematic of per-instance procedural evaluations 124
Figure 60 Illustrates the cookie cutter concept for instance generations 126
Figure 61 Depicting variation between pedestrians in Grand Theft Auto IV (GTAIV) 130
Figure 62 Illustrates object duplication in games 131
Figure 63 Variation present within early video games 132
Figure 64 Illustrates a variation strategy that is frequently used in games 132
Figure 65 Various tessellation strategies 136
Figure 66 Illustrates non-uniform (or adaptive) tessellation 137
Figure 67 Quality comparison between tessellation strategies 138
Figure 68 Illustrates seam-gap artifacts which occur when levels of deformation between adjacent triangles differ 140
Figure 69 Illustrates the severity of seam artifacts under normal circumstances 141
Figure 70 Illustrates adjacent geometry that is accessible during triangle processing 142
Figure 71 Illustrates the NPD algorithm’s gap prevention strategy 148
Figure 72 Shows how triangles that require no deformation are copied through internal passes 152
Figure 73 Comparison of parallelism in each NPD algorithm approach 153

Figure 74 Illustrates the role of support buffers for data flow in the revised NPD shader 159

List of tables

Table 1 The typical chronology of visual quality in game franchises....1

Table 2 Unit sales for current generation console hardware as of 2010 3

Table 3 A summary of PM's in a range of games....25

Table 4 Code excerpt showing features of mesh packaging and transmission iteration 59

Table 5 Material types supported in RTCE....60

Table 6 Auxiliary attributes assigned to objects by and for the RTCE plug-in 63

Table 7 Summary of interaction events in Maya that the RTCE responds to 66

Table 8 Development chronology of the RTCE interface....72

Table 9 Code excerpt showing typical features of an HLSL shader 78

Table 10 Code excerpt showing main features of a simple shader declaration 82

Table 11 Shows how pre-processing capabilities of HLSL are used to deliver shaders which are adaptive to arbitrary vertex formats 83

Table 12 Code excerpt showing main features of a simple shader declaration in the GRC's shader system 88

Table 13 Code excerpt shows how 'blocks' of code are conditionally introduced to a shader at compile time, to deliver procedural composition....94

Table 14 Comparison of shader types, showing how geometry shaders facilitate variable data output, unlike other shader types 104

Table 15 Shows how variable output is achieved via flow control and the output 'structure' of geometry shaders in HLSL....105

Table 16 Per-instance parameter structure emitted from instance shader....106

Table 17 Projection equation used by the RTGI algorithm's 'sub triangle' extraction calculation 112

Table 18 The marching process exposes this data to the instance generation phase 123

Table 19 Code excerpt from the instance shader implementation, illustrates the integration of the 'cookie cutter' feature....128

Table 20 Code excerpt from the deformation shader implementation which shows how adaptive vertex interpolation is achieved in order to compute the 'split vertex' 146

Table 21 Pass structure of revised NPD shader 155

Table 22 Code excerpt showing the 'technique structure' of the NPD shader 157

Chapter 1: Introduction

The global entertainment industry has shown significant growth in recent decades; a trend that is expected to continue, according to analysts and market researchers (Business Wire, 2007) (The Financial Express, 2007). Consistent with this trend, are significant increases in consumer spending throughout the entertainment software industry (Riley, 2008). According to the Entertainments Software Association (ESA), the U.S entertainment software industry grew 17% between 2003 and 2005. Furthermore, the entertainment software/gaming sector has shown particular growth compared to adjacent industries; namely the films and music industries (Anderson, 2007).

These solid industry trends and market projections indicate a clear opportunity for commercial revenue in gaming software. This yields a competitive commercial climate, where remaining at the forefront of games technology and development techniques, is paramount to the success of game development studios. More specifically, improvements to the quality of game experiences will continue to underpin a game studio's success.

		
Wolfenstein 3D 1992 (id Software: Wolfenstien 3D and Spear of Destiny, 2001)	Return to castle Wolfenstein 2001 (id Software: Return to Castle Wolfenstein, 2001)	Wolfenstein 2009 (Wolfenstein | Media, 2009)
Table 1 The typical chronology of visual quality in game franchises		

In most games, the visual element is the primary channel through which the game experience is conveyed. Although elements such as game play, audio and social interaction are significant components of a game experience, the graphic component is arguably the most influential, particularly in terms of sales influence, impressiveness and overall impact. Games graphics continue to serve as a cornerstone for ongoing innovation and development in the games industry. Thus, the alluded motivations for this are primarily based on "[increased consumer] *expectation for greater realism* [in] *the visual quality of the game content*" (Scheidt, 2005).

Significant improvements in the graphics of games have been evident, particularly in the past two decades of the games industry. Table 1 shows this trend in Id Software's Wolfenstein 3D video game franchise, which spans over two decades and exhibits considerable improvement in graphics during this time (Wolfenstein 3D, 2010).

Despite these achievements, prominent figures in the games development industry have indicated that opportunity still exists for further improvement in visual realism and the quality of games.

Tim Sweeny, founder and technical director of Epic Games and arguably one of the industry's leading contributors to game technology design and development, suggested in an interview with Benj Edwards of Gamasutra in 2009 that "[games are] *about a factor of a thousand off from achieving* [photo realism] *in real-time*" (McLean-Foreman, 2001) (Sweeney, 2009). This estimate indicates significant opportunity for continued research and development in real-time graphics.

Developing graphics rendering functionality that is consistent with quality standards of current games however, is already a non-trivial task. Game 'rendering functionality' is typically integrated into a 'graphics engine' subsystem, within the game's 'engine technology'. In addition to geometry rendering, modern graphics engines typically integrate functionality that delivers special effects and animation.

Common special effects offered by rendering engines typically include 'high dynamic range' and 'motion blur', which are simulated in real-time (Rosado, 2008) (Green & Cebenoyan, 2004). Design characteristics which facilitate optimization, hardware acceleration, data processing and priority management, as well as parallelism, are important in game render engines. Underlying these staple elements is an emerging requirement for 'cross architecture' support.

More specifically, the design and implementation of commercial game rendering systems is often necessary for use on most, if not all, current generation gaming architectures; namely the PC, Playstation® 3, Wii™, and Xbox360®. The reason for this is usually motivated by economic factors, given that the increased market exposure which results from multiplatform game deployment maximizes the product's revenue prospects (Simpson, 2009).

As of 2010, the market composition for console gaming hardware indicated a reasonable balance in the user bases of each of the three current generation console systems (see table 2). Thus, the motivation for developing multiplatform games and technology is clear, given the significant user base across each of the console platforms.

Playstation® 3	33.5 million (SCEI, 2010)
Xbox360®	40 million (Ingham, 2010)
Wii™	70.6 million (Nintendo, 2010)

Table 2 Unit sales for current generation console hardware as of 2010

Unfortunately, cross platform development and console game development in general, hinders progress towards improved visual quality of games. This is because the hardware specifications of consoles (such as graphics functionality), remain fixed throughout a console's product lifetime. Despite the ongoing pursuit for improvement in the graphics of games, the overriding economic motivations for console based development confine current development to the limitations of console hardware.

Hardware specifications for PC's are obviously more dynamic and provide greater opportunity for improved graphics quality. This was emphasised by Tim Sweeny who in 2009, stated that "[PC] *video cards, have about 10 times the graphics horsepower of* [today's] *console*" (Sweeney, 2009).Furthermore, the memory/storage capacity of current PC's is significantly higher than that of current generation consoles (see Appendix B).
Despite the potential for improved visual quality via high-end PC gaming systems, a recent study indicated that the PC platform/market as a whole, only accounts for 16% of total consumer spending in entertainment software sales (Warman, 2010). Thus, developing games tailored to high-end PC systems within this small market share is often commercially unviable, given that the production costs for single platform games average at ~$10 million (Crossley, 2009).

The 'static' technological climate of the games industry presents a major challenge for game developers working towards better graphics quality/realism in games. Although superior hardware in game platforms that succeed the current generation consoles is beneficial towards visual improvement in games, it is likely that specifications of future consoles will also remain static. Note that 'step wise' improvements to the hardware specifications of consoles have been characteristic of the seven iterations/generations of console hardware (Video game console, 2010) (The Home Video Game Console, n.d). Thus, merit exists in identifying strategies and algorithms that promote further improvement to game graphics, despite the fixed hardware of target platforms.

Compelling visual experiences in games are dependent on the quality of game media and content that encapsulates the game's underlying rendering techniques and technology. Delivering better game graphics not only requires improved rendering techniques (that comply with technological constraints), but also complex and highly detailed content, such as game characters, environments and props. Furthermore, the composition and density of games content is also fundamental in the delivery of believable game experiences; namely the placement of props in game scenes.

The creation of digital art and content for modern games is renowned for the huge workload that it represents in the game production process; this often leads to high proportions of artists in game development teams.

Larger teams of game artists obviously account for bigger overall game development teams, which are partly responsible for increasing budgets that typically range between $10-100 million (Crossley, 2009) (Ashrafi, 2008). This tends to oppose economic preferences of game production which aim to minimize production budgets. Thus, content creation strategies which make better use of artists' time are desirable both economically, and in terms of prospects for improved visual quality.

The implication of equipping artists' with processes and strategies that permit more effective content creation, in the scope of a fixed production timeline, allows greater opportunity for refinement of game content and/or the introduction of additional detail. Thus, by providing artists with efficient techniques that facilitate asset composition in game environments, namely for increased density/population of objects in scenes, significant reductions to an artist's workload and overhead are anticipated.

Integrating these techniques directly into artist workflows is likely to maximize the impact they have on the game production process. This was emphasised by Gregor vom Scheidt, vice president of Computer Graphics at Avid© when he spoke at the 2005 Game Developer Conference (GDC05) in San Francisco on game content creation; "increasing time and budgetary constraints [in games] are fuelling the demand for content creation tools that integrate seamlessly into existing production pipelines and empower game developers to work more efficiently" (Gregor vom Scheidt, 2005) (Scheidt, 2005).The production and technical constraints which hinder graphics development in games represent core focal points of this research. Thus, ideas and algorithms that are developed in this research are inherently influenced by these two themes.

The practical element of this research is the design and development of algorithms that enable artists' to improve visual complexity, detail and realism of game objects and scenes, in an efficient manner. These algorithms achieve improved artist productivity, while complying with technical constraints which are relevant to the current technological landscape of the games industry. Both of these algorithms execute in 'real-time', making them suitable for direct integration into a target games' rendering technology. Furthermore, the real-time element of these algorithms makes them suitable for integration in an 'interactive workflow', allowing artists to immediately see the outcomes of their work in the 'target game'. In other words, this real-time characteristic supports the notion of 'interactive content creation' that is contextually orientated.

Improvements to content creation in the developed algorithms stem from the notion of an 'interactive content creation' paradigm (which Gregor vom Scheidt alluded to at the GDC05). The essence of interactive content creation, is to expose 'interactive production workflow' to artists during content creation; coupling various elements such as game rendering technology, 'immediate feedback' to artist interaction and artist collaboration during production. These elements are evidently gaining prominence in content creation methodologies, technology and workflows, used by many major game studios.
By integrating this workflow paradigm, the practical research outcomes are consistent with current trends in the games industry. In addition, this workflow paradigm also provides an environment that facilitates the introduction of new ideas/concepts which might be impractical in non-interactive content creation workflows.

The interactive basis of this research aims to makes less conventional data sources/representations more feasible for use by artists in content creation. In particular, the use of 'procedural methods', for game content creation is explored.
Procedural data representations are attractive given that they maintain a low memory footprint to data volume ratio. This makes data sources based on 'procedural mechanisms' well suited to current platforms and technologies; namely game consoles, where storage capacity is a particular constraint. By integrating procedurally based content production algorithms into an interactive workflow, this research aims to amalgamate the beneficial characteristics of procedural functions, with a content creation environment that encourages experimentation and refinement, while imposing minimal workflow 'overhead' on artists. This 'combination' therefore, aims to advance boundaries in the complexity and detail of

game graphics by using procedural methods, under artist control, to generate game graphics data.

As mentioned the algorithms of this research are integrated into an interactive artist workflow. For reasons that are subsequently discussed, a workflow was designed and implemented to accommodate research specific algorithms. This 'workflow' consists of several software components and is referred to as the 'interactive tool chain'.
The algorithms integrated into this tool chain are essentially abstractions for common classes of respective content authoring tasks that face artists. The motivation for delivering 'abstract' algorithms is to provide content creation mechanisms that are capable of covering a wide range of applications and scenarios for various content authoring tasks. Although a number of obvious applications for these algorithms exist, the abstract nature of the research's algorithms aims to 'free' the creativity of artists', allowing them to be used for a range of content/game development scenarios.

This thesis begins with a review of the core elements that underlie the practical outcomes of this research with the concept of procedurally generated graphics data explored in greater detail. In addition, the history and usage trends of this concept are reviewed, establishing precedent for the use of procedural generation. An emerging trend in the application of procedural methods for other purposes in modern games is also investigated; providing insight into the diverse nature of procedural data generation. This demonstrates that in addition to explicit game graphics, procedural data can fulfil other purposes in the delivery of game experiences. In addition, technologies and strategies used and developed by numerous game development studios are reviewed; this identifies emerging features of content creation strategies. These points illustrate the relevance of this research in interactive content creation scenarios, for artists'.

Following the review, the project design chapter outlines key points that directed the investigation and implementation tasks of this research. The chapter provides detail of the development criteria which underlies subsequent work, in addition to an overview of the 'abstract' algorithms which represent methods of improving game graphics, in the context of an underlying 'interactive workflow'/tool chain.
The specific procedural algorithms developed, namely procedural/algorithmic 'geometry instancing' and procedural 'geometric object variation', are introduced and briefly described. Accompanying these descriptions are implementation specifications for the algorithms to

ensure they meet research objectives, as well as the level of flexibility required by artists during use.

The largest section of this thesis is the 'implementation chapter', which consists of three main sections. The order of these sections reflects the chronology of the implementation process.
The first section describes the tool chain that was developed. Two software components are covered; a module that integrates into the artist's content authoring environment and a game-like rendering context for displaying artists' work/content. Design decisions and features of the developed tool chain component that integrate with the target 'content authoring environment', are also discussed. This provides insight as to how the tool chain compliments the artist's creative process, while maintaining 'transparency' to encourage an uninterrupted, fluid process of content creation.
The implementation underlying the game rendering context provides specific detail and justification for rendering technologies and features used. These are explained in the context of the objectives for minimized space complexity, as well as high flexibility and configurability for artists.

The second and third sections of the implementation chapter describe the 'procedural instancing' and 'procedural object variation' algorithms. As indicated, these algorithms represent the research's main strategies for improving the productivity and efficiency that underlie a common task for artists; namely the population of game scenes with objects.
Often artists are tasked with placing thousands of objects (such as props) throughout scenes of modern games as part of the creative process, representing an obvious and significant workload. The section illustrates how procedural instancing aims to minimize this overhead, while simultaneously abiding to system capacity constraints by the integration of procedural methods. Unlike 'conventional' 'static' object instancing, this implementation of instancing has a real-time basis, allowing it to respond to artist interaction in real-time via the tool chain's interactive context.
The 'object variation' section also assumes a similar integration of its algorithm within the research's game rendering interactive/context tool chain. Again, the motivation for this integration is to yield real-time, responsive feedback to artists that use the variation algorithm. As the section title suggests, this algorithm aims to achieve (procedurally based) geometric variation between objects. The motivation for this algorithm is explained in the literature review and requirements chapters.

In addition to describing each algorithm's implementation, these sections also provide a brief overview/discussion of features in modern graphics hardware, relevant to each algorithm. As is discussed, the motivation for applying these hardware features is to demonstrate that the ideas presented in this research can be expressed in high performance architectures therefore making them suitable as resident components of a game's rendering technology.

Following this chapter, a number of 'game inspired' test cases that demonstrate the ideas and algorithms of this research are showcased in image form. These demonstrations depict the interactive and 'productivity enhancing' aspects of the content creation tool chain and algorithms, that were developed.

Chapter 2: Literature review

The following discussion covers a variety of topics relevant to this thesis, expanding on challenges currently facing game developers; namely production and technical constraints. This research explores the use of 'procedural methods' (PM) as an avenue for continued video game improvement, beginning with a look at previous and current roles of PM's in a number of prominent games. In addition, the implications of PM's as a feature of the game-production process is also explored. Due to the influence that 'game content creation' has on the game production process, 'game content creation systems' that are currently used in industry, are reviewed. Although varied, each of these systems shares a common characteristic; a move towards 'interactive' and 'real-time' content-authoring to assist game artists.

This analysis provides a premise of this research which is to amalgamate the benefits of PM's with the game production process. In addition, by making direct use of PM's in game technology, this research considers the implications of smaller memory usage and storage (as offered by PM's) towards the delivery of rich and detailed game experiences.

Procedural methods

This review will begin with a brief introduction to the concept and theory of PM's. As the literature review develops, the introduction will serve as a backdrop for subsequent discussion of project specific ideas that are based on PM's.

Elements of procedural methods

PM's are diverse, abstract concepts that modify 'input data' to generate results systematically, via an internal algorithm/ 'characteristic'. Thus, the inherent diversity of PM's makes them applicable to a variety of situations.

Despite the diversity of PM's, all are unified by a number of factors; namely, the functional nature of procedural output. That is, procedural results are a direct side effect of evaluating a procedural function. In addition, PM's are 'referentially transparent' and thus, should always yield the same computation result for given 'input parameters' (Sondergaard & Sestoft,

1990). To illustrate referentially transparency, consider the ‘sine’ function which is deterministic for any specified ‘phase offset’.

The structure and process of a PM typically involves operation on input data by the procedural's internal generator or 'descriptor'. This is followed by the return of procedural results that reflect characteristics of the internal descriptor, as well as the specified input parameters. The descriptor is essentially the PM’s implementation and hence, it dictates the procedural's output.

Input parameters are a typical component of PM’s. Non-parameterized procedurals do exist however, which satisfy the formal definition of a PM. These ‘marginal’ examples are limited to two classes of PM’s; pseudo random generators and constant functions. This limitation illustrates the functional nature of PM’s in that they are more flexible when input data is specified.

The output of a non-parameterized, procedural random generator is however, still deterministic (or ‘pseudo’ random). That is, under equivalent system states the same result will be produced by the function, therefore maintaining the characteristic of referential transparency.

PM’s tend to be most effective for situations where they are parameterized with contiguous data. The classic example of this is pattern/image generation across screen pixels. When parameterized with contiguous pixel data, namely the screen coordinates of each pixel, PM’s are capable of expressing images/textures results which retain distinct characteristics.

In contrast to this, PM’s evaluated in the context of ‘scrambled data’ will typically struggle to produce legible results as PM’s implicitly reflect their incoming data, as well as any inconsistencies within a parameter ‘neighbourhood set’ (figure 1). This will be expanded on further in subsequent discussion.

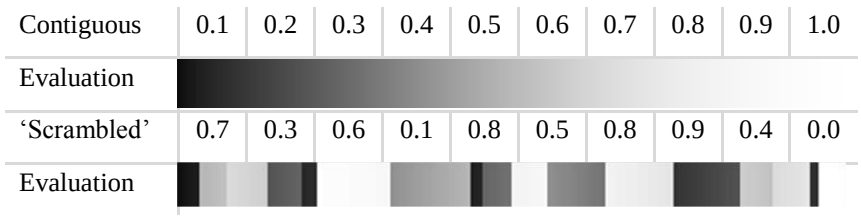
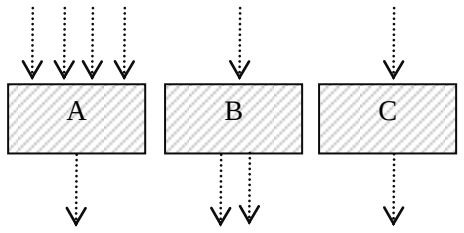


Figure 1 Depicts the ‘functional nature’ of PM’s. Legible’ procedural data generation often relies on consistent/contiguous input data

Another characteristic of PM's is that no correlation between the amount of input and output data is required, as figure 2 shows. A PM can therefore, inflate or reduce the volume of data that passes through it. To illustrate this, consider a procedural function that generates wave forms. This function may have a number of parameters, such as amplitude and frequency. The evaluation of this procedural, processes these parameters, reducing them to a single wave form offset (i.e. one scalar value). In this example, the reduction of incoming data during evaluation, demonstrates the principle of data volume 'independence'.

One obvious constraint that applies to all PM's is that procedural functions must always return data following evaluation (see figure 2). This 'axiom' reiterates the fundamental nature of PM's in that they never terminate data but rather emit or channel it.



Data the amount of data that flows through procedural functions (boxes) can be, deflated, (B) inflated, (C) or retained. Note that all procedural functions must return data.

Figure 2 Illustrates the different types of data flow through procedural functions

Practical usage

The following section discusses the concept of PM's in more depth via the use of computer graphics examples. Consider the previous example where a simple procedural texture was produced. When PM's are used for texture generation, the procedural function typically generates colour values for each 'texel' in the texture. The generated colour values can be influenced by input parameters that direct the PM.

Perhaps the most common parameter(s) supplied to procedural textures are the coordinates of each pixel that is processed. These coordinate parameters provide the procedural texture with contextual information about a pixel's position within coordinate space. Hence, this information is used to direct the procedural's output into the texture.

Although coordinate space is arbitrary, texture coordinates are usually normalized between the range of 0.0 and 1.0. It is therefore convenient for procedural functions to operate on coordinates that are clamped between these ranges.

Additional parameters can be supplied to the texture PM depending on the function's design/requirements. Additional 'auxiliary' parameters enable developers and designers to externally access and control the PM's internal behaviour and functionality. Parameters therefore, represent the diversity of PMs' in a different sense. A single PM for example,

could be parameterized in different ways; each of which yields different results that makes the function applicable to a variety of situations.

Examples of simple procedural textures

Consider an example where a gradient texture is procedurally generated. For simplicity, the function is solely parameterized by the pixel's coordinate. To achieve a gradient that sweeps across the vertical dimension of coordinate space, the descriptor simply returns the fractional component of the pixels 'x coordinate'. This is illustrated by (A) in figure 3.

	Coordinate	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
A	frac(x × 2)	0.0	0.2	0.4	0.6	0.8	0.0	0.2	0.4	0.6	0.8	0.0
												
B	step(frac(x × 6))	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0
												

Figure 3 Correlation between numerical result and visual representation

Another simple procedural that generates a 'stripe' texture could be achieved by the 'modulo' of a pixels 'y-coordinate'. When evaluated across the 'y axis', the operator would yield uniform alternation between zero and non-zero. By this interpretation, a final texture depicting alternating 'stripes' could be achieved (B in figure 3). Returned function results range numerically between 0.0 and 1.0, and are visually represented by black through shaders of grey to white.

Advantages of procedural methods (explained via textures)

To illustrate some incentives for using PM's, consider the gradient example from the previous section. As illustrated, this simple PM offers a robust mechanism for delivering gradient textures. In a video game, this gradient PM could tint the sky of the game environment for instance, simulating atmospheric effects of the real world. A common, alternative strategy would be to use a sky gradient texture (image) which would be 'sampled'/mapped onto the sky's surface. As subsequent discussion illustrates, an advantage of graphics strategies which are based on PM's is that their data memory/storage requirements are minimized.

The resolution characteristics of procedural functions are preferable to equivalent 'functions' based on discrete data. Consider again the gradient procedural function. In theory, this PM is capable of delivering an arbitrary level of resolution (quality). This is due to the 'decimally

infinite' nature of the function; a consequence of its numerical basis (Gowers, 2004). In practice however, the resolution and quality of a procedural texture is limited by the arithmetic precision of the underlying hardware/technology used. Although recent graphics hardware can support high precision processing (64-bit), most hardware functionality is limited 32-bit number representations (NVIDIA Corporation, 2010) (Brown, ARB_gpu_shader_fp64, 2010). This precession usually offers a sufficient level of colour resolution and quality for most graphics situations.

To achieve results with equivalent quality using conventional data storage, an array containing the full spectrum of 32-bit values between the range of 0.0 and 1.0 would need to be available for 'sampling'. This would require an array containing 4.2×10^9 elements to match the quality of the procedural gradient. Despite this memory consumption, no gains in final texture quality or resolution would be made.

In practical situations where storage constraints are applied, degradation in quality usually occurs. This is due to the discrete nature of data when stored in memory. Although 'filtering' mechanisms exist to supplement these constraints, limitations and low resolution tend to still be noticeable. Figure 4 illustrates the effect of filtering on low resolution data.

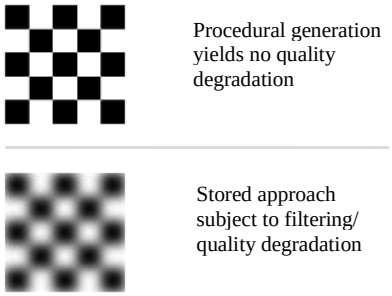


Figure 4 Quality comparison between procedural and conventional reproduction

Composition of procedural functions

Although simple, the previous example illustrated a key benefit behind PM's; this being high-resolution results coupled with a small memory footprint.

In practical applications, procedural textures are typically represented as a composition of many procedural elements which, when combined appropriately, produce more interesting final results. The structure of this composition can be represented as a tree or DAG.

Procedural generators/functions typically occur as leaf nodes in these structures and return data which contributes to the final result. During evaluation of these structures, output from child/leaf nodes flows towards the structure's root via inner/parent nodes (as the arrow in figure 5 illustrates).

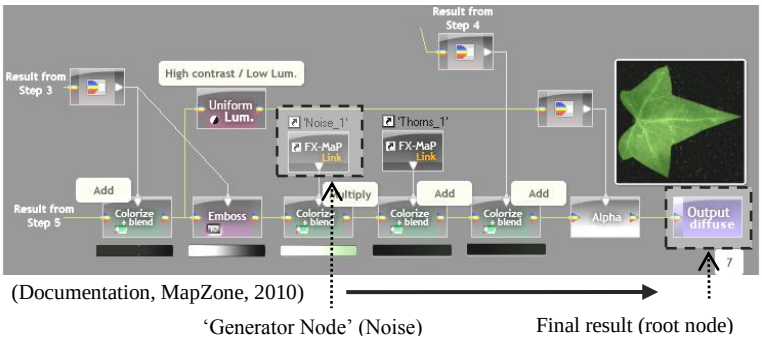


Figure 5 A procedural tree structure composed in 'MapZone'

Processing inner nodes (that bind the generator nodes) combines procedural results/units such as colour tuples, in some predetermined manner. Many combination schemes exist; examples include computing the 'multiplication', 'difference' or 'average' of each component within input data/units. The results of these operations propagate 'up' the tree, towards the root node (see figure 5). These combination schemes provide artists and developers with a powerful avenue for control over the design and final effect of a procedural composition.

Common procedural functions

Despite many procedural classes existing, those classified as 'noise' are most often used in film, simulation and games (Perlin, Making Noise, 1999). Noise procedurals are well suited to these applications, particularly when natural or organic effects are required. By taking advantage of noise within a gaming context, high levels of detail can be efficiently introduced into game scenes and objects.

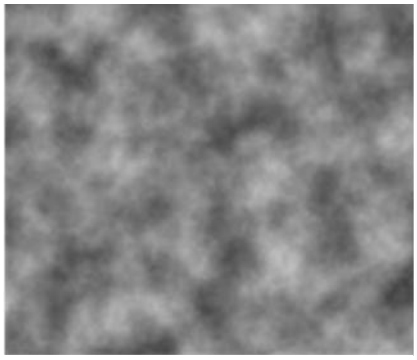
(Bathroom, Allegorithmic, 2010)

Figure 6 Grittiness and grime achieved via procedural noise

Noise's algorithmic quality enables artists to introduce greater levels of detail into game content without crafting it by hand. As figure 6 shows, noise is often compounded onto game objects to introduce an appearance of grime, grittiness and variation. This producing results that are more consistent with the real world.

Procedural noise produces 'pseudo random' results that are similar in nature to the 'random generators' previously mentioned (Perlin, Band

limited repeatable 'random' function, 1999). Procedural noise tends to yield consistent characteristics such as ‘structure’ and ‘form’ which are inherit in the final noise result (see figure 6). In contrast, random generators produce arbitrary results which yield no consistency or correlation between adjacent ‘samples’ of the generator in sample space. When the output of a random generator is expressed as an image, it tends to look like ‘television static’.



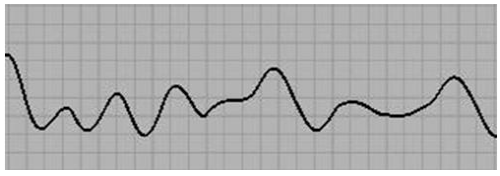
Visual representation Perlin noise expressed in a two dimensional image. Note the structure that is evident in the noise result. (Misc Perlin Noise, 1999)

Figure 7 Visual representation of Perlin noise.

Procedural noise however, yields more distinguishable results which express and emphasise unique characteristics of the function. Useful procedural noise functions are those that possess natural, seemingly ‘organic’ and homogeneous qualities; namely Perlin noise, as shown in figure 7 .

The distinct characteristics of noise procedurals are achieved by different combinations of simple mathematical functions and techniques, including dot product and clamping/bounding operations, as well as trigonometric functions.

Note that the characteristics of noise implementations are always preserved, regardless of where the noise function is evaluated in sample/parameter space. Furthermore, noise implementations must be robust in that a correlation between procedural evaluations exists, when the procedural is evaluated against contiguous ‘parameter values’. For example, consider the Perlin based wave of figure 8. Although variation is evident, intermediate consistency between adjacent ‘segments’ on the wave (each of which corresponds to a point on the contiguous number line), exists. These characteristics are particularly important for creative applications where artists and designers often rely on the preservation of traits (such as structure), within the procedural result.



(DeWolf, 2000)

... 0.21 0.22 0.23 0.24 0.25 ...

Figure 8 Illustrates continuous nature of Perlin noise (1D) when evaluated against contiguous parameters

Case Study: Perlin Noise

To examine these ideas in further detail, consider ‘Perlin noise’. Perlin noise is perhaps the most commonly used implementation of procedural noise, boasting wide spread usage; particularly in the films industry (Perlin, Perlin Noise, 1999). It was developed by Ken Perlin in 1983 and offers a robust method for generating controlled and referentially transparent noise (Perlin, Controlled Random Primitive, 1999).

Perlin's noise algorithm is based on computations between the current 'position' in the sample space and a 'uniform grid' of pre-computed auxiliary vectors. In the case of 2D Perlin noise, the evaluation of noise at each point on the 'image plane' begins with simple vector arithmetic. Four directional vectors which extend from the sample position to the closest intersecting points on the 'uniform grid', are computed (figure 9). The dot product is then calculated between each direction vector and the corresponding 'auxiliary vector' from the 'uniform grid' that implicitly overlay the noise result.

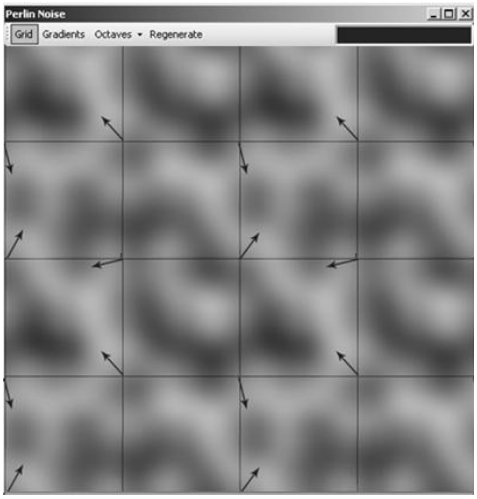


Figure 9 Illustrates a demonstration program that was developed. The 'uniform-grid' and 'auxiliary vectors' which underlie Perlin noise are depicted.

Following this, the leftmost scalar products (with respect to the uniform grid), are interpolated with their rightmost counterparts. This interpolation is based on the sample's horizontal position relative to the enclosing 'cell' of the uniform grid. Perlin's original implementation based this interpolation on an 'S-curve' (Perlin, Noise and Turbluence, 2009). The s-curve essentially blends the scalar values to yield a result similar to 'Gaussian blur' (Perlin, Algorithm, 1999).

The process concludes by repeating the interpolation process on the pair of 'scalar interpolations' previously calculated. This interpolation is based on the sample's vertical position within the enclosing cell. This final interpolation gives the noise sample at this specified current point in sample space.

An interesting subtlety of this implementation is expressed through a property of the scalar product. As samples approach 'cell' corners (or intersecting points of the 'uniform grid'), the scalar product approaches zero. This observation is an example of 'clamping' within a noise procedural.

Regardless of how well vectors of the scalar product align, the results of this operation near cell corners will always approach zero. This property produces a 'radial falloff' around cell corners which contributes to the isotropic characteristic of structure in Perlin's noise (Perlin, Controlled Random Primitive, 1999).

Like most procedural functions, Perlin noise functions often expose parameters to influence the inner noise calculation and thus, the final noise result. Typical parameters include 'amplitude' and 'threshold' (offset) which alter the 'brightness' of the procedural result. In

addition, resolution in the uniform grid can be increased, which yields greater ‘granularity’ in the noise result.

Case Study: Sparse Gabor Noise

Another variant of procedural noise is based on the distribution of ‘Gabor convolutions’ (Lagae, Lefebvre, Drettakis, & Dutr', The Gabor Kernel, 2009). Like Perlin noise, noise based on sparse Gabor convolutions (SGC) incorporates ‘random’ distribution to achieve variety (Lagae, Lefebvre, Drettakis, & Dutr', Procedural Noise using Sparse Gabor Convolution, 2009). The use of a ‘pseudo random’ function for spatial distribution of Gabor samples implies that SGC noise is referentially transparent (Lagae, Lefebvre, Drettakis, & Dutr', Procedural Evaluation, 2009). This approach differs from Perlin noise, and is achieved by accumulating the distribution of simple ‘Gabor convolutions’. For simplicity, SGC noise will be explained as a texture in the context of a 2D plane.

Like other noise implementations SGC has characteristics such as structure and ‘orientation’. SGC noise exposes a number of parameters to control these characteristics, enabling a variety of noise results to be achieved. Most parameters of SGC are directly associated with those of the Gabor kernel(s). Gabor kernels can also be parameterized either uniformly or on an individual basis to produce/achieve other structural characteristics.

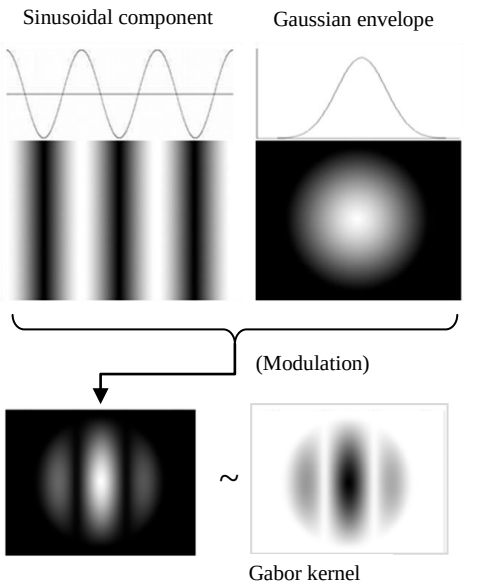


Figure 10 Composition of the Gabor kernel

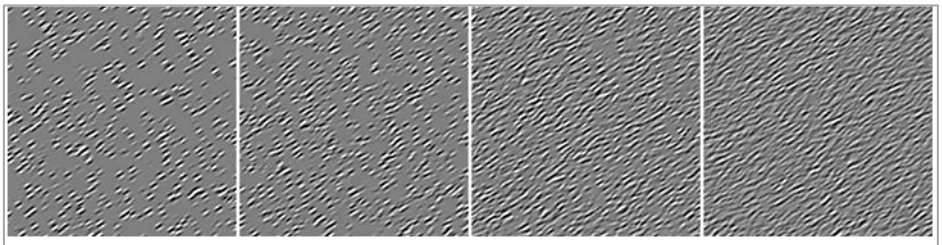
In a two dimensional plane, Gabor kernels represent the modulation between a sinusoidal/harmonic function and the Gaussian function (see figure 10). When expressed as images, Gabor kernels have an appearance of structure and orientation which is a side effect of the sinusoidal element.

More specifically, it is the repetition/oscillation of sinusoidal functions that introduces structure into the kernel. This is because Gabor kernels usually incorporate at least one full phase of the sinusoidal element.

Gabor kernels expose parameters for phase offset and frequency, both of which directly control the kernels sinusoidal element. The orientation of Gabor kernels is manipulated by rotating the ‘axes’ of the sinusoidal function relative to the sample plane. The influence of

kernel orientation on SGC noise, in terms of the resulting isotropy, will be explored in subsequent discussion.

The second component of the Gabor kernel is its Gaussian function. As mentioned, this is applied to the sinusoidal element as an 'envelope' around the kernels 'origin'. This produces the effect of a soft 'fall-off' that encloses the kernel; a feature which makes the Gabor kernel suitable for SGC noise.

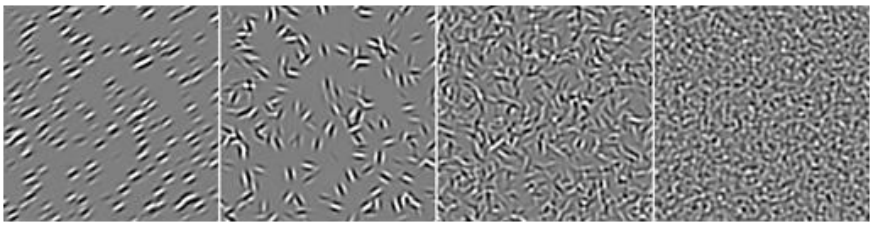


A B C D

'Splatting' Gabor kernels in the "*spatial domain*" GRC (Lagae, Procedural Noise using Sparse Gabor Convolution, 2009). Note that a diagonal 'grain' characteristic emerges as the kernel density increases

Figure 11 The noise result achieved by 'splatting' Gabor kernels

As figure 11 illustrates, SGC noise is achieved via an accumulated 'random' distribution (or 'splatting') of Gabor kernels throughout the image plane (Lagae, Lefebvre, Drettakis, & Dutr', Procedural Evaluation, 2009). Note that the scale and orientation of kernels can also be randomized to achieve different results, as will be discussed. Individual kernels contribute little to the procedural result and thus, a relatively dense distribution of kernels (small Gabor kernels most often being used), is required. It is important however, that the sinusoidal element of a scaled kernel still be perceivable. This is because the sinusoidal element affords 'energy' and structure in the resulting noise.



(Lagae, Procedural Noise using Sparse Gabor Convolution, 2009)

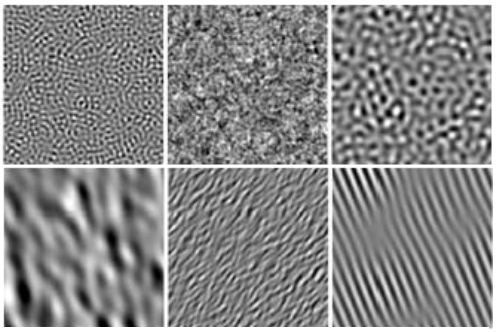
A: Sparse, unidirectional kernels	B: Sparse random direction kernels	C: Random direction kernels	D: Dense accumulation yields final result

Figure 12 Achieving isotropic noise via random distribution of Gabor kernel orientations

'Direction', which can be varied, is characteristic of SGC noise. Direction is achieved when the orientations of sinusoidal elements in each Gabor kernel partially align throughout the overall image. By applying wholly random distribution to the orientation/rotation of kernels

as shown in figure 12, isotropic SGC noise can be produced by essentially 'dissolving' any notion of directional structure in the SGC noise result. The overall affect of misaligned kernels eliminates any directional structure, therefore producing an appearance similar to Perlin noise.

The Gaussian 'envelope' of each kernel is also significant in terms of contribution to the final image. This feature maximizes the 'entropy' of a kernel, while maintaining 'harmony' between neighbouring or partially overlapping kernels in the image plane (i.e. the 'soft' kernel envelop blends with other overlapping kernels). This also ensures that the contribution of kernels in the final result is achieved without making individual kernels distinguishable or noticeable.



(Lagae, Lefebvre, Drettakis, & Dutr', Procedural Noise using Sparse Gabor Convolution, 2009)

Figure 13 Illustrates a range of results that can be achieved via GRC noise

Procedural methods in games

During the history of game development, procedural methods (PM) have been applied to games in a variety of ways with varying degrees of importance. PM's were used in games to deliver volumes of content that were too large to store on distribution media and/or system memory. In this sense, PM's were used as a form of data compression within games.

(Brooks, 2010)

Figure 14 The Sentinel uses procedural methods to generate a large set of playable levels

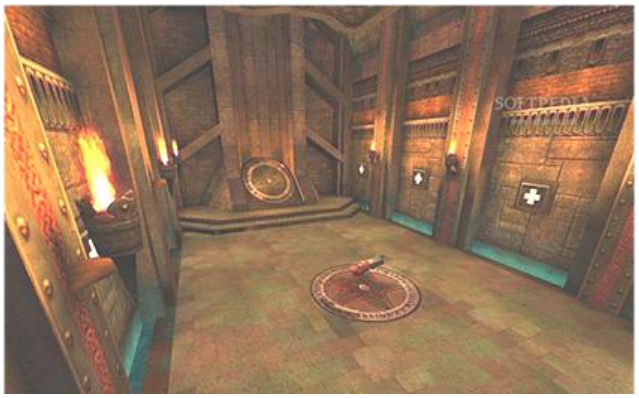
A well known example of this was 'The Sentinel'; a game which was published in 1986 and capable of providing players with up to 10,000 procedurally generated levels while running within 64kb of memory (The Sentinel , 2010). Although the main motivation for using PM's in 'The Sentinel' was data compression, other reasons for this application of PM's may have also existed. By automating the process of content generation, much of the burden of content creation was transferred from the game's developers to the system. Thus, PM's

present an opportunity for data compression in games, in conjunction with improved game production processes. These ideas constitute the direction and theme of this research.

In comparison to today's video games, 'The Sentinel's' application of PM's was central to the game's implementation and delivery. Historically, this option was viable due to the climate of the game's market with lower consumer expectation. PM's served as an attractive and efficient method for authoring games content, given that the approach satisfied game/production quality milestones. During this era, PM's were also used for audio, graphics and challenges/game play. Hence, PM's tended to play a central and highly influential role in most aspects of the game experience.

In contrast, today's games are largely based on (static) content that is manually prescribed by artists and designers. The transition from generative content to prescribed content started in the 1990's and was based on a number of factors; namely significant improvements to graphics processing and storage capacity of consumer gaming hardware (Kudler, 2007).

The use of PM's in game graphics was consequently replaced with image based 'texturing' as graphics hardware became capable of storing and rendering images at reasonable resolutions. With the advent of 3D graphics acceleration in the mid to late 1990s, painted textures and hand crafted geometry quickly became staple elements of game art (GeForce 256, 2010) (GPU, 2010).



(Quake 3 Arena Screenshots, 2006) (Quake III Arena, 2002)

Figure 15 Quake 3 Arena applied textures via improved capabilities of consumer graphics hardware

As a result, graphics techniques and algorithms orientated around these art forms, were developed by the industry (Lilly, 2010). One other hardware development played a vital part in this paradigm shift; namely the widespread adoption of CD/ DVD media. These media provided significant distribution space for texture and geometry data, making the use of 'prescribed' game content more feasible (Optical disc, 2010).

During this transitional period, PM's remained a feature of the game production process. These applications were often in 'pre-baked' forms however, meaning that the procedurals were pre-evaluated before being introduced into the game. A common example of this is the use of procedural functions during the creation of textures. In these situations, procedural operations are often compounded into a texture result that is crafted by the artist (Ahearn, 2006). Another example is the application of procedural modifiers provided in modelling packages such as Maya and 3D Studio Max (Matossian, Ms, 2001). These modifiers apply noise, waves and other distortions to target geometry, and are applicable to a variety of modelling situations. Despite PM's being present in the production process, these are somewhat 'superficial' applications because they are compounded into a static form.

This research however, aims to integrate PM's so that the benefits of evaluation at runtime, such as compression, are achieved.

As mentioned, the mid 1990's saw a rise in game content that was manually crafted by artists and designers; an approach that differed significantly from the previous decade. Despite this significant shift, some game titles still used procedural techniques to achieve effects and phenomena of their game experience. A typical example was the use of a 'noise function' (usually a pseudo random number generator) to compute vectors with random orientation (tr_noise.c, 2005). These vectors can be used to give debris a random initial velocity following an explosion, resulting in a seemingly natural distribution of debris.

Procedural methods are also used in 'Age of Empires' (AOE), a real time strategy game published by Microsoft (Age of Empires, 2010). AOE provides players with causal game modes that take place in procedurally generated levels. Although procedural levels are not part of the games storyline/campaign, the feature offers additional game play via algorithmic map generation. Note that AOE's procedural map generation process is also parameterized by simple criteria such as terrain type and foliage density (Microsoft Age of Empires, 1998).

(Age of Empires,Features, 1998)

Figure 16 Illustrates terrain, procedurally generated by Age of Empire's level editor

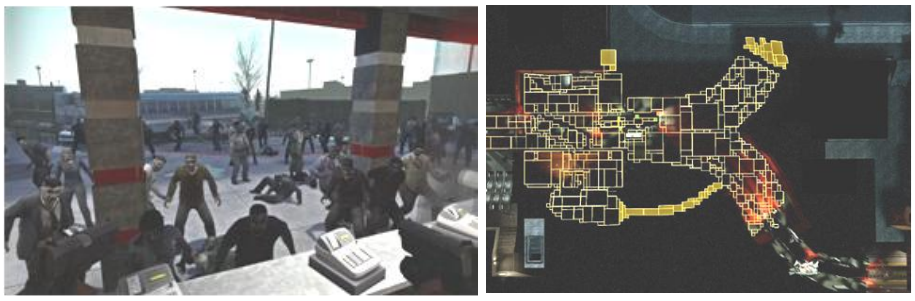
These examples show an interesting 'relationship' between procedural functions and the roles they fulfil. Although each is expected to yield desirable characteristics, the actual evaluated outcome is at the discretion of the procedural function itself. Thus, the integration of these

procedurals hinges on procedural characteristics being manifested, rather than a specific 'layout' of characteristics being produced. In terrain generation for example, a particular arrangement of hills isn't necessarily required provided, that variation across the landscape exists.

Procedural functions are therefore selected, based on the characteristics that they manifest. This selection criterion still seems to apply to modern games that use PM's. Noteworthy examples include acclaimed titles such as 'Left 4 Dead' and 'FarCry 2', both integrating PM's in sophisticated ways to deliver richer game play experiences (Far Cry 2, 2010) (Left 4 Dead, 2009). Because these games make central use of PM's, it is critical that procedurals are carefully selected, tweaked and integrated, to ensure solid game play and end-user experiences. The subsequent section examines relevant cases in more detail.

Case study: Left 4 Dead

Left 4 Dead is an action game where players fight against large numbers of 'infected zombies' in the aftermath of a "zombie apocalypse" (Left 4 Dead, 2009). High action game play is achieved in Left 4 Dead by procedurally instantiating enemy zombies beyond the players' line of site via 'Structured Unpredictability' (Booth, 2009). Thus, by basing the instantiation and placement of enemy zombies on PM's, Left 4 Dead delivers variation in the game experience.



(Booth, slide 62, 2009).

Figure 17 Illustrates the result of procedurally based enemy placement and control in Left 4 Dead.

This is achieved by 'The Director', a subsystem of the game that coordinates events and situations to avoid stale and repetitive game play (Left 4 Dead, 2009). An advantage of The Director is that it minimizes the need for the definition of scripts throughout an entire game. In a typical game, scripts are usually provided to control the placement and behaviour of enemies. Thus, the task of scripting is essentially offloaded to The Director, which incorporates rule systems, scene analysis and various heuristics, to automatically perform

zombie management (Booth, 2009). The outcome of this procedural system is dynamic behaviour throughout many aspects of the game, which reduces the need for script development.

In addition, The Director adds a further dynamic to the game experience, namely difficulty scaling (Booth, 2009). If the system detects a high error rate by the player for example, The Director can dynamically respond to this by minimizing the game's difficultly at runtime to suit that particular player.

Case Study: FarCry 2

FarCry 2 uses PM's in a different way to achieve unique content generation. The game is capable of delivering massive and detailed game environments, as well as a huge variety of game characters/enemies, through the integration of a procedurally based content generation system (Far Cry 2, 2009).

It is interesting to note the revival of traditional uses of PM's in FarCry 2. Recall from previous discussion the hardware constraints that faced developers of The Sentinel. These constraints led to PM's playing a central role in delivering large volumes of data for the game. In FarCry 2, this same situation is manifest through its objective to deliver a diverse population of in-game characters beyond the storage capabilities of gaming hardware (Breckon, FarCry 2 Preview, 2008). Thus, FarCry 2's procedurally driven character generation system is capable of delivering this variation by dynamically generating game characters 'on-the-fly'.

The obvious distinction between the use of PM's in FarCry 2 and The Sentinel however, is FarCry 2's use of 'artist prescribed' content. Thus, FarCry 2 merges PM's with artist prescribed content, to deliver a 'hybrid' character generation system. This strategy allows the game to achieve a high level of quality, realism and scale, despite being implemented on game consoles with tight memory constraints (~512mb) (PS3Focus, 2005).

Case Study: Roboblitz

Roboblitz was released in 2006 and is a 3D action game which makes explicit use of PM's for much of its game art (RoboBlitz, 2010). It integrates a variety of procedural functions such as noise, pattern and shape generators, to compose a variety of textures for effects and environmental surfaces.

In recognizing a relationship between procedural compositions and the game's intended art style, developers could deliver the product in a 50Mb package (RoboBlitz, 2007).

(RoboBlitz, Gallery, 2006)

Figure 18 Illustrates the use of procedurally generated textures in Roboblitz

Despite its small size, critics have estimated that Roboblitz offers approximately 5 hours of game play (Brudvig, 2007). This is noteworthy given that comparable games in excess of 1000Mb typically only offer 8 to 12 hours of game play. Roboblitz's size is attributed to its use of PM's as a substitute for 'conventional' game media data.

Roboblitz locally 'unpacks' the game's 'procedural data' into conventional (uncompressed) forms, which are then made available to the game. This significantly offsets the game's initial size, while still enabling it to deliver a solid visual and interactive experience.

From a marketing perspective, smaller software sizes are advantageous as this makes deployment through channels such as the internet, feasible. Roboblitz capitalizes on its small size, exclusively using digital/internet stores such as 'Steam' and 'Xbox Live Arcade' for marketing exposure and sales (RoboBlitz, 2010).

Although Roboblitz takes advantage of PM's to improve distribution prospects, it doesn't apply or evaluate PM's for texture generation during runtime. Thus, the small size of PM's does not benefit Roboblitz during runtime and thus, the game is subject to the same 'runtime capacity constraints' that face typical games.

The following table summarizes these case studies. It also provides an outline of PM's within other popular games, therefore illustrating the diverse range of functions they fulfil.

Game Title	Application	Integration
The Sentinel	Generative game levels	Procedural functions for level generation are evaluated at runtime.
Roboblitz	Texture generation for game surfaces	Procedural functions are deployed with the game but are expanded /evaluated before runtime. The unpacking process yields procedurally generated texture images which are used in a conventional way during runtime (Postmortem: Naked Sky Entertainment's RoboBlitz, 2007).
Left 4 Dead	Placement and behaviour of enemies	Procedural functions are used to position and control opponents within the game world. These procedural functions are evaluated at runtime (Booth, 2009).
Left 4 Dead 2	Placement and behaviour	Similar to Left 4 Dead.

	of enemies	
	Dynamic level generation	Procedurally driven decision making is used to dynamically configure levels (Champandard, 2009). Procedural decision making is evaluated at run time and is influenced by factors such as player skill. (Runtime Random Level Generation, 2009).
Far Cry 2	Environment generation	Procedural functions were used during development to generate massive game environments, with physically accurate characteristics. This process however, 'bakes' the procedural data which is used by the game at runtime, in a conventional way. This therefore, represents a static integration of PM's in the game (Far Cry 2, 2009) (Making Far Cry 2's Africa, 2008).
	Dynamic character generation	The integration of procedural character generation. Evaluation of these procedural functions to achieve character variety with minimal memory footprint (Far Cry 2, 2009).
	Dynamic skies	Procedural functions are used to produced dynamic skies within game environments of Far Cry 2 (Rossignol, 2008).
.kkrieger	Generative levels, opponents, textures, sounds, effects	Procedural functions are deployed and evaluated at runtime to generate most of the game's media.
Spore	Character colouration	One of Spore's developers stated that "*Spore uses a procedural paint system* [for game characters]" (Hecker, 2009).
	Animation	The game generates animation based on arbitrary 'creatures' that are created by players at runtime (Procedural generation, 2010).
	Music	Spore integrates a software component that is based on procedural functionality called 'The Shuffler'. This component "*accepts input based on the game's parameters* [and] *can turn even a small combination of samples into a composition which will never repeat, no matter how long you will play the game*" (Whiting, 2007).
Quake 3 Arena	Special effects	Perlin noise is used to generate animated water distortion effects (tr_shade_calc.c, 2010).
Table 3 A summary of PM's in a range of games		

As these examples illustrate procedural functions, namely noise procedurals, serve a variety of purposes in games. Noise is a characteristic that is inherent at all levels in the real world. Thus, noise is often manifested in the accumulation of 'detail' in the world. When comprehending the levels of detail in characteristics such as dirt, vapour or rust for example, the compounded effect of these are often perceived as 'noise'.

Thus, it is perhaps this observation that explains noises wide spread use in games. Furthermore, it is the ubiquitous presence of noise in the real world that makes the sensible application of procedural noise within games, a convenient way for integrating more convincing and believable elements into a game's overall experience.

Motivations for procedural methods

In the current climate of game development, the application of PM's as a primary source for graphics rendering is rare, particularly for high budget titles. As demands for improved visual fidelity continue to rise, the push to maximize the capabilities of gaming systems is however, projected to proportionally increase. To accommodate this trend, alternative forms of data representation (i.e. PM's), are more likely to play a pivotal role in the graphics of games.
Key factors in this change are limitation of storage capacity and a growing need for web based distribution. Despite the current generation of gaming consoles supporting media capacities from between 9GB (Xbox 360) and 33.4GB (Playstation 3), the issue of compression and data organization is becoming increasingly important in modern game projects (Orry, 2005) (Ivan, 2010).

In 2008 Id software's lead designer Tim Willits, spoke to this issue directly during an interview on one of the company's current high end game projects called 'Rage' (Breckon, id: Rage Content Cut due to Xbox 360 Size Limit, 2008). In this interview Willits mentioned the negative economic implications of distributing their game across multiple discs for the Xbox 360. A more recent statement made by game development studio 'Naughty Dog', indicated that the Playstaion 3's Blu-Ray media had been fully utilized in order to deliver their latest action game, 'Uncharted 2' (Bantick, 2009).

Steps towards PM's being a viable possibility are being made however, as is evident through the development of new middleware technologies. This 'technological shift' places a particular emphasis on the design and implementation of authoring tools that underlie the 'content creation pipelines' for games.

Middleware development and avenues for procedural integration

Given the minimal storage requirements of PM's, their widespread application in games seems imminent, if improvements are to continue in visual quality. At the forefront of this endeavour are software companies such as Allegorithmic. Allegorithmic has developed a

sophisticated set of authoring tools, focused on procedural texture composition, that are directly used in game graphics (Allegorithmic, About, 2010). One tool in particular being 'MaPZone', enables artists to harness the power of PM's to create highly detailed and realistic textures that are completely generative (Allegorithmic, Products, 2010). Allegorithmic's technology evaluates procedural textures at runtime, therefore imposing a relatively small runtime memory footprint.

Furthermore, MaPZone permits "higher resolution textures", through its basis of PM's (What is MaPZone?, 2010).

High resolution textures which incorporate high levels of detail are relevant to many games. Examples include flight simulators and first-person shooters, where high resolution textures can reduce 'texture tiling' and/or 'texture filtering' (blurring), when underlying surfaces are viewed from certain vantage points.

It is important to note that high resolution textures in games must be complemented by equivalent levels of geometric complexity in game graphics, in order to unify the game's overall visual delivery.

Content creation pipelines

Achieving increased levels of detail in games obviously introduces a new range of technological and production challenges for developers. This is particularly true for artists and modellers, when creating game environments as it significantly increases their workload. To match current and projected production demands, game studios are realizing the need for new development strategies; particularly through the optimization of 'content creation pipelines'.

Given the variable characteristics of game projects, studios tend to place emphasis on different aspects/strategies of the content creation process. The following section shows some 'pipeline features' that are important to the ambitions of game projects and/or companies.

Crytek, 'LiveCreate'

'Crytek' is an industry leader in game/'game technology' development and as mentioned is responsible for developing the infamous first person shooter, 'Farcry' (Far Cry, 2010). Since Farcry's release in 2004, Crytek has remained a strong and well respected competitor in the development of its game technology, 'CryEngine'.

The CryEngine is renowned for delivering high quality visual experiences in games (CryEngine, 2010). One important feature is the engine's cross-platform capability

(3DVision, 2010). In addition, the company offers a software system called 'LiveCreate' as part of a software suite which enables licensee's of the technology to easily and efficiently harness the features of the CryEngine.

In August 2009, Crytek demonstrated LiveCreate at the European Game Developers Conference (Jube, 2009). LiveCreate was presented as a solution to the issues of content creation that hinder the game development process, particularly the tedious flow of game data in production (Jube, 2009).

As the name indicates, LiveCreate enables developers to create game content in a 'live' and responsive way. This is achieved by centralizing the role of game engine technology in the creation process. Thus, the CryEngine's real-time renderer is used to provide a live display of the content being created by artists. The process of 'manually' transferring art content from authoring tools into a game engine/project is thus, eliminated.

The low-latency, real-time nature of this system has other positive implications for the authoring process, particularly 'content prototyping'. Providing a content creation environment that 'connects' to the game's renderer, provides greater opportunity for artist experimentation, as well as quality tuning.

LiveCreate's real-time feedback is also beneficial for other aspects of production, particularly shortening project duration. The cumulative effect of efficient data flow in the content creation pipeline has positive implications towards project deadlines being met.

In addition to these benefits, LiveCreate addresses another major challenge that has plagued game development studios in recent years, that being cross-platform development.

With the major gaming platforms offered by Sony, Nintendo and Microsoft (as well as the PC) serving as principal avenues for market exposure, studios often seek to maximize potential income by ensuring their games are available on most, if not all, platforms. Due to hardware variation between platforms however, this presents a series of technical challenges that must be considered by developers in order to preserve the game play experience for all customers. This has had negative implications on the outcomes of projects, particularly in terms of production cost, duration and overall quality. LiveCreate's cross platform capability however, simplifies the technical and artistic issues that face developers of high definition, cross-platform games. By applying LiveCreate in the development process, only a single development 'pathway' is necessary to deploy a game across the three major gaming platforms (PC, Xbox 360 and Playstation 3). LiveCreate eliminates the need for separate development teams within a studio where each sub-team would traditionally be dedicated to delivering the same game project on each target platform.

The main area of interest in LiveCreate however, is the high level of feedback that it offers to artists during the content creation process. This aspect of LiveCreate is not only consistent with trends seen in other game studios, but it is also relevant to the focus of this thesis.

Id Software: 'IdStudio'

Texas based software company 'Id Software', also sits at the forefront in the development of highly integrated content creation pipelines. Id Software has a reputation for innovation which stems from its introduction of the 'first person shooter' gaming genre (Cifaldi, 2006). Since the early 1990's, Id Software has remained at the cutting edge of graphics technology and visual quality in games. The company is responsible for developing the infamous video game series'; Doom and Quake (Id Software: Final Doom, 2001) (Id Software: Quake, 2001). In addition, Id Software is known for innovation in the graphics and development in games. Thus, Id Software has stated an interest in the use of content creation systems that offer a high degree of feedback to designers and artists.

'IdStudio' is the company's proprietary tool set which is deployed with the company's current generation of licensed game engine technology, and is designed "primarily with artists in mind" (Accardo, 2007). Id Software's technical director John Carmack, specifically states that IdStudio gives artists "*as much creative freedom as possible*", during the creative process (Accardo, 2007). Furthermore, Carmack has indicated the tool's ability to allow artists to "*paint*" or "*scrub out*" areas of a game environment in an interactive, in-game context (Carmack, 2010). In addition, IdStudio places emphasis on artist and designer collaboration throughout the creative process, thus allowing for parallel and efficient development.

IdStudio's emphasis on collaborative content creation is relevant to an emerging trend of 'large scale' and 'open world' games in the industry. A consequence of this trend is the increased need for collaboration between teams of artists and designers, in order to keep such projects feasible. Interestingly, Id Software's current project 'Rage', fits this 'open world' criteria. Through the use of IdStudio's collaborative capabilities, the studio is able to more effectively develop the title by enabling paralleled development by teams of artists working on the project.

Like the CryEngine, 'production builds' of Id Software's 'tech5' engine are capable of running games across all HD gaming platforms with consistent performance and visual quality (Carmack, 2010). In contrast however, IdStudio has not demonstrated the ability for real time development across all HD gaming platforms, as is possible with LiveCreate.

Epic Games: 'Unreal Engine'

As shown, prominent studios have sought to synthesise game technology and content authoring to streamline production workflow. Interestingly, each case study shows that emphasis has been placed on different aspects of the tool chain; namely cross-platform support and integrated collaboration. Despite these differences, both IdStudio and LiveCreate are unified by the same underlying concept; that being a deep integration of technology in the production process. This concept however, is well established in the game industry's timeline.

In 1998 the first person shooter 'Unreal' debut and was perhaps the first product to incorporate an integrated tool chain (Unreal, 2010). Unreal was shipped with additional software on disc; the studio's own world/level authoring tool, 'UnrealEd' (Unreal, 2010). The addition of UnrealEd was well received by the game modification community and as the Unreal franchise developed, so too did the accompanying tools.

UnrealEd's continued development led to the introduction of a revolutionary concept; a tighter integration of games technology with authoring tools. In 2003, a significant revision to the third version of the tool was debut which integrated the games rendering technology into the level editor's interface (UE2:UnrealEd 3, 2008).

This was a notable milestone in game production processes, which arguably started a new trend in the implementation of game development tools. Over the following decade, further developments were made to UnrealEd which included the introduction of physics simulation and hardware accelerated graphics (Golding & Nalezynski, 2010).

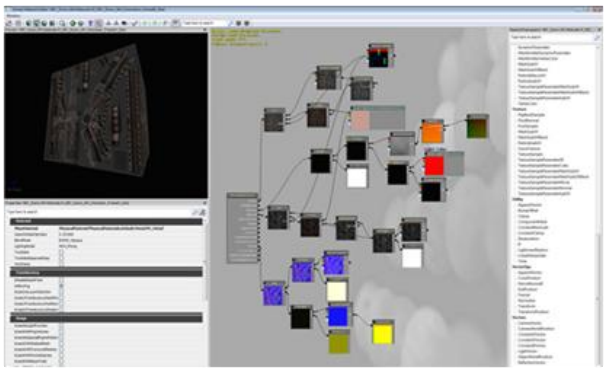


Figure 19 The interactive shader/material creation offered in Unreal 3's tool suite

The current generation of Unreal's tools and technology have harnessed the integrated concept to its fullest extent, allowing artists and designers to fully play/test levels and content directly from UnrealEd's 'viewport' (UnrealEd, 2010).

Furthermore, users of UnrealEd for the 'Unreal 3.0' engine can compose complex materials and surface textures via the editors 'shader assembly' system (figure 19). Developers can also specify and test complex physics simulations directly within UnrealEd, due to its close integration with the 'Unreal 3.0' technology.

UnrealEd also inherits a series of graphics features from the engine. Through this, complex lighting effects can be directly manipulated by artists within the UnrealEd interface in real-time. This demonstrates a new application for accelerated graphics hardware, wherein the hardware's capabilities are focused on enhancing the game production process.

Summary

The topics that have been discussed in this section convey ideas and theory that are relevant to this thesis. This discussion reviewed the general concept of PM's, including a detailed analysis of common/relevant procedural functions. The discussion outlined the main characteristics of PM's while placing an emphasis on the seemingly natural/organic effects that certain classes of PM's can algorithmically produce.

Further discussion showed the important role of PM's throughout the history of games and notable issues concerning modern game development. In particular, the role and influence of art/content creation in game development was explored, highlighting an emerging theme of 'interaction' in modern game creation strategies.

In addition, this discussion highlighted the opportunity for a feasible and potentially beneficial integration of PM's into the modern game development process. This opportunity suggests the untapped potential of PM's in game content creation.

In keeping with the identified trends of modern game development, this research explores the integration of PM's into interactive game content development. Thus, the motivation is that PM's serve as a mechanism to enhance and automate aspects of content production for games.

Chapter 3: Project design

A noted trend in development strategies used by prominent studios of the games industry is ‘interactive processes’ that underlie game art and content creation. The ‘interactive element’ gives the artist the ability to visualise production content in the context of game rendering technology, during the production process. This introduces a high level of ‘feedback’ for artists’, which enables tailored content creation that suits the game and it’s rendering technology. Perhaps the most important implication of this strategy is the potential for gains in artist productivity. The element of high feedback also provides increased opportunity for ‘artistic prototyping’ which has positive implications towards the final quality of game art.

(Adams, 2009)
Id Software’s current game project ‘Rage’ depicts large scaled environments, which are likely to benefit from IdStudio’s collaborative elements

Figure 20 Image of Id Software’s current game project ‘Rage’

Given the ongoing goal of improved visual quality and detail in games shared by many development studios, it is not unreasonable to assume that these ‘common objectives’ would ‘unify’ the industry. As the literature revealed however, studios place emphasis on the unique functions of their own tools and content creation processes.

The highlighted feature of CryTek’s ‘LiveCreate’ tool chain for example, is that it allows concurrent game content development across all target platforms, in real time. This delivers responsive, immediate feedback and serves as a valuable indicator to ensure visual consistency of artwork across different platforms. LiveCreate’s core authoring capabilities

are however, orientated around the use of its proprietary content authoring software for creating game environments called ‘SandBox’ (CryENGINE® 2 Specifications, 2010).

In contrast to LiveCreate’s focus on cross platform capabilities, Id Software’s ‘IdStudio’ reportedly places more emphasis on artist collaboration, enabling large teams of artists to work simultaneously on assets and content within the same project. This therefore, emphasises the distribution of tasks amongst artists. When considering Id Software’s current open environment game ‘Rage’, the collaborative features are obviously well suited to this project. Thus, the game’s large scale facilitates concurrent development between teams of artists, given that the game’s vast environment minimizes risk of artist conflict or interference during production. It would be difficult to imagine Id Software integrating this functionality if the game project itself didn’t justify the need.

Although the solutions are both relevant to the game development process, they are tailored to the agendas of each studio/company and thus, don’t fully address a series of emerging and fundamental challenges which face game developers; namely capacity constraints of target platforms, in conjunction with focused productivity strategies for content creation.

As mentioned however, the games industry is interested in new and improved approaches to game development; particularly in content creation. These novel strategies strive to address bottlenecks that hinder content production by centralizing the role of interaction and feedback in artists’ workflow. It appears however, that an opportunity for further development to these ‘artist centric’ content development strategies exists; namely through integration of automated methods. By combining the interactive element of contemporary content creation, with procedurally driven mechanisms for automation and enhancement, it’s plausible that further gains could be made.

Development constraints

The literature review revealed issues common to many studios, particularly ‘capacity constraints’ that overshadow console development. Numerous studios have commented on the limitations concerning memory and storage capacity that arise when developing for both the Playstation® 3 and Xbox 360®. This work aims to address issues that affect all studios. The goal is that by identifying universal factors, the outcomes of this research may be widely applicable to most, if not all, game development processes.

Another long standing universal problem that has faced software development is that total project duration tends to exceed deadlines. In the context of game development, this issue takes on a new dimension as deadlines can be violated by the production of game art/content. Thus, investigation into strategies that minimize the overhead of content creation is high on the agenda of this research. Although systems such as 'LiveCreate' and the Unreal 3 engine provide responsive content authoring environments that address some issues, there exists an opportunity for investigation into the integration of responsive work flow with methods that address the issue of content capacity.

This presents an opportunity for the use of procedural methods in the graphics of games, which has the potential to address game production constraints and hardware limitations. As the literature review discussed, PM's provide diversity, small memory footprint and the capability to deliver results with high precision.

However, the perceived shortfall of PM's is that their application substitutes the traditional skill domain of artists (typically based on digital painting and brush strokes), with the specification and tweaking of procedural parameters. It is conceivable however, that in the context of an interactive tool chain, a process of parameter tweaking would be acceptable, given the high level of visual feedback that results. The integration of PM's in this way should reduce the penalty/overhead incurred by iteratively tweaking not only procedural functions, but also the geometry of game objects. The central theme of this research therefore, is the integration of PM's into a responsive and interactive tool chain.

Industry consultation

As part of the preliminary work for this thesis, programmers and technical artists at Wellington based game studio 'Sidhe' were interviewed (Sidhe, 2010). This interview reiterated that the themes and objectives of this research for improved game graphics via effective content creation strategies are well aligned with the needs and climate of the games industry.

Although Sidhe studio doesn't currently integrate a responsive/interactive tool chain, the positive implications that this would have towards Sidhe's internal projects was appreciated by the studio's staff. Furthermore, Sidhe's positive response to the concept of an interactive tool chain suggests that the integration of PMs into such a system would also be well received. This is based on the understanding that the integration of PMs in an interactive tool chain would permit interactive parameter tweaking/alteration.

During this interview, staff at Sidhe also spoke about issues regarding the shortfalls of PM's that were mentioned previously. A number of inherit limitations in PM's were discussed; particularly scenarios where 'explicit' artist control is required in specific 'portions' of the procedural/evaluated results. If for example, an artist requires a grimy 'noise-like' texture for a metal panel surface, which consists of details such as 'rivet heads' and bolts, a noise procedural would typically not be used, given that noise makes no provision for these 'prescribed' details and features.

Hybrid solutions were subsequently discussed, to address these situations. This involved a combination between artist prescribed 'elements' and PM's to deliver 'solutions' that retain the benefits of procedural functions, in conjunction with conventional 'artist control'. The general consensus between staff at Sidhe was that a hybrid solution such as this would be sufficient for a range of situations.

In addition, the integration of PM's as a mechanism to compliment 'conventional' game art was also suggested. The primary example of this involved a combination between 'painted textures' and 'detailed procedural methods' in order to procedurally enhance the resulting texture.

The underlying theme of this discussion however, was that the level of control offered to artists and designers via traditional artistic methods, is still highly valued. Furthermore, Sidhe's developers reiterated that the required levels of artistic control offered by traditional artistic methods, outweighs the negative implications that they may impose on production fidelity (i.e. greater memory footprint, lower resolutions).

The previous section provided insight into the challenges and needs of the games industry, revealing an element of tension between these factors. Improving game fidelity requires 'capacity friendly' data strategies to deliver richer, more compelling gaming experiences. As the literature review illustrated, PM's have historically served as an attractive option for delivering large volumes of data to achieve this. In terms of game art/content however, discussion with industry members showed a significant emphasis is placed on high levels of 'artistic control'; which PM's tend to lack.

An opportunity for further improvement in game development could therefore exist, by achieving greater synthesis between these game development factors. Through the use of an interactive development environment that encapsulates 'hybrid' techniques with 'automated' content creation strategies, the benefits of PM's in game content creation may be better realized.

Automated object placement

Automated content creation strategies provide an opportunity for use of PM's in the content creation process. This section looks at object placement in game scenes. Based on current trends, the density and complexity of geometry in typical game environments can be expected to increase far beyond the already high levels of detail in current games. As a consequence, artists are burdened with the task of highly prescribed or even manual object placement within scenes; a tedious process which is capable of consuming considerable production time.

For many of these situations however, it would be satisfactory to automate this process algorithmically. An example of this might be procedurally driven placement of foliage across terrain. This would avoid the need for 'prescribed' and tedious placement of foliage objects, while potentially retaining a 'natural' overall appearance.

(Woody, 2010)

This scene depicts the natural distribution of foliage. By applying procedural algorithms, this process could potentially be achieved in an automated fashion. Fable 2, was developed by Lionhead Studios (Lionhead Studios, 2010).

Figure 21 Image of a forest scene in Fable 2

By coupling such an algorithm with an interactive tool chain that exposes procedural function parameters, the potential for an accelerated and highly configurable object placement strategy exists.

Artists would set parameters for a procedural function in order to algorithmically control the instantiation of objects in a scene. Consider the previous example, where foliage is distributed across a hill side. Starting with foliage and ground surface geometry, an artist would associate the foliage asset with the ground.

The system would internally 'bridge' this association with an artist specified procedural function, such as Perlin noise (figure 22). This association would avoid the tedious process of

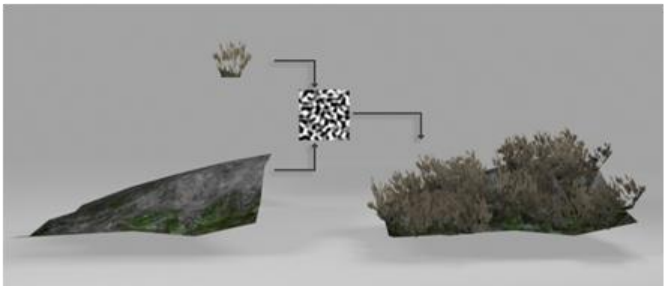


Figure 22 Visualization of the algorithmic instancing concept, showing the relationship between geometric elements and procedural functionality

manually placing foliage across the ground surface, by offloading the task onto the system which would instantiate foliage at discrete positions determined by the noise function. As the procedural is evaluated across the ground surface, the evaluations could be reduced to Boolean values, to 'mask' the instantiation of foliage at discrete points. This 'mask' could yield a non-uniform and thus, seemingly natural distribution of foliage across the ground. By tweaking parameters of the Perlin noise procedural, the density and regularity of foliage could be controlled. Because this algorithm is implemented in an interactive tool chain, parameter changes could be made interactively. This would enable artists to rapidly identify a suitable configuration which aligns with the artistic vision for the scene.

Direct access to game rendering technology would play a central role in this concept. This is because the technology has real-time rendering capabilities which can immediately show the algorithm's results. Thus, the tool chain's game renderer should encapsulate the instancing implementation, therefore yielding responsive, visual feedback to artists during instancing alteration/crafting. The ideal integration of this algorithm would therefore compute object instancing 'on-the-fly' to enable interactive authoring of object instantiation by artists.

The concept of procedural instancing serves as a compelling case for using PM's in games. Furthermore, the concept would naturally extend to give artists' control over 'data channels' that orientate and scale instanced objects, via procedural functions.

A B

Increased realism from standard instancing (A) can be achieved by adding other channels of procedural data; namely scale and orientation to the instancing computation/process (B)

Figure 23 Visualization of other channels of procedural data in procedural instancing

As an aside, 'channels' represent a widely understood concept amongst those in digital/game art communities. Perhaps the simplest and most common example of channels in graphics is manifest in 'colour'; an accumulation of independent channels that describe intensities of red,

green and blue. Multiple sources (or ‘channels’) of data are often required in graphics content, to achieve certain effects or rendering results. Figure 23 provides a visualization of how this ‘channel extension’ could work to allow more sophisticated and realistic instancing results.

In an ideal implementation of this instancing algorithm, artists would be able to specify unique/independent characteristics for rotation, scaling and masking, amongst a population of instanced objects.

For most situations where instancing is applicable, the need for artist specification of more than one instancing ‘channel’, is almost always necessary. Consider the foliage instancing example in figure 23; this depicts a scenario ubiquitous to games. Image (B) shows how the cumulative effects of variety amongst the orientation and scaling of instances improves realism compared to image (A).

As discussed in the literature review, procedural functions are inherently abstract and can be manifested/expressed in many ways. When applied to different information channels however, procedural functions must be appropriately interpreted. Figure 24 shows how procedural functionality interpreted as a mask in object instancing, could yield a straightforward and legible relationship.

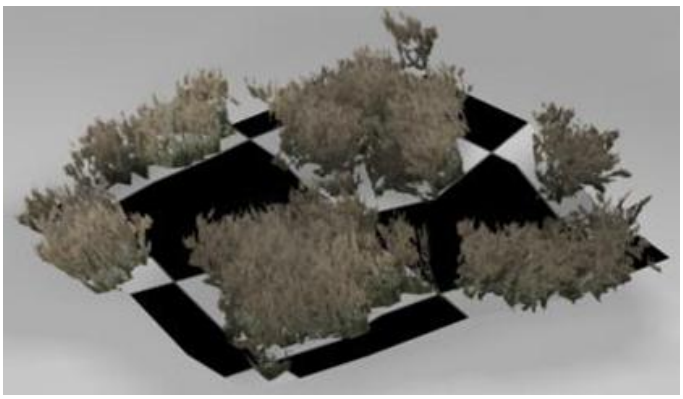


Figure 24 Abstract illustration showing a clear correspondence between a checker procedural and the instancing outcome

This would have positive implications regarding the user’s/artist’s comprehension of masking procedurals in the instancing context.

Expressing the orientation and scale of instances by procedural functions might however, yield less clarity/correspondence than procedurally driven instance masking. A factor in this might be that the orientation and scale channels are based on ‘continuous’ data; this differing from a ‘mask procedural’ which reduces to a Boolean value.

Thus, although these channels enable more variation between instances, it could be difficult to clearly understand the correlation between procedural functions and their manifestations in these channels. Figure 24 illustrates this through the application of Perlin noise to foliage rotation/orientation. Although 'variety' is evident, it is difficult to discern characteristics of the noise function in the distribution of foliage orientation.

It is likely that integration in an interactive tool chain would resolve this however. If parameters of procedural functions were exposed to allow interactive control of these channels during the authoring process, artists could work by experimentation and iterative adjustment. Providing this interaction is therefore, an important aspect of this implementation.

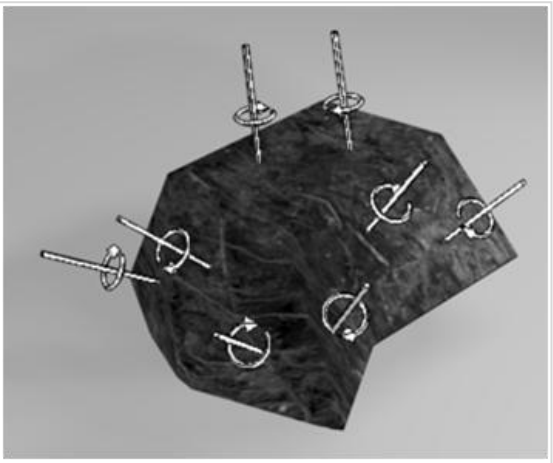


Figure 25 Illustrates 'local rotation axis' for instances. These axes are equivalent to corresponding surface normals

For most situations, arbitrary orientation of instances about all rotation axes is inappropriate. When considering trees and foliage for example, the orientation of each 'instance' is essentially constrained about its 'local' vertical axis. 'Local axes' for 'object instances' could be derived from the 'orientation frame' at points on the underlying surface. Thus, rotation about the normal vectors of the underlying manifold would be suitable (figure 25). The magnitude of rotation could therefore be dictated by the evaluation of a procedural function.

As earlier discussion suggests, the instancing concept coupled with 'channel' extensions, can offer a flexible automated object placement solution for use in a range of content creation scenarios. Situations exist however, where the proposed solution would be unsatisfactory; namely where explicit, highly 'granulated' control over the instantiation of instances on the 'manifold' is required. For example, consider generation of dense foliage in a scene containing elements such as buildings. By applying the algorithm to the ground surface, 'collisions' between foliage instances and scene elements are inevitable. These collisions would permit unnatural intersections between foliage and scene elements, which would be unacceptable.

In order for instancing algorithms to be applicable to situations like these, the artist would have to resolve geometric conflicts in one of two ways; identify masking procedural

parameters that yield no conflicts with other scene elements, or reposition elements to 'fit around' instanced foliage. Each of these 'solutions' is unsatisfactory.

For scenes that contain complex and prescribed arrangements of elements (such as buildings), it is unreasonable to expect that parameter configurations for a masking procedural, that comply (avoid 'conflicts') with the scene elements, exist. Even if parameter configurations that 'complied' with a scene's elements did exist, identifying these would typically be impractical, given the large permutation space for possible parameter configurations. Thus, relying on an optimal 'masking configuration' that suits a scene's elements is impractical.

Alternatively, rearranging scene objects to suit a procedural instancing result, although seemingly suitable, does not account for the dependency of other game aspects on the position of scene elements. Depending on the role of scene elements, repositioning may have negative implications on game production. The layout of scene elements often directly links with factors such as game play timing and difficulty, as well as the game's overall 'narrative'. Thus, repositioning buildings to comply with the procedural instancing of foliage may undermine important layout decisions established by the game's designers. Resorting to this means of 'conflict' resolution between static and instanced elements is therefore likely to open a new branch of conflict between artists and designers during a game's production; which is obviously undesirable.

To resolve this, a method that allow explicit control over instancing in a game scene, would be required. The proposed method will be discussed in the instancing algorithm section of the implementation chapter.

As discussed, procedural instancing automates the tedious task of manual object placement in game scenes. The implications of this algorithmic approach to object placement are that vast and complex game scenes become more feasible for developers, both in production and technically. Harnessing the capabilities of procedural functions in this algorithm, could therefore provide artists with a mechanism for delivering realistic and dense game settings.

Automated object variation

The idea of 'object variation', which maintains similar themes to the instancing concept, will be subsequently explored.

A typical strategy for increasing the realism of game scenes is to populate them with large numbers of props and entities (see figure 26). Although this approach is reasonable, the extensive reuse of 'prop collections' for game scenes often leads to an artificial and

unconvincing result. This is an obvious side effect of reusing identical scene props; particularly those which consist of distinct features and forms.

(Banks, 2009)

Figure 26 Modern games achieve increased realism by populating environments with many props and objects

As discussed in the literature review, some studios have implemented 'variation' systems into games, as a method of delivering more believable graphics and game play experiences. A noteworthy example of this is the game studio 'Ubisoft Montreal', which integrated procedurally based character variation into FarCry 2. Through this, the game delivered a unique population of game characters to better reflect the real world being depicted. This was a highly specified system however, that was only suited to achieving variety between game characters. To achieve generalized variation, many studios have employed 'manual' approaches to object variation, to diminish the sense of prop repetition in game scenes. This approach requires artists to produce numerous versions/variations of a single asset or prop which manifest uniqueness (i.e. unique damage to the same vehicle). Thus, rather than rely on a single prop to populate a game scene, variants can be used throughout the scene. The cumulative effect of this is an element of variety and hence realism, in the scene's final composition. Although this approach tends to yield significant improvement over situations where a single object is reused, the strategy has numerous shortcomings as will be discussed.

Variations between different objects in game scenes tend to be subtle. Thus, a set of object variations can essentially be represented by the same 'base geometry'. The implication of this is that geometric data is mostly duplicated. This obviously adds pressure to the 'capacity constraints' of game development in terms of memory usage and/or distribution media.
In addition, the requirement for variations of base geometry imposes additional workload on artists who must manually craft these variations, which often inflates project duration and/or requirements for larger teams of artists. Ultimately, this has negative implications on potential revenue of a game project.

With these negative implications as a premise, the research's final objective aims to provide an 'automated' and generalized mechanism for object variation. In keeping with the theme of this research, such a mechanism would transfer the overhead of this process from artists to a procedurally based algorithm.

Developing systems that generate and apply object variation is already a significant branch of computer science research; one example being the simulation of growth via L-systems (L-system, 2010). To deliver a universal solution that covers most variation scenarios for artists' and designers', a complex solution based on sophisticated rule sets and geometric modifiers might be required.

Although conceivable, an approach such as this would diverge from an objective of this concept, to minimize production overhead on artists, whilst retaining artist control and design in real-time.

A sufficient solution could be to provide artists with a simpler and intuitive method of applying variation to objects, in the context of an interactive tool chain. The goal of this would be to develop a simple and intuitive variation technique that can work under a range of circumstances. Thus, by providing a parameterized and algorithmic object variation system in an interactive tool chain, an avenue for improved game content production exists.

Furthermore, this variation system should take advantage of the interactive tool chain context, to provide immediate feedback/response to artist application and parameterization of algorithmically based object variation.

In this concept, object variation would exclusively apply to a base object's geometry. During the variation process, the base object would be subjected to a series of 'temporary' alterations. Following variation, rendering would immediately take place and thus, the 'alterations' would appear on screen. The alterations to the base object's geometry would be discarded, leaving it available for reprocessing in a subsequent 'render cycle'. These alterations should take advantage of graphics hardware acceleration, making the deformation process more feasible in a real-time context. Depending on the amplification of this function, objects may possess subtle or extreme distortion in accordance with the artist's requirements.

Procedural functionality is integrated into the deformation process, by 'evaluating' the procedural function at each point on the geometry. Thus, procedural data is returned at all points on the geometry's surface, by implicitly supplying 'sample coordinates' for the procedural function that 'map' to the geometries surface.

The objective of the variation system is to yield ‘authentic/unique variation’ between instances of the same base geometry. By definition, procedural functions cannot achieve this from local coordinate data given that they are referentially transparent. A sufficient level of variation can be achieved however, by manipulating the procedural’s sample coordinates.

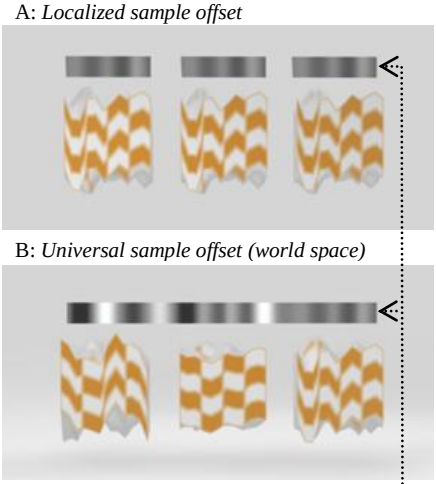


Figure 27 Compares different parameterization schemes for a deformation procedural function

Image (A) in figure 27 provides a visualisation of geometric deformation applied to three objects that share the same base geometry. Although deformation is evident, the repetition of features in the deformation is obvious. This is because the deformation procedural is evaluated by sample coordinates that are relative to each object. Because each ‘point’ on the object is unique, the object exhibits deformity. However, when each object evaluates the same procedural function with the same relative ‘offset coordinates’, the deformation effect is consequently identical between instances, yielding a result that is obviously undesirable.

Thus, by uniquely offsetting the deformation sample for each object, this should produce the desired inter-object variation. One method for acquiring ‘unique sample offset’ between objects could be to base the offset on the object’s position in ‘world space’, as shown in image (B), figure 27. Because each object has its own position, this can be used to uniquely offset the sample coordinates in sample space for a procedural deformation.

An unfortunate side effect of this approach is that movement of the object through its environment would cause an ‘animated’ effect in the object’s deformation. Thus, this scheme would require that objects remain static in game scenes. This should however, be sufficient for a number of game development scenarios.

As discussed, the proposed concept manipulates ‘base geometry’ according to a procedural function, to generate surface variation. To produce good deformation results however, the ‘base geometry’ would require sufficient geometric complexity. For this algorithm, geometric manipulation means the manipulation of ‘vertices’ that comprise the geometry. Thus, for simple base geometry such as (A) in figure 28, the effects of variation via this approach are vague and difficult to discern. In contrast, highly ‘tessellated’ base geometry, as shown by

(C) in figure 28, yields a clear deformation result. This variation system would therefore, rely on high levels of geometric complexity in its 'operand', to deliver the full effect of variation.

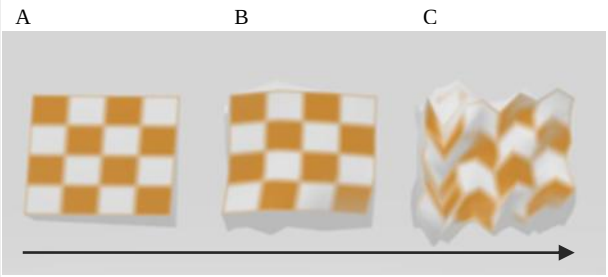


Figure 28 Shows how increased levels of 'geometric tessellation' yield more legible deformation results

A naive implementation of the proposed concept would require that highly tessellated base geometry is always supplied. This would implicitly require that extra (tessellation) data be stored within 'base-object' geometry; an imposition that opposes the system's objective of minimizing space/storage complexity. From this, another implementation requirement is established; that the variation system must compensate for simple base geometry, by adaptively tessellating specified geometry during the variation process.

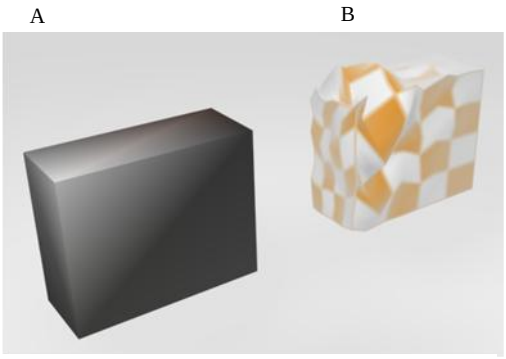
Although promising, the proposed system still lacks a fundamental element of artist prescribed controls over deformation across a base object. Many situations exist where high levels of artistic control are required to deliver prescribed game art and content. To keep the proposed variation system relevant to more content creation situations, the system should allow artists to control where algorithmic variation occurs in specified 'base geometry'.

Thus, permitting 'non-uniform deformation' across a base object, would allow greater control over the integration of variation into games content, rather than enforcing a level of uniform variation across an entire base object.

Consider the delivery of a post-apocalyptic game setting for example, where a suburban street is littered with props such as rubbish cans, debris and vehicles. To accurately reflect the prior events of the scene, these props would appropriately display damage, making them ideal candidates for the variation system.

For simple props, uniform deformation may be appropriate. Complex props such as vehicles however, may require a prescribed distribution of deformation. Depending on the scene's narrative, damage/deformation may only be appropriate on certain surfaces of vehicles (i.e. upward-facing surfaces). Generative 'damage' could be confined to these selected surfaces via non-uniform deformation, allowing algorithmically generated variation on the object to be consistent with the object's setting.

To remain ‘intuitive’ the variation system could incorporate non-uniform deformation via a ‘per-vertex weighting’ scheme. Vertices in the base geometry would store weighted values which would indicate the level of tessellation to apply at respective points. Image (B) in figure 29 shows the result of ‘higher tessellation/deformation’ in the upper corner of the geometric operand. Note the non-uniform deformation throughout the overall object in image (B), and the correlation of deformation to the ‘colour weighting’ in image (A) (where white areas yield more deformation that black). From the artist’s perspective, this concept could be presented as ‘auxiliary’ greyscale colour in the base geometry as image (A) of figure 29 shows. Artists would follow the convention of ‘painting deformation colour’ onto the base geometry, at portions of the geometry where tessellation/variation is required.



A. *Artist ‘paints’ deformation weighting to corner of base geometry*
B. *Deformation system applies adaptive deformation to corresponding section on geometry operand*

Figure 29 Visualization of the ‘painting’ metaphor when assigning deformation to geometry

It is important to note that these per-vertex weightings would bear no influence on the actual colour of the object during rendering. Rather, the intensity of this colour, as seen in the artist’s content authoring environment, would be internally used by the variation system for tessellation control as described.

Given the interactive tool chain environment, the effects of this painting process would ideally be immediately visible to artists. This constitutes the final implementation requirement for the variation system; an adaptive variation scheme that is dictated by per-vertex weightings in the base geometry.

Chapter 4: Implementation

This chapter details the implementation and algorithms which underlie the developed interactive content creation system. The chapter consists of three sections which cover the main areas and ideas of this research:

- Development of an interactive tool chain
- Process and implementation of procedural geometry instancing
- Process and implementation of procedurally driven object deformation for unique geometry generation

The tool chain component provides a platform for the implementation of algorithms described in the other two sections. This integration allows each feature to inherit the framework's interactive and responsive characteristic. As discussed, the motivation for basing this work on an interactive context follows from observations in the literature review, of the qualitative value towards productivity and quality in games production. Thus, the motivation for this process was to explore the potential implications of generative functionality towards more effective content creation, when coupled with an interactive context.

Interactive tool chain

This section presents a detailed discussion of the interactive tool chain that was developed. The tool chain itself, consists of three main software components; the 'communication plug-in', 'real-time game renderer' and a network library. Each of these software components were developed as part of this research, to allow practical evaluation of research ideas. The subsequent sections focus on the 'plug-in' and 'renderer' components of the tool chain.

As a foreword, the network library (which underlies the tool chain) uses standard techniques and communication protocols, namely TCP/IP and threading/queuing. The Winsock 2 library was used, (through which the TCP/IP protocol is exposed) given that it offers robust, flexible and efficient data communication (Windows Sockets 2, 2010).

The network library encapsulates Winsock functionally into an asynchronous, thread safe layer which is invoked by the tool chain. The motivation for an asynchronous layer is to avoid 'blocking' in the calling application, which is a side effect of the Winsock functions

used (namely '`recv()`') (recv Function, 2010). Thus, a simple queue structure was integrated into this layer to manage data transfer between the calling application's 'thread', as well as associated 'network activity' threads.

Overview

As discussed in the literature review, the interactive tool chain paradigm has been gaining significant momentum in game development studios in recent years. Although many integration models exist for interactive content authoring, they share a number of core elements; namely constant, low-latency feedback during the authoring process. In addition, interactive tool chains integrate game rendering technology, which provides artists' with access to an 'authentic context' for displaying content during development.

These tool chains provide artists' with visual feedback regarding the appearance of game content in the context of the 'target' game. This can be particularly useful for games that use specialized/optimized lighting systems or unique special effects that may influence the appearance of the content. Enabling the artist to see and work with content in a 'project specific' context therefore, offers benefits in production efficiency, particularly where game content needs to be 'tailored' to an established game setting. This tool chain paradigm also capitalizes on the 'real-time' quality of game rendering technology, to deliver a creation environment that is immediately reactive to modifications made by the user/artist(s).

Interactive tool chains: Integrated model

During the planning phase, two interactive tool chain models were considered as a basis for this project. The first model integrates game rendering technology directly into the content authoring software used by artists. This model is essentially used in the 'Unreal 3 engine's' tool chain. Recall that Unreal's real-time rendering technology is directly integrated into the system's proprietary content authoring environment. A similar tool chain structure, where Autodesk's' 3D modelling application 'Maya' substitutes Unreal's authoring environment, was considered for this project (Autodesk: Maya, 2010).

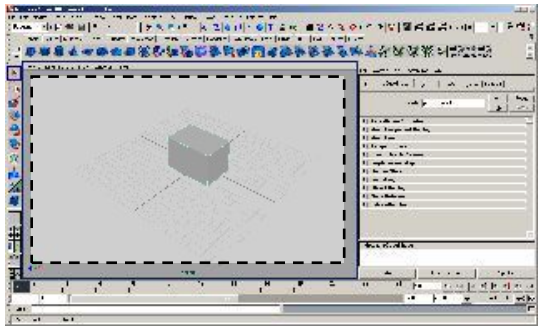


Figure 30 Screenshot of Maya, highlighting the software's 'viewport' interface element

Maya is a popular 3D modelling and animation package which is used by many industries, particularly the game development industry (Autodesk: Games, 2010). Examples of game development studios that use Maya include Insomniac Games, Sidhe, and Id Software (Sidhe Interactive Online, 2009) (Id Software, n.d) (Murdock, 2010). Integrating industry standard development tools such as Maya make the project inherently relevant to its target industry.

Maya is highly customizable and integrates bindings to two scripting languages; MEL and Python (Autodesk Maya: Features, 2010). It also exposes API's for environment customization and plug-in development (Autodesk Maya: Features, 2010). In particular, Maya exposes a software interface (`MViewportRenderer`) which allows the 'modelling viewport' to be re-implemented (MViewportRenderer Class Reference, 2010). As figure 30 illustrates, the viewport element is central to Maya's interface and provides the user with a clear visual representation of the modelling project.

Re-implementing the '`MViewportRenderer`' interface would provide an avenue for integrating the ideas and rendering algorithms of this research, directly into Maya. In addition, this integrated approach would see rendering techniques and effects, which are present in games, also being introduced into Maya. This strategy would allow artists to create content in a familiar authoring environment, which is directly visible in the context of the target game's rendering technology. Thus, many benefits of 'Unreal 3's' tool chain would be inherent in this solution. Furthermore, this integrated strategy would provide a platform for incorporating the algorithms and ideas of this research.

Maya is a cross-platform modelling package with a large user base that extends over all major computer platforms. To maintain compliance with its user base, the reimplementation of Maya's viewport would ideally adhere to cross platform standards. This would require the use of 'OpenGL', a popular cross platform graphics API that exposes hardware acceleration (Segal & Akeley, 2010) (Neider, Davis, & Woo, 1994).

As discussed, the motivation of this is to introduce game rendering technology directly to the content authoring context. Using OpenGL for game rendering purposes however, is not reflective of current trends in the game industry, where Microsoft's Direct3D API is predominantly used (Rosen, 2010). Microsoft offers exclusive support for Direct3D on its own platform; the 'Windows' line of operating systems. This is inconsistent with Maya's cross-platform nature and thus, the integration of Direct3D into Maya is problematic.

In addition to widespread use in industry, Direct3D has other advantages over OpenGL; particularly in terms of cross-vendor functionality. Despite the benefits of OpenGL as an open platform, this has hindered standardization of the API, presenting a challenge to

developers that require modern graphics features in software which can run across hardware supplied by different graphics vendors (i.e. NVidia and ATI) (NVidia, 2010) (ATI Technologies, 2010). Although standards are updated to unify OpenGL (vendor) implementations, the specification of standards has historically lagged behind the introduction of new graphics features.

In recent years, OpenGL's standards/specifications have trailed behind those of Direct3D; noteworthy to this, was the motivation for the recent OpenGL 4.0 specification which aims to maintain parity with Direct3D (Bright, 2010).

Thus, utilizing modern graphics features via OpenGL typically requires use of the API's 'extension' architecture (Kilgard, Section 5.0, 2000). This involves querying the availability of hardware features (or 'extensions') at runtime, imposing the need for different code-paths and/or reduced software compatibility (Kilgard, Section 5.0, 2000). Because most extensions are vendor specific, graphics applications often become complex when integrating unique graphics functionality offered by different hardware vendors (Kilgard, Section 3.0, 2000)

In contrast, the Direct3D platform maintains a static and universal 'specification' per version release. All hardware vendors must comply with this specification to gain DirectX certification. Thus, when working with Direct3D (version 10) on certified hardware, the availability of "*Direct3D 10's base feature set is guaranteed*" (Overview of the Major Structural Changes in Direct3D 10, 2010). This therefore, offloads the burden (which faces OpenGL developers) of integrating modern hardware capabilities across different vendors/hardware.

The capabilities of modern graphics hardware play a central role to the delivery of algorithms and ideas in this research, hence the motivation for using the Direct3D API.

In addition, this model depends on the extensibility of content authoring tools, requiring that a tool's viewport interface element be customizable. However, a review of the API documentation for Autodesk's other popular modelling package, '3D Studio Max', indicates that no developer interfaces exist for custom viewport implementation (Autodesk: 3ds Max Products, 2010) (3ds Max 2011 SDK, 2010). Thus, if a studio's authoring tools do not allow custom viewport implementation, a tool chain based on the integrated model would be inapplicable.

Interactive tool chains: Connection model

In response to the integrated model's shortcomings, the 'connection model' was selected as the basis for this project. This 'connection model' delivers the core functionality of an interactive tool chain while abiding to the previously mentioned constraints.

In comparison to the integrated model, this approach moves game rendering functionality from authoring tools (i.e. Maya) into an external, stand alone application. Although this sacrifices the elegance and simplicity of the integrated model, the approach provides greater opportunity for artist collaboration, as well as cross-platform support within the tool chain. What's more, the connection model does not depend on viewport extensibility in the selected authoring software, as is the case with the integrated model.

The connection model naturally extends to offer cross-architecture support as well. This is similar to that demonstrated by Crytek's 'LiveCreate' content authoring system, which is based on a connection model.

Recall that LiveCreate enables artists to produce content while maintaining a synchronized view of the content on the Playstation® 3, Xbox 360™ and PC simultaneously. Achieving this level of console integration in the integrated model is not possible, due to the rendering technology of consoles being proprietary to respective manufacturers.

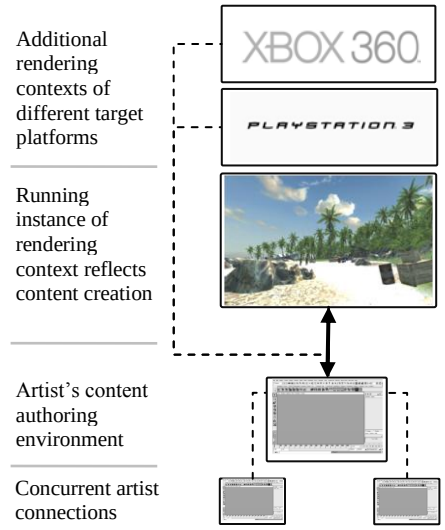
As alluded, the connection model is flexible and can accommodate a variety of configurations. Because game rendering technology exists in a standalone application, the tool chain doesn't depend on the content authoring software being concurrently active. In this sense, the model could therefore directly integrate with a studio's game project, providing artists with an authentic context for content prototyping and development. Furthermore, by integrating elements of the real-time tool chain directly into the game, the project can utilize the algorithms under normal game play circumstances.

In addition to improving production processes, the project aims to make production of visually complex games more feasible from a technical perspective.

Integrating procedural algorithms directly into games, aims to offset issues of size complexity in games content, by taking advantage of the processing power in modern GPU's. Furthermore, by exposing the parameters of procedural functions to artists, the research also aims to allow better utilization of procedural data generation, promoting more effective content production. Subsequent discussion will explore these ideas further.

This selected tool chain model also supports artist collaboration for shared/concurrent asset development. The model's inherent flexibility makes it capable of hosting many 'connections' between artist and 'game rendering instances'. Thus, configurations where multiple artists share a single rendering/prototyping context are possible. Because artists connect to a shared, remote rendering context, each individual artist maintains control at their local 'authoring' workstation. Facilitating multiple artist connections to an external rendering context is therefore, feasible.

Thus, by maintaining a connection to the rendering context, any changes made by artists in their local authoring environment, are immediately reflected in the 'remote' game rendering context.



(IGN: Crysis Screenshots, 2007)

Figure 31 Schematic diagram, illustrating work flow configurations via the 'connection model'

As figure 31 illustrates, the communication model has two main components; the 'real-time content encoder' (RTCE), which is embedded in an authoring tool and transmits data to the second component, the 'game rendering context' (GRC) which displays game content. The following sections explore each component in more detail.

Real time content encoder (RTCE)

The RTCE is a custom plug-in developed during this research for the Maya modelling package. The plug-in's main function is to provide an interface that encodes and transmits data/content from Maya to the responsive/interactive tool chain. In addition, the RTCE allows artists to specify parameters for procedurally based graphics algorithms, in a production efficient manner; namely the 'real-time generative instancing' and 'unique object deformation' algorithms.

As identified, production efficiency for PM's has traditionally been hindered by tool chains that require slow and tedious content transfer processes (between authoring tools and game technology). A key objective of this framework is to produce a responsive tool chain system that performs this process at interactive rates. This requires that all components of the system perform efficiently to avoid bottlenecks in data flow. The RTCE is arguably the most important component of the system from a performance perspective.

To achieve solid performance, the plug-in takes advantage of Maya's C++ API. C++ is often used in situations where high performance is needed, particularly when compared to scripting languages. For this reason, core functions of the plug-in were built using software interfaces exposed by the C++ API. In addition to C++, the plug-in also uses the 'Maya Embedded Language' (MEL) to construct its user interface.

RTCE characteristics

The following list summarizes the RTCE plug-in components, and serves as an outline for the implementation discussion:

Functional:

- Cross platform compatible
- Perform at interactive/real-time rates
- Responsive to user interaction in Maya's modelling context
- Access, encode and transmit data internal to Maya for use in the respective tool chain
- Add and maintain plug-in specific data to game content/geometry

Interface:

- Maximize consistency with Maya's user interface and conventions
- Expose parameters and functionality for research specific algorithms
- Use modelling conventions where possible

Technical development

The RTCE's implementation is based on a simple modular code design. Each module addresses one or more of the listed functional characteristics. A number of factors influenced this design choice; namely the software interfaces exposed by the Maya API, as well as the need for code flexibility during development of this 'prototype'.

RTCE structure, data access

Maya is capable of producing and representing sophisticated virtual scenes and objects with high fidelity. To efficiently maintain, store and access this complex data, Maya internally uses a graph structure. Interestingly, Maya's user interface maintains a close mapping to the software's internal data representation (figure 32).

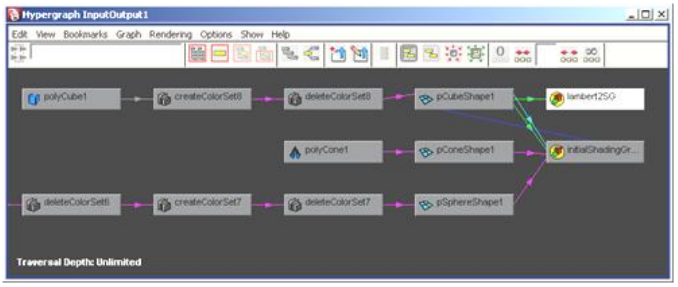


Figure 32 Example of scene representation in Maya 2008 using abstract graph structure

The RTCE plug-in accesses Maya's internal data structures, namely DAG's (Directed Acyclic Graphs), to gain access to scene data relevant to user interaction (MFnDagNode Class Reference, 2010). User interactions in Maya can occur in different software contexts and with different data types. Thus, Maya's internal data structures provide a runtime efficient mechanism that allows scene data to be accessed, encoded and transmitted in an interactive timeframe.

Maya has approximately one thousand data types which are collectively referred to as the 'function set' (MFn Class Reference, 2010). In Maya's API, 'data objects' are managed by developers as 'handles'. Casting a Maya handle to a relevant 'function set' (data type) exposes the inner functionality and data that the handle implicitly represents. At runtime, the RTCE plug-in locates data (or 'nodes') from the scene's internal DAG, using a subset of API class structures ('iterators') to achieve filtered iteration of certain node types in the current Maya scene.

To simplify RTCE's overall implementation however, a 'utility library' was developed which provides methods for DAG access and traversal. These library functions further simplified the use of the Maya API, avoiding the need for repeated instantiation of 'iterators', element looping and key comparison. Such functionality constitutes the 'utility' library which is a module within the plug-in's structure.

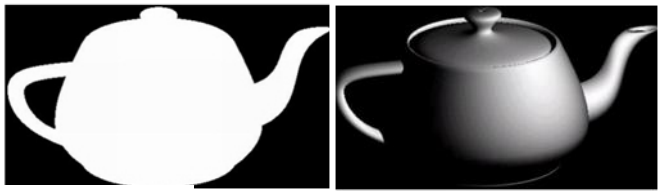
Data types, packet identification

The RTCE accesses and encodes a range of data types; namely 'Mesh' (geometry), material and texture data. It also transmits plug-in specific auxiliary data that the RTCE 'appends' to scene objects. This project uses a simple identification/key convention, through which transmitted data is associated with 'objects' in the GRC. Thus, when a mesh data packet is sent to the GRC, the plug-in assigns 'identification' to the packet. When the packet is received by the server (or GRC), the 'identification' data is used to channel the packet/data

into the correct ‘game object’. In Maya, most DAG nodes (i.e. mesh objects) have a unique ‘name’ value which is suitable for packet identification.

Mesh packet

Mesh packets constitute the main data sent from the Maya plug-in to the GRC. This is because mesh data typically consists of auxiliary information, in addition to raw geometry. Examples of auxiliary data include ‘per-vertex’ normal vectors, texture coordinates and colour data. ‘Per-vertex’ data associates different ‘channels’ of information with each vertex position, through which a variety of rendering effects are achieved. Real-time lighting for example, is typically achieved via rendering calculations that rely on per-vertex normal vectors, in geometry (see figure 33).



(Seyringer, 2003)

No per-vertex normals yield simple ‘ambient’ lighting

Per-vertex normals enable improved lighting results

Figure 33 Illustrates how improved lighting can be achieved via computations that use per-vertex normal vectors

To simplify the process of interpreting mesh data at the GRC (for real-time rendering), it is transmitted from the RTCE in an organized and ‘interleaved’ state. The motivation for this is to take advantage of functions provided by the Maya API, that enable efficient vertex data interleaving, rather than defer the process to the GRC where interleaving would incur processing overhead.

In the current implementation, updating an object in the GRC involves accessing and transmitting all mesh data for the corresponding object in Maya. For large geometric data sets, this simple approach would represent a performance bottleneck within the system, which would obviously impact on performance. This is because any modification to the geometry from Maya would force the entire object to be sent to the GRC. Although this would be impractical for ‘real world’ scenarios that use complex geometry, the approach has proven sufficient for conceptual development. It leaves however, an opportunity for future improvement to the system.

An element of optimization is however, present in the RTCE’s mesh transmission component. The aim of the optimization is to reduce pressure on the network bandwidth between Maya and the GRC. As stated, per-vertex data associates extra pieces of information

with each vertex of a geometry collection. Thus, interleaving per-vertex data with geometry effectively multiplies the size of a geometry buffer. The RTCE allows artists to choose the per-vertex elements that are required for a specific game object(s). This functionality is encapsulated in the RTCE's 'mesh transmission' function, which is partially listed in table 4 (page 59).

An object's 'vertex element configuration' is also referred to as it's 'vertex format'. The vertex format for an object is used during the mesh transmission process to selectively interleave elements of required per-vertex data, into the mesh packet. This feature maintains user interface consistency with well established modelling conventions, as will be subsequently discussed.

Maya's objects' usually consist of multiple 'materials', each of which is bound to a geometry subset of the object. To reproduce unique material effects across an object, each geometry subset of a multi-material object is typically rendered under the context of respective materials. The mesh transmission process takes multi-material scenarios into account; namely by transmitting multi-material geometry in a series of nested loops. The outer loop of the encoding process iterates over the materials that are assigned to the object, providing the inner process with a 'material specific' context. Thus, the encoding process is adaptive to arbitrary material configurations in Maya's objects.

The inner loop iterates over the polygons/triangles of each material in the object. This inner loop contains functionality that extracts data (geometry, auxiliary, etc) from the target Maya object, into a 'byte stream' which is subsequently transmitted from the RTCE by the network interface. Note that the 'extraction' process is dictated by the target object's 'vertex format', which is assigned by the artist.

The reason for this is to avoid unnecessary object data from being appended to the byte stream, as previously discussed. The code excerpt in table 4 shows how the extraction process is adaptive to/dependant on, the object's vertex format.

To access subsets of an object's geometry based on individual materials, the process makes use of 'mapping' functionality provided by the Maya API. Similar mapping functions are also used to extract texture coordinates, normals, etc, from the object.

In modelling software, an object's texture coordinates are usually shared by a variable number of geometric vertices. Texture coordinates specify a 'mapping' of an image/texture across each point on 3D geometry. Thus, texture coordinates for similar/equivalent vertices of adjacent triangles in the 3D geometry, are often the same. From an artists' perspective, texture coordinate manipulation is greatly simplified when a single coordinate shared by

many vertices can be modified, rather than requiring the repeated manipulation of coordinates for each vertex.

As a result, the number of texture coordinates in an object is not necessarily the same as the number of geometric vertices. Thus, if the vertex format requires texture coordinates, Maya's mapping functionality is used to correctly assign texture coordinates to vertices in the transmitted 'byte stream'. Texture coordinate data is then directly interleaved to the byte stream, as (A) in the code excerpt of table 4 shows. Similar mapping processes to this, are used to interleave other per-vertex data into vertices of the byte stream.

The code excerpt in table 4 illustrates how bitwise masking is applied to the object's vertex format, during each vertex iteration, to determine the required per-vertex elements (i.e. per-vertex colour, normals, texture coordinates). This makes dynamic vertex formats of this project 'order dependant', requiring consistent use of 'element precedence' between vertex elements of the Maya plug-in and GRC.

Mesh Transmission Excerpt

```
bool Net::Transmit_Mesh(MObject& targetObject,
bool shouldRemove) {

  char* pByteStream = NULL;
  int sendSize = 0;
  int vertexFormat = 0;
  MPlug attribute;

  ...

  if(MAYA_FAIL(attribute.getValue(vertexFormat)))
       throw tException("Failed to get vertex format.");

  ...

  /*
  Iteration through the object's geometry begins here
  */
  MItMeshPolygon polygons(targetObject);
  for(uint i = 0; i < partCount; i++) {

    ...

    uint polygonCount = polygons.length();
    for(uint j = 0; j < polygonCount; j++) {

        ...

        /*
        At this level of nested iteration, the process is           iterating
      through vertices of the current object
        */

        int vertexIndicies[3] = {0};
        if(MAYA_FAIL(meshObject.getPolygonTriangleVertices(j, k,
        vertexIndicies)))
               throw tException("Failed to get polygon triangle
```

```
        vertices.");

    uint polyVertexCount = polygons.polygonVertexCount();
    for(uint m = 0; m < polyVertexCount; m++) {

      ...



      /*
      If per-vertex position data is required in vertex format, then
      extract and interlace position data into byte stream
      */
      if(vertexFormat & VERTEX_POSITION) {

        float position[4];
        tVector3* pPosition = (tVector3*)pByteStream;
        pByteStream += sizeof(tVector3);
        ...

        pPosition->x = temp[0];
        pPosition->y = temp[1];
        pPosition->z = temp[2];
      }

      /*
      If per-vertex normal data is required in vertex format,
      then extract and interlace normal data into byte stream
      */
      if(vertexFormat & VERTEX_NORMAL) {


        tVector3* pNormal = (tVector3*)pByteStream;
        pByteStream += sizeof(tVector3);
        ...

        pNormal->x = normal.x;
        pNormal->y = normal.y;
        pNormal->z = normal.z;
      }

      /*
      If per-vertex tangent data is required in vertex format,
      then extract and interlace tangent data into byte stream
      */
      if(vertexFormat & VERTEX_TANGENT) {

        float tangent[4] = {0.0f, 0.0f, 0.0f, 0.0f};
        ...

        tVector3* pTangent = (tVector3*)pByteStream;
        pByteStream += sizeof(tVector3);

        pTangent->x = temp[0];
        pTangent->y = temp[1];
        pTangent->z = temp[2];
      }

      /*
      (A)
      If per-vertex texture coordinate (0) data is required in vertex
      format, then extract and interlace texture coordinate (0) data into
      byte stream
      */
```

```
        if(vertexFormat & VERTEX_UV0) {

          tVector2* pUV = (tVector2*)pByteStream;
          pWriteVertex += sizeof(tVector2);
          ...

          pUV->x = uv0[0][uvIndex];
          pUV->y = -uv0[1][uvIndex];
        }

        ...

        /*
        If per-vertex colour data is required in vertex format,
        then extract and interlace colour data into byte stream
        */
        if(vertexFormat & VERTEX_COLOUR) {

          tVector4* pColour = (tVector4*)pByteStream;
          ...

          pColour->x = colour.r;
          pColour->y = colour.g;
          pColour->z = colour.b;
          pColour->w = colour.a;
        }

        ...
      }
      ...
    }
    ...
  }

  /*
  Send off the geometry data and clean up heap as necessary
  */
  if(!VNet::g_Client.SendDataPacket(pByteStream, sendSize,
  UPDATE_GEOMETRY))
    throw tException("Failed to send geometry data.");

  ...
                                          }
```

This function accesses geometry data from the '`targetObject` 'handle' and gathers it into data structures that are used by the tool chain. Because geometric complexity of objects is arbitrary, the 'gathering' process exists in a variable loop. Note that data is 'interlaced' into the byte stream ('`pByteStream`') depending on the current '`vertexFormat`' of the object being transmitted.

Table 4 Code excerpt showing features of mesh packaging and transmission iteration

Material packets

Textures and materials are staple features of game graphics. Hence, they are an integral part of this project and constitute a major part of the RTCE's functionality. As discussed, this research explores the implications and benefits of integrating PM's into games. Following this theme, a branch of this research explores the conventional use of PM's for materials/texture of games. The objective is not to replace conventional surface textures, but to supplement the texture with procedurally introduced detail.

This integration of material/texture also explores the implications of a responsive tool chain on the integration of PM's in games content. By joining a responsive tool chain with artist exposure to procedural texture/material composition, the project aims to make the use of PM's within games content, a viable and attractive option.

To achieve this, the RTCE plug-in is responsible for accessing and transmitting data from a range of 'texture types' within Maya. In addition to transmitting 'raw' material data, the RTCE must associate system specific data with each material packet. This system data indicates the material's 'usage' in the real-time rendering context.

Textures/materials are used by game rendering systems to achieve a variety of visual results, as well as special surface enhancement effects. Thus, given that modern games make extensive use of 'auxiliary' texture data, similar capabilities have been integrated into this project. A side effect is that each material packet requires additional data to indicate the material's role in the rendering process.

Materials are also used by the RTCE system for 'procedural instancing'. The procedural instancing algorithm makes use of procedural materials for up to three different 'data channels'. Detailed discussion on the procedural instancing algorithm can be found in the instancing algorithm chapter (page 100).

Table 5 provides a summary of the PM's (as well as the raw image source) which are available in Maya and supported by the RTCE.

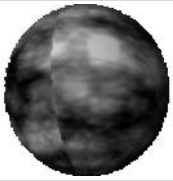
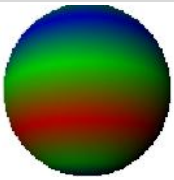
Type	Grid	Perlin Noise	Image data
			
Parameters	Colour vectors Line width	Amplitude Frequency Octaves Threshold	Raw binary data
Type	Ramp	Wood	Checker
			
Parameters	Colour vectors Sine amplitude	Colour vectors Ring frequency	Colour vectors Contrast

Table 5 Material types supported in RTCE

The RTCE plug-in closely integrates with Maya's internal 'material' system and its associated data structures; a design decision which provides a number of benefits. A benefit

of using Maya's own data structures is that material information used by RTCE, is saved into the project file when Maya is shutdown. This use of Maya's built-in storage functionality also advocates project 'portability', because no dependency on external data is introduced.
Another side effect of this close integration is that Maya's conventions for material creation and composition are inherited by the RTCE. This supports the plug-in's objective for 'transparency', by permitting the reuse of skills established by Maya users.

Maya's materials are complex data structures that host 'connections' to data sources (i.e. procedural functions, textures), as determined by the artist. Material connections allow arbitrary data sources to supply the material's 'channels', such as colour and transparency. Like most authoring software, Maya supports the notion of 'channels' in many of its materials. Channels allow independent control/specification over different characteristics of a material, usually through assignment of different data sources to each channel. When a material is 'rendered', data sources assigned to channels are usually interpreted based on the channels characteristics. For example, data assigned to the 'colour' channel is expressed as the material's explicit colour. A data source assigned to a material's 'transparency' channel however, would typically dictate the presence of translucency in the rendered material.

As figure 34 illustrates, the 'material node' makes provision for these channel data connections. The RTCE plug-in takes advantage of Maya's 'material structure' to enable artist composition of different surface characteristics in GRC rendered geometry.

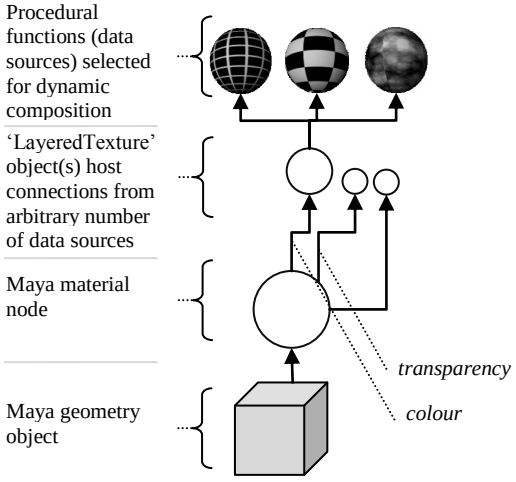
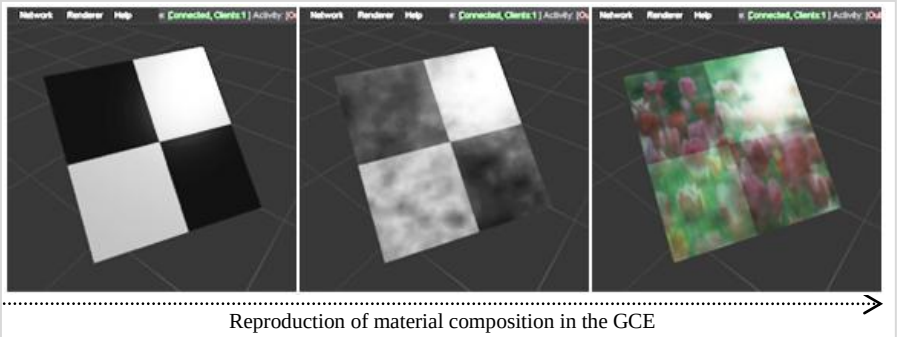


Figure 34 RTCE's material hierarchy for showing custom composition of procedural functions for a material channel

For each material channel in an object's hierarchy, the RTCE plug-in automatically creates and inserts a 'LayeredTexture' node. Figure 34 shows the position of LayeredTexture nodes within the hierarchy. Maya's documentation states that "*the LayeredTexture node can be used to layer multiple textures on top of one another to produce a single texture result*" (layeredTexture node, 2010). The RTCE however, doesn't use the LayeredTexture node for this purpose but instead, takes advantage of the arbitrary number of connections that it supports. This property makes LayeredTexture's ideal for the system's 'procedural

composition' feature, as the RTCE can interpret these nodes as 'containers' for procedural functions. Thus, prior to transmitting a material to the GRC, the plug-in iterates each connection of a LayeredTexture node, sending the associated texture/procedural function data at each connection. Because the material hierarchy is built from Maya's data structures, this permits intuitive material composition for artists, while also supporting efficient internal access to procedural/texture data via DAG traversal.



Reproduction of material composition in the GCE

Figure 35 Illustrates an enhanced surface material via composition of numerous procedural elements in the RTCE plug-in

Other data

As discussed, the RTCE plug-in achieves a high level of integration with Maya's conventions and data structures. Some aspects of the plug-in are unique to the project however, therefore requiring that data is added to Maya's objects. The Maya API supports this with dynamic object 'attributes' which provide the mechanism for adding data to nodes in Maya's DAG structures (Maya, 2010). The RTCE introduces a series of custom object attributes through which system-specific data and materials are associated with objects in Maya. Table 6 provides an outline of this system specific data. Note that at runtime, the RTCE automatically adds default attribute fields to objects if they are missing.

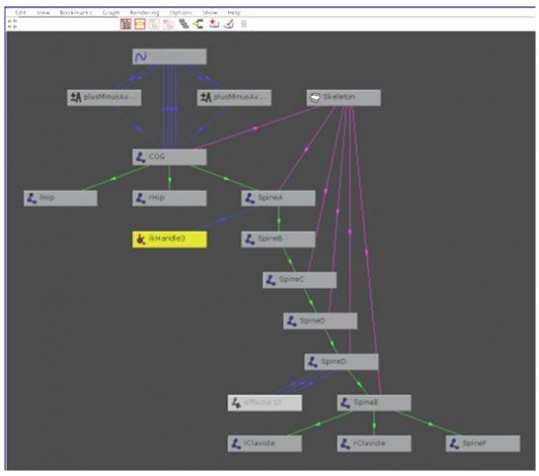
Attribute	Type	Purpose
Shader filename(s)	String(s)	Enables the artist to specify 'shader'(s) that are used by the GRC to render subsets of (material) geometry in an object.
Vertex format	Byte	Bit field that represents the per-vertex elements in the object's vertex-format/structure.
Instanced	Boolean	Toggles whether the object is available for instancing by 'procedural instancing algorithm' (see the instancing algorithm chapter, page 100).

Deformable	Boolean	Toggles whether the 'unique deformation algorithm' is applied to the object (NPD, see object , page 130)
Instancing: ▪ Mask Procedural ▪ Orientation Procedural ▪ Scale Procedural	Strings(s)	Establishes string based association between the object and material nodes that are used for instancing. This association doesn't explicitly use node 'connections' and therefore, conventional 'traversal' is not possible.
Instancing 'Cookie cutter'	String	Enables the artist to choose an 'image' to explicitly mask instancing in the 'procedural instancing algorithm' (see the integration of instancing 'cookie cutter' section, page 125).
Deformation scale	Float	Enables artist control over the scale of deformation amplitude in the object (only available if 'Deformable' attribute is true).

Table 6 Auxiliary attributes assigned to objects by and for the RTCE plug-in

Event mechanism

As mentioned, the RTCE plug-in has numerous modules, one of these being the 'synchronization module' which is responsible for handling user interaction/events. The synchronization module underpins the plug-ins 'network communication' and essentially invokes all data transmissions to the GRC. The module's main functionality exists in 'call back' routines, which are bound to Maya's software 'events'. At runtime, the 'sync' module initializes a special call back that is invoked when Maya's scene graph/DAG is modified. This call back therefore 'captures' the events for object addition/removal in a Maya scene.



(Diffuse minus Specular, 2009)

A typical DAG structure in Maya, which represents the relationship and hierarchy of scene elements

Figure 36 Illustration of DAG structures in Maya

Through this, the plug-in receives notification when an artist introduces geometry, lights or materials into the scene. During notification events for object insertion, the RTCE binds relevant call

backs to the new node. Examples include ‘call backs’ that are invoked when an artist modifies properties or data of the particular node.

For example, the plug-in receives notification for events relating to artist interaction with scene elements such as geometry. Thus, if the artist moves a vertex of a 3D object in Maya, this triggers a sequence of ‘events’, invoking respective RTCE functionality. The high resolution offered by Maya’s event system enables the plug-in to transmit data from Maya to the GRC, in a highly interactive/responsive fashion.

This mechanism ensures that the RTCE can respond to all relevant user interaction events for each object/node that is inserted by an artist, into a Maya scene. The approach offloads all interaction monitoring from the RTCE onto Maya, allowing for code that aligns with Maya’s extension interfaces.

Maya’s call back interfaces can be/are invoked for several different ‘event types’. The side effect of this is that simple interactions with Maya can cause several invocations of the same call back. This is undesirable, given that call backs directly invoke data transmission over the network interface. To maintain efficient use of network bandwidth, irrelevant event types are filtered by the plug-in. Table 7 summarizes the events that the RTCE responds to, as well as the relationship between the ‘sources’ of an event (in the context of Maya), and the way that the system responds to them.

Event Source	Interaction/Event	Description
Geometry	▪ Translation ▪ Rotation ▪ Scaling	When a geometric object in Maya is moved, rotated or scaled, the RTCE responds to these interactions in real-time by transmitting the object’s ‘transformation’ data to the GRC.
	Vertex manipulation	Manipulating a vertex (or triangle) of a Maya object, triggers the RTCE to immediately transmit the modified geometry which results in the changes being interactively reflected in the GRC.
	Texture coordinate manipulation	Manipulating texture coordinates in Maya objects, triggers a similar ‘transmission’ event as in vertex manipulation. Thus, the GRC interactively reflects any changes to texture coordinates of a Maya object.

	Normal manipulation	Manipulating normal vectors in Maya objects invokes a similar ‘transmission’ event, as vertex manipulation. The GRC interactively reflects this type of object modification.
Light	Translation Rotation	If the artist moves or rotates a light source of a Maya scene, the RTCE responds to this interaction in real-time. The transformation of the corresponding light source in the GRC is interactively updated.
	Attribute change	When attributes of a Maya light source, such as brightness and colour are changed, these events are immediately reflected in the GRC.
Material	Channel Modification	As discussed, Maya’s materials consist of different channels. Thus, when a data source is added, removed or replaced for a material channel, the new material structure is immediately transmitted to the GRC. The side effect of this is that objects in the GRC which apply the corresponding material, visually reflect the channel changes.
Texture	Attribute change	Table 5 shows the parameters associated with each texture type, supported by this tool chain. As ‘attributes’ of textures in Maya are modified by the artist, the tool chain immediately responds by transmitting the texture’s attributes as ‘parameters’ to the GRC. Thus, materials of the GRC that use the respective texture as a data source, immediately yield an updated appearance that reflects the

		changed attribute.
Scene	Selection changed	These events typically occur when Maya's 'interaction context' changes. When for example, the artist selects a different object in Maya this indicates that the artists' current 'subject of interest' in the scene has changed. Thus, the RTCE invokes synchronization functionality, to ensure that the selected object's 'counterpart' in the GRC, is correctly displayed.
	DAG changed	As discussed, Maya's scenes are internally represented by a DAG structure. Changes to this structure usually indicate that a node (i.e. geometry, material, etc) has been added or removed from Maya's scene. The RTCE responds to these events, to maintain consistency between Maya and the GRC scene. Through this event, all supported node types, namely geometry, materials, textures and light sources, are synchronized with the GRC.

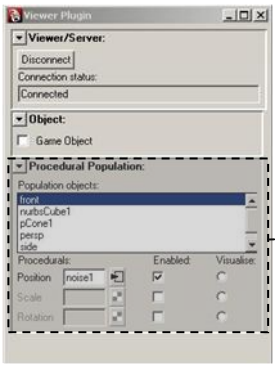
Table 7Summary of interaction events in Maya that the RTCE responds to

User interface

RTCE's final module implements the plug-in's user interface. This module consists of two components; initialization/management of the RTCE's 'user interface scripts', and a custom 'message handler' that responds to user interactions with the interface.

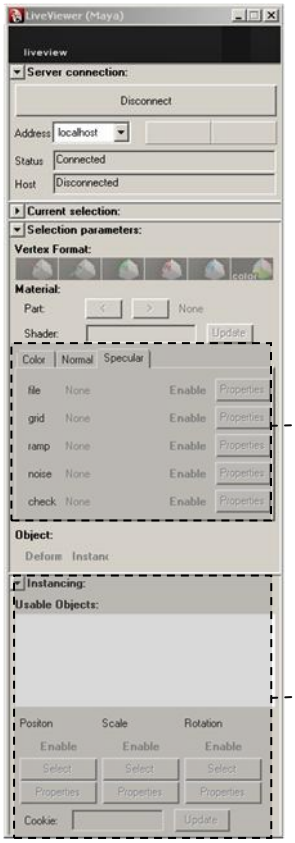
Most of Maya's GUI is written in Maya's own scripting language 'MEL'. Thus, for consistency and integration purposes, the RTCE's user interface was also implemented via MEL script. Because MEL is proprietary to Maya, language specific methods and syntax had to be studied; this represented a significant overhead in the development process. An iterative

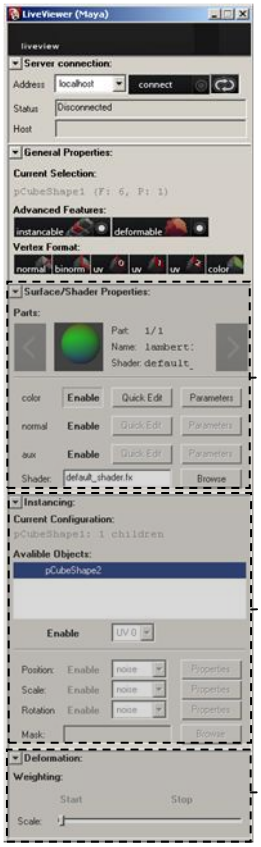
approach was taken to development. The following table (table 8) presents a brief summary of design features and limitations for each of the interface's main iterations.

First Design Iteration	Discussion
	The first iteration represents a highly experimental development phase. At this stage, the basic concepts of interface development via MEL are being studied. This early iteration does however, integrate and expose elements that are relevant to this research; namely for parameterization of the 'procedural instancing algorithm' (see subsection A in diagram). The motivation for early integration of these features was to support concurrent development/experimentation of the 'procedural instancing algorithm'. The interface also includes 'networking features' to support the tool chain's 'connection model'. Layout mechanisms (provided by MEL's interface library) underpin these features, providing better organization within the interface.

Second Design Iteration	Discussion
	A better understanding of MEL's user interface library is manifest in this more sophisticated interface. This revision is the first to have subsections that parameterize the core features of the GRC application/research. The interface subsections that have been introduced are shown in the accompanying diagram.
	A: Material Composition
	These components exposed early system functionality which allowed the artist to choose and apply materials to geometry in the tool chain. As shown, this interface subsection provides visual feedback regarding the currently bound material. The sub-interface also makes provision for material 'toggling' which represents the earliest stage in material composition functionality. 'Tab' panels are introduced into this section as a means of categorizing similar components for different 'channels' of the same material. This allows artists to specify a procedural function or texture for up to three material channels (colour, 'normal' and 'specular') (Owen, 1999).
B: Procedural instancing parameters	
These components reflect significant development that has taken place in the project's 'procedural instancing algorithm'. The revision to this subsection now integrates the main features that are required to fully parameterize/utilize the objects that are instanced by the algorithm; namely a list interface component. The list is populated with information for scene objects that are available for use in the 'instancing algorithm'. Upon selection of a list item, the artist is able to toggle and specify procedural functions that control the object's 'instancing behaviour'.	
C: Unique deformation parameters	
This simple subsection is introduced into the interface to expose basic functionality for the 'deformation algorithm'. A noteworthy feature is the 'Paint' button which	

introduces the notion of ‘deformation painting’ into the RTCE plug-in. When invoked, Maya enters ‘painting mode’, which allows artists to interactively ‘paint’ black/white onto the geometry of the scene object. This process adds a new ‘layer’ of data to the geometry, which is used by the ‘deformation algorithm’. For more information, see the object section (page 130).

Third Design Iteration	Discussion
	This iteration focused on improving the ‘material composition’ and ‘procedural instancing’ subsections of the interface. In addition, changes to interface components were made to improve usability. An example is the increased size of the ‘Connect/Disconnect’ button, which is justified by the component’s frequent use. Other additions include ‘material/part scrolling buttons’, shader selection and icons for vertex format controls. Most additions in this revision were required by concurrent developments taking place in other areas of the project. No changes were made to interface parameters of the ‘deformation algorithm’ at this stage.
	A: Material Composition
	Significant revision to this subsection was made to enable artist control over ‘material composition’. The design continues the use of ‘tab panels’ as a way of expressing procedural composition for the ‘channels’ of an object’s material. Each tab panel in this sub interface now lists procedural functions that can be used in a composition, with simple controls to toggle the presence of the procedural function in a material composition.
B: Procedural instancing parameters	
The list component in this design has been simplified to remove ‘redundancies’ in the previous iteration. In this revision, the list is populated with just the names of objects that are available for instancing. Other data that was previously listed is no longer present. This subsection also provides control of the selection of ‘instancing procedurals’, in a similar way to the ‘Material Specification’ sub section. Finally, the design introduces the ‘Cookie’ text field. This allows artists to specify a ‘cookie image’ which invokes cookie cutter functionality in the procedural instancing algorithm. Details of this functionality are covered on page 125 of the instancing algorithm section.	

Final Design Iteration	Discussion
	This image represents the current interface of the RTCE plug-in. The interface now provides parameterization for all features and algorithms of the interactive tool chain. The design also aims to improve usability and ‘transparency’ of the plug-in within Maya. Changes in this iteration introduce notable ‘graphic enhancements’ to several of the interface’s components. The motivation for these enhancements is to follow a core principle in HCI; improved usability via symbolic association/affordance with interface functionality (Lidwell, Holden, & Butler, 2003). The design also follows the style of Maya’s own user interface(s) (namely those of material and texture creation). This targets artists/users that are familiar with Maya’s conventions, potentially making the functionality of RTCE to be quickly understood via familiar interfaces. Thus, this revision focuses on improving the plug-in’s usability rather than the introduction of functionality or ‘parameter controls’.

A: Material Composition

This material composition interface merges elements from previous interface revisions, to deliver a flexible and user centric solution. Notable changes include the removal of the ‘channel’ tabs, which have been replaced by a single interface panel. This panel contains high level controls for each of three configurable material channels.

Interface controls for procedural composition have been transferred from the RTCE’s main interface, into a separate dialog box. This simplifies the overall material composition interface, allowing space for a ‘swatch’ display to provide local visual feedback of the material being composed.

The components for ‘material scrolling’ have been moved to this subsection. The motivation was to clarify the relationship between multi-material objects and the interface’s material composition features.

As discussed, Maya’s ‘LayeredTexture’ structure underpins data storage for material composition in RTCE. Maya already provides user interfaces for configuration of ‘LayeredTexture’s’ and thus, these were reused to maintain user familiarity. This seems appropriate, given the plug-in’s objective of integration and ‘transparency’ within Maya. The

RTCE now provides ‘shortcut buttons’ to give direct access to Maya’s respective procedural/texture configuration interfaces. Using the conventions and interfaces that experienced users of Maya are familiar with, helps to minimize learning overhead.
B: Procedural instancing parameters
This subsection inherits most design elements from the previous design iteration. The layout of components in this section has however, been reorganized to establish consistency with the ‘Material Composition’ subsection. In addition, the controls which allow selection of procedural functions for the instancing algorithm have been revised. In previous designs, buttons were available that opened Maya’s procedural function ‘catalogue’. Because the instancing algorithm only supports a subset of Maya’s procedural functions, it is sufficient to provide a drop down list with just the supported procedural functions. By not presenting the procedural function ‘catalogue’, this improves workflow as only relevant/supported procedural functions are exposed to the artist. This revision also allows artists to specify the ‘texture coordinate set’ (of an object) to use with the procedural instancing algorithm. Note that objects can have more than one ‘texture coordinate set’ (or channel). Multiple texture coordinates for objects are supported in this tool chain by the dynamic vertex format feature. The new control allows the artist to choose the set which ‘drives’ the procedural instancing algorithm. For more information on the influence of texture coordinates in the instancing algorithm, refer to page 119 of the instancing algorithm section.
C: Unique deformation parameters
The deformation interface has also been changed by the introduction of ‘deformation scaling’. During software testing, the need for control over deformation ‘amplitude’/scale became apparent. This is mainly because the scale of 3D worlds/scenes is arbitrary. In an effort to address this, a ‘scaling’ parameter of the deformation algorithm was exposed in this subsection of the RTCE interface. Thus, when procedural deformation is applied to an object, the artist can interactively adjust the scale/amplitude of deformation to achieve the required result.
Table 8 Development chronology of the RTCE interface

Game rendering context (GRC)

As discussed, this project implements an interactive tool chain that allows artists to apply PM’s to a range of authoring processes, to enhance their production workflow. Recall that the tool chain uses a ‘connection model’, with independent content authoring tools and game

rendering technology components. This section covers the design and implementation of the tool chain's 'game rendering component' (GRC).

In a commercial setting, the GRC would ideally be the engine of a studio's game project. In this research however, a custom 'prototype renderer' has been implemented as a substitute for the game engine. The main reason for this is to facilitate experimentation with rendering features in modern graphics hardware, which is relevant to algorithms of this research. This research explores the possibility of exposing 'adaptive'/customizable procedural elements to artists, in an interactive environment. This requires that a renderer with specific low level capabilities, particularly in relation to shader integration, be implemented.

Before an approach via a custom prototype was selected, a survey of available game rendering/engine technologies was carried out. The survey concluded that features and functionality offered by 'renderer candidates', didn't fully align with the requirements of the research; mostly in terms of their lack of Direct3D 10 support. As discussed, Direct3D 10 offers a comprehensive feature set, which provides features specifically required by this research's algorithms. Of the engines surveyed, the majority of these lacked support for Direct3D 10. Furthermore, those that did offer support were either unavailable (due to commercial licensing) or limited in terms of extensibility/modification. On this basis, the custom 'prototype' framework was implemented. Refer to Appendix A for a summary of the rendering technology survey.

GRC characteristics

The following lists the GRC's software characteristics, and introduces some features that will be fully described in the following sections:

Technological:

- Based on Direct3D 10
- Uses Shader Model 4.0 (SM4.0) (Shader Model 4, 2010)
- Programmed in C++

Functional:

- Shader-based rendering
  - Multi-pass shader support
  - Expose data 'Streamed' from shaders (see page 103)
  - Support 'Hardware Instancing' (see page 106)
  - Real-time procedural/material composition
    - 'Multi channelled' materials
    - Dynamic procedural functions
  - Adaptive shader system
    - Variable input data/geometry vertex formats
    - Allow custom shader behaviour
- Responsive network interface/tool chain
  - Update object data interactively:
    - Geometry
    - Shaders
    - Materials
  - Update object transformations interactively
  - Update scene lighting interactively
  - Add/remove objects on demand
- Resource management
  - Allocate and manage:
    - Textures/Material parameters
    - Geometry
    - Shaders
  - Material composition management/tracking
  - Shader management/tracking
- Maintain runtime performance and interactive rate
- Standalone application that is independent from other software components in the tool chain

GRC structure

Similarly to the RTCE, the GRC has a modular code structure and design. Despite the GRC's iterative development, the 'modules' that comprise the software have remained consistent throughout the project's duration. The following provides a brief overview of the GRC's modules:

Module	Description
Window	Encapsulates functionality that invokes, manages and displays the application's 'window'. This module is also responsible for basic event handling (namely mouse/keyboard interaction) which is communicated to the GRC.
Network	Represents the communication interface between the internal GRC application and external 'client workstations' (i.e. RTCE instances). When network data is received, this module decodes the data and invokes functionality of relevant, internal sub-systems of the GRC. The network module is essentially the GRC's 'event mechanism', given that it handles tool chain related events.
GUI	Provides a simple user interface that overlays a portion of the GRC's 'rendering canvas'. This GUI exposes the GRC's basic functionality and inherits the project's 'visual identity'.
Core	A simple 'container' that hosts other modules of the GRC, namely the 'Scene' and 'Renderer'. The core also centralizes event/data handling from the 'Window' module, as well as the network interface.
Scene	A 'container' that manages the game objects in the GRC's 'game scene'. This module is responsible for invoking update and rendering functionality on all registered game objects. In addition, the scene module manages data for any active light sources in the GRC.
Renderer	The GRC's most sophisticated module that incorporates an abstract 'render' interface, through which all rendering functions are invoked. Direct3D 10 functionality is integrated into the GRC via a 'D3D10' implementation (implements Direct3D 10 functionality) of this interface. The rationale of this abstraction is to facilitate extensions to the GRC; namely the future introduction of a Direct3D 11 based renderer. In addition, the interface is partially implemented via a Direct3D 9 based renderer. Note that this was implemented in the initial stages of the research project as a 'placeholder renderer'. The Direct3D 9 implementation now serves as a 'fallback' option for system configurations that lack Direct3D 10 level graphics hardware. This module also incorporates management of rendering resources/data such as 'auxiliary buffers', geometry and textures. A reference counting strategy is used for efficient memory use and robust resource sharing.

Materials	In addition to resource management, this module is capable of receiving and decoding material data from the network interface. Incoming network data is delivered to the material module in binary form. The module follows project wide conventions to extract parameters and data from an incoming data stream (that represent 'procedural/texture compositions'). In addition to updating materials, this module also manages material objects and integrates a reference counting scheme to permit data sharing throughout the application.

Rendering module

As mentioned, the rendering module incorporates an abstraction, for future extensibility and compatibility. However, in keeping with the current requirements, a Direct3D 10 implementation of the interface abstraction has been developed. The following discussion describes its implementation.

A side effect of 'shader based renderers' is that two different software architectures are required; these being the CPU and GPU. This introduces a need for two codebases.

The CPU based component is directly embedded into the application, in this case, the GRC's C++ code implementation. This code exists in the render module, which integrates/communicates directly with the other modules of the GRC.

The primary responsibility of the render module's CPU based component is accommodating and interfacing with functionality that is executed on the GPU architecture. As indicated, custom GPU functionality exists in 'shaders'. Traditionally, shaders were simple assembly programs supplied by developers to control stages of the rendering process (Shader, 2010).

As the need for more sophisticated rendering behaviour has arisen, flexible 'human readable', 'high level' shader languages were consequently developed. Notable languages include Microsoft's 'High level shading language' (HLSL) for use with modern Direct3D API's, as well as the 'OpenGL Shading Language' (GLSL), these both consisting of 'C' like syntax (HLSL, 2010) (Kessenich, Baldwin, & Rost, 2010).

The code excerpt in table 9 shows the definition of a complete shader written in HLSL. This excerpt also demonstrates various language features and syntax. Note that HLSL also offers a 'C pre-processor'; a feature which will be frequently referred to in subsequent discussion (Preprocessor Directives, DirectX HLSL, 2010).

A simple HLSL shader

```
float4x4 g_matrixWorld;
float4x4 g_matrixViewProjection;
float3 g_vectorLightDirection;
float g_scalarTime;
```

A

Declaration of shader variables, which are available for use by the shader. These remain constant during the shader's execution. They are usually specified prior to the shader's execution

```
void GetFastTime(out float time) {

       time = 2.0f * g_scalarTime;
}
```

Simple example, showing how 'user defined' functions can be defined in an HLSL shader

```
void MyVertexShader(
  float4 inVertexPosition : POSITION0,
  float2 inVertexTextureCoord : TEXCOORD0,
  float3 inVertexNormal : NORMAL0,
  out float4 outHomPosition : POSITION0,
  out float2 outHomTextureCoord : TEXCOORD0,
  out float4 outHomColor : COLOR0
)
{
  /*
  Transforms the vertex postion into homogeneous space
  Transforms the vertex normal into world space
  Assign texture coordinate to output
  Perform basic diffuse lighting for vertex color
  */
  float4 worldPosition =
    mul(inVertexPosition, g_matrixWorld);
  float4 worldNormal =
    mul(inVertexNormal, g_matrixWorld);

  outHomPosition =
    mul(worldPosition, g_matrixViewProjection);
  outHomTextureCoord = inVertexTextureCoord;
  outColor = saturate( dot( normalize(worldNormal),
  normalize(g_vectorLightDirection)));
}
```

C

Input and output parameters for vertex shader

Definition of the custom vertex shader

Geometry transformation and projection into homogenous coordinates, etc

Returning output from the vertex shader

```
void MyPixelShader(
  float4 inHomPosition : POSITION0,
  float2 inHomTextureCoord : TEXCOORD0,
  float4 inHomColor : COLOR0,
  float3 outPixelColor : COLOR0
)
{
    /*
    Shows how the custom function is invoked
    */
    float time = 0.0f;
    GetFastTime(time);

    outPixelColor =
      inHomColor * (sin(time) * 0.25 + 0.5);
}
```

D

Shows the definition of custom pixel shader functionality. The collective image for pixels of this shader will yield the appearance of lighting from a 'global' light source. The objects brightness will also oscillate according to a sine wave

```
technique MyTechnique
{
  pass MyFirstPass
  {
    VertexShader =
    compile vs_4_0 MyVertexShader();
    PixelShader =
    compile ps_4_0 MyPixelShader();
  }

  pass MySecondPass
  {
    VertexShader =
    compile vs_4_0 MyVertexShader();
    PixelShader = NULL;
  }
}
```

B

Definition of a 'shader technique'. Shaders can have numerous 'technique' blocks however this shader only has one. A technique typically associates vertex/pixel shaders together, in a 'pass' (or stage). The following technique shows a multi-passed shader. Note that passes can independently assign shader functionality

A 'pass' block of the technique (referred to in this thesis as a stage)

Table 9 Code excerpt showing typical features of an HLSL shader

As indicated, the main function of the renderer's CPU component is to control and invoke rendering functionality on the GPU/graphics hardware, to deliver high quality, real-time rendering. Additional responsibilities of the CPU component include initialization of the graphics device/hardware. This takes place during the GRC's 'window creation' phase, and is essentially a 'run once' process.

At runtime, the CPU component of the renderer is repeatedly invoked (~30 times per second) during the GRC's application loop. One of the CPU component's main responsibilities is 'binding' required data resources to the graphics device in preparation for rendering. In

addition, ‘shaders’ are systemically invoked/executed by the CPU component, to render geometry in the context of customized shader/rendering functionality.

With respect to the GRC, rendering resources include raw geometry and render state information, as well as texture data. The GRC also makes use of ‘data binding’ functionality to specify parameters that are subsequently used by shader based procedural functions. This is equivalent to specifying the data for shader variables, such as those shown in (A) of table 9.

The binding process is often referred to as ‘uploading’, given that it represents data transfer from the CPU/host system, to the graphics hardware/GPU. The approach to resource binding is typically not important, provided that all required resources/parameters are bound prior to the specific rendering process.

Once the CPU has bound/uploaded a shader and its associated rendering resources to the graphics device, the CPU then specifies basic information about the rendering task; namely the number of triangles (or ‘primitives’) that must be drawn. Within the GRC process, this operation takes place during the ‘render’ routines of each registered ‘game object’.

Given that rendering takes place in the context of a ‘game object’, the design elegantly parameterizes the device with relevant information, internal to that object. The GPU then immediately carries out the rendering process. This typically causes the GRC game object to be ‘rasterized’ (‘painted’ from geometry) to the screen (Rasterisation, 2009).

Note that the GRC incorporates auxiliary functionality which invokes the GPU under different rendering ‘configurations’. The reason for this is to process and yield different types of data from the GPU, which is used at subsequent stages in specialized rendering processes. For more detail on this functionality, refer to page 103 of the instancing algorithm section.

The GRC’s standard geometry rendering process can therefore be summarized as the amalgamation of rendering data by the GPU, to yield a buffer of pixel colours that represents all scene geometry in image/bitmap form. This conventional use of rendering hardware is central to the GRC’s delivery of real-time rendering.

The notion of ‘multi-staged’ rendering is fundamental to two specialized ‘rendering’ algorithms which have been developed during this research. Further detail on the implementation of these algorithms (and their specific application of multi-staged rendering) can be found in the instancing algorithm section (page 100) and object section (page 130) of this thesis. Because these algorithms are integrated into the GRC, the rendering module must therefore, support multistage rendering. Further discussion on multi-staged rendering, as well as their implication on advanced rendering techniques and effects, can be found on page 101.

In summary, multi-stage rendering enables ‘complex’ rendering processes that are represented by a number of intermediate ‘shading’ steps. As the ‘technique’ (B) in the HLSL except of table 9 shows, incorporating multiple shader stages or ‘passes’ can be invoked/accumulated to a final rendering result, during invocation of a single technique of an (HLSL) shading/rendering process.

The CPU component of the GRC’s renderer supports multi-staged shaders/rendering via a loop structure, where each loop iteration corresponds to a stage in an HLSL technique. These shader stages are incrementally invoked during the rendering process of a multi-staged shader. To achieve this, the GRC queries information from a data structure that encapsulates the HLSL shader. More specifically, the GRC queries the number of stages in the shader/technique. This is used to specify the number of loop iterations. In the Direct3D 10 API, the data structure which encapsulates shader code and provides these query functions, inherits the ‘Effect’ interface (Effect System Interfaces, 2010) (ID3D10Query Interface, 2010).

‘Effect’s’ simplify the use of shaders within a rendering application, essentially encapsulating functionality for binding shader code and rendering resources to the associated graphics hardware. When an HLSL shader is compiled by the Direct3D 10 API, an ‘Effect’ object is returned which internally contains shader functionality that corresponds to the specified shader code (D3DXCreateEffectFromFile Function, 2010).

As indicated, shaders play a central role in the GRC’s delivery of real-time rendering. Coincidentally, shaders also underpin other central features of the tool chain; namely dynamic ‘vertex formats’ and ‘procedural compositions’. These features are highly influential towards the implementation and integration of the GRC’s shaders.

Adaptive vertex format for shaders

As mentioned, the tool chain supports arbitrary ‘vertex formats’ in the geometry that it manages and renders (via the GRC). Recall the motivation for this is to make efficient use of ‘data bandwidth’ at different stages of the tool chain. Efficient bandwidth use is an important part in the tool chain’s delivery of an interactive/responsive workflow; particularly given its basis on a network connection.

As discussed, configurable vertex formats allow artists to arbitrarily select vertex ‘elements’ that are ‘interleaved’ into the geometry of game objects’ in the tool chain. Through this, the transfer and management of redundant vertex information can be avoided. Thus, smaller geometry buffers are transferred across the network connection that exists between software components of the tool chain.

In addition to improved network bandwidth use, smaller geometry buffers improve 'upload' efficiency between the CPU and GPU, during the GRC's real-time rendering process. Recall that rendering via the GPU, involves data transfer between the CPU and GPU. Although this is a reasonably fast process, it often represents an undesirable 'time expense' relative to the real-time nature of rendering applications. Thus, by minimizing the net size of data (geometry) that is uploaded, this aims for faster runtime performance in the GRC.

Shader programs have traditionally been applied to two different parts of the rendering process. This is illustrated by the HLSL shader structure in table 9, where a 'vertex shader' (C) (i.e. geometry processing/transformation), and 'pixel shader' (D) (rasterizing processed geometry into pixel buffers/bitmaps) are defined.

During the vertex shading process, the shader is applied to each vertex of the geometry buffer which is subjected to the shader that is bound to the respective graphics device. Following this, an accompanying pixel shader is invoked which controls the way pixels that represent the geometry (in bitmap form) are coloured/shaded.

The elements of each vertex (i.e. per-vertex position, normal, texture coordinates, etc) in a geometry buffer must 'align' with the declared input parameters of a vertex shader's definition. The code excerpt in table 10 shows the declaration structure for a typical vertex shader ('`vertex_shader`') in HLSL. This vertex shader requires each vertex of the geometry being 'shaded' to consist of a position, normal and colour vector, as well as two texture coordinate 'channels' (see A, table 10). The shader also assumes that vertex elements are specified in this 'order' for data that is being streamed to the shader.

Simple vertex shader

```
void vertex_shader(
  float4 vertex_position : SV_Position,
  float3 vertex_normal : NORMAL0,
  float4 vertex_color : COLOR0,
  float2 vertex_texcoord_a : TEXCOORD0,
  float2 vertex_texcoord_b : TEXCOORD1,
  out float4 transformed_position : SV_Position,
  out float3 transformed_normal : NORMAL0,
  ...
)
{
  /* vertex_shader implementation */
  transformed_position = ...
  transformed_normal = ...
  ...
}
```

A

Input parameters of the vertex shader. These must 'align' with the elements of the associated vertex that is being rendered/shaded

Processed data can be returned from a HLSL via 'output' parameters. Note that the types/elements of input parameters do not have to correspond to output data

Table 10 Code excerpt showing main features of a simple shader declaration

As discussed, shader declarations (such as that in table 10) must be compiled prior to being used in the rendering process; this obviously makes the shader 'static' and only compatible with a single vertex format/structure (corresponding to its input parameters). In other words, when this shader code is compiled into GPU assembly, the resulting shader program will only operate on a buffer of vertices that match these input parameters.

To facilitate dynamic vertex formats, the input parameters of the GRC's shaders must be adaptable to arbitrary vertex data/structures. This was achieved by taking advantage of HLSL compilation functionality provided by the Direct3D 10 API, as well as HLSL's language/syntax features.

Recall that the vertex format of an object can be changed by artists during runtime; this consequently alters the 'layout' of that object's geometry buffer. To handle this, shaders' of the GRC must be adaptable to arbitrary 'vertex data' formats in the geometry data received from the RTCE. Furthermore, shaders must be adaptable 'on demand' during runtime.

If notification of a changed vertex format is received, the GRC updates the HLSL shader that is associated with the changed geometry. This 'update' process corresponds to the shader being recompiled by Direct3D 10's HLSL compiler, in order for the shader to 'comply' with the revised geometry format. When shader recompilation occurs, in response to an 'updated vertex format', this consequently results in the input parameters (A, table 10) of that vertex shader being added/removed to reflect the format of the underlying geometry.

Note that under some circumstances, shader recompilation can be avoided. If for example, the render module already has a cached instance of the required shader, that particular shader instance will be reused. The caching system will be briefly discussed in subsequent discussion.

<table>
<tr><th>Vertex shader prior to pre-processing phase</th><th>Vertex shader following pre-processing</th></tr>
<tr><td></td><td>The PPD list used in this example:
{ VERTEX_POSITION, VERTEX_NORMAL } B</td></tr>
<tr><td>void vertex_shader(
A
#ifdef VERTEX_POSITION
/* Position always required */
float3 in_position:SV_Position,
out float4 out_position:SV_Position
#endif

#ifdef VERTEX_NORMAL
,float3 in_normal : NORMAL
,out float3 out_normal : TEXCOORD5
#endif

#ifdef VERTEX_COLOR
,float4 in_color : COLOR
,out float4 out_color : COLOR
#endif
...
/* Other availible parameters (tangent, uv1, uv2, uv3, uv4, etc) omitted*/
)
{

... /* Implementation */

#ifdef VERTEX_POSITION
out_position = ...;
#endif
#ifdef VERTEX_NORMAL
out_normal = ...;
#endif
#ifdef VERTEX_COLOR
out_color = ...;
#endif
...
}</td><td>void vertex_shader(
C

float3 in_position:SV_Position,
out float4 out_position:SV_Position

,float3 in_normal : NORMAL
,out float3 out_normal : TEXCOORD5
)
{
/* Implementation */
...
out_position = ...;
out_normal = ...;
}

Note that the pre-processed shader omits parameters from the ‘shader template’ (left coloumn) that are not included in the PPD list.</td></tr>
</table>

Table 11 Shows how pre-processing capabilities of HLSL are used to deliver shaders which are adaptive to arbitrary vertex formats

The GRC achieves interchangeable input parameters for its shaders, via the use of pre-processor definitions (PPD). As shown in table 11, this is achieved by 'strategically' embedding PPD's throughout the definition of a GRC shader (namely in the parameter declaration). Prior to a shader's recompilation, the GRC generates and passes a list of PPD's to the HLSL compiler, which correspond to the artist specified vertex format. Section (B) in table 11 shows an example of a typical PPD list that is generated by the GRC and passed to the HLSL compiler. The PPD list directs the pre-processor which in this case, yields HLSL code that corresponds to (C) in table 11. Thus, table 11 shows how HLSL code is dynamically generated to yield shader's where input parameters and functionality are tailored to align with the specified vertex format.

Thus, the generation of a PPD list is pivotal to this 'dynamic shader system'. Note that this functionality resides on, and is executed by, the CPU. As an aside, the performance of the pre-processing and shader recompilation processes, were found to be adequate. Under typical circumstances, shader compilation took ~5 seconds. This is reasonable given that the geometry vertex format is not frequently altered.

The interchangeable 'parameter interface' (i.e. shader input) described, is fundamental to a shader's compatibility with arbitrary geometry formats in this tool chain. There are however, other equally significant elements to this 'shader system'; namely a structure that enables a high level of customization to facilitate the needs of both artists and 'technical artists'. The objective of this, is to provide a shader system that fulfils artists' typical usage requirements, while also permitting unique rendering behaviour that is often required to deliver distinct visual experiences in games.

In essence, the system amalgamates the arbitrary shader parameter interface, with a 'structure' capable of invoking custom functionality. To deliver this, the structure makes integral use of 'global function declarations' within the system. The role and integration of these functions will be covered in subsequent discussion.

This structure is relevant in a tool chain such as this, where many permutations of 'input parameter' configurations exist. From the developer's perspective, implementing custom shader functionality in this tool chain would require manual definition of the custom shader for all possible parameter configurations. This would be impractical, given that games tend to consist of many shaders.

The shader system counteracts this however, by providing an intuitive development interface that allows custom shader functionality to be easily implemented, while retaining compatibility with arbitrary parameter/vertex format configurations. Thus, the system 'hides' the complexity of 'configuration permutations' when implementing custom shader behaviour, requiring that only two functions be implemented. These functions contain the custom shading behaviour for vertex and pixel shading, respectively.

As mentioned, this system facilitates the needs of two 'developer bases'; the artist, and the technical artist. Through the system's adaptive/runtime characteristic, the needs of artists during typical usage scenarios, namely the creative process (involving specification of vertex formats, etc), are fulfilled. In addition, the system also caters to the specification of custom shader functionality, often required by the studio's technical artist(s). Because the specification of custom shader code is a 'less common' occurrence however, the tool chain does not directly expose shader development interfaces in the authoring environment. Instead, the technical artist provides implementations for the mentioned function pair, via a text editor. The schematic in figure 37 shows the structure of the shader system that has been described.

Recall that the artist interface (i.e. Maya plug-in) of this tool chain exposes functionality that allows an HLSL shader to be assigned to a geometric object. Thus, the 'technical artist' can conveniently develop and apply custom shader functionality during runtime, effectively taking advantage of the system's capability for runtime shader recompilation. When a newly modified shader is applied to an object in Maya, the tool chain 'packages' the shader code into a network packet, sending it from the Maya plug-in to the GRC. When the shader is received by the GRC, the code is unpacked, installed, compiled and applied to the GRC's rendering process respectively. This process occurs interactively during run time, and aligns with the tool chain's objective to provide interactive 'development', coupled with maximized developer flexibility.

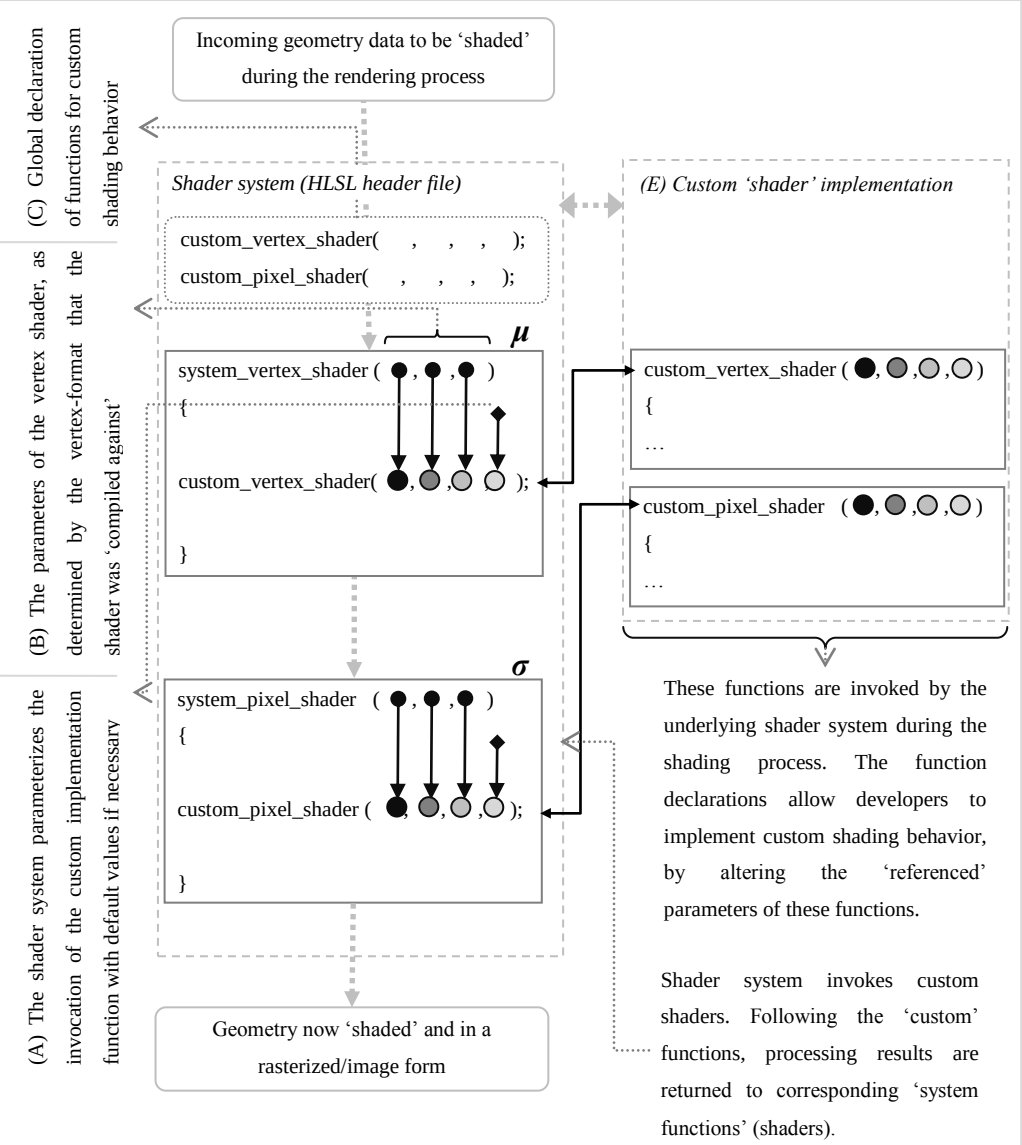


Figure 37 Schematic of the GRC's shader system that supports custom shader functionality in conjunction with dynamic vertex formats

This system effectively 'abstracts' the notion of conventional shader development. As discussed, the developer provides custom shader code/implementation for functions that are globally declared by the shader system (see item C in figure 37).

Note that the shader system being described resides in an HLSL header file, as figure 37 illustrates. Thus, when developing a custom shader for the GRC, the shader's source needs to 'include' the 'shader system's' header file, as well as implementations for the globally declared vertex and pixel shading functions ('custom_vertex_shader' and 'custom_pixel_shader'). The code excerpt in table 12 shows the source code for a shader, developed for this system. Item (E) in figure 37 represents a custom shader incorporating the

function definitions that encapsulate custom shader behaviour. Note that these are invoked by the underlying shader system.

Semantics and structure required for an HLSL shader, such as the 'technique blocks' and passes, are defined in the shader system header that is included. As discussed, the 'custom functions' are globally declared (see C in figure 37) in this system's header file. Because the technical artist/developer provides definitions for these global declarations in the subsequent 'custom shader file', the system therefore inherently relies on the linking capabilities of the Direct3D 10's HLSL compiler. Note that during compilation, code for the shader system (which exists in the header file) is 'expanded' into the custom source file, defined by the developer.

Recall that items (C) and (D) of the HLSL shader in table 9 (page 78), represent the vertex/pixel shaders that are executed by the GPU, during rendering/shading. Within this shader system, these are represented by the 'system_vertex_shader', (μ) and 'system_pixel_shader', (σ), shown in figure 37. For this discussion, these 'system_*_shader' functions (μ and σ) are referred to as 'wrappers'.

Thus, as figure 37 shows, each wrapper invokes the corresponding 'custom shader' function which is explicitly provided by the developer.

Implementing custom shader behaviour in the GRC's shader system

```
#include "lib_system.fxh"

void custom_vertex_shader(
  float3 in_position,
  float3 in_normal,
  float3 in_tangent,
  float2 in_uv1, float2 in_uv2,
  float2 in_uv3, float2 in_uv4,
  float4 in_colour,
  float4x4 in_transform,
  out tOutput out_data
)
{
  /*
  Applies custom vertex shading behaviour to 'warp' the geometry. This
  illustrates how customized vertex manipulation/behaviour is integrated
  via the described shader system.
  */
  float4 warp_position
    = in_position.x + sin(g_time);
```

```
  /*
  Prepares vertex data for rasterization, namely by transforming and
  projecting vertex positions to homogeneous coordinates.
  Assigns input vertex data to shader's output
  */
  out_data.os_position = warp _position;
  out_data.normal      = mul(in_normal, (float3x3) in_transform);
  out_data.tangent     = mul(in_tangent, (float3x3) in_transform);
  out_data.ws_position =
    mul(float4(warp _position,1.0f), in_transform);
  out_data.h_position =
    mul(float4(warp _position,1.0f),mul(transform,g_viewproj));
  ...
}

void custom_pixel_shader(
  float3 in_os_position,
  float3 in_ws_position,
  float3 in_ws_normal,
  float3 in_ws_tangent,
  float2 in_uv1, float2 in_uv2,
  float2 in_uv3, float2 in_uv4,
  float4 in_colour,
  out float4 out_colour
)
{
  /*
  Applies custom colour manipulation to the pixel shading process
  (tinted red in this case). This illustrates how custom pixel shading
  behaviour via the described shader system, is achieved.
  */
  float4 pixel_colour  = in_colour;
  pixel_colour.g              = 0.0f;
  pixel_colour.b              = 0.0f;

  ...

  /*
  Returns the computed colour for this pixel in the rasterized result.
  */
  out_colour = pixel_colour;
}
```

Table 12 Code excerpt showing main features of a simple shader declaration in the GRC's shader system

As discussed, the GRC's shader system combines 'adaptive' shader interfaces with structure that facilitates custom pixel and vertex shading functionality. Recall the motivation for this is to simplify the task of implementing custom functionality for technical artists, under an 'adaptive' shader context.

Achieving this however, requires that ‘adaptive’ shader code elements (similar to those used in the adaptive ‘shader interface’) be incorporated into the shader system’s ‘wrappers’. Shaders of the GRC feature adaptive input parameters, making them compatible with arbitrary geometry/vertex formats. The following discusses how the arbitrary shader parameter interface is ‘assembled’ with corresponding ‘custom shading functionality’ which itself, consists of a static interface.

The subsequent discussion explains this assembly, in the context of the system’s vertex shading capabilities. Note that these ‘principles’ are similarly applied to the system’s pixel shading functionality.

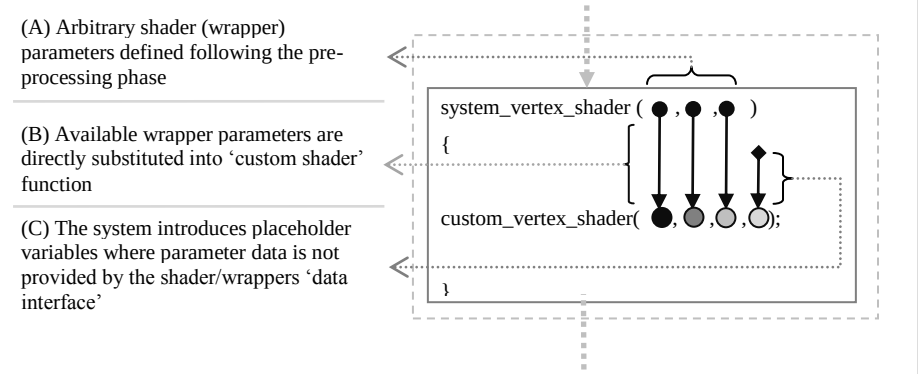


Figure 38 Illustrates the adaptive mechanism which handles the automatic substitution of parameters for custom shader functions when required

Item (A) in figure 37figure 38 depicts an ‘arbitrary’ set of input parameters for a vertex shader/wrapper of the system. The wrapper’s parameters are directly passed to the corresponding ‘custom vertex’ function. As discussed, the definition for this custom vertex shading functionality is supplied by the developer in the accompanying shader source file.

These ‘custom’ shading functions are declared with parameters that correspond to every type of vertex element that is available in this tool chain. These functions are therefore capable of ‘facilitating’ any set of parameters that correspond to any vertex format that an artist could specify for an object.

A shader’s vertex format typically only incorporates some of the available vertex elements, offered by the tool chain system. Thus, pre-processed shaders will usually only provide some parameters for the wrapper’s invocation of the ‘custom shader’ function, as figure 38 shows. Note that the ‘parameter list’ of the pre-processed shader (A of figure 38) only natively supplies some of the parameters required by the ‘custom shader’ function.

Under normal circumstances, a compilation error would occur due to the ‘custom shader’ function not being fully parameterized. The shader system handles this by introducing

‘placeholder’ variables into the wrapper’s body which are used as necessary (see item C in figure 38). These placeholder variables (initialized to default values) prevent syntax errors occurring during the shader’s compilation phase. As mentioned, this placeholder mechanism is also applied to the shader system’s ‘pixel shading’ stage.

In summary, this aspect of the GRC shader system provides a means for linking custom shader functionality to an arbitrary data interface for shader parameters. Furthermore, this system feature is fully automated in that no specialized intervention is required by artists during the shader compilation process. This represents a significant part of the GRC’s shader system structure.

In addition, the GRC’s shader system consists of another major component, which is also based on compile time assembly/PPDs, namely shader based procedural material composition. This ‘component’ encapsulates sophisticated procedural functionality in the context of the shader system, through a suite of high level ‘combiner’ functions. These functions are provided by the system for use by developers/technical artists, when defining custom shader functionality, using this system.

Shader based procedural composition

The previous discussion illustrates the GRC shader system’s sophisticated use of HLSL to deliver ‘adaptive’ shaders. Adaptive shaders based on pre-processor based shader compilation, proved to be robust during initial development and thus, these principles have been reapplied to deliver interchangeable, shader based ‘material composition’.

Recall that the ‘Maya plug-in’ (RTCE) provides user interfaces and functionality that permit material creation via arbitrary composition of procedural functions. In other words, these RTCE interfaces expose parameters that enable specification of materials which are used in geometry, during the GRC’s rendering process. Complex material compositions can amount to sophisticated calculations which must be evaluated in real-time, for each pixel that represents the material’s underlying geometry. For this reason, it is appropriate to harness the parallel processing capabilities of the GPU, to deliver this process at real-time/interactive rates.

Because material composition is provided in this ‘interactive’ tool chain context, changes to material compositions during runtime, must be immediately reflected following artist interaction with the system. Subsequent discussion shows how this was achieved, via the reuse of established elements of the GRC’s shader system.

Although the majority of the material composition system is based in shader code, some elements of this component are CPU bound. This is necessary in order for material parameters to be stored between render cycles of the GRC application.

Recall that rendering processes based on the GPU architecture, require data to be 'uploaded' to the GPU prior to the GPU based rendering process. Because the shader based material system is parameter driven, it therefore requires that parameters be specified (uploaded) on the GPU prior to the material being rendered/applied.

Thus, when a material composition is rendered by the GPU, it is merely a reflection of the uploaded 'parameter/data context'. Given that parameters are uploaded to the GPU on a per-frame basis this therefore, underpins the immediate/interactive response of the material system to parameter changes invoked from the RTCE. This 'interactive side effect' serves as another motivation for providing a GPU based material composition system.

Recall that the RTCE permits advanced material composition for use on geometry. In addition to 'composition' of procedural functions, the material composition system also allows artists to specify unique compositions for different 'channels' of a material. The material channels that are supported by this system are:

- Colour channel
- Bump channel
- Auxiliary data

Using this system, an artist could for instance, combine the 'Perlin noise' function with a texture image, to dynamically compose a 'gritty' variant of the texture image. The artist can choose to express this composition via the material's 'colour channel'. This would result in the composition being explicitly visible as colour, across the geometry to which it is applied.

In addition, the tool chain/material system allows artists the option of applying a procedural composition to the material's 'bump channel'. Thus, the material's 'bump' surface enhancement could for example, be based on a noise procedural composed with a procedural checker pattern.

Note that the material system is based in the GRC's pixel shading stage; this making the delivery of per-pixel shading effects, such as 'bump mapping', feasible. Bump mapping emulates additional surface detail on geometry, by altering the interaction of scene lighting across geometric surfaces (Elias, 1998). In this system, this 'alteration' is typically based on the results of a procedural composition, in the materials bump channel. Because bump

mapping occurs on a per-pixel basis, it naturally integrates into this pixel shader based material composition system.

Thus, the material system takes advantage of this context, to deliver and express data in a range of ways via different channels.

As indicated, this tool chain natively supports a third 'auxiliary' channel. The auxiliary channel provides an additional source of data for use in the development of customized pixel shading functionality.

An example of the auxiliary channels application could be to specify transparency in the material's rendering result. Thus, the technical artist could implement a shader which interprets data in the auxiliary channel, to allow artists to procedurally specify the transparency of geometric surfaces which apply the material/shader.

The schematic in figure 39 provides a visual representation of the GRC's multi-channel material composition system.

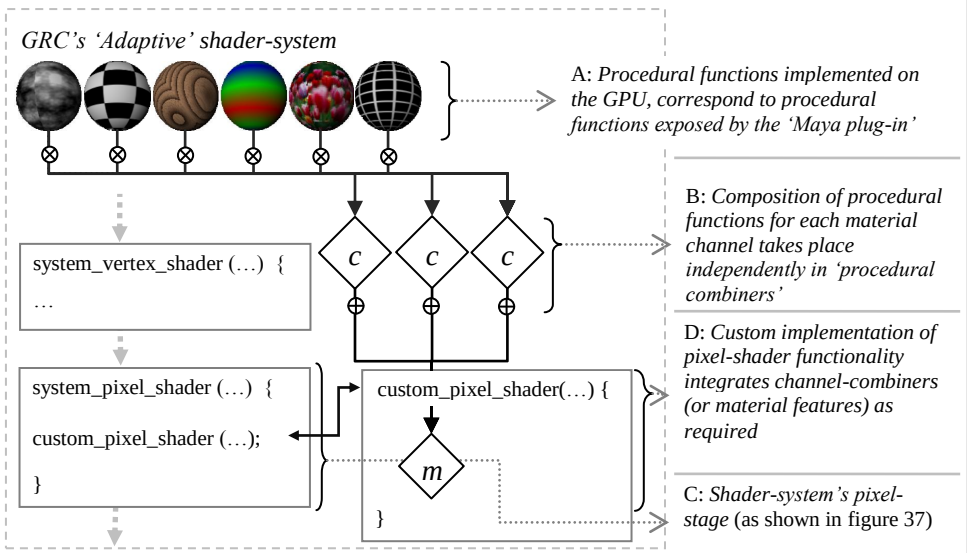


Figure 39 Illustrates the GRC's shader based material composition system

Figure 39 reiterates the concept of per-channel procedural composition, as interaction between items (A) and (B) of the schematic show. Each of the available procedural functions, as shown in (A), can be used during a composition or 'combining' process (B) of a material channel. Item (D) represents the integration of custom pixel shader functionality. Note that 'combining processes' for each of the three material channels can be optionally invoked/incorporated into the 'custom' implementation of a pixel shader.

To achieve arbitrary ‘procedural compositions’ in the pixel shader, the system reuses the concept of pre-processor directed shader compilation. Thus, when an artist specifies a composition for a material channel, the system responds by recompiling the respective shader.

As discussed, the recompilation process yields a shader that corresponds to the artist’s changes; in this case, a procedural composition that reflects the artist’s actions in the RTCE. Thus, shader recompilation underpins the interchangeable/customizable nature of material compositions in the system.

Note that although material composition could be achieved via composition structure which is based on ‘conditional statements’, there is a major disadvantage to this approach. Composition based on switches/conditional statements would obviously require invocation to each procedural function to exist throughout the body of the shader.

Recall however, that HLSL compilation implicitly expands the code that underlies function calls, into the accommodating shader. Thus, the accumulated effect of a switch based approach would be the compilation of overly complex shaders. Aside from taking longer to compile, most of the shader’s complexity would typically be unutilized.

The approach taken in the GRC’s shader system, applies PPD’s to yield smaller and more concise pixel shader code, which is executed by the GPU during rendering. As in the ‘arbitrary parameter interface’, pre-processor driven compilation ‘culls’ unnecessary code from the shader source that is passed to the HLSL compiler. A beneficial side effect of this, is less computational processing required at the pixel shading stage of the rendering process; this having positive implications on the GRC’s runtime performance.

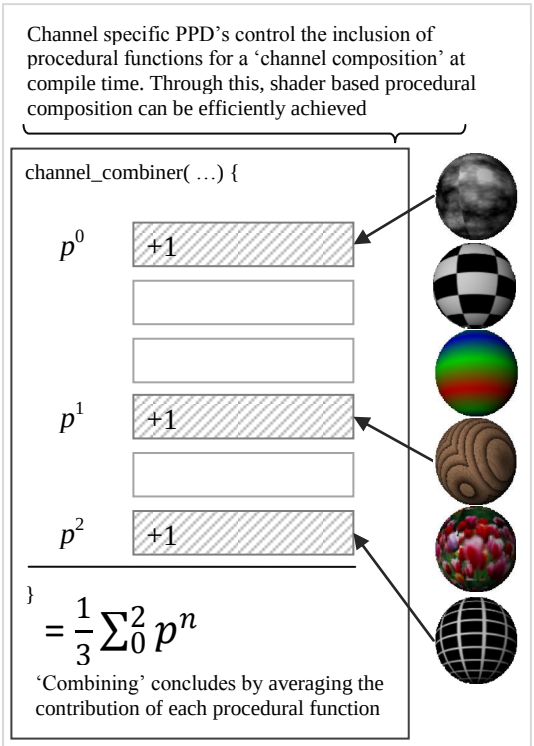


Figure 40 Shows how procedural functions are included/excluded in a procedural composition during shader compilation

When the GRC receives network notification to update a material, it responds by generating a list of PPD’s which correspond to the new material composition. These PPD’s instruct the HLSL compiler’s pre-processor as to which ‘blocks’ of ‘procedural functionality’ to include/exclude from source code that is complied. The inclusion/exclusion of

‘procedural blocks’ constitutes the system’s ability to arbitrarily combine procedural functionality in a shader. An abstract illustration of this ‘block based’ combination system is shown in figure 40. Note that systematic organization of source code, coupled with consistent use of conventions for procedural function definitions, is used to deliver a compiler driven combination mechanism, in the context of a shader.

Compile time material composition, driven by PPD’s

```
float denominator = 0.0f;
float3 result = (float3)0.0f;

...

#ifdef PROCEDRUAL_CHECKER_CHANNEL_COLOUR
        sum += Procedural_Checker(coordinates, checkerParam_lv_colour);
        denominator++;
#endif
#ifdef PROCEDRUAL_PERLINNOISE_CHANNEL_COLOUR
        sum += Procedural_PerlinNoise(coordinates, rampParam_lv_colour);
        denominator++;
#endif
#ifdef PROCEDRUAL_GRID_CHANNEL_COLOUR
        sum += Procedural_Grid(coordinates, gridParam_lv_colour);
        denominator++;
#endif

...

result /= (denominator == 0.0f ? : 1.0f : denominator);
```

Table 13 Code excerpt shows how ‘blocks’ of code are conditionally introduced to a shader at compile time, to deliver procedural composition

As the code except in table 13 shows, all procedural functions of the shader system’s procedural library, are explicitly incorporated into the source code. Note that this code represents the implementation of a ‘combiner function’ (see channel combiner in figure 40). The result of each ‘invocation’ of a procedural function is accumulated into a single variable (‘`result`’) that is declared in the scope of the combiner function. PPD’s enclose each procedural function, enabling the provided PPD’s to dictate which of the available procedural functions contributes to the ‘combined’ result.

Note that functions of this procedural library follow a ‘contract’, where the returned (scalar) values must comply to the range of [0.0 - 1.0]. This convention was selected, as it constitutes the range of colour intensity that most graphics hardware can express (for red, green and blue colour channels).

Note that the accumulation of most ‘procedural combinations’ is likely to yield a result that exceeds the noted colour range. To yield correct visual results, the combiner function ‘averages’ the contribution of the procedural functions in a procedural combination.
This requires a denominator to ‘divide’ the accumulated combinations to a combined ‘average’. Because the complexity of combinations is arbitrary, the denominator is consequently variable.
The ‘compiler driven’ combination function integrates a special code structure to handle this automatically. The code excerpt in table 13 is taken from the GRC’s shader system and shows the main elements of this compile time averaging strategy. Each time a procedural function is included in the shader’s source code by PPD’s, the local ‘`denominator`’ function is incremented. The accumulation of procedural functions concurrently yields a denominator value that corresponds to the number of procedural functions involved in the composition. As table 13 shows, the denominator is used by the combination function to compute the ‘average’ or ‘combination result’, of all procedural functions selected for the material composition.
Test cases and examples of outcomes produced by the discussed shader based material composition system can be found in the demonstration chapter.

In addition to the objectives of this research, characteristics of this system are inherently influenced by a variety of sources, namely research material covered in the literature review, as well as consultation with industry (see page 35 in the project design chapter).
As discussed, the system facilitates a high degree of ‘runtime configuration’, enabling arbitrary geometry ‘data formats’, in conjunction with complex material composition and support for custom shader behaviour. These characteristics align with the research’s objective to deliver ‘artist centric’ content creation that integrates and exposes the capabilities of PM’s throughout the content creation process.
Furthermore, the system integrates PM’s into the research’s tool chain, to facilitate more detail in games content, while abiding with identified constraints that face game developers.
As implied, the procedural functionality that underlies this material composition system exists in a generalized shader based ‘function library’. Part of the motivation for this generalized library, was to make the procedural functionality usable by other algorithms developed during this research.

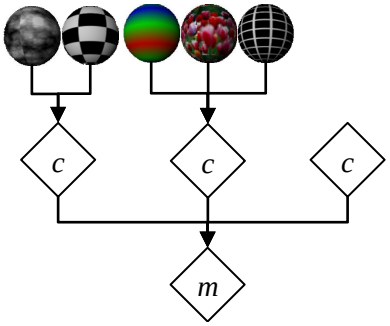
Material/Shader management

Recall initial discussion in the 'rendering module' section, that reviewed shader integration in real-time rendering applications; namely for the GRC. Shader based rendering applies the processing power of GPU hardware to accelerate graphics rendering for interactive graphics software.

Despite the GPU's central use in the GRC's rendering process, much of the process is still underpinned by the CPU. This is because the CPU is responsible for controlling the GPU, as well as supplying it with resources/data during the rendering process.

The GRC's 'CPU bound' component therefore, plays a central role in the delivery of 'adaptive' shaders and material composition. Recall for example, that adaptive shaders rely on a recompilation process, as well as the generation of PPD's, prior to shader recompilation. This process is carried out on the CPU which is followed by CPU based invocations to the Direct3D 10 API. As mentioned, the CPU component is specifically responsible for data management and supply of material/shader parameters during the rendering process. The following discussion provides a brief overview of shader/material integration in the GRC's CPU component.

In keeping with the tool chains interactive characteristic, the CPU component of the GRC's renderer emphasises external specification of data/parameters through the design/implementation of internal data structures. The 'GRC structure' section introduced the GRC's material module, and provided insight into its role in the software. A primary role of this module is to interface with network traffic streaming from 'connected' instances of the RTCE/Maya plug-in, during the content authoring process. In addition, the module stores material data which is 'uploaded' to the GPU during the rendering process.



This example shows how procedural functions can be independently combined across different channels, yielding highly customizable materials in the GRC

Figure 41 Illustrates the GRC's material structure

Figure 41 shows how materials of this tool chain represent arbitrary compositions of procedural functions in up to three separate channels. Thus, materials can potentially represent sophisticated data structures.

In an effort to promote tool chain interactivity, the material module has been designed to minimize network traffic during the specification of materials/procedural functions from the RTCE. Thus, rather than send entire material

structures from the RTCE, the RTCE only sends the parameters of individual procedural functions; thus, deferring procedural function association with material structures, to the GRC.

The motivation for this is that certain ‘controls’ of Maya’s user interface, which parameterize procedural functions, are capable of frequently invoking the network interface during artist interaction. An example of this being ‘sliders’, which parameterize certain aspects of ‘Perlin noise’ functions. Because data is interactively transmitted from Maya as the artist drags the slider interface control, it is preferable to minimize the amount of data that is transmitted by these events.

In response to this minimized ‘network data schema’, the material module is therefore, based on the storage and management of individual ‘procedural functions’, rather than entire material structures.

As artists modify a material composition which is presented in Maya/RTCE, only modified ‘procedural function’ parameters are transmitted to the GRC’s material module. Thus, transmission of the entire material structure is typically avoided.

In addition to making efficient use of network bandwidth, this approach also encourages shared procedural data in the GRC, as figure 42 shows. The actual material data structures, which represent the assembly of procedural functions, are stored in each ‘game object’ of the GRC. This approach to storing material compositions is appropriate, given that materials are unique to game object instances (as implied by the RTCE). Note however, that the data which underlies ‘procedural functions’, still remains in the material module. This data is therefore ‘externally referenced’ by ‘game objects’, as required by the object’s respective material composition.

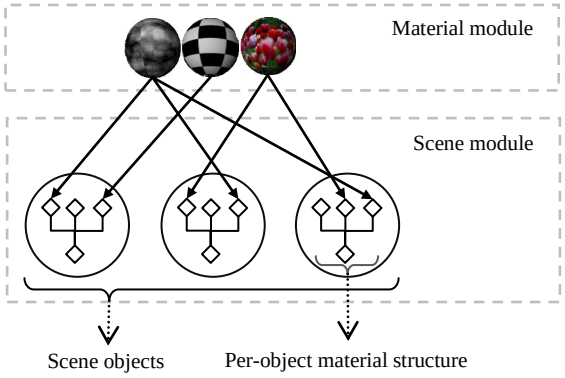


Figure 42 Illustrates integration of ‘material module’ and objects of the ‘scene module’

As an aside, each ‘procedural function’ stored in the material module, is associated with a unique identification. These identifiers are consistent between both the RTCE and GRC, enabling the tool chain to ‘target’ the transmission of data from the RTCE, to specific modules and data structures on the ‘remote’ GRC application.

As table 5 (page 60) shows, each ‘procedural function’ consists of arbitrary parameters. Recall from earlier discussion, the material module’s network interface ‘decodes’ and

extracts procedural parameters from a binary stream of network traffic. Given the 'arbitrary nature' of procedural parameters/data, achieving this required the 'global' use of data structure representations of procedural functions, throughout the tool chain.

Thus, the module 'casts' the incoming byte stream (network traffic), against a globally defined 'packet header'. Note that this header is also used by the RTCE when data is transmitted to the GRC.

This 'casting process' reveals the key information about the network traffic; namely the traffic's 'target identifier', as well as data concerning its representation as a procedural function. Within the material module, additional data is extracted from the stream. This data identifies the 'type' of procedural function that the network traffic/stream represents. The material module defines data structures that correspond to each of the procedural functions and thus, the extracted data is again, cast to the appropriate material module data structure, depending on the identified procedural type of the network traffic.

Note that if the data packet's identifier matches that of a procedural function already stored in the material module, the data of that procedural function is replaced. If no matching procedural function is found, a new procedural 'datum' is created.

Recall that when the parameters of a procedural function are 'replaced', the appearance of objects' that refer to the procedural data is immediately updated to reflect the change. This is due to the parameter driven nature of the shader based material system, as discussed.

The module is also responsible for uploading parameters of procedural functions to the GPU, prior to the rendering of respective geometry. As mentioned, procedural functions are expressed in the context of a material structure, as figure 41 (page 96) shows. Thus, when parameters for a procedural function are uploaded to the GPU, this contextual information must be provided so that procedural parameters are applied to the correct 'material channel'.

As discussed, material structures are stored in the GRC's game objects. Thus, 'procedural parameters' are uploaded to the GPU during the game objects 'drawing' process. Game objects therefore, explicitly invoke the parameter upload process, given that the game object internally stores the material structure (or context of the procedural function).

When the material module's parameter upload functionality is invoked, it internally links the procedural function 'type' with the specified material channel. This data context 'maps' the parameters to appropriate 'shader registers', which correspond to data used by procedural functions in the shader based material composition system.

Similar management strategies are also used for ‘shaders’ in the render module. As mentioned, shaders are ‘adaptive’ to the vertex/geometry formats of game objects. Thus, instances of shaders tend to represent shader code that is specific for a particular game object. Opportunity exists however, for the sharing of shaders in the GRC; this optimizes memory use in situations where multiple game objects use the same material structure and vertex-format.

To take advantage of these situations, the GRC employs similar identification strategies to shader objects, as applied to procedural functions of the material module.

The render module therefore, associates additional data with shaders; specifically the ‘vertex-format’ and material composition which the HLSL source code template was compiled against. Thus, prior to any shader recompilation event, the render module searches its internal ‘shader record’, for any shader instances that match the shader ‘recompilation’ request. If a match is found, the cached/found shader is reused. In addition to better memory usage, this approach also avoids the need for unnecessary shader compilation, which can momentarily stall the interactive authoring process. As mentioned, a reference counting scheme is maintained by the ‘renderer’ for each shader object. This prevents runtime errors that would otherwise arise, if a shared shader was ‘released’ by a referring object.

The demonstrations chapter provides a series images, depicting software components of the implemented tool chain, as well as its core functionality.

Instancing algorithm

This section describes the real time generative instancing (RTGI) algorithm. This algorithm provides ‘instance’ generation during the rendering process. Recall that a motivation for real-time instancing is to achieve better integration with a responsive tool chain that aims to improve productivity.

The instancing functionality being discussed is primarily implemented on the GPU and is heavily dependent on functionality that is relatively new to the architecture; namely ‘geometry shaders’ and ‘hardware instancing’.

Geometry shaders were introduced in 2006, with the advent of ‘shader model 4.0’, while hardware instancing was introduced via ‘shader model 3.0’ in 2004 (Efficiently Drawing Multiple Instances of Geometry, 2010). The main motivation for choosing the GPU architecture for development is the potential that it offers for high runtime performance, due to its inherently parallel nature (Owens, 2007). Subsequent discussion will explain how RTGI can be elegantly expressed on the GPU.

Structural overview

RTGI is a multi-stage rendering algorithm. Multi-stage (or multi-pass) algorithms are frequently used in real-time graphics. Examples of multi-staged rendering algorithms include special effects such as simulated depth-of-field and motion blur (figure 43), as well as reflections and refraction. Although distinct, these particular examples share a similar ‘flow of data’. That is, data which has been processed by the GPU in one stage circulates through the GPU in subsequent stages (or passes). For example, ‘per-pixel motion blur’ is achieved by two rendering passes; the first ‘renders’ an object’s velocity, with the second performing ‘conventional’ rendering of the object under the influence of previously rendered velocity data.

(Motostorm, 2007)

Figure 43 Illustrates motion blur as a post-processing effect in Motostorm

RTGI begins with an ‘instance generation’ stage which takes arbitrary geometry (‘manifold’) and computes transformations for other object ‘instances’ across the geometry’s surface. The second stage renders many ‘copies’ or ‘instances’ of other geometric objects, where each is transformed by the previously generated transformation matrices. This is achieved by the use

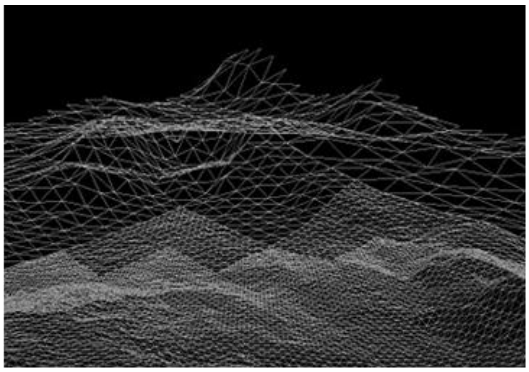
of ‘instancing’, which provides an efficient method for rendering large ‘populations’ of objects. Further details on hardware instancing will be provided in subsequent discussion.

Note that stages/passes in RTGI are used in a different way to the special effect examples. In those examples, passes operate in the ‘image space’ domain on a per-pixel basis to produce a composed rendering outcome. Such multi-pass processes are referred to as ‘post processing’ effects using data stored in pixel buffers. In RTGI however, stages are concerned with processing geometric data. In addition, RTGI does not encode generated data as is the case with the post processing effects. Passes of post processing effects tend to produce data that has internal meaning to the algorithm. RTGI differs, in that data emitted during the first stage is ‘generalized’, and can be reused at other points in the application’s rendering process.

Algorithm specific passes

Each pass of the RTGI algorithm uses different features of the GPU; the first of which is ‘geometry shading’. Geometry shaders introduce functionality to consumer graphics hardware that enables ‘geometric data’ to be generated during the rendering process (NVidia: GeForce 8800, 2010). This differs from conventional rendering processes, where all geometric data had to be provided prior to rendering.

Because geometry shaders provide a means of adding geometric complexity during rendering, they are often used to increase the detail of rendering. In the case of terrain for instance, high quality results can be efficiently achieved via geometry shaders. Geometry shaders enable ‘selective tessellation’ to be implemented entirely on the GPU, providing improved visual results with minimal overhead on the hosting processor. Selective tessellation means that primitives/triangles of the terrain that satisfy a tessellation criteria (such as falling within a certain vicinity of the viewing position) are subdivided, yielding a more convincing visual appearance.



(QuadTerrain LOD: Finished, 2008)

Figure 44 Illustrates adaptive terrain, where tessellation is a function of view position

The overall result is tessellation that only occurs in portions of the terrain where necessary, avoiding ‘redundant’ tessellation at distant or non-visible areas of the scene. Because the tessellation is a function of the viewer’s position, it reacts to viewer movement and thus, as the viewer moves through the scene, portions of the terrain increase and decrease

in detail according to the ‘tessellation criteria’. When selective tessellation is implemented via geometry shaders, it offers a good balance between efficiency and visual quality without imposition of overhead on the host processor.

(Hagedoorn, 2007)
Figure 45 Sophisticated fluid flow achieved by applying geometry shading to a particle system

Another common application of geometry shaders is to create or enhance ‘particle systems’. The particle system is a widely used abstraction that underpins many special effects in games such as rain/precipitation, fire, explosions and fluid. To remain practical for real-time graphics, particle systems simulate large volumes of particles by substitution of a small number of visually complex particles. This trade off (between particle quantity and quality) tends to require careful optimization by the developer. With the advent of geometry shaders however, a new ‘dynamic’ quality is introduced to particle systems that allows massive volumes of particles to be added/removed from a system while maintaining CPU independence. This offsets the mentioned trade off quite significantly, as processing additional particles is entirely offloaded onto the GPU.

For the instancing algorithm however, geometry shaders are applied in a different way for an entirely different purpose. Rather than use geometry shaders to enhance the appearance of rendered geometry/phenomena directly, the algorithm takes advantage of data generation within the geometry shader, coupling it with ‘output-streaming’ functionality (Stream-Output Stage, 2010). Actual scene enhancement takes place after the geometry shading stage, and uses the data it generated by the geometry shader for efficient ‘scene population’.

The aspect of data generation in geometry shader’s is specifically used to generate per-instance information for the subsequent ‘scene population’ phase. To achieve this, the rendering pipeline needs to be configured to enable ‘output-streaming’. By default, geometry shaders propagate any generated data/geometry forward through the rendering process, towards the final rasterization stage. This happens in the two examples previously discussed, where generated data is passed forward for immediate onscreen rendering. For this algorithm however, generated data needs to be channelled ‘back to the system’ in order for it to be reinterpreted/reused as instancing data in the subsequent rendering process.

Streaming

As mentioned, RTGI relies on data that is generated by the geometry or 'instancing' shader, being accessible to the host system. The streaming capabilities enable data buffers that are accessible to the host system/CPU, to be filled by the output of geometry shaders. This requires reconfiguration of the rendering pipeline (Stream-Output Stage, 2010).

As an aside, the pipeline configuration used for RTGI demonstrates the flexibility of the Direct3D 10 graphics API. Rasterization is not required at the instancing stage of RTGI and thus, 'pixel buffers' (or 'render-targets') are not bound to the device prior to the first pass (Samyn, 2009). Instead, the pipeline is configured so that render-targets are substituted with generic data buffers, which are populated with instance data during the streaming process. Although the geometry shader still 'renders' the manifold geometry, no visual outcome is produced. Instead, the 'rendering' process fills the instance buffer in a similar way to the rasterization of a render target by a pixel shader.

In comparison to other shader types such as vertex shaders, geometry shaders allow more control over the population of output buffers; particularly in terms of the amount of data issued per shader call. This is possible because geometry shaders integrate a 'list-like' data structure through which data is output (Stream-Output Object, 2010). This, in conjunction with hardware based flow control/branching (introduced in 'shader model 4.0'), provides explicit control over the amount of data that a geometry shader can output (Blythe, The Direct3D 10 System, 2006). This is the mechanism for variable output data from geometry shaders**.**

Other shaders such as vertex shaders impose that a static quantity of data be outputted during execution. When a geometry shader operates on a triangle however, arbitrary amounts of data can be independently emitted. Table 14 contains code excerpts written in HLSL, that illustrate these shader characteristics.

Vertex Shader	Geometry Shader

void vertex_shader (
  ...
  out float4
  out_position:SV_Position,
  ...
){
 	
  /*
  shader output 'fixed' registers
  that have been explicity declared
  */
  out_position = position_data;
  ...
}
	void geometry_shader (
	  ...
	  inout PointStream<data_struct>
	  shader_output
	){
	
	  /*
	  shader output can be 'appended'
	  independantly
	  */
	  ...
	  shader_output.Append(data);
	  ...
	}

Table 14 Comparison of shader types, showing how geometry shaders facilitate variable data output, unlike other shader types

Arbitrary data streaming, based on dynamic branching in geometry shaders, lies at the heart of RTGI. In RTGI, the evaluation of a procedural function is used to decide where objects are instantiated across the manifold. This function will be referred to as the 'Mask' procedural. Instances are emitted or suppressed at discrete points across the manifold, depending on the Boolean reduction of the mask procedural at that point. Flow control is therefore central to this aspect of the shader. Table 15 contains the partial definition of a geometry shader which illustrates the use of flow control for data output in the context of HLSL.

Geometry Shader

```
[maxvertexcount(128)]
void inst_geometry_shader(
  ...
  inout PointStream<inst_type> inst_shader_output
) {
  ...
  inst_type data;
  ...
  if(mask_result) {

    /*
    Note that the stream structure can only be appended to and not read from, despite
    the 'inout' semantic in the structure's declaration
    */
    inst_shader_output.Append(data);
  }
  ...
}
```

Table 15 Shows how variable output is achieved via flow control and the output 'structure' of geometry shaders in HLSL.

As this definition shows, the '`PointStream`' mechanism is provided to capture data generated by the geometry shader. The '`PointStream`' is one of three 'list' representations available in HLSL geometry shaders (Stream-Output Object, 2010). This list mechanism enables the output of arbitrary amounts of data via the '`Append`' intrinsic.

HLSL provides three 'stream types' that are designed to simplify the output of geometric data from the shader; these being the '`PointStream`', '`LineStream`' and '`TriangleStream`' (Stream-Output Object, 2010). Interestingly, input and output primitive types within HLSL geometry shaders are independent. This characteristic is particularly useful for RTGI given that output instance data doesn't correspond to input data.

As an aside, note the '`maxvertexcount(128)`' decorator, in the shader declaration of the previous excerpt. This instructs the HLSL compiler as to how many data output registers (from a maximum of 1024), should be allocated for the shader (Geometry-Shader Object, 2010).

In terms of instance generation via the geometry shader, any streaming 'type' can be used to output data, given that the data is not immediately/directly used for rendering. This assumes however, that data accumulated from all calls to the geometry shader are correctly 'aligned' via the selected stream type. Because the RTGI algorithms subsequent rendering stage

utilizes ‘hardware instancing’, alignment of instance data is crucial. This is because hardware instancing assumes consistency in the layout of instance data, in order to efficiently ‘batch’ rendering of multiple objects. This will be discussed in more detail in the following section.

Instance data provides information that allows similar objects to be uniquely parameterized as a basis for inter-object/instance variety. Although the instance parameters are defined according to the needs of an application, they usually include an affine transformation in order for instances to maintain unique position, orientation and scale (Weisstein, Affine Transformation., 2010). For these situations, 12 or 16 floating points are required to represent the transformation of instance.

For the RTGI algorithm however, the specification of 12 floating points can be avoided due to the algorithms design and implementation. Thus, only 8 floating points are required to parameterize each instance due to assumptions; namely uniform per-instance scaling and constrained rotation (about a single axis), being applied to instances of this algorithm. The motivation for minimizing per-instance data is to make efficient use of capacity and bandwidth of the graphics hardware. Table 16 shows the per-instance parameter structure that RTGI generates.

Purpose	Components	Data type	Data size	Packet
Instance position	*x,y,z*	‘`float4`’	16 bytes	Per instance description
Instance scale	*w*			
Manifold normal	*x,y,z*	‘`float4`’	16 bytes	
Instance rotation (yaw)	*w*			

Table 16 Per-instance parameter structure emitted from instance shader

Instancing

As mentioned, ‘instancing’ provides a mechanism that efficiently renders the same geometry multiple times. This efficiency is achieved by minimizing the number of invocations to the graphics device during the rendering process, than would traditionally be required. Introducing more geometry into the final scene allows an increased level of realism in the graphics of games to be achieved. Note that for many situations, quality gains can be made by redrawing instances of the same geometry, where each instance is subject to unique ‘per-instance parameters’.

Traditionally, rendering multiple geometric instances (prior to hardware instancing) was avoided, due to the negative implications that this had on runtime performance. To render geometry multiple times required copies of the geometric data be sent to the GPU from the

CPU. The cumulative effect of this was a significant load on inter-processor bandwidth, consequently hindering runtime performance.

To take advantage of GPU instancing, two data sources are required; an 'instance buffer' (containing all per-instance parameters) and the base geometry being instanced. As indicated, an instance buffer is a sequentially organized buffer containing the per-instance data that describes/parameterizes each object instance. The amount and type(s) of data in an instance 'packet' is arbitrary, allowing instancing to cater to many applications.

GPU instancing typically begins by binding and uploading the instance and geometry buffers to the rendering device. When this data has uploaded from the CPU to the GPU, mass rendering of instances can be efficiently performed. This is a side effect of all instancing data being locally available to the GPU in its own memory. Thus, fast iteration of the instance buffer by the GPU, is possible. For each iteration, the parameters for a single instance are applied to the base geometry that is also locally cached in GPU memory. The GPU then immediately renders that parameterized instance geometry. The superior performance gained by this approach is based on the instancing process occurring solely on the graphics hardware, without the need for CPU/external intervention.

Due to the ambitions for real-time performance in this research's interactive tool chain, efficient rendering techniques such as this are significant. Furthermore, because hardware instancing is predominantly used extensively in games, the technique is also relevant to the tool chain's application domain. Hence, the motivation for applying GPU instancing within RTGI is obvious.

Motivations & considerations

The widespread application of hardware instancing in games, may bring into question the need for procedurally based instancing in a games engine. Procedurally based instancing however, retains a distinct and powerful characteristic; that it performs the entire task of instancing 'on-the-fly'.

Given that this approach generates instancing data in real-time, this makes it inherently different from typical methods that are based on pre-baked static data. The consequence of this 'standard' approach is that the parameters for each instance must be explicitly stored, thus increasing the game's overall size. Furthermore, the standard approach doesn't advocate real-time responsiveness (to parameter change) in the same way that procedurally based instancing does, given the static nature of underlying data. The implication of 'static data' is that the production processes that underlie standard instancing are likely to be tedious and

slow due to the process of ‘baking’ static instancing data. The ‘baking’ process usually involves manual specification of instances by an artist, or the generation of stored instance data by a ‘fixed algorithm’.

Procedural instancing improves issues related to instance authoring and storage due to its generative, ‘on-the-fly’ characteristic. Given the performance capabilities of current GPU’s, coupled with the architectures solid performance increase projections, GPU based real-time instance generation serves as an increasingly feasible strategy to reduce space complexity and artist/authoring overhead (Tech ARP, 2010).

Despite the benefits of this generative approach, some memory considerations must still be taken into account. Furthermore, streaming functionality that offers the levels of flexibility previously discussed must also be natively available. Due to the way that stream-out functionality is currently implemented in mainstream graphics API’s, the algorithm requires that memory be available upfront and in full during rendering, to store streamed data.

In principle however, instancing data generated by the geometry shader could be passed directly to an ‘instancing-like’ rendering process for immediate rendering. Due to the capabilities of current graphics APIs, namely Direct3D 10, the use of geometry shaders for instance generation, requires streaming functionality to be available in order for generated data to be channelled to a temporary storage buffer.

During the process of data streaming from a geometry shader, current API implementations require the stream destination buffer be pre-allocated and of a fixed size (Stream-Output Stage, 2010) (Lichtenbelt, Brown, & Werness, 2008). In OpenGL and Direct3D 10 however, it is possible to resize the stream-out buffers during runtime, provided that this takes place outside of the streaming process. This capability is taken advantage of in RTGI to accommodate varying numbers of instanced objects. A number of strategies are available for calculating the size of a dynamic buffer used to store stream-out data.

Although crude, the first strategy can be applied to data streaming in both OpenGL as well as Direct3D 10. Furthermore, it is conceptually simple. The strategy is based on basic information about the geometry being rendered; namely the number of triangles it consists of. To render geometry, it is necessary to know the number of primitives/triangles that it consists of. Graphics API's provide functionality or documentation indicating the maximum amount of data that a primitive can emit during execution of a geometry shader (Geometry-Shader Object, 2010). Recall that geometry shaders implemented in HLSL require that data output size be explicitly declared (Geometry-Shader Object, 2010). Thus, by allocating a buffer for

stream-out data by the following equation, the application can guarantee enough buffer space for every possible streaming scenario.

$$\text{stream buffer size} = \text{object primitive count} \times \text{maximum data output per geometry shader}$$

Given that geometry shaders (such as the instancing generation shader) tend to emit variable amounts of data, the average stream buffer utilization for this approach is poor. Figure 46 illustrates this.

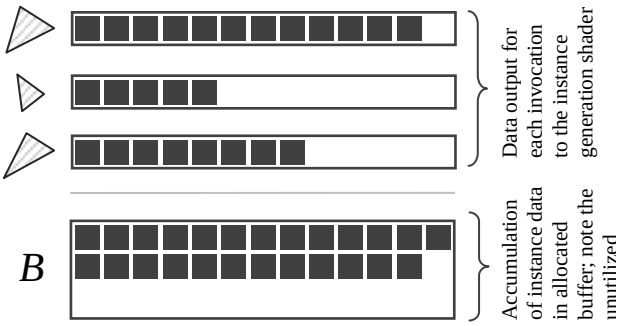


Figure 46 Illustrates poor utilization of stream via first strategy

An alternative approach is to allocate stream-out buffers based on the exact amount of data emitted during the geometry shading process.

Achieving this via Direct3D 10 requires use of the 'Asynchronous Query' interface which enables statistics to be captured at certain stages throughout the rendering process (ID3D10Asynchronous Interface, 2010). Although this interface can determine the amount of data emitted (i.e. the number of instances generated), this information is only available after the shading/streaming process has taken place. Thus, the stream-out buffer can only be allocated based on the volume of streamed data from the proceeding frame/application loop. As a consequence, the potential for inter-frame glitches exists. These would come as a result of stream-out buffers being resized in response to the dynamic number of instances generated.

In stating this, prediction based strategies could be implemented to avoid these glitches by minimizing the frequency of reallocating the destination buffers for output streaming.

For situations where the destination buffer is too small, 'excess' data is simply discarded by the hardware/API. In addition to 'inter-allocation glitches', these situations may also arise if generated data exceeds the memory resources provided by the underlying system. Thus, the artist must remain conscious of platform limitations when working with this implementation of the RTGI algorithm.

Obviously, the visual side effect of discarded data in the context of RTGI is that respective instances are not rendered in the algorithm's final stage.

Implementation of first pass

The following section describes the essence of the RTGI algorithm, which is predominantly based in the instance generation geometry shader. Thus, it assumes that a software environment is present which correctly initializes the instancing geometry shader with geometric, procedural and other necessary runtime data.

Because this work is of an experimental nature, a number of different implementations for this algorithm were produced. These implementations were essentially iterations towards a final algorithm that is presented as the procedural instancing solution.

Although unified shading languages such as HLSL provide rich and flexible instruction sets, there is no native functionality that directly assists instance generation across a geometric manifold.

Recall that the RGTI geometry shader operates on triangle primitives. Thus, the instancing shader's primary function is to generate instance data that corresponds to the surface orientation and position of arbitrary triangles being processed. Furthermore, generated instance data must reflect the triangle's topology in addition to procedural functions that influence generated data. Figure 47 provides a schematic overview of the process.

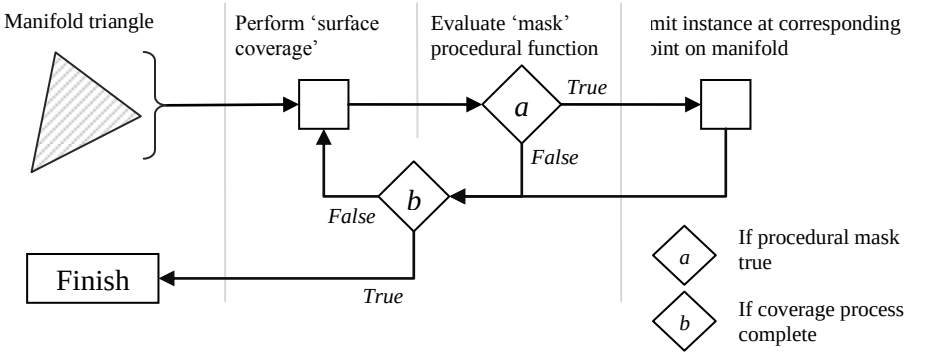


Figure 47 Schematic diagram illustrates general flow control in the instancing shader

Thus, when a triangle is being processed, the shader must evaluate its surface to calculate where instances will be placed. This process is encapsulated by the 'surface coverage' stage in figure 47.

Defining a solution for this stage was a non trivial task, and constituted much of the effort in developing the algorithm. Interestingly, the 'coverage stage' has similar functional characteristics to rasterization, in the sense that 'plotting' takes place across the triangle similarly to pixel 'plotting' across a triangle during rasterization. Rasterization is a complex

process that often merges sophisticated mathematics with heavily optimized scan line algorithms (Rasterisation, 2010).

The coverage process required for this algorithm is also inherently complex, due in part to the arbitrary nature of input geometry. Furthermore, the coverage process must incorporate custom instance generation functionality that is invoked during each ‘sample’ of the process; this adding another layer of detail to the algorithm. As a foreword, it is important that the coverage process yields ‘stable’ output. That is, instance generation should not be stochastic between frames.

A ‘marching’ mechanism underpins each of the surface coverage solutions that were explored. Marching begins with an ‘anchor point’ on the triangle’s surface being selected, from which ‘stepping’ across the triangles surface incrementally takes place (see figure 48). Marching therefore, uses a loop to drive the sample position across the triangle. The sampler loop also allows the number of samples for a triangle to vary; this being a function of the triangle’s surface area.

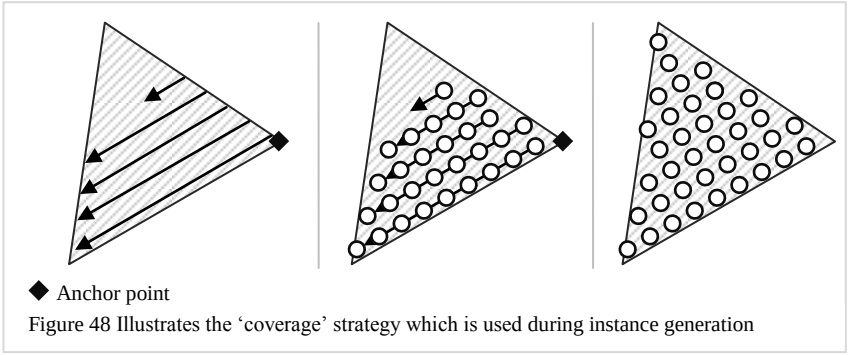


Figure 48 Illustrates the ‘coverage’ strategy which is used during instance generation

First revision

The first approach to this coverage problem began with the subdivision of an incoming triangle into two simple right-angled sub triangles. By performing this dissection, the resulting right-angled triangles allowed assumptions to be made during the subsequent processing of the sub-primitives. Using right angled triangles helps to keep the ‘march based sampling’ approach as simple as possible.

The ‘dissection’ phase begins by determining the longest edge of the incoming triangle. Once this edge has been found, the process finds the adjacent vertex (that is not on the triangle’s longest side). The process uses this adjacent vertex as the ‘splitting point’ for dividing the initial triangle into two right-angled sub triangles, as shown in figure 49.

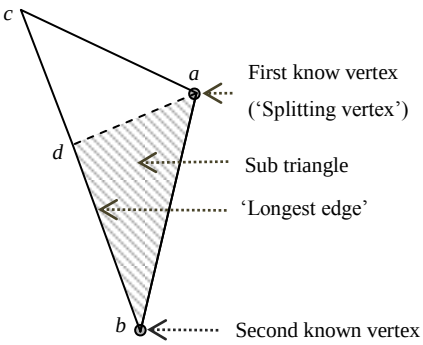


Figure 49 Illustrates what information is known about a triangle at the start of the instance generation process

Because the incoming triangle is arbitrary, it is essential that the 'selected edge' be the longest side of the triangle. Thus, the 'splitting vertex' cannot be randomly selected from vertices of an arbitrary triangle, as doing so could yield non-right angled sub triangles following the dissection process.

At this stage in the process, two of the three vertices for each sub triangle are known. Figure 49 illustrates this, with one of the known vertices being a point on the longest edge, and the other being the 'splitting vertex' that is adjacent to the longest edge.

To find the third vertex which is shared by both sub triangles, simple 2D projection is applied. The following equations show how 2D projection is used to obtain '*d*' in figure 49 (note the shared use of variables between the image and formulae).

$$\bar{v} = c - b$$

$$\bar{u} = a - b$$

$$p = \left(\left(\frac{\bar{v}}{\|\bar{v}\|} \right) \bullet u \right)$$

$$d = p \left(\frac{\bar{v}}{\|\bar{v}\|} \right)$$

Table 17 Projection equation used by the RTGI algorithm's 'sub triangle' extraction calculation

Projection is used to determine the line that passes through the adjacent 'splitting vertex' and is perpendicular to the longest edge. The projection process itself, produces a scalar magnitude (p) that represents the offset of the intersection along the longest edge. This intersection point defines the final vertex of the vertex tuples that are the two right-angled sub triangles.

Once these sub triangles are found, the coverage process takes advantage of the right-angled characteristic of both sub triangles. The orthogonal sides of each are used to define a 'virtual rectangle' that encloses the sub triangle, as shown in figure 50. This rectangle represents the geometric region that will be iterated by nested loops during the sampling process.

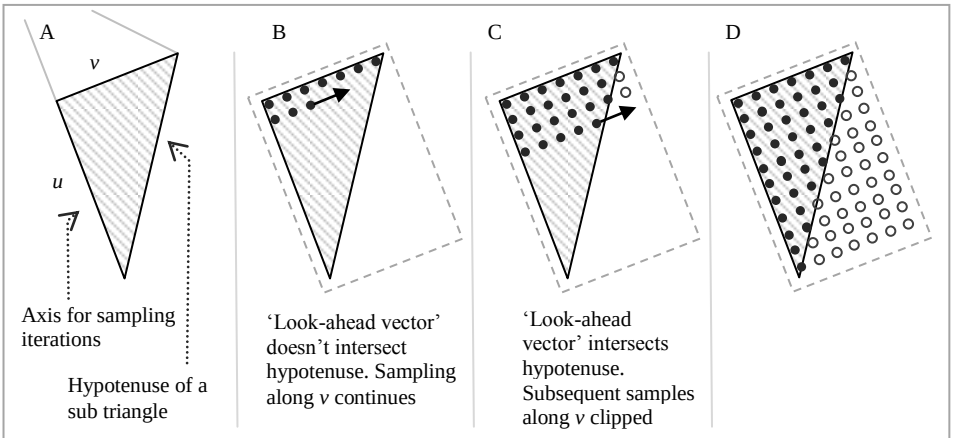


Figure 50 Illustration of sample clipping during iterative sampling of sub triangle

The outer loop moves along one edge of the virtual rectangle. The nested loop iterates in the orthogonal direction, parallel to the other edge of the rectangle. Thus, full coverage of samples across the space that encloses the sub triangle is achieved.

Iteration based on the 'virtual rectangle' alone will still produce incorrect results however. This is because approximately half of all samples will appear beyond the hypotenuse edge of the sub triangle (see image D in figure 50).

Therefore, a strategy involving line intersection is used to ensure samples are contained within sub triangles. Each nested iteration computes a position that represents the 'sample position'. By also computing the sample position of the next iteration, a 'delta vector' can be deduced. This represents the change vector between the current and subsequent sample positions. An intersection test takes place between this delta vector and the sub triangle's hypotenuse. This intersection determines if the coverage/sampling process is bordering the bounds of the sub triangle. If the hypotenuse is encountered during iteration, the nested loop is terminated. This prevents samples from exceeding the hypotenuse of the sub triangle.

It should be noted that the hypotenuse is the only edge that requires intersection tests during this phase. This is because the (nested) iterations are implicitly constrained by the sides of 'virtual rectangle', which are directly derived from the orthogonal sides of each sub triangle. This makes intersection tests with the adjacent/tangent edges of each sub triangle unnecessary, as images (B) and (C) of figure 50 illustrate.

Marching across the geometry shader's incoming triangle is achieved by applying this coverage process to both sub triangles. This marching scheme provides a means for extracting uniformly distributed positions from the surface of the triangle. For this instancing

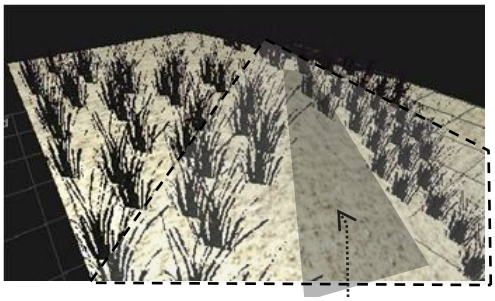


Figure 51 Illustrates visual artefacts that resulted in the first implementation of the instance generation process

algorithm, these samples represent the positions of possible instances that are constrained to the surface of the underlying manifold geometry.

Although this concept appears to be theoretically sound, practical testing revealed a series of weaknesses with this approach. Perhaps the most severe of these were notable 'alignment artefacts' which produced inconsistencies in the distribution of instances between sub triangles.

In other test cases, there were artefacts such as visual overlapping of instances or irregular spacing between instances, along the 'seam' of both sub triangles. Figure 51 shows a severe case of 'alignment inconsistencies' of instances (grass) along sub triangle seams in arbitrary manifold geometry (terrain).

Because samples were uniformly spaced elsewhere on the sub triangles, this made inconsistencies along the sub triangle seams particularly apparent. Following extensive testing with more complex data, another short coming was revealed; inconsistencies between the distribution of instances on the manifold, and the manifold's texture coordinates. Thus, texture coordinates in the manifold had no influence on the distribution of instances.

It is essential that the final marching scheme accounts for texture coordinate data. As discussed, texture coordinates within geometry provide a flexible mechanism for mapping two dimensional data, namely texture images, onto arbitrary geometric surfaces. Because the concept of instancing over a geometric manifold is analogous to 'texture mapping', the integration of texture coordinates into the sampling process is appropriate. Integrating texture coordinate data into the sampling process enables artists to harness instancing in a similar way to texture mapping. The advantage is that artists are able to reapply existing texture mapping skills, enabling them to quickly take advantage of the instancing concept.

Furthermore, a correlation between the marching process and manifold texture coordinates is also relevant to the integration of PM's in the instance generation process. In order to achieve consistent integration of PM's, texture coordinates must be present in order to parameterize the PM's during evaluation. Although PM's can be evaluated in this context without texture coordinates, this would impose limitations on the artist, therefore diminishing the capabilities of this concept.

Second revision

The first approach introduced the core elements and necessary considerations for the instance sampling procedure. In the second approach, the sample triangle 'marching' idea remains. Alterations have been made however, that eliminate the 'distribution uniformity' issues that affected the first approach.

The previous method subdivided each incoming triangle into two right angled triangles. By extracting right-angled elements from the initial triangle, a simple nested loop process could easily 'march' across each primitive.

In practice, this yielded instance distributions which were noticeably inconsistent around the sub triangle seam.

To address this issue, a method that avoided triangular subdivision was explored. The new method differs to the previous in that it immediately computes a 'virtual rectangle' that encloses the incoming triangle (figure 52). By computing this 'virtual rectangle', the marching 'boundaries' for the subsequent sampling are therefore established.

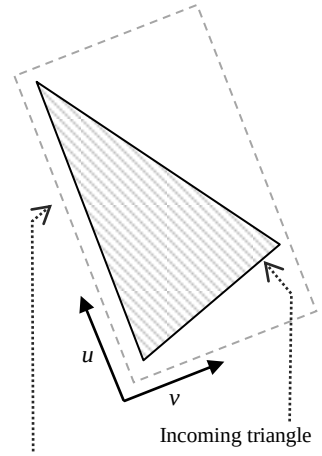


Figure 52 Illustrates the 'virtual rectangle' which encloses triangles processed by the instance generation phase

As previously illustrated, another side effect of computing the 'virtual rectangle' is that the axes by which triangle marching is based, are implicitly supplied (see '*u*' and '*v*' in figure 52). This therefore, eliminates the need for subdivision of the incoming triangle and thus, the noted sampling inconsistencies are avoided.

Computing the virtual rectangle begins by calculating two orthogonal vectors from the incoming triangle; known as the bitangent and tangent vectors (Weisstein, Binormal Vector, 2010). These two vectors form as basis vectors of a Euclidean space referred to as 'Tangent Space' (Tangent space, 2010). Tangent space represents the orientation of the surface at a given point (Tangent space, 2010).

For 3D geometry, tangent space is described by three basis vectors. Two of the basis vectors lie in the manifold surface at a specific point. The third basis vector is equivalent to the surface normal at that manifold point.

The concept of tangent space is not novel to game development, having been used in industry for many years. Perhaps the most notable application of tangent space is its use in achieving surface enhancing effects for real-time rendering. Examples of such effects include 'normal mapping' and 'parallax mapping' which simulate additional surface detail on simple

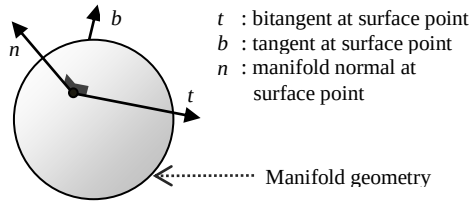


Figure 53 Illustration of tangent space on arbitrary manifold. Note the orthogonal nature of this space.

geometry via the use of tangent space (Normal Map, 2010). Figure 53 illustrates the basis vectors of tangent space, with respect to a manifold surface.

Thus, bitangent and tangent vectors define the vectors underlying the sides of the triangle's 'virtual rectangle' (see figure 52). Although tangent space is orthogonal, orientation of the space about the normal vector is essentially arbitrary. That is, the bitangent and tangent vectors can arbitrarily rotate about the normal vector, while remaining orthogonal.

In the context of 3D graphics, it is often necessary to maintain consistency in tangent spaces throughout the manifold. Thus, bitangent and tangent vectors are typically aligned to texture coordinates in the 3D manifold geometry.

A benefit of tangent space based on a manifold's texture coordinates, is that it can be computed algorithmically (Mittring, 2006). For most applications in real-time rendering, tangent space basis vectors are typically computed and stored for each vertex of a geometric object. Computing this space information has usually been an 'offline' process that occurs prior to runtime. This is because the algorithm requires vertex adjacency information (i.e. information about neighbouring vertices within the geometry), information that has typically been unavailable at runtime.

With the advent of Direct3D 10 and its revised shader specifications, this adjacency information can now be made available at the geometry shader stage in the rendering process (Shader Stages, 2010). Tangent space can therefore, be determined directly in the instance generation geometry shader, for use in the triangle marching process. For more information about computing tangent space within this context, see Appendix C.

As noted, the orientation of tangent space is influenced by the texture coordinates of the manifold geometry. In this sense, the sampling process satisfies the requirement to integrate texture coordinate data into the sampling process. The objective is to achieve instancing distributions that can be manipulated by artists during the modelling process, via adjustment to the texture mapping/coordinates.

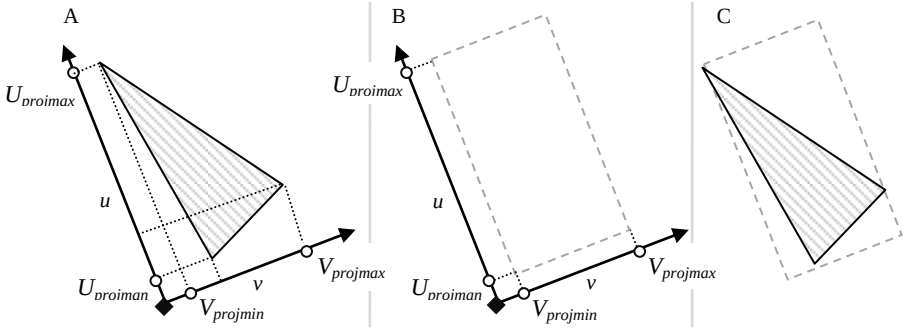


Figure 54 Illustrates the process of 'virtual rectangle' computation via tangent space and orthogonal projection

The process continues by calculating the 'extent' of the virtual rectangle about the subject triangle. A series of 'point-line' projections are computed between the triangles vertices, and the 'side vectors' of the enclosing rectangle. As image A of figure 54 shows, each vertex of the triangle is projected orthogonally onto each 'side vector' (bitangent, tangent vectors). Each projection produces a scalar value which represents the offset of the project point from the side's origin. Through this, maximum and minimum projection bounds can be determined trivially. Via simple vector arithmetic, the corners of the enclosing rectangle can be computed by the following equations:

$$C_a = vV_{projmin} + uU_{projmin}$$
$$C_b = vV_{projmin} + uU_{projmax}$$
$$C_c = vV_{projmax} + uU_{projmin}$$
$$C_d = vV_{projmax} + uU_{projmax}$$

Where:

C_a, C_b, C_c, C_d	corners of the enclosing rectangle
u, v	tangent/bitangent vectors
$U, V_{projmin,max}$	maximum and minimum projections onto U, V from each vertex

The process uses the 'virtual rectangle' in a similar way to the previous surface coverage/marching approach. That is, the enclosing rectangle is a geometric representation of the area that is iterated during the marching phase. Nested iterations are used to march the sampling position over this rectangular space, incrementing the sample position by scaled u and v vectors.

The scale of increments along the u and v vectors are influenced by the texture coordinates of the manifold geometry. If for example, the texture coordinates are 'small', the sampling

reflects this by relatively large increments, giving a smaller number of samples in the rectangle. ‘Larger’ texture coordinates yield smaller increments and thus, ‘condensed’ samples/marching. Refer to figure 55 for a visual illustration of this concept.

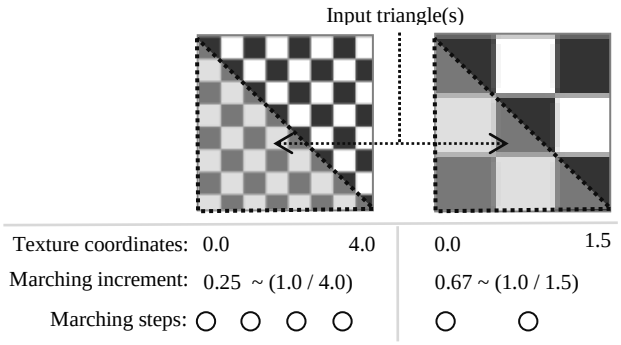


Figure 55 Illustrates how texture coordinates ranges influence marching ‘density’

As with the previous approach, iterating the enclosing rectangle requires ‘clipping’ of instances that fall outside of the triangle. If no clipping were to take place, the sampling procedure would produce instances both inside and outside of the triangle. To resolve this, a ray casting variant of the ‘Point in Polygon’ algorithm is used (Haines, 1994). This algorithm determines if a point is contained within a polygon by counting the number of intersections between edges of the polygon and an overlapping ray-segment. An odd number of intersections indicate that the point is contained within the arbitrary polygon and an even number indicates no containment (Haines, 1994).

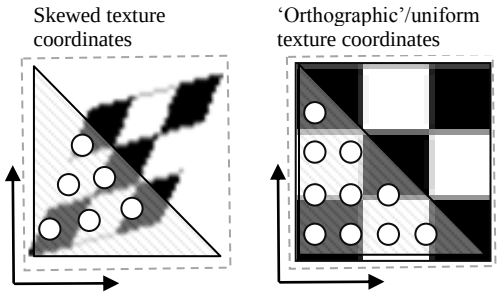
This elegant algorithm lies at the heart of the stepping process’s ‘clipping’ solution. The process applies the ‘point in polygon’ algorithm for each sample of the marching process. The algorithm’s ‘ray-segment’ is substituted with any vector that defines an edge of the ‘enclosing rectangle’. For each sample step that is iterated, the ray-segment’s origin is offset to the sample’s position in the enclosing rectangle. Intersection tests between the offset segment and all edges of the enclosed triangle are then performed. If an even number of intersections occurs, the sample is determined to be outside the triangle and is immediately clipped. If an odd number occurs, the sample is used to generate instance data.

This approach is a significant improvement of the predecessor. Not only does it integrate texture coordinates into the marching process, it also addresses the ‘seam inconsistencies’ that occurred in the first approach.

Despite its improvements however, a series of issues were discovered during subsequent testing. Again, these were related to the process’s interpretation of texture coordinates in the manifold geometry. Unpredictable results were produced under circumstances where skewed or tapered texture coordinates were assigned to the manifold geometry. This is due to an assumption that texture coordinates ‘match’ the topology of the ‘virtual rectangle’ (i.e. uniform, right angled). As **Error! Reference source not found.** shows, the distribution of

amples across a triangle containing 'irregular' or skewed texture coordinates, is inconsistent with the triangle's topology.

This assumption cannot be made, based on the requirements of artists. Many techniques involve the clever manipulation of texture coordinates, which can optimize both production and runtime efficiency of game/game content. Thus, to deliver a powerful and flexible instancing solution that is based on geometric as well as texture coordinate data, a solution that correctly handles these situations is required.



This instance generation approach is dependent on uniform/ 'orthographic' texture coordinates in the manifold (right), to yield a correct instance distribution

Figure 56 Illustrates limitations of instance generation in the second instancing approach

Final implementation

The final revision to this sampling process amends all of the issues discovered in previous attempts and delivers a robust method of mapping instance samples across arbitrary triangles.

As an aside, it is encouraging to see that many elements from initial revisions of the sampling process are still present in this final implementation. This indicates a good initial understanding of the problem and the direction that subsequent exploration would need to pursue.

The developed algorithm is fundamentally based around the manifold's texture coordinates, unlike the previous approaches that primarily base sample distribution on the manifold's geometry. This approach serves as a robust and generalized solution that yields correct sample distribution for all valid manifold configurations. This general robustness is arguably due to the solution's basis in abstract mathematics, namely linear algebra, as the following discussion shows.

The process utilizes linear transformations to distribute instance samples that correspond to the incoming manifold's geometry and texture coordinates. Again, the 'virtual rectangle' concept is used in this process. Rather than apply it to the geometric coordinates in 'object space' as in previous revisions, the enclosing rectangle is applied to the manifold's 'texture space' coordinates, as shown in figure 57. The implication of this is that iterative 'sample marching' now takes place with respect to texture space, instead of 'object space'.

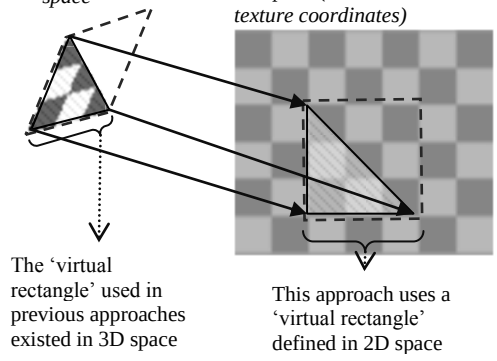


Figure 57 Shows how a texture coordinate based 'virtual rectangle' can be used for instance generation

Texture coordinate based samples, that are deemed to be 'valid', are transformed to object space via a series of intermediate transformations. The advantage of 'sample marching' in texture coordinate space is that any skewing, orientation and/or translation of texture coordinates across the 3D geometry, is implicitly represented in a normalized plane; that being 2D texture space. Consider the triangle in image (A) of figure 57. For an arbitrary geometric triangle that consists of texture coordinates, a linear transformation exists to express its corresponding form in 2D texture coordinate space (i.e. image B of figure 57). By inverting this linear relationship, the sampling/marching process can reflect arbitrary transformation and arrangement of texture coordinates when distributing samples across the triangle's geometry. What's more, this mechanism is robust enough to handle arbitrary texture coordinate 'ranges' (as illustrated in figure 55, page 118), which influences sparse/condensed marching behaviour across a triangle.

The algorithm begins by establishing the minimum and maximum bounds of the texture coordinates assigned to the manifold geometry. As mentioned, this produces an enclosing rectangle in 2D texture space. To achieve sample coverage across the triangle, nested iterations increment the sample position through each dimension of this sample space. Because iteration occurs in texture space, tangent and bitangent vectors are not required to direct the sampling position this stage in the process.

The 'point-in-polygon' algorithm is still used to determine if a sample in texture space is contained within the triangle that is defined by the texture coordinates of the manifold geometry. Samples that are outside of this triangle are immediately clipped, ensuring that no instance is generated.

For samples that are contained in the 'texture space triangle', information about the geometric context, as well as the texture coordinate that represents the sample position, are used for subsequent transformations. The goal here is to transform a two dimensional texture coordinate (i.e. the current contained coordinate) into a corresponding three dimensional position on the manifold geometry. Thus, a mapping is required to a point on the object's surface from a coordinate in the object's underlying texture coordinates. The mapping/transformation will be illustrated by breaking it into two logical stages.

First, consider the concept of texture space. In real-time graphics, the visual representation of coordinates in this space is that they ‘wrap’ around the associated 3D manifold. Thus, for each coordinate in this space, there exists a corresponding three dimensional point (or points) on the manifold surface. It is possible to acquire this information by introducing an intermediate transformation that maps texture coordinates to coordinates in tangent space.

As previously discussed, tangent space represents the ‘frame of orientation’ for a given point on the manifold geometry. Recall that, of the three axes of tangent space, one is already known to be the surface normal. Given that this axis is already known, the intermediate mapping only needs to orientate two basis vectors in tangent space; the tangent and bitangent. The orientation for both of these vectors is obtained from texture coordinate vectors derived from the respective triangle.

Consider a Euclidean matrix with two basis vectors defined by two sides of the triangle’s representation in texture coordinates/space. Using this matrix, coordinates in tangent space can be transformed into texture space. As an aside, this matrix underpins the process’s ability to correctly represent arbitrarily skewed and orientated texture coordinates in the final sampling result. By transforming coordinates from tangent space to texture space, based on the axes defined by the triangle’s texture coordinates, there exists no dependence on right-angled compositions or strict orientations of texture coordinates. Rather, this transformation, which is based on an arbitrary composition of texture coordinates, provides a robust and elegant mapping between tangent and texture space.

This is sufficient for orientating/skewing tangent space coordinates to texture space. However, it does not account for the translation of the triangle’s texture coordinates in texture space. Thus, the described Euclidean matrix should be encapsulated within an affine transformation to account for coordinate translation. The translation component of this affine transformation is based on the texture coordinate that is shared by both ‘texture coordinate sides’ that were used to define this matrix.

As noted, this affine transform represents a mapping from tangent space to texture space. By inverting this matrix, a mapping from texture space to tangent space is acquired. This enables the iteration process that is based in texture space to express valid samples as coordinates in tangent space.

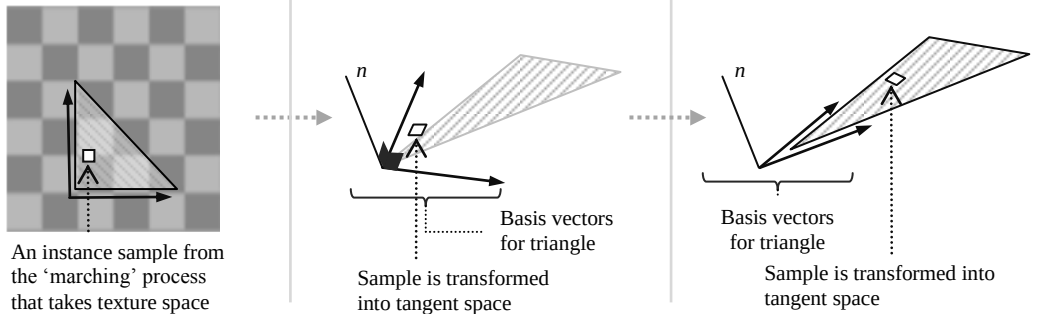


Figure 58 Illustrates the sample transformation process for the final triangle coverage algorithm

To conclude the overall mapping process, a transformation that maps coordinates from tangent space to model/triangle space is required. Acquiring this transformation is trivial given that tangent space is orthogonal. Because this transformation is consistent with the overall 'direction' of the mapping process, inversion of this transformation is not required.

To construct this transformation, basis vectors are identified which constitute the triangle's 'space' (see image C in figure 58). Because this space aligns with the triangle's surface, two of the three basis vectors are defined by geometric sides of the triangle. Note that this basis pair must correspond to the 'sides' of the triangle that were used to derive the 'tangent to texture space' transformation.

The third basis vector of this space is represented by the normal vector of the triangle's geometry as image (C) in figure 58 illustrates. Note that this triangle space is not orthogonal. The side effect of this is that coordinates from tangent space are appropriately 'warped' to the geometric topology of the triangle (space). This is illustrated between images (B) and (C) in figure 58, where the sample position is 'warped' into the geometric bounds of the triangle.

Within the actual implementation, both of the mentioned transformations are constructed and then composed into a single transformation. This combined transformation represents the entire mapping process and is illustrated by the following equations.

$M_c = M_{to} \cdot (M_{tt})^{-1}$ $c_o = M_c \cdot c_t$	
Where:	
M_c	Mapping composition
M_{tm}	Tangent space to model/triangle space
M_{tt}	Tangent space to texture space
c_o	Coordinate in model/triangle space
c_t	Coordinate in texture space

Integration of procedural methods

As mentioned, a central motivation of this algorithm is to expose PM's to game artists in new and novel ways. In this section, the full integration of the mechanism for instance/sample distribution is described. Thus, the following discussion is concerned with the instancing shader's generation of parameters that represent 'generated instances'. Note that this generation process takes place, following a 'valid sample' of the previously described marching process.

The marching process was designed to calculate and provide data that specifies the position of instances on the manifold geometry (in 3D space). In addition, it provides information that is required to parameterize the PM's, invoked during the sample/instance generation. Table 18 provides an overview of the data that is provided in this context.

Input Data	Dimensions	Space
Sample Position	3	Object (Affine)
Manifold normal	3	Object (Euclidean)
Texture Coordinate	2	Texture
Output Data		
Data Stream (List)	-	Object (Affine)

Table 18 The marching process exposes this data to the instance generation phase

As mentioned, the algorithm aims to achieve as much 'transparency' as possible for the artist during the creative process. The motivation is to avoid unnecessary learning overheads, by maintaining consistency with techniques and concepts familiar to artists. For this reason,

effort was made to preserve the conventional interpretation of texture coordinates in manifold geometry, yielding greater control over the use of PM's for instancing.

The primary purpose of the sampling process is to choose positions for instances on the manifold geometry. However, for reasons that will be discussed, it is also useful to have texture coordinates available for each sample, with respect to the triangle of the manifold geometry.

Fortunately, these texture coordinates are actually computed as part of the process for choosing the instance sample positions and therefore, no extra computation is necessary. The texture coordinates of an 'instance sample' are the interpolation of texture coordinates explicitly assigned to the three vertices of the manifold triangle. These interpolated coordinates enable procedural functions to be evaluated at any intermediate point on the manifold geometry. This provides intermediate data values over the surfaces that can be used for evaluation of procedural functions. Importantly, this doesn't add significant complication when integrating procedural functions.

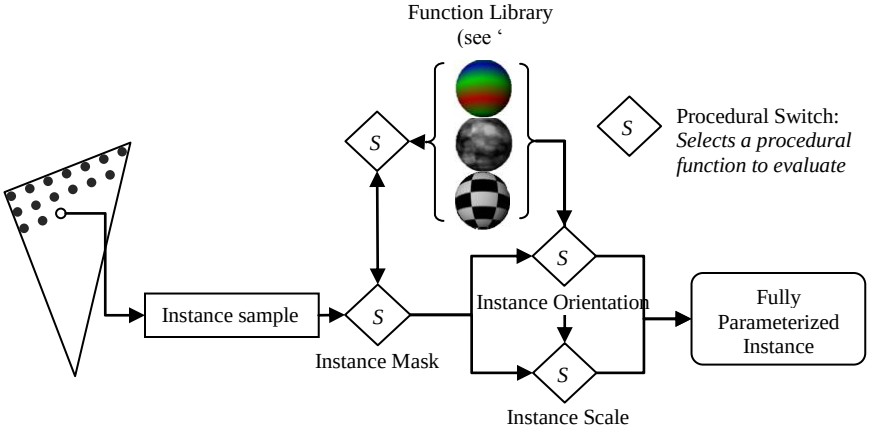


Figure 59 Schematic of per-instance procedural evaluations

Although up to three procedural functions can be involved in the generation of an instance, provision has only been made for two functions to specify instance parameters (note that the third function is applied as the instancing 'mask'). In this experimental implementation, the two procedural functions are used to parameterize the orientation and scale of an instance (see figure 59), each of which utilizes the interpolated texture coordinates mentioned.

Procedural functions generate a single scalar value which can range between 0.0 and 1.0. Recall from the project design chapter that the parameterization of an instance's orientation is achieved by scaling this 'range' between 0.0 and 2π. This implementation only alters orientation about the normal vector at that point on the manifold's surface. In games, orientation about this axis alone is sufficient for most situations where instancing is used.

The algorithm takes advantage of this ‘convention’ by only specifying the magnitude of rotation around this axis. As discussed, minimizing the instance data makes more efficient use of data bandwidth between the GPU and hosting system during the algorithm’s ‘stream out’ phase, improving overall space/time efficiency.

To demonstrate the scope of the instancing concept, the implementation allows different procedural functions to be specified by artists for each instance ‘parameter’. This is achieved via switches which are integrated into the instance generation stage of the shader. These switches are controlled by the hosting application, and specify which procedural function to evaluate for a particular instance parameter. If the ‘Perlin noise’ procedural were selected for instance masking for example, the shader would evaluate that procedural function using appropriate procedural parameters. A similar process is followed for orientation and scaling, for which procedural functions are independently evaluated.

Integration of instancing ‘cookie cutter’

PM’s allow the application of RTGI to a range of scenarios. Recall initial discussion on procedural instancing in the automated object placement section of the project design chapter which demonstrated how PM’s could greatly contribute to the realism of algorithmic instancing. Despite this, procedurally driven instancing still lacks an important level of ‘control’ that is required for a number of scenarios. Thus, for many situations, the previously described algorithm would be insufficient. To elaborate, consider the example in automated object placement section (page 37) where algorithmic instancing is used to populate a ground surface with foliage.

Typically, a procedural mask is applied in situations like this, to add variety to foliage distribution. In many cases, this would provide a sufficient level of ‘artist control’ over the resulting algorithmic instancing.

Scenarios exist however, where instance distribution based on a procedural function is insufficient; namely for situations where specific ‘features’ in the distribution of instances are required. If for example, buildings and other prescribed elements were added to the scene containing procedurally instanced foliage, intersection (conflicts) between these elements would be inevitable. As the density and complexity of foliage/instancing increases, the likelihood of conflicts between scene elements and instances also obviously increases. Given the pursuit of realism in game graphics, intersections between instanced geometry and other elements of a scene are unacceptable.

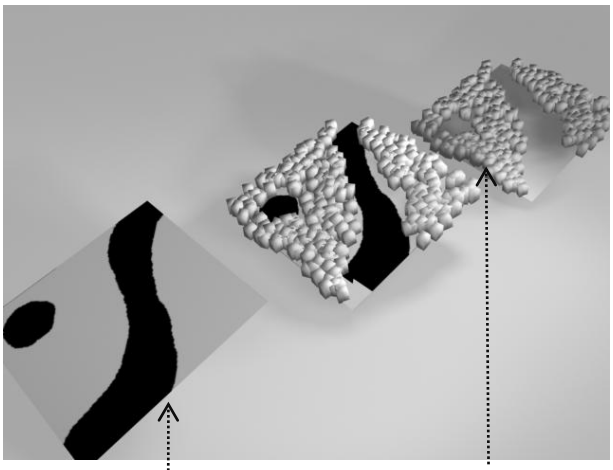
Note that this issue is consistent with findings from the industry consultation section (page 35) of the project design chapter. Staff at Sidhe emphasised the importance of 'artist control' during the content authoring process. The use of procedural functionality to replace 'meticulous' authoring tasks is therefore avoided by Sidhe's developers, due to the limited control that PM's tend to offer. Conflicts between instanced objects and scene elements, is obviously a side effect of automated instancing based solely on procedural functions. As the interview with staff at Sidhe illustrates however, 'hybrid' strategies exist; these can be employed to supplement the limited control that PM's naturally offer.

This hybrid strategy merges 'conventional game art' with PM's to provide artists with explicit control over where the instanced geometry can appear on the manifold. Through this, the RTGI algorithm can be used in scenarios such as the described scene, without conflicts occurring between instances and 'obstacles' that exist on manifold geometry.

The solution introduces another mask into the RTGI implementation, referred to as the 'cookie cutter'. This mask enables artists to explicitly define regions on the manifold where geometry instantiation can occur. Note that the algorithms underlying procedural instancing behaviour is still maintained.

The cookie cutter (or cookie) introduces a high level override that controls instantiation of instances, regardless of the algorithm's normal behaviour. The cookie offers a level of control to the instancing process which is equivalent to 'per-pixel' painting in an image.

The cookie 'image' is implicitly mapped over the manifold geometry. Placement of 'features' of the cookie dictate where instances can appear on the manifold. The cookie image is typically applied to the manifold geometry in the same way that textures are applied to geometry. Note that the concept of mapping image/texture data to 3D geometry is well established amongst the digital/game art community.



The cookie cutter image, mapped to the manifold geometry. Under normal circumstances, this is not explicitly visible.

The corresponding result of 'cookie cutter' based instancing

Figure 60 Illustrates the cookie cutter concept for instance generations

The RTGI algorithm 'samples' the cookie image during the instance generation process. Each sample corresponds to the position of an instance in the cookies image

(‘texture space’). In other words, this process yields the cookie ‘colour’ that a particular instance ‘sits upon’ on the manifold. If the colour value matches or exceeds a threshold (in this implementation the threshold is 1.0 or white), the instancing process continues as normal. If this criterion is not satisfied, the particular instance is suppressed.

Integrating the cookie feature into the final RTGI implementation was a straightforward process. As discussed, the instancing process involves iterative ‘marching’ across the surface of each triangle that comprises the manifold. Recall that texture coordinates are provided in the context of each instance iteration of the manifold geometry. In the context of shader programming, image/texture sampling requires the specification of texture coordinates to ‘direct’ the sampling process (HLSL:Sample, 2010) (OpenGL, 2010). Accordingly, at each ‘instance iteration’, the texture coordinates for that point on the manifold are used to sample the cookie image. Conceptually, the sampling result can be viewed as the ‘pixel colour’ that ‘sits below’ an instance on the manifold. As mentioned, the sampled colour must satisfy the predefined criteria, in order for an instance to be generated at that point on the manifold.

Integrating the cookie feature into RTGI was a trivial process. This is due to shader languages, namely HLSL, readily exposing texture sampling functionality. The code excerpt in table 19 shows how the revised algorithm integrates the cookie feature via texture sampling (HLSL:Sample, 2010).

Instancing Shader

```
[maxvertexcount(128)]
void gs_instancing(    ... )
{
  /* Initialization and sample marching iteration */
  ...
  {
    /*
    This code is executed for a marching sample that is ‘valid’ w.r.t. the current
    triangle, etc
    */

    /* Interpolated texture coordinates for sample on triangle */
    float4 sample_texcoord = float4(x, y, 0.0f, 1.0f);

    /*
    The cookie cutter texture is sampled, colour ‘beneath’ marching
    sample is returned (tex2Dlod is a texture sampling function in HLSL)
    */
    float4 sample_cookiecolor = tex2Dlod (g_cookie, sample_texcoord);
```

```
    float sample_cookievalue = (float) sample_cookiecolor;

    /*
    If the cookie cutter sample satisfied criteria, or cookie cutter
    feature inactive, generate the marching sample's instance
    */
    if(sample_cookievalue >= 1.0f || g_cookieexists == false) {
      ...
      GenerateInstance(sample_position, x, y, n);
    }
  }
  ...
}
```

Table 19 Code excerpt from the instance shader implementation, illustrates the integration of the 'cookie cutter' feature

As illustrated, the 'cookie cutter' takes advantage of sampling functionality that is native to the shader architecture, to deliver an elegant solution for the 'conflict' problem previously described. Furthermore, given that texture/image sampling is a relatively efficient process on graphics hardware, this solution is well suited to the algorithm's 'real-time' nature.

As the shader excerpt in table 19 shows, 'cookie cutting' has been integrated into the RTGI algorithm as an optional feature. The system only invokes the feature when a cookie image has been specified by the artist during runtime. Although this feature offers a considerable level of control over the instancing outcome, artists must take into account space/capacity issues when using the feature. This is because the cookie represents a bitmap image and thus, the storage of a cookie cutter(s) has the potential to introduce significant memory use.

Low resolution cookies should be used where possible, to preserve the algorithms ambition of minimal memory/capacity utilization. For situations where a high level of instancing 'granularity'/control is required, it is recommended that artists manually place these objects to avoid the need for a high resolution cookie.

Implementation of second pass

The previous discussion provides a detailed overview of the RTGI implementation's first shader pass. The purpose of this pass is to generate data which can be used in subsequent stages of the rendering process for instanced rendering. As discussed, the second pass applies 'hardware instancing' to efficiently draw geometric objects that correspond to the generated instance data. The following section briefly describes the implementation of the second pass in the context of this research project.

Recall that ‘hardware instancing’ utilizes special purpose GPU functionality to efficiently render large ‘batches’ of the same geometry (Instanced Geometry, 2010). To achieve this, the GPU is supplied with two sources of data; the geometry being ‘instanced’, as well as the data that describes each unique instance.

Because this project is based on the Direct3D 10 API, invoking hardware instancing functionality is relatively straight forward. As discussed, the Direct3D 10 API exposes functionality for rendering geometric primitives (Myers, 2007). In addition, Direct3D 10 introduces a similar subset of rendering functions that explicitly exposes instanced rendering to developers (Input-Assembler Stage, 2010). These functions are therefore, invoked during the second phase of the RTGI algorithm.

As discussed, the instance generation pass is executed in the context of a rendering pipeline that is configured for data ‘output-streaming’. Output-streaming exposes instance data generated in the algorithm’s first pass, making it available for use in the second, independent rendering stage. The concept of ‘streamed’ data is typically represented as the flow of shader generated data, from GPU to CPU memory via the connecting data bus. The accessibility of streamed data to the CPU enables it to be used across multiple rendering stages. For RTGI, streamed data is ‘rebound’ to the graphics device as instancing data for hardware instancing.

In practice, the ‘flow’ of data across the noted data bus may not necessarily occur. This is because Direct3D 10 effectively exposes buffers as pointers, which are used and manipulated by the CPU during interaction with the rendering device (Resource Types, 2010). Between each pass of the algorithm, the implementation’s CPU bound component, unbinds and rebinds the same ‘instance data’ pointer to the graphics device.

Thus, it is conceivable that ‘streamed’ instance data generated by the first pass, remains resident in GPU memory for direct use in the second pass. The transfer of instance data across the data bus could potentially be avoided, therefore having positive implications on runtime performance.

With the exception of ‘instancing’ draw routines that are invoked, much of the RTGI’s implementation for the second stage is indistinguishable from the project’s standard rendering process. Instance rendering in the algorithm’s second stage is essentially equivalent to the rendering process described on page 76 of the interactive tool chain chapter. Some additions were made however, to shader code that underlies standard geometry rendering. The purpose of these additions was to provide a separate ‘shader code path’ that facilitates and processes the instance data supplied prior to instance rendering.

Page 170 of the demonstrations chapter illustrates the core functionality of the final RTGI algorithm, via the implementation that was integrated into this research's tool chain.

Object variation algorithm

As mentioned in the project design chapter, this research explores the use of PM's to achieve variation in game content, without placing excess burden on artists. Ideally, total development effort for artists producing varying content should be that only a single 'base object' need be made.

In addition, use of PM's reduces storage requirements necessary for delivering object variation. This is because duplication of any data isn't needed, as is the case with traditional object variation schemes. The simplest and most common strategy for 'geometric variation' between game objects is to produce multiple copies of an asset where each has subtle/unique geometric variation. If variation for an object is created by an artist, each 'version' of the asset is then deployed via the game's distribution medium, for use in the game. The proposed algorithm substitutes data duplication with a procedural function which generates variety between different instances of the same geometric 'base object'.

Invalid source

Figure 61 Depicting variation between pedestrians in Grand Theft Auto IV (GTAIV)

The literature review identified the use of PM variation techniques in commercial games. For example, Far Cry 2 integrates a procedurally based 'character generation system' into its game engine technology (Breckon, FarCry 2 Preview, 2008). This system generates/constructs characters at runtime, delivering realistic crowd scenes. By basing this system on PM's, pseudo random character generation can be achieved. The 'randomness' must be bound however, given that systems such as this require that sensible parameters are generated, ensuring correct and in this case, believable characters. By using PM's to specify

character parameters, developers can tune the presence of desirable variety traits in a population of generated characters.

Because PM's are referentially transparent, the data they yield retains 'consistent characteristics' for any specified input parameters. This offers artists/designers a level of 'bound control' which in this case, guarantees against undesirable outcomes in character generation. PM's therefore, provide a robust and reliable mechanism to deliver 'controlled variety', in contrast to generative variety based on runtime probability systems, which could yield erroneous parameters (or a biased distribution of parameters). These 'variation systems' therefore, enable games to better reflect the settings/environments being simulated; an obvious benefit to player experience.

(Wagner, n.d)

Duplication of props is common in games such as Half-life 2 (above). Props such as these serve as good candidates for the use of a geometric variation algorithm

Figure 62 Illustrates object duplication in games

Like many variation systems however, Far Cry 2's variation system is engineered towards a specific application (i.e. character variety). The concept being proposed aims to generalize the notion of 'generative variation' in game objects while also substituting conventional variation techniques. Keeping with the theme of this research, this proposed system also aims to deliver artist specified variation, in an interactive content creation context. The idea should be suitable for yielding algorithmic variation between most game assets/objects, rather than specific asset types as with Far Cry 2's variation system.

The proposed system achieves generalized variation by altering the geometric form of 'base object's' that it operates on. It should be noted that the algorithm implemented focuses on geometric variation between target objects, rather than variation of other 'embedded' object data, such as colour and/or materials. The reason for this is expanded in following section, 'objectives and overview'.

Like other algorithms of this thesis, PM's play a central role in delivering this concept. As with procedural based instancing, the variation system reuses procedural functionality which was implemented for the project's material system (see Perlin noise in table 5 on page 60).

The system implementation uses a feature of modern graphics hardware; namely the geometry shader. A motivation for this is to maximize real-time responsiveness of the system, in the context of an interactive tool chain. Thus, the algorithm remains within the interactive tool chain paradigm, computing geometric variation for objects in real time via the

GPU. This algorithm therefore enables artists to create and develop content with associated content variation, in an interactive development environment.

Objectives and overview

As mentioned, this variation system focuses on geometric modification to alter the form of 3D objects. Although other types of variation in game objects is possible, alterations to geometry are arguably more novel within the current climate of game development.

The concept of variation is not new to games and for many years, games have incorporated dynamic variety between objects via comparatively trivial methods. For example, dynamic colour and material composition on the same 'base content'. Strictly speaking, techniques such as this have been present in games dating back to the era of Pacman, where variation between the visual appearances of 'ghost' opponents was achieved by altering the colour of each ghost instance (figure 63) (DeMaria & Wilson, 2003). Although the idea's sophistication developed in games over subsequent decades, the fundamental concept still remains a popular method for integrating variety into games.

(Mackie, 2008)

Figure 63 Variation present within early video games

This technique for example, was and still is used in the Grand Theft Auto (GTA) franchise which debuted in 1997 (Grand Theft Auto 1, 2010). Since the first version of GTA, the game has been capable of delivering variety in vehicles via dynamic colour/material composition. The likely implication of this game feature, from an artist's perspective, is that enabling new versions of a vehicle would only require the creation of an additional 'colour mask' for that vehicle.

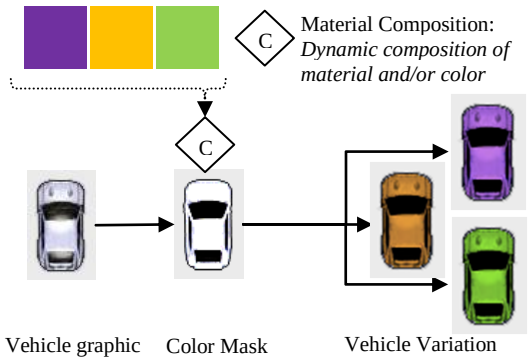


Figure 64 Illustrates a variation strategy that is frequently used in games

Colour masks are usually black and white images (see figure 64), where white pixels permit 'dynamic qualities' (variety) in the composed 'template'. The convention follows with black pixels of the mask preventing dynamic appearance in the composition.

In terms of overall production for GTA, having this mask substitutes the manual production of many cars for each colour

variant required. As figure 64 shows, the mask is modulated with an arbitrary colour hue to yield dynamic car colouration between instances of the car 'template' graphic. Note that black portions of the mask, such as the car's windshield, produce no variation in the final composition. This obviously offloads 'duplicative' production overhead to the underlying game technology.

As mentioned, the system being developed takes advantage of features in modern graphics hardware to deliver variation via similar principles, which are instead manifested in the geometry of assets/objects. This solution aims to remain as transparent to the content authoring process as possible, by encouraging the use of common modelling concepts while minimizing additional workload for asset variation.

To satisfy these criteria, a series of modelling scenarios, where this algorithm might be applicable, were considered. This led to an interesting insight; that the proposed algorithm is intrinsically similar to variation strategies based on material and colour variety.

As figure 64 shows, concepts such as 'colour masks' are provided in variation systems to give artists additional control over variation in game objects. Similarly, geometric variation in the proposed algorithm must expose a suitable level of control for artists. The 'variation system' must allow artists to selectively subject portions of a complex 3D object to the variation process. This selective variation will be referred to as non-uniform geometric deformation.

Non-uniform deformation

Non-uniform object deformation is an essential feature in a viable, procedurally based content 'variation system', as most situations only require subsections of target geometry to exhibit variation.

Recall the 'post apocalyptic game scene' concept that was presented in the automated object variation section of the project design chapter. In this setting, variety and 'damage' is procedurally added to scene objects, in accordance with the scene's narrative. The scene's narrative suggests certain characteristics in the way 'damage' should be manifested in props.

Non-uniform object deformation would naturally apply to these props, enabling artists to specify areas that are procedurally deformed in a consistent fashion with the scene's narrative. If the algorithm only enabled uniform variation across a target object, its use in this situation would yield visual results that were inconsistent with the scene's narrative, reducing

realism. Non-uniform deformation however, keeps the algorithm relevant to a wide range of situations.

This algorithm aims to maximize the level of control that artists have over non-uniform procedural variation in 'base geometry'. Thus, non-uniform variation was introduced at an 'atomic' level in the algorithm's 'data context', via vertices of the base geometry.
As mentioned, the integration of non-uniform deformation in this algorithm employs the sample principle as the masking technique, illustrated in figure 64. The obvious exception however, is that pixels in the masking technique are substituted with vertices of base geometry in this geometrically orientated system.

Applying, representing, and integrating variation data

Two major implementation strategies were explored during development of the variation system, each requiring different data and structures. Aspects common to the two strategies will be explained first, followed by details of each of the two approaches being covered.

In keeping with maximum artist control, the variation system requires independent and controllable levels of object variation across manifold geometry, to provide artists with control over the extent and location of deformation. Both versions of the algorithm subdivide or 'tessellate' the 'base geometry'. As discussed in the automated object variation section of this thesis (page 41), tessellating base geometry adds geometric resolution to the object. This added resolution allows more recognizable/detailed geometric variation in the processed result. Storing deformation data in each vertex of the base geometry allows flexible distribution of deformation data throughout the object. Furthermore, storing per-vertex deformation as continuous/floating point data, allows 'variable' levels of deformation in the base geometry.

As discussed in the interactive tool chain section (page 66), familiar/intuitive content authoring techniques exist which allow artists to easily assign deformation data to base geometry. Given that modern games contain objects consisting of thousands of triangles and vertices, it is essential that deformation data can be applied to geometry in a simple and flexible way. As mentioned, the proposed variation system uses the 'painting metaphor'. This metaphor exists in many modelling tools and is used by digital/game artists for a variety of authoring situations; namely assignment of per-vertex material/colour data to geometry and geometric sculpting/shaping. Several tools that are heavily used in industry, such as

Autodesk's 3D Studio Max, XSI (available in 'ICE' attributes) and Blender, natively integrate vertex-painting functionality (Matossian, 3DS Max for Windows, 2001) (ICE Attribute Reference, 2009) (Doc:Manual/Materials/Vertex Paint, 2009). Given the concept's widespread inclusion in a range of relevant tools, the algorithm's dependence on per-vertex colour data, painting conventions and the painting metaphor is reasonable.

This metaphor introduces additional 'colour' information to manifold geometry, which the system numerically interprets as a 'deformation weighting'. The algorithm's interpretation of 'deformation weighting' will be detailed in subsequent discussion. With respect to the authoring process, this colour data can be displayed to artists in the modelling environment, as a monochrome 'intensity' on base geometry. The intensity or 'brightness' represents the degree of geometric tessellation and vertex offset that will be applied at specified portions of the geometry, when the object is rendered in the game engine. Thus, 'colouration' constitutes an intuitive representation of this abstract idea, for artists working in the context of a content authoring environment.

The implication of per-vertex deformation weighting is that an auxiliary 'colour channel' is introduced into the vertex structure of base geometry. Thus, deformation data is held throughout the geometry. The deformation colour channel is exclusively used by the algorithm, to control the non-uniform variation process. Note that the 'colour deformation' channel doesn't replace other 'conventional' per-vertex colour channels, which can simultaneously exist in the geometry of this tool chain.

Achieving variety

Like the procedural instancing algorithm, the variation system is primarily implemented in a geometry shader. The motivation for this is that the shader/GPU architecture can offer good runtime performance, making this system suitable for integration in game rendering technology and interactive tool chains.

The tessellation and deformation process takes place in the context of individual triangles that comprise the target/base geometry. Per-vertex deformation data is encountered by the algorithm when the vertices of each triangle are accessed. Depending on the 'tessellation criteria', the algorithm responds to the cumulative deformation data of a triangle by either tessellating the triangle, or preserving its original form. 'Preserving' the triangle obviously yields no variation to the 'area' of the manifold that is enclosed by the triangle. If tessellation is applied however, new vertices are introduced to the original (incoming) triangle, producing the effect of triangular tessellation. Note that further detail on the tessellation approach that is used for this system, is provided in subsequent discussion.

Typically, geometric deformation is applied to vertices that are introduced during the tessellation process. The algorithm's deformation effect is achieved by offsetting the position of target vertices, by the triangle's scaled normal vector. The magnitude of this scaling is controlled by two factors, these being the variation system's associated noise procedural function, and the user specified 'amplitude' factor. Note that the amplitude factor comes from 'deformation amplitude', which was introduced during discussion of the tool chain's interface implementation (table 5, page 60 and table 8 page 72).

This 'vertex offset' mechanism constitutes the system's strategy for geometry deformation. Although this deformation approach is sufficient for many situations, it is important to note that scenarios exist where the 'offset' approach is inadequate. For example, situations that require severe or highly specified alteration to the form of target geometry may not be achievable via this approach. This is because the offset method is inherently limited by the original form and features of target/base geometry. Consider the situation where objects in a game scene have been subjected to a powerful explosion. To yield the impression of a local explosion amongst damaged variants, portions of 'object varieties' might be removed from the underlying 'base geometry'. Achieving variation of this nature is beyond the scope of the proposed system. Situations like these are reserved for future research and/or the use of 'conventional' approaches to object variation. For situations where object variation preserves the features and underlying form of 'base geometry', the proposed solution is a good candidate, allowing for space efficient, scalable variety amongst game objects. The term of 'generalized deformation' therefore applies to variation situations where variation retains the base object's form.

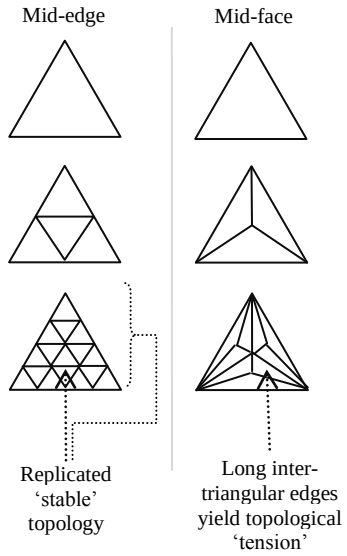


Figure 65 Various tessellation strategies

Triangle tessellation

As illustrated, triangle tessellation is central to the deformation process. Because a number of tessellation strategies exist, exploration into the most appropriate strategy for non-uniform procedural deformation (NPD) was necessary. Interestingly, the two predominant tessellation methods identified are both inherently recursive. These are referred to as mid-edge and mid-

face tessellation (see figure 65). The use of recursive tessellation enables algorithms such as NPD, to be elegantly expressed via recursion.

To illustrate this, consider NPD where mid-edge tessellation is applied to a triangle of the manifold geometry. The algorithm typically responds by tessellating the triangle into four sub triangles as illustrated in figure 66. Because the algorithm supports variable levels of deformation, sub triangles themselves, can also be tessellated. The overall deformation process tends to produce optimized tessellation across arbitrary manifold geometry. This is because the algorithm adaptively expands 'recursive sub trees' (tessellates), depending on the deformation criteria. This criterion is checked on a per triangle basis and determines whether tessellation is suppressed or applied to that triangle. Note that tessellation can apply to triangles at any level in the 'recursive' tree. In addition to the recursive nature of mid-edge tessellation, it also maintains 'stable topology' making it a good 'tessellation candidate' for this algorithm (see figure 65).

Subdivision topology

The tessellation strategy used for this algorithm must preserve the 'outer boundary' of each original input triangle. This ensures that the processed geometry maintains structural fidelity with the original base geometry. This means that inner angles of tessellated and un-tessellated triangles are inherited from the original triangle.

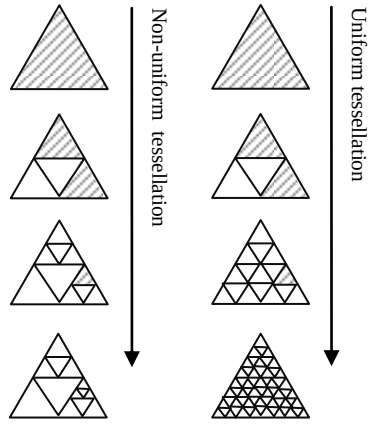


Figure 66 Illustrates non-uniform (or adaptive) tessellation

Furthermore, to prevent other rendering artefacts (such as holes and seams), sub triangles that are introduced during tessellation must be packed together within the bounds of the original input triangle (see figure 66). As figure 65 shows, both mid-edge and mid-face tessellation, satisfy this 'geometric criteria'.

Any complex polygon can be represented by a combination of simpler polygons or triangles (O'Rourke, 1994). For a given 'combination', this can be referred to as a 'topological representation' of the polygon. It is good practice to maintain sound 'triangular topology' when working with polygons in 3D geometry; namely by avoiding elongated, 'degenerate' triangles (Simmons, 2008). Good topology tends to produce robust objects, which are better suited for situations like animation and arbitrary warping. If animation is applied to a poorly structured polygon, 'visual artefacts' such as those shown in

image (A) of figure 67 may result, as the polygon's surface is subjected to arbitrary warping and tension.

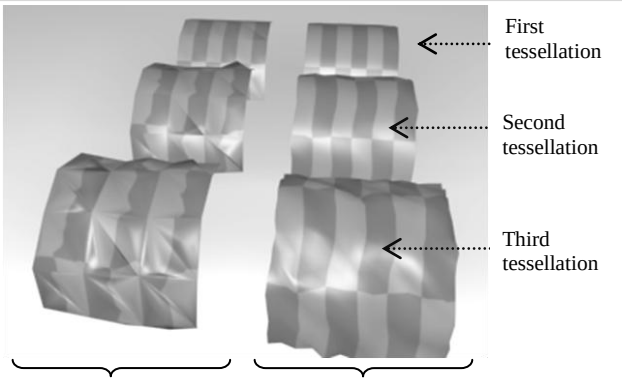


Figure 67 Quality comparison between tessellation strategies

The NPD algorithm aims to achieve high quality rendering results and thus, sound triangular topology is desirable for geometry processed and tessellated by the algorithm.

Good geometric topology is often achieved by minimizing the elongation of triangles (i.e. degenerate-like triangles), that comprise a complex polygon. As figure 65 (page 136) shows, mid-face tessellation introduces imbalanced 'tension' along the boundaries of adjacent sub triangles. In contrast, the mid-edge tessellation scheme illustrates a 'stable' form of tessellation, with 'uniform tension' along sub triangle boundaries.

Mid-edge tessellation also encourages continuity between adjacent triangles. This is because vertices that are introduced when an edge is 'split' tend to be 'matched' by equivalent 'split vertices' on adjacent triangles. When the deformation procedural is applied to adjacent 'split vertices', the boundary between adjacent triangles (of the original manifold geometry), remain indistinguishable. In contrast, mid-face subdivision fully preserves the outer edges of triangles being subdivided, as figure 65 illustrates. Although mid-face tessellation maintains consistency between adjacent triangles, the effect of edge preservation in the context of deformation unfortunately reveals the 'triangular substructures' of tessellated geometry (see figure 67).

For these reasons, mid-edge tessellation was consistently used for both approaches to the NPD algorithm.

Data interpolation

As discussed, tessellation is fundamental to this algorithm. Recall that this tool chain system integrates 'variable vertex structures' through which arbitrary data/data channels can be

interleaved in object geometry. As figure 65 illustrates, sub triangle vertex positions are 'interpolated' across the edge of a triangle during the mid-face tessellation process. Because other data typically exists in vertex structures, the NPD algorithm must also interpolate this data in a similar way to interpolation of vertex positions.

Thus, when a triangle's edge is subdivided, any per-vertex data associated with vertices of that edge is interpolated and assigned to the introduced 'mid-vertex'. Most per-vertex data of the mid-vertex can be expressed as the average of per-vertex data in vertices that define the edge. This assumption holds, because mid-edge tessellation subdivides edges exactly halfway across the face of the triangle, as figure 68 (page 140) shows. For some per-vertex data however, averaging alone is insufficient and thus, additional processing is necessary.

Per-vertex 'normal' vectors for example, must be re-normalized following the tessellation process. This is because 'vertex normals' must maintain unit length, to ensure correct surface response to light sources during the final rendering phase. If the normal vector of a mid-vertex were not normalized, incorrect lighting effects would result, given that vertex averaging would yield non-unit normals under many circumstances.

Applying data interpolation during the tessellation process preserves consistency between triangle data values supplied to the shader as input, and the tessellated sub triangles that are generated by the NPD algorithm.

Vertex adjacency and non-uniform deformation

Maintaining data consistency of internal triangle tessellation is essential to ensure that correct geometric visual results are produced by the NPD algorithm. Similarly, the process must also yield correct external/inter-triangle consistency between primitives of the base geometry. Because the algorithm supports non-uniform distribution of tessellation throughout base geometry, inconsistencies can arise between adjacent triangles that have different levels of tessellation/deformation.

As discussed, shader programs operate on individual primitives independently. This makes GPU based shader algorithms autonomous. The main advantage of 'autonomous' data processing architectures, is that they provide a robust basis for scalable parallelism. The GPU architecture achieves this by limiting 'accessible data' to the current primitive that is being processed. Random access of the entire manifold geometry is not possible, from the context of a shader.

To achieve seamless deformation across the boundary of primitives with different levels of tessellation, 'autonomous' shaders need to be 'aware' of the local geometry neighbourhood

for a given triangle. This presents a problem which is also seen in other GPU based algorithm implementations. The shadow volume algorithm presents a classic example of a situation in which neighbouring/adjacency information is useful. Dynamic shadows are achieved by 'extruding' back surfaces of occluding objects into/through the remaining scene, where shadows are required (Everitt & Kilgard, 2002). Adjacency information is required to identify the occluding objects 'silhouette edges', where extrusion is applied (Everitt & Kilgard, 2002). Implementing shadow volumes in a shader required that information about adjacent triangle vertices be made available in the context of a vertex shader. To make adjacency available in this context, additional vertex shader complexity as well as duplicate data in the manifold geometry, is required. Note that if the vertex shader determines a vertex to be on the silhouette edge, the vertex's 'extrusion' is typically computed by the shader. Similarly, this algorithm also requires that adjacency information be available at runtime. In contrast to shadow volumes however, this algorithm operates on triangles and thus, requires information about adjacent triangles during execution.

A feature of geometry shaders, as introduced by the latest graphic API's, is exposure of 'adjacency information' in the context of shaders (Shader Stages, 2010) (Brown, EXT_geometry_shader4, 2009). To take advantage of this functionality in Direct3D 10, the manifold/base geometry must be passed to the GPU in conjunction with its corresponding indice data (buffer). Indice buffers are arrays of integers that typically map triangles and polygons from a buffer of vertices. During the geometry shading process, this data is used by the graphics API's to efficiently expose adjacency information in the context of the geometry shader, without the need for data duplication, etc.

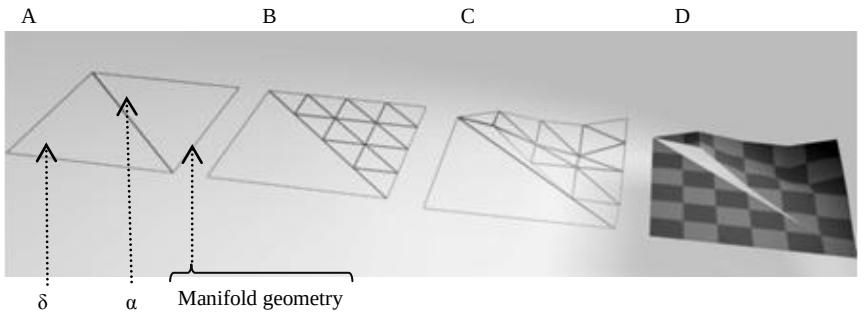


Figure 68 Illustrates seam-gap artifacts which occur when levels of deformation between adjacent triangles differ

Adjacency information is used in this algorithm to avoid 'seam inconsistencies' between adjacent triangles that have different levels of tessellation. (D) of figure 68 illustrates the visual 'gap artefacts' that result when adjacent tessellation is not correctly accounted for. These 'gap artefacts' are unacceptable, given the algorithm's objective of visual quality. (B)

in figure 68, shows the presence of non-uniform tessellation in simple manifold geometry, without the effects of 'vertex offset'/deformation. When the procedural function 'offsets' internal vertex positions (C, figure 68), 'gap artefacts' result along the seam of the original manifold primitives. This side effect is caused by a number of factors that derive from different 'geometric resolution' between adjacent primitives (δ and α, figure 68). Because the procedural function for vertex offset is autonomously applied to geometry, primitives 'implicitly assume' that neighbouring primitives are tessellated to the same extent.

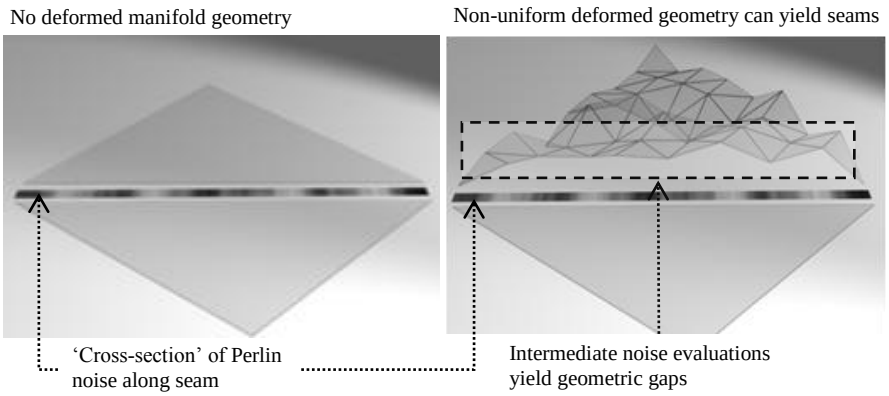
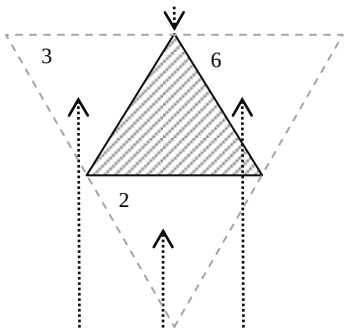


Figure 69 Illustrates the severity of seam artifacts under normal circumstances

Given the continuous nature of procedural functions, namely Perlin noise, more refined reproductions of procedural functions can be produced when applied to tessellated geometry, as figure 69 shows. In addition, this image also shows the implications of a continuous procedural function being used to deform triangles with different levels of tessellation.

As mentioned, tessellated sub triangles that lie along edges of the original manifold geometry are independently subjected to procedural offset, regardless of matching/non-matching tessellation in the adjacent geometry. Recall that procedural functions of this project are parameterized and evaluated by texture coordinate data, which is associated with vertices of manifold geometry. As mentioned, the interpolation of vertex data takes place during the tessellation of triangles into smaller sub triangles. This interpolation applies to per-vertex texture coordinates and thus, unique procedural noise/geometric offset can occur at intermediate points along the original manifold geometry edge.

To avoid gap artefacts along the boundaries of inconsistent primitives, the NPD algorithm makes use of 'deformation weightings' in both the current manifold triangle, as well as triangles in the local geometric neighbourhood (see figure 70).



Adjacent triangles in local manifold neighborhood (numbers represent the levels of tessellation that have been calculated for triangles)

Figure 70 Illustrates adjacent geometry that is accessible during triangle processing

In essence, the need for tessellation (as determined by a ‘tessellation criteria’) is computed for the current and adjacent triangles which are supplied to the NPD shader by Direct3D 10’s ‘input assembler’ (Input-Assembler Stage, 2010). Through this, the algorithm can ‘detect’ the deformation weightings in neighbouring triangles and thus, compute the levels of tessellation in adjacent geometry. Typically, if the tessellation level of an adjacent triangle is less than the current triangle, the tessellation of the current triangle is conformed to the lowest adjacent tessellation level.

As mentioned, two approaches were taken when implementing the NPD algorithm; each consisting of unique implementation characteristics to handle these tessellation scenarios. Further discussion on the implementation details of each approach is provided in the following sections.

First tessellation approach

As discussed, the NPD system requires non-uniform, mid-face tessellation. Multiple levels of tessellation can be elegantly expressed via recursion and thus, the algorithm was implemented recursively. For this approach, the deformation/tessellation algorithm was directly implemented into a ‘single pass’ shader. A motivation for this was that a single pass shader would easily integrate into the project’s rendering framework.

Unlike single pass algorithms, multi-pass algorithms require more intervention from the host system to control the execution and order of each pass, performed on the GPU during rendering. In addition, a multi-pass implementation of NPD would also require allocation of additional buffers, to temporarily store tessellation data between passes. This would impose extra responsibilities on the host system in terms of resource allocation and management, as well as resource ‘binding’ (to the graphics device). Thus, an NPD implementation that is encapsulated in a ‘single pass’ shader was attractive.

Unfortunately, recursion is not natively supported in HLSL (Function Declaration Syntax, 2010). It is possible however, to implement a customized stack data structure, in a shader to simulate recursion (Fryazinov & Pasko, 2008). This underpins the first implementation/approach of NPD.

The shader begins by allocating a static array of data structures which represent the ‘recursive stack’. In a similar way to other stack implementations, this data array is traversed during the recursion process. ‘Traversal’ of the stack is tracked by a ‘stack index’ (or ‘stack pointer’). The stack pointer represents the current position of the algorithm/shader within the ‘recursion tree’. Due to HLSL’s syntactical similarity to other languages, the stack’s core implementation is relatively straight forward.

As figure 66 (page 137) shows, each ‘level’ of the recursion represents a sub triangle in the tessellation process. The first recursive level processes the triangle that is passed to the NPD shader. The algorithm calculates the tessellation level of the current triangle from deformation data in the triangle’s vertices. A triangle’s tessellation level is represented as the average ‘deformation weighting’ of each vertex. If the tessellation level for the current triangle is greater than the current level of recursion, tessellation of the current triangle takes place. This represents the ‘tessellation criteria’ for this approach of the NPD algorithm.

As discussed, this involves the interpolation (averaging) of vertices on each side of the triangle. The newly interpolated vertices provide the corners of four ‘sub triangles’ to the current triangle (see figure 66). Recursion continues by traversal of the four sub triangles, where each sub triangle is stored/assigned to the stack (relative to the current stack index/pointer). The stack pointer is then updated to point to the ‘top’ of the stack following these assignments. The stack pointer will increment and decrement through the stack, as the recursive sub trees for each sub triangle are traversed.

For each level/triangle of the recursive process, the algorithm maintains an additional variable which ‘tracks’ the sub triangles that have and haven’t been recurred/processed. When a sub triangle has been recursively processed in full, the tracker variable that is associated with its parent triangle is incremented. Given that tessellation yields four sub triangles, the algorithm checks this tracker variable following recursion of a sub triangle. If the tracker equals four, the algorithm knows that the current triangle has been fully processed and the stack pointer decrements.

If the current triangle’s tessellation level is less than or equal to the current level of recursion, then the triangle is immediately ‘emitted’ (returned) from the shader. Thus, triangles that require no further tessellation are recognized as primitives that constitute the final tessellation result. If the recursion process reaches a level beyond the ‘depth’ that the pre-allocated stack can hold, then the current triangle is also emitted and the recursive process ‘decrements’. Recall that data which is emitted from HLSL geometry shaders is ‘appended’ to an output list, which is exposed in the shader’s context. A side effect of this algorithm’s

implementation is that the order of sub triangles emitted from the NPD shader is essentially arbitrary. The arbitrary order of triangles in this list (representing the tessellated base geometry) yields no irregularities in the subsequent rendering (rasterization) outcome.

As shown in the demonstrations chapter (page 180), the NPD algorithm achieves non-uniform levels of tessellation via that tessellation which occurs variably, on a 'per-triangle' basis. As mentioned, the interpolation of per-vertex data is an integral part of 'edge splitting' in the tessellation process. Recall that deformation data exists in each vertex throughout the manifold geometry. When an edge is split, a new vertex is introduced which represents the split (mid) point. As discussed, most data assigned to the split vertex is the interpolation (or average) of values in the vertices that define the edge being subdivided. The per-vertex deformation value that is assigned however, represents the minimum deformation value between the 'edges' vertex pair.
This has the effect of 'conservative' tessellation, given that smaller deformation weighting's yield lower tessellation levels for sub triangles. As a side effect, the tessellation criterion is less likely to be satisfied during subsequent recursion, therefore minimizing overall tessellation.

Size and compile time

As mentioned, the GPU is a parallel architecture that is capable of delivering high processing performance. Thus, the architecture is an attractive platform for algorithms such as NPD, where real-time operation is required.
Recall that the NPD's integration into this GRC application requires that the algorithm be executed in real-time during the hosting application's 'render cycle'. The algorithm's real-time performance enables NPD to be applied to game objects/geometry dynamically and interactively. Despite the performance advantages of this architecture, some aspects of GPU/shaders are limited. As noted in the tool chain implementation section, the interactive tool chain temporarily stalls when a shader is recompiled during runtime. As discussed, shader recompilation is necessary during runtime following a number of artist interaction events; namely when geometry (vertex) formats are reconfigured (page 80) and/or procedural material compositions are changed (page 90). Recall that the tool chain allows artists to independently configure 'vertex formats' for game objects/geometry. The NPD algorithm integrates with this feature of the tool chain and thus, requires that artists activate 'deformation weightings' in an object's vertex-format in order to take advantage of the NPD system. This requires that artists' invoke functionality on the tool chain's interface, which

was previously discussed in the ‘real time content encoder (RTCE)’ section on page 66. Given the tool chain’s responsive nature, the object’s associated shader is recompiled to include NPD functionality following the specification of ‘deformation’ in the object’s vertex format. Unfortunately, recompilation of this NPD shader is nontrivial for Direct3D 10’s HLSL compiler. This is mostly due to the recursion stack, which is central to the shader consisting of complex flow control, nested in a conditional loop.

Compiling shaders that integrate deformation functionality therefore, took approximately 2 – 2.5 times longer to compile, than compilation of their non-deformation counterparts. Note however, that the specification of deformation functionality is a ‘low frequency’ interaction event and thus, delays that result from these shader recompilations are somewhat acceptable.

In addition, the output of a typical deformation shader introduces over 200 more instructions than the ‘non-deformation’ equivalent. The specifications for shader model 4.0 (used for NPD) require that graphics hardware offer at least 65536 usable instruction slots, essentially removing the shader size limitations of prior ‘shader models’ (Blythe, The Direct3D 10 System, 2006). Larger shaders however, tend to incur longer compile times which obviously hinders the fluidity sought in this interactive tool chain.

In addition, the number of micro instructions used by an NPD shader correlates to the complexity of the specified vertex-format. As mentioned, shaders of this system are adaptable to arbitrary vertex structures. Because interpolation is central to triangle tessellation/subdivision, the NPD shader must correctly interpolate all per-vertex data. For implementation readability, this adaptive interpolation behaviour was encapsulated into a single HLSL function, as the code excerpt in table 20 demonstrates.

Deformation Shader

```
void edgeInterpolation(
  tDefVertex vertexA,
  tDefVertex vertexB,
  out tDefVertex vertexSplit
) {
  /*
  Initalize deformation vertex to default values
  */
  vertexSplit = (tDefVertex)0.0f;

  /*
  This function is specific to deformation and thus, it is safe to
  assume that per-vertex deformation data is present in the vertex
  structure.
  */
```

```
  vertexSplit.deformation.xyz = ...
  vertexSplit.deformation.w =
    min(vertexA.deformation.y, b.deformation.y);
  /*
  Systematically check the vertex format being compiled against, include
  shader instructions to interpolate per-vertex data as necessary.
  Note that 'lerp(a,b,s)' is an HLSL intrinsic function that
  interpolates input parameters (a,b) by a scalar (s)
  */
  #ifdef VERTEX_POSITION
  vertexSplit.position =
    lerp(vertexA.position , vertexB.position, 0.5f);
  #endif

  #ifdef VERTEX_COLOUR
  vertexSplit.colour =
    lerp(vertexA.colour,  vertexB.colour, 0.5f);
  #endif

  #ifdef VERTEX_UV1
  vertexSplit.uv1 =
    lerp(vertexA.uv1, vertexB.uv1, 0.5f);
  #endif

  #ifdef VERTEX_UV2
  vertexSplit.uv2 =
    lerp(vertexA.uv2, vertexB.uv2, 0.5f);
  #endif
  ...
  /*
  Per-vertex normals and tangents require normalization
  */
  #ifdef VERTEX_NORMAL
  float3 interpolatedNormal =
    lerp(vertexA.normal, vertexB.normal, 0.5f);
  vertexSplit.normal =
    normalize(interpolatedNormal);
  #endif

  #ifdef VERTEX_TANGENT
  float3 interpolatedTangent =
    lerp(vertexA. tangent, vertexB.tangent, 0.5f);
  vertexSplit.tangent =
    normalize(interpolatedTangent);
  #endif
}
```

Table 20 Code excerpt from the deformation shader implementation which shows how adaptive vertex interpolation is achieved in order to compute the 'split vertex'

This function is invoked three times (for each triangle side) during each iteration of the shader's loop based recursion. Unfortunately, the HLSL compiler resolves (non-intrinsic) function calls, by 'inlining' the function's 'body' in the shader. The implications of this for the NPD shader are that micro instructions underlying the subdivision function (shown in table 20), are duplicated three times throughout the NPD shader's loop. Thus, the '`edgeInterpolation`' function quickly becomes larger as vertex structures gain complexity, therefore increasing the shader's total instruction count and compile time.

Adjacency considerations for non-uniform deformation

By default, non-uniform deformation yields 'gaps' along the seams of adjacent manifold triangles that have different levels of tessellation. To solve this, per-vertex 'support' variables are introduced to this NPD implementation. These support variables provide 'contextual information' about a vertex, with respect to its tessellated geometric 'neighbourhood'. As tessellation takes place, the vertices of sub triangles inherit geometric context from the triangle being tessellated. This context is initially obtained from 'adjacency information' which is provided by the Direct3D 10 API. Because recursive tessellation steps in this algorithm are independent, triangles need to 'deduce' the level of tessellation present in surrounding triangles. By deducing adjacent tessellation, the algorithm can determine where 'vertex offset' (procedural deformation) is appropriate, therefore avoiding 'gaps' in the final tessellation result. Through this, adaptive tessellation is achieved over arbitrary manifold geometry configurations, without geometric artefacts/gaps occurring.

To elaborate, consider the scenario depicted in figure 71 , which shows the use of per-vertex tessellation context through three levels of adaptive subdivision.

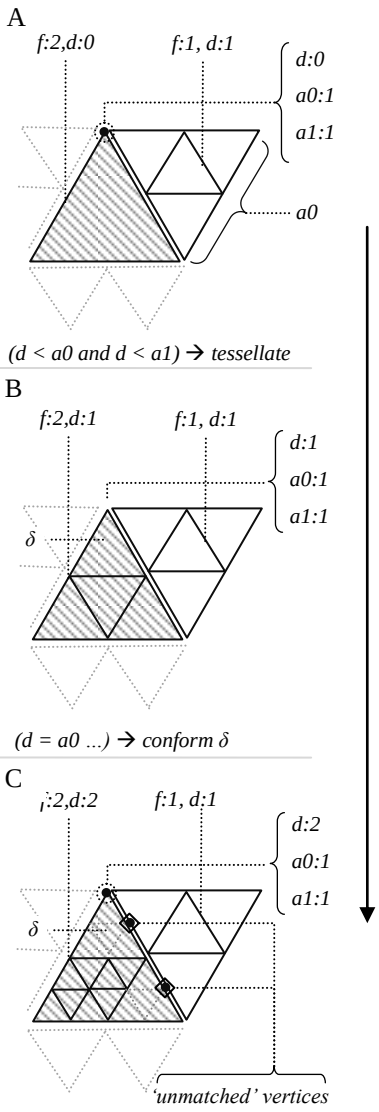


Figure 71 Illustrates the NPD algorithm's gap prevention strategy

If subdivision for the current triangle is required, recursive tessellation occurs as images (A) to (C) in figure 71 shows. Before the process beings however, the tessellation level of adjacent triangles is stored in the vertices of the current operand. Image (A) of figure 71 shows the storage of tessellation context in the topmost vertex of the shaded triangle (operand). This vertex stores the level of tessellation in adjacent triangle '*a0*' which, in this example is 1 (*f:1*). The operand of image (A) is tessellated into four sub triangles, given that its target level of tessellation is 2 (*f:2*). Newly created sub triangles (i.e. *δ*, in image B, figure

71) are now autonomous; relying on inherited contextual information to deduce surrounding triangle tessellation.

During recursive tessellation, each sub triangle tracks its current level of tessellation via the variable *d*. When an operand tessellates, the values of *d* for its sub triangles are incremented from *d* of the current operand. Therefore, as a triangle is processed, the value of *d* is tested against context variables that store adjacent tessellation levels (i.e. *a0, a1*). If adjacent tessellation levels are greater than *d* then the algorithm deduces that tessellation is appropriate. Otherwise, tessellation is suppressed as illustrated by sub triangle δ, between images (B) and (C) of figure 71.

Where sub triangles have different levels of adjacent tessellation, the tessellation/deformation situation must be handled differently. This requires sub triangles (i.e. δ) to negotiate between differences in adjacent tessellation, by 'adaptively suppressing' the effect of procedural vertex offset. If a sub triangle detects differences in adjacent tessellation, it will simply tessellate 'towards' its own target level of tessellation. This introduces 'unmatched' internal vertices which exist on seams/boundaries between the current triangle and adjacent triangles with lower tessellation (see image C of figure 71). To avoid gap artefacts, procedural offset is not directly applied to unmatched vertices. Rather, unmatched vertices are 'clamped', aligning them to the offset that will result in the adjacent tessellation.

Thus, the algorithm handles this 'negotiation' by maintaining its own tessellation level, while adaptively suppressing 'vertex offset' (deformation) of introduced vertices, as deemed necessary by contextual tessellation data. 'Conformance' to adjacent triangles with greater levels of tessellation is not required. This is because they too are subjected to 'conservative' tessellation and thus, will conform to the tessellation level of this triangle. By uniformly applying conformance in 'one direction', simple and robust inter-triangular deformation was achieved.

An interesting effect of this 'contextual data' is that, conformance to adjacent tessellation is 'generalized' for all sub triangles. Thus, the same context driven tessellation behaviour applies to all sub triangles, regardless of their position, with respect to the original manifold geometry. For example, sub triangles that exist on the boundary of original manifold triangles employ the same tessellation functionality as 'inner' sub triangles of tessellated manifold geometry.

Limitations and issues

From a functional perspective, this NPD implementation fulfils its core requirements. In addition to delivering non-uniform deformation, this algorithm delivered adequate runtime performance on a range of test scenes, on midrange graphics hardware. Unfortunately, the full extent of geometric deformation could not be achieved with this first implementation approach. This is due to fixed limitations relating to emitting/streaming data from geometry shaders.

Recall that geometry shaders can currently emit up to 2048 bytes of data via output-streaming (Stream-Output Stage, 2010). This presents a fundamental problem to the NPD algorithm described. As discussed, the recursive algorithm is implemented in a single pass geometry shader. This was advantageous because it simplified the integration of NPD into the project's rendering system. The consequence of this approach however, is that it assumes the stream-out capacity of geometry shaders is sufficient to capture all tessellated output data. Following tests, it became obvious that the capacity of data streaming in geometry shaders would be insufficient to deliver the full scope of tessellation resolution sought.

To elaborate, consider a vertex structure consisting of per-vertex position, normal and colour vectors, as well as a texture coordinate channel. The minimum size for this vertex structure would be approximately 48 bytes:

$$vertex_{size} = \left(position_{xyz} + normal_{xyz} + colour_{rgb} + coordinates_{xy}\right)$$
$$vertex_{size} = (3 + 3 + 3 + 2) \times 4$$
$$triangle_{size} = vertex_{size} \times 3$$
$$triangle_{size} = 144\ bytes$$

Assuming a standard vertex structure such as this, in conjunction with the stream-out limitations, each manifold triangle processed by the NPD geometry shader would only be capable of tessellating (emitting) approximately 14 sub triangles. Thus, only one level of tessellation could be achieved (as one level requires 4 triangles and two levels require 4×4 triangles etc). Recall that if a geometry shader expends its stream-out capacity, excess data is discarded by the hardware. The consequence of this, in the context of NPD, is that discarded triangles yield the appearance of 'holes' throughout the final deformation geometry.

Obviously tessellation applied to geometry consisting of more sophisticated vertex structures (i.e. per-vertex tangent space, additional texture coordinates) will yield larger vertex/triangle sizes, therefore further limiting the shader's tessellation capabilities.

Perhaps the most useful scenarios/applications for NPD are those where target 'base/manifold geometry' consists of large, planar surfaces. Examples of these that arise in many game objects/props include crates, containers, buildings and vehicles. By generating high levels of tessellation and deformation algorithmically, the potential for increased deformation detail, with minimal time investment by artists, exists. Given that the stream-out limitations hinder the quality and 'granularity' of variation detail in these 'cases', an alternative method for implementing NPD was explored. As the following section illustrates, the new approach avoids the mentioned limitation, therefore enabling higher levels of tessellation and thus, more detailed object variation.

Second tessellation approach

This approach shares many features with the preceding implementation, in particular the use of recursive tessellation. As mentioned, the shader architecture's stream-out limitations hindered the previous NPD implementation. In order to resolve this, the design has a significantly revised structure. The new structure can be elegantly expressed on the GPU/shader architecture. A number of other benefits and issues arise from this revision, which will be subsequently discussed.

Structure and simplicity

As with the first revision, the majority of this NPD algorithm exists in a geometry shader. A notable distinction in this shader implementation however, is the absence of a 'stack'. Recall the first approach implemented a stack to emulate shader based recursion. This was necessary as current programmable hardware does not support recursion (Function Declaration Syntax, 2010). Although recursion was achievable, the simplicity that was originally sought after through the use of recursion was undermined by the complexity of implementing a stack. Furthermore, the addition of 'gap elimination' functionality in this custom stack, led to a final implementation that was difficult to maintain and debug (despite debugging capabilities of shader authoring tools such as PIX) (PIX, n.d.).

In this approach, the ‘recursion stack’ is replaced by this shader’s multi-pass structure which is illustrated in figure 72. Each ‘pass’ of this multi-pass solution represents a level of tessellation in the ‘recursive tree’.

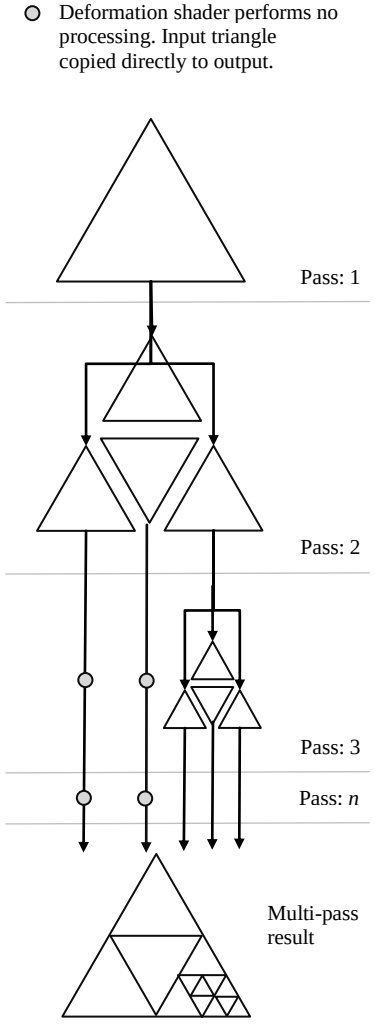


Figure 72 Shows how triangles that require no deformation are copied through internal passes

In contrast to the first implementation, passes of this solution constitute smaller, simpler and more manageable code modules. Each shader pass emits a fixed volume of data; either one or four triangles. Recall that the previous NPD shader was designed to emit a variable number of triangles from a single shader pass. By minimizing the volume of streamed data from each invocation of the NPD shader however, the algorithm avoids the stream capacity constraints which hindered the first approach. Multi-pass tessellation requires that geometry emitted from a shader be streamed directly into temporary storage, in between passes. The role of this storage (buffer) will be elaborated in subsequent discussion.

As figure 72 indicates, this geometry shader operates on single triangles. Thus, the emission of four triangles from a geometry shader represents tessellation in this NPD solution. The geometry shader is also capable of emitting a single triangle, being a duplicate of the input triangle. Emitting a single triangle involves no intermediate processing and thus, incurs little overhead on the GPU. Via these two forms of output, the NPD shader delivers non-uniform tessellation. The cumulative effect of these behaviours is shown between passes one and three of figure 72. Note that triangles independently tessellate throughout the multi-pass process.

As illustrated, mid-edge tessellation is applied to input triangles of the NPD shader that are eligible for tessellation. Similarly to the previous implementation, the tessellation applies interpolation to vertex data to deliver subdivision. This interpolation process possesses some unique characteristics however, which will be expanded in later discussion.

As noted, this NPD implementation maintains key differences to the first approach; that shader instances emit less geometric data while also reducing variation in the volume of data emitted from each execution instance. As well as greater functional flexibility, the following section illustrates how consistent data output has positive implications for the shader's runtime performance.

Parallelism and load balancing

Modern GPU's achieve massive parallelism by simultaneously executing shader code across hundreds of 'cores' (What is GPU Computing?, 2010) (Hwu, 2009). At runtime, cores are assigned work/tasks based on the shading context. For example, during the pixel shading process, the graphics hardware would assign 'cores' to pixels, therefore allowing batches of pixels to be concurrently processed.

This NPD implementation capitalizes on the GPU's parallelised architecture, via its 'modularized' and multi-pass structure. 'Instances' of the shader operate on individual primitives/triangles of the manifold geometry. When the algorithm is applied to complex manifolds, the hardware can naturally distribute NPD processing to all available cores, achieving parallel execution that is typical of good, GPU orientated algorithm implementations.

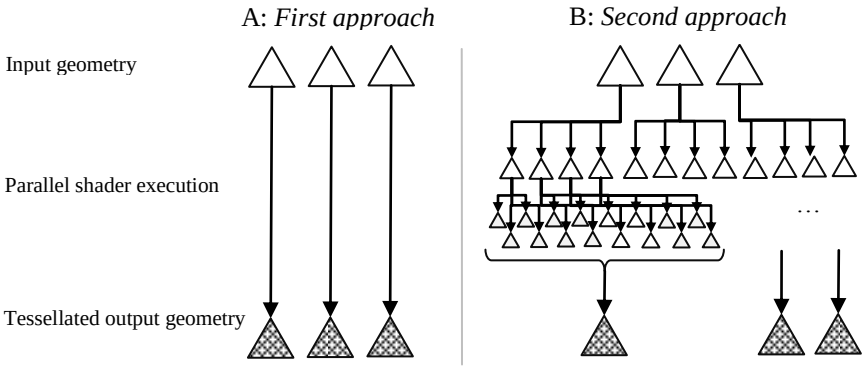


Figure 73 Comparison of parallelism in each NPD algorithm approach

The NPD shader's core implementation is similar to that of a single level of recursion in the 'first' NPD approach. As the following discussion illustrates, this represents a 'generalized' shader (and functionality), which is reused by multiple passes, during an NPD shader's execution. Data that is emitted during an NPD pass typically circulates through other instances of the same NPD shader, in subsequent passes. This is illustrated in figure 73, where each 'arrow' in the diagram can be interpreted as an instance of NPD shader execution.

Figure 73 illustrates how parallelism is utilized in each NPD approach. The parallelism illustrated for the ‘first approach’ (A), shows parallelism that is achieved as a consequence of implementing an algorithm on the GPU. This is because the first approach assigns ‘cores’ to triangles of the base geometry, which then serially perform the tessellation process.

In contrast, image (B) of figure 73 shows how the second approach makes better utilization of the GPU architecture’s parallelism. As the geometry tessellates (in the second implementation), many instances of the shader operate on emitted sub triangles. The approach scales well, via the cumulative effect of ‘branching’ smaller processing tasks, which run concurrently.

As a result, this design allows improved ‘load balancing’ amongst cores, in comparison to the previous implementation. Parallel systems achieve high performance by allowing multiple processing tasks to be carried out simultaneously. Unfortunately, the performance benefits of parallel systems can be hindered when synchronization between parallel processes is not achieved.

Consider a situation where ‘cores’ of a synchronized parallel system are assigned different processing workloads. The implication of this is that some cores will complete processing before others. To maintain synchronization, all cores must wait until each workload is complete. Thus, varied workloads obviously cause some cores to idle during the concurrent process, which represents unutilized processing capacity; this obviously being undesirable. Thus, ‘load balancing’ is an important consideration for parallel development.

The second NPD approach improves task distribution, given that the ‘workload’ of each task has only one of two values; that either one or four triangles are generated and emitted.

In contrast, the implementation of the first NPD algorithm used flow control by means of a ‘variable loop’, through which arbitrary volumes of triangle data were emitted. Thus, the opportunity for significant variation in runtime duration existed.

Final shader implementation

The shader itself is divided into six individual passes. Table 21 summarizes this structure.

Pass	Name	Purpose
1	Initialization	Prepares ‘generic’ geometry for use in subsequent tessellation passes, namely by adding ‘support data’ to manifold geometry which enables correct non-uniform tessellation.
2	Tessellate	These passes typically granulate incoming geometry to a higher tessellation. If no tessellation is required, the incoming triangle is directly copied to the shader’s output.
3		
4		
5		
6	Transformation & rasterization	Transforms/projects final tessellation geometry and renders this to the active pixel buffer/render target

Table 21 Pass structure of revised NPD shader

Recall that the first NPD approach achieved non-uniform deformation, using per-vertex support variables in the vertex structures. As discussed, this enabled autonomous operation on primitives, each retaining an ‘awareness’ of local/adjacent geometric tessellation. With this context information, the algorithm could determine when to apply procedural offsets to vertices, or when to conform local vertices to adjacent deformation. This system was necessary to avoid ‘gap’ artefacts along edge boundaries of manifold geometry, where levels of tessellation differ.

The first step in the ‘gap prevention’ mechanism is initialization of per-vertex ‘context data’. The initialization takes advantage of triangle adjacency information provided in Direct3D 10. This context data is interlaced into a copy of the manifold geometry that is created during the initialization pass, to be used in subsequent passes.

In contrast to the first NPD shader, this shader uses only two context (support) variables in each vertex of the manifold geometry. Recall that the first NPD shader introduced one variable to track the vertex’s tessellation level, in addition to variables that store the tessellation level of each adjacent triangle.

For this approach, one variable tracks the current level of tessellation for the respective triangle and the others hold ‘adjacency tessellation/context’. The insight behind this

simplification was that only the lowest level of tessellation in adjacent triangles, needed to be 'carried' with a vertex.

'Conservative tessellation' of the first NPD algorithm limits subdivision to the lowest level of tessellation in local and adjacent geometry. The revised algorithm performs this calculation once, during the shader's initialization pass. Therefore, the lowest adjacent tessellation of a manifold vertex is propagated through the subsequent tessellation process. The use of additional per-vertex support variables therefore becomes unnecessary. Importantly, the same 'gap prevention' behaviour (which depends on this context information) is maintained. An obvious side effect of this is a simplified NPD implementation, which only compares/tests a single 'tessellation context/support variable'.

As before, tessellation occurs if the triangle's 'cumulative tessellation value' is greater than the 'pass index' (and less than adjacent tessellation). Again, this represents the tessellation criteria for the NPD implementation. Note that cumulative tessellation is the sum of artist assigned deformation weightings (per-vertex) for a given triangle.

Because triangular tessellation is autonomous, the implementation needs a way of determining a triangle's current 'level' in the recursive (tessellation) tree. To achieve this, the NPD shader took advantage of another feature of HLSL; allowing generalized shader functionality which is 'implicitly aware' of the tessellation level. HLSL shaders consist of 'technique blocks' which group/encapsulate shader passes (Effect Technique Syntax, 2010). The code excerpt in table 22 shows how the NPD shader takes advantage of this structure; explicitly declaring shader passes for each level of tessellation that can be achieved with this implementation.

NPD Shader Structure

```
technique10 Shader_Technique
{
  #ifdef VERTEX_DEFORMATION
  pass pass_initialization {
    SetPixelShader(NULL);
    SetVertexShader(CompileShader(vs_4_0,vs()));
    SetGeometryShader(
      ConstructGSWithSO(
        CompileShader(gs_4_0, npd_initialize()), SO_FORMAT));
  }
  /*
  Tessellation passes do not rasterize and thus, no pixel shader function is assigned.
```

```
  Geometry shaders are assigned and are stream out capable ('ConstructGSWithSO'). Each
  tessellation pass invokes the same, generalized tessellation functionality
  ('npd_tessellate'). Note that each invocation to 'npd_tessellate' is parameterized
  with a value that corresponds to the pass index.
  */
  pass pass_tessellation_level_1 {
    SetPixelShader(NULL);
    SetVertexShader(CompileShader(vs_4_0,vs()));
    SetGeometryShader(
      ConstructGSWithSO(
        CompileShader(gs_4_0, npd_tessellate(1.0f)),SO_FORMAT));
  }

  pass pass_tessellation_level_2 {
    SetPixelShader(NULL);
    SetVertexShader(CompileShader(vs_4_0,vs()));
    SetGeometryShader(
      ConstructGSWithSO(
        CompileShader(gs_4_0, npd_tessellate(2.0f)),SO_FORMAT));
  }

  ...

  pass pass_tessellation_level_4 {
    SetPixelShader(NULL);
    SetVertexShader(CompileShader(vs_4_0,vs()));
    SetGeometryShader(
      ConstructGSWithSO(
        CompileShader(gs_4_0,npd_tessellate(3.0f)),SO_FORMAT));
  }

  pass pass_rasterize {
    SetPixelShader(CompileShader( ps_4_0, npd_ps()));
    SetVertexShader(CompileShader(vs_4_0,vs()));
    SetGeometryShader(NULL);
  }

  #else
  /*
  Non-deformation shader equivalent goes here.
  */
  ...
  #endif
}
```

Table 22 Code excerpt showing the 'technique structure' of the NPD shader

As table 22 shows, the tessellation function (‘`npd_tessellate`’) is invoked at each pass of the shader and is parameterized with the pass index (or tessellation level). The pass index is accessed by the ‘generic’ ‘`npd_tessellate`’ implementation, and is internally used to determine if triangular tessellation should be applied or suppressed.

If the pass index value is less than a triangle’s cumulative tessellation, the shader function proceeds to tessellate the incoming triangle. Like the previous implementation, vertex data is interpolated to yield mid-edge vertices.

Geometric deformation is applied to tessellated geometry in a similar way to the first approach. That is, deformation is applied to sub triangles as permitted by the triangles context information. If for example, the context information indicates a lower tessellation in an adjacent triangle, then generated sub triangles are clamped to adjacent geometry. If adjacent deformation is greater than or equal to the current triangle, then geometric deformation by means of ‘vertex offset’ is applied. Recall from earlier discussion that offset is achieved by evaluating a noise procedural function. The evaluated result is used to move a vertex position along the normal vector of the associated triangle.

Note that this structure requires explicit declaration of passes, which imposes a static ‘upper bound’ on the NPD’s maximum tessellation level. This limit however, is not particularly problematic as four levels of recursive tessellation tend to yield sufficient levels of geometric resolution. Strategies which involve dynamic NPD shader construction prior to shader recompilation could potentially be integrated, to provide a variable level of maximum tessellation.

The final pass shown in table 22 (‘`pass_rasterize`’) is responsible for transforming and rasterizing the final geometric tessellation of the NPD process. Thus, it incorporates functionality that is similar to standard shaders. The pass’s vertex shader projects the NPD’s tessellation geometry to homogenous (screen) coordinates in preparation for rasterization. The pass concludes with invocation of an assigned pixel shader (‘`npd_ps()`’), through which rasterization of each triangle in the tessellation result is achieved. The distinction between this pass, and that of a non-deformation (‘standard’) shader, is that it processes the ‘vertex-structures’ from the NPD process.

Supporting structure and related details

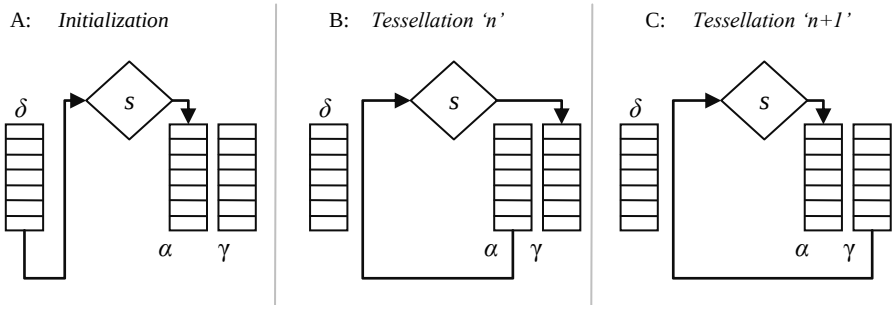
As explained, the revised NPD implementation has several differences to the original NPD implementation. These changes were motivated by a serious limitation in the first solution’s functional capacity which limited data streaming. The revised multi-pass solution overcomes

this issue, while also making better use of parallelism in the GPU architecture. Aside from shader reimplementation, this change also required significant revision of the CPU code which operates and supports the NPD shader.

As discussed, the single pass implementation could be executed by the same 'CPU support code' used to execute standard rendering shaders. Thus, the CPU simply bound geometry to the hardware device, prior to the rendering (and tessellation) phase. The shader tessellated bound geometry accordingly, and streamed the tessellation result into the GPU's rasterization unit. Thus, the entire tessellation process could be achieved by a single call to '`DrawAuto()`'; a function exposed by Direct3D 10's 'draw API' (DrawAuto Method, 2010).

In contrast, the multi-pass approach is less independent and requires greater intervention by the host system/CPU to function. The CPU must iterate and invoke each pass of the shader independently which imposes data management/manipulation responsibilities on the CPU. As discussed on page 76 in the 'interactive tool chain' section, the tool chain's game rendering context (GRC) facilitates 'standard' multi-pass shader rendering by default.

The main hindrance in the revised approach, is its dependence on auxiliary 'resource buffers' (managed by the CPU), which are required to temporarily store inter-pass tessellation data. Recall that the algorithm achieves deformation/tessellation by conditionally breaking triangles into four sub triangles. By reapplying this process to emitted triangle data, high levels of tessellation/deformation resolution are achieved. Each pass of the algorithm however, must channel generated tessellation data back into subsequent NPD shader passes.



δ: 'generic' buffer containing manifold geometry

α,γ: support buffers

Figure 74 Illustrates the role of support buffers for data flow in the revised NPD shader

As figure 74 shows, the multi pass NPD algorithm requires two data storage buffers (α,γ) to support the deformation process. The role of these buffers allows 'data circulation' between passes of the tessellation process.

Recall that the shader begins with an 'initialization' pass. As image (A) in figure 74 shows, this pass is compatible with 'generic' manifold geometry data and is responsible for inserting NPD specific data (i.e. 'tessellation context' information) into a duplication of the manifold geometry. Following this, the CPU invokes the first tessellation pass ('`pass_tessellation_level_1`' of table 22) that operates on the output of the initialization pass. Note that tessellation shader passes are only compatible with geometric data that has been pre-processed by the initialization pass.

To achieve this inter-pass flow of data, the initialization pass (image A, figure 74) streams output data into the first available 'support buffer' (α). Once this data has been captured, the destination buffer (α) is then bound to the hardware device in preparation for the next pass. In addition, the second support buffer is bound as the destination buffer for streamed data. The previously streamed data now acts as the data source for the subsequent tessellation pass (image B, figure 74). Upon completion of this tessellation pass, the second buffer (γ) is bound to the device as the data source (image C, figure 74). The first support buffer (α) is again bound to the device to capture the next 'batch' of tessellation data. This 'alternating' process repeats until all passes of the NPD shader have been executed.

This illustrates added overhead on the host application/CPU in contrast to the first NPD implementation. This is because the host application must store and manage the support buffers. In this tool chain, the GRC's rendering module (page 76) maintains responsibility for allocating and maintaining these buffers. In addition, the render module alternates the support buffers between shader passes, during the NPD rendering process.

The dependence on storage buffers effectively limits the tessellation approach to memory capacity of the hosting hardware/system. Because these storage resources are used exclusively by objects during the tessellation process however, this represents better utilization of the underlying memory, thus allowing for higher levels of tessellation/deformation detail.

Given that high visual quality is a fundamental objective in this research, the revised approach is presented as the final NPD implementation. Page 180 of the demonstrations chapter illustrates the functionality and application of this NPD algorithm, in the context of this research's tool chain.

Chapter 5: Demonstrations

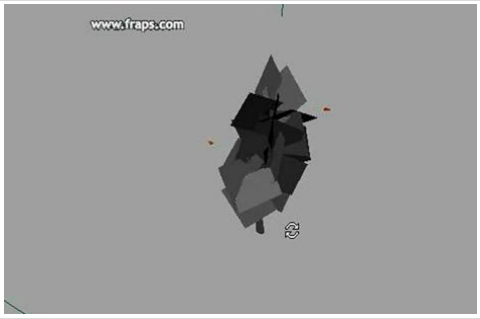
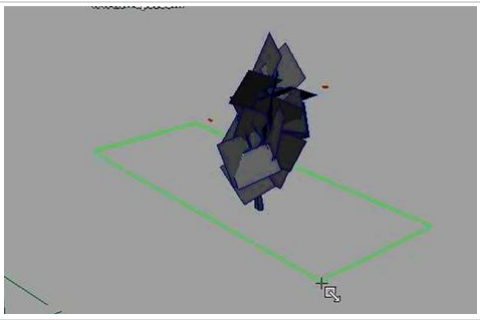
This chapter shows functionality of the interactive tool chain that was developed for this research. Each of the core functions in the tool chain are illustrated; namely procedural material composition, real-time generative instancing and non-uniform procedural deformation. These algorithms are demonstrated through different examples that could be appropriate for computer games.

Images in the leftmost column represent the artist's view of a game scene/content in the context of the tool chain's authoring environment (Maya/'real-time content encoder'). They show views and media which are visible at various stages during the authoring process.
The rightmost column shows the reproduction of corresponding content in the tool chain's 'Game Rendering Context' (GRC). These images show how the system responds to artist interaction in the tool chain. As discussed, the GRC incorporates specialized functionality which allows rendering of content in a real-time, game specific rendering context. The images in this column exhibit the implementation of the procedural methods used.

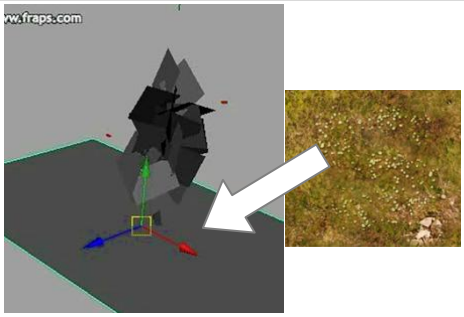
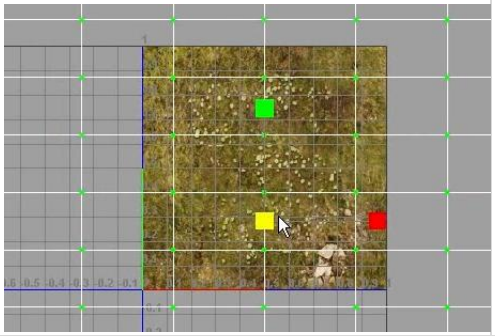
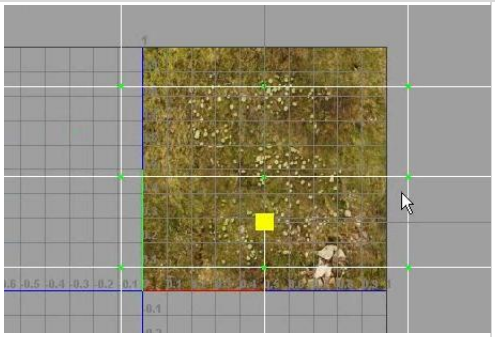
Tool chain interaction and material composition

Basic geometric manipulation of game content in the GRC is illustrated by the following diagrams. The sequential organization of these images demonstrates the interactive and responsive nature of the tool chain. The example shows how content can be built and manipulated in the tool chain.

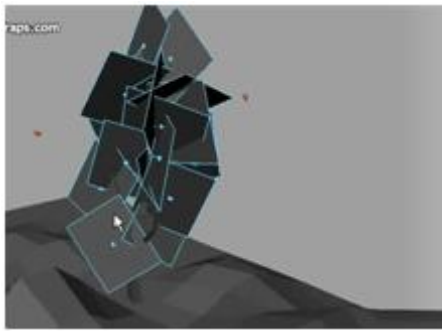
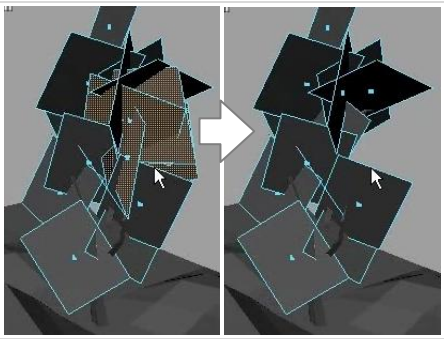
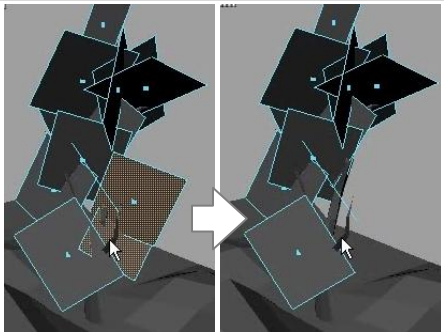
The initial images show ‘terrain’ being built in the ‘context’ of an existing game content (i.e. a tree).

	RTCE (Maya)	GRC (Game renderer)
1		
	Existing ‘scene element’ is loaded into Maya.	The GRC immediately synchronizes with the RTCE’s state. The GRC renders the tree as it would appear in the game.
2		
	The RTCE transmits geometry during it’s insertion into the scene, by the artist (ground plane).	The geometry being inserted is interactively shown in the GRC.

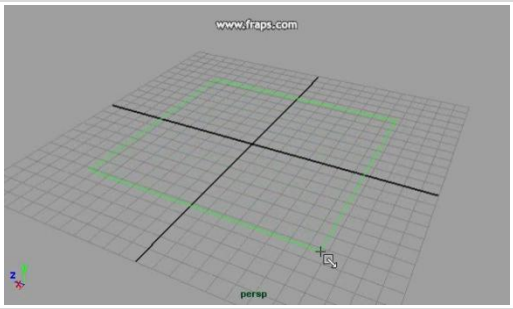
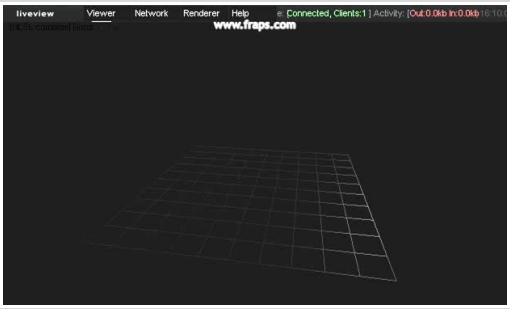
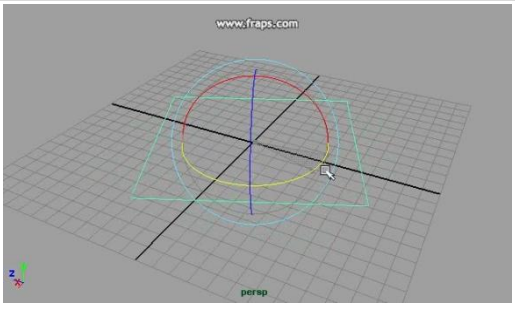
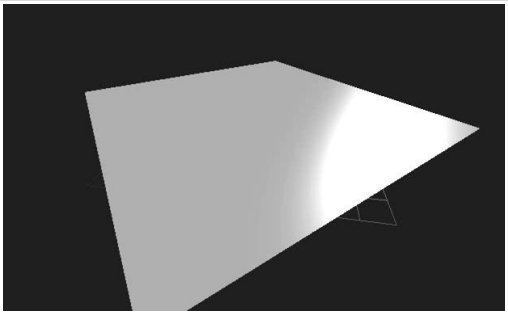
In addition to geometry, the tool chain synchronizes other ‘channels’ of game object data. The following shows how content in the GRC immediately reflects an artist’s assignment of texture data and texture coordinates, in the RTCE.

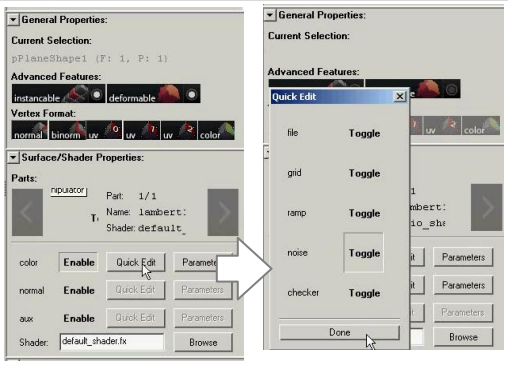
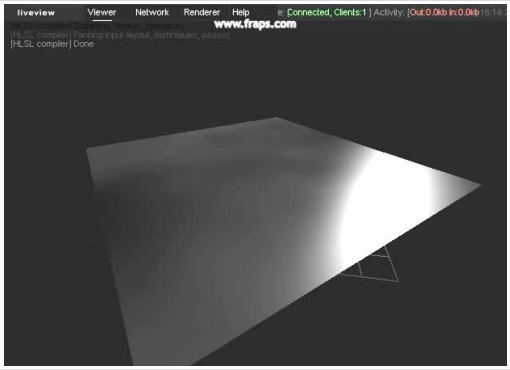
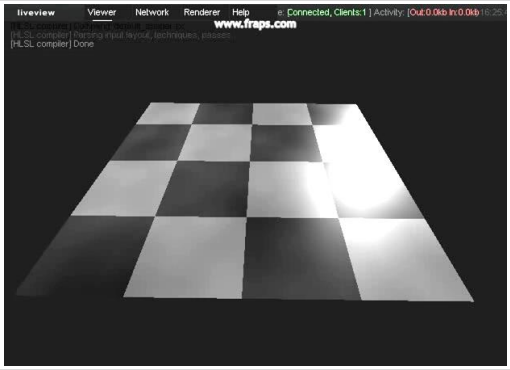
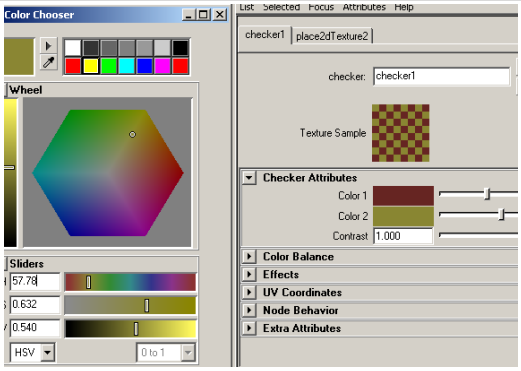
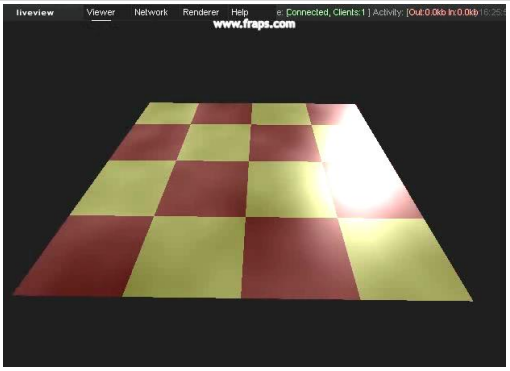
3		
	The RTCE responds to the association of texture data with geometry by transmitting relevant data to the GRC.	The relationship between texture data and the ground surface is immediately displayed in the GRC context.
4		
	As mentioned, the tool chain responds to artist manipulation of texture coordinates. This shows the artists' view of the ground geometry's texture coordinates in Maya.	The 'mapping' of the ground's texture image corresponds to the ground's underlying texture coordinates. Note the 'scale' of the ground surface texture.
5		
	From Maya, the artist has increased the scale of the texture coordinates which underlie the ground geometry.	The GRC interactively responds to these changes. Note that the texture image on the ground surface appears to have 'condensed'.

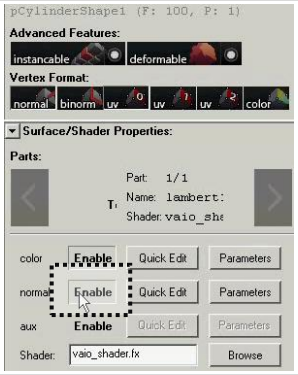
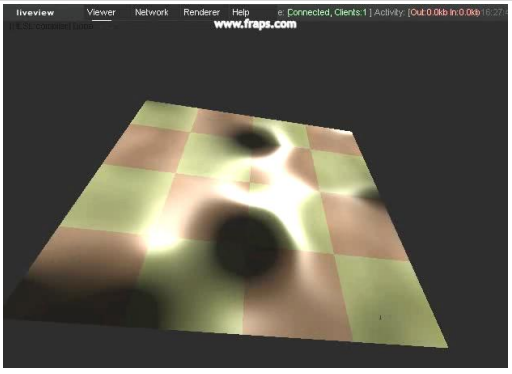
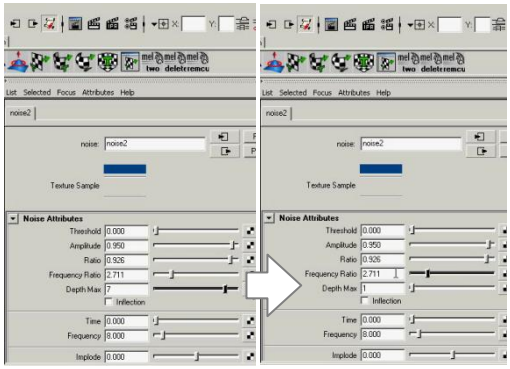
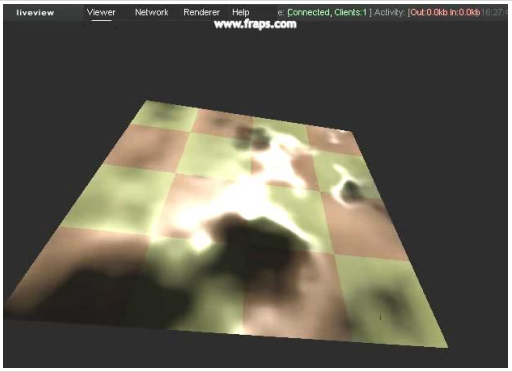
The tool chain is robust and facilitates direct/explicit modification of game objects. As mentioned, modifications to game objects are immediately reproduced in the GRC. The following images demonstrate the tool chain's interactive response when an artist removes geometric elements from the tree object.

6		
7		
	The artist selects and removes elements of the game object in the context of the RTCE/Maya.	The highlighted region shows how the interactions manifest in the GRC's rendered result.
8		
		The changes are consistently and interactively displayed.

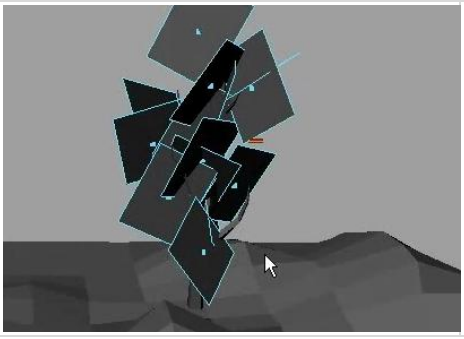
The next example demonstrates the use of the tool chain's material composition feature. An abstract example is presented which shows the sequence of artist interactions required when composing a simple material. Practical examples of the composition feature are shown subsequently.

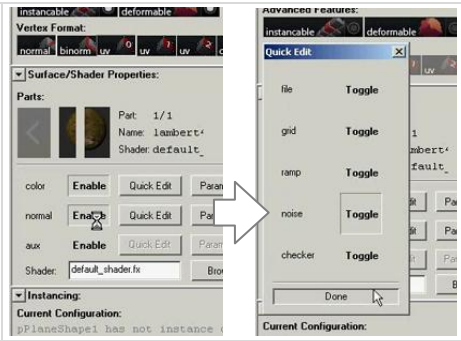
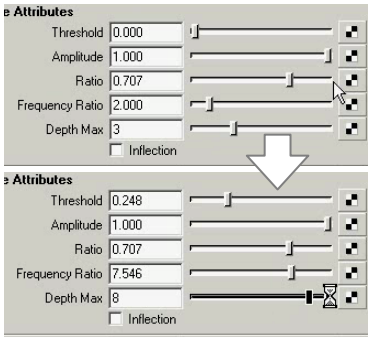
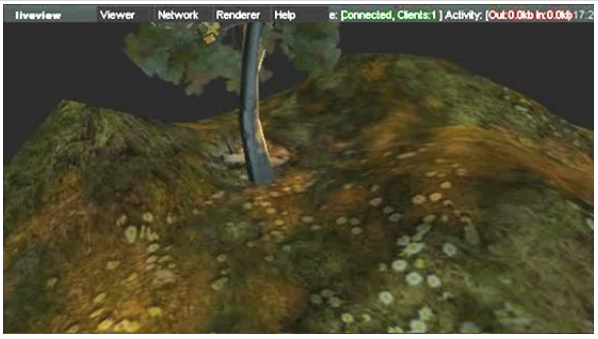
1		
	A geometric surface is created in Maya to facilitate the composition demonstration.	
2		

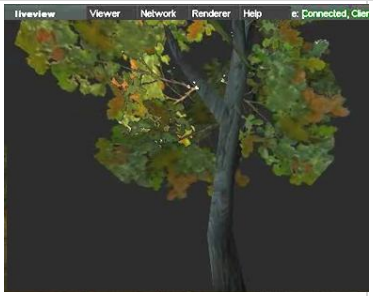
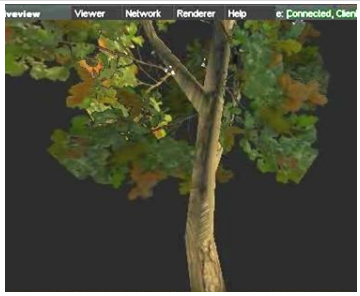
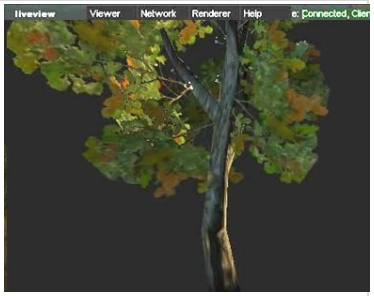
3		
	This shows a portion of the RTCE's 'material composition' interface. Material composition starts by 'enabling' the colour channel. Following this, the RTCE's 'Quick Edit' feature is invoked, to assign a noise procedural to the material's colour channel.	The GRC immediately reflects changes to the material's composition. Perlin noise is now displayed across the geometric surface. Note that the noise is interpreted as 'colour' across the geometry.
4		
	A 'checker' procedural is then introduced to the colour channel, again via the RTCE's material composition functionality.	The GRC interactively combines the noise and checker procedurals producing a result that corresponds to the specified 'material composition'.
5		
	Changes made to parameters of the composition's procedural functions are interactively transmitted.	These parameter changes (i.e. checker colour) are immediately reflected in the GRC.

6		
	The 'normal channel' is activated in the RTCE. Procedural functions can now be interpreted as 'surface normals' to allow different surface characteristics to be achieved.	The GRC reproduces the new material composition. The surface now exhibits characteristics that simulate surface contour by manipulating corresponding surface normals by the assigned procedural function.
7		
	Changes made to parameters of the noise procedural assigned to the material's 'normal channel', are interactively transmitted to the tool chain via the RTCE.	The surface's material updates, to reflect the parameter changes, increasing the noise's granularity (or 'octaves').

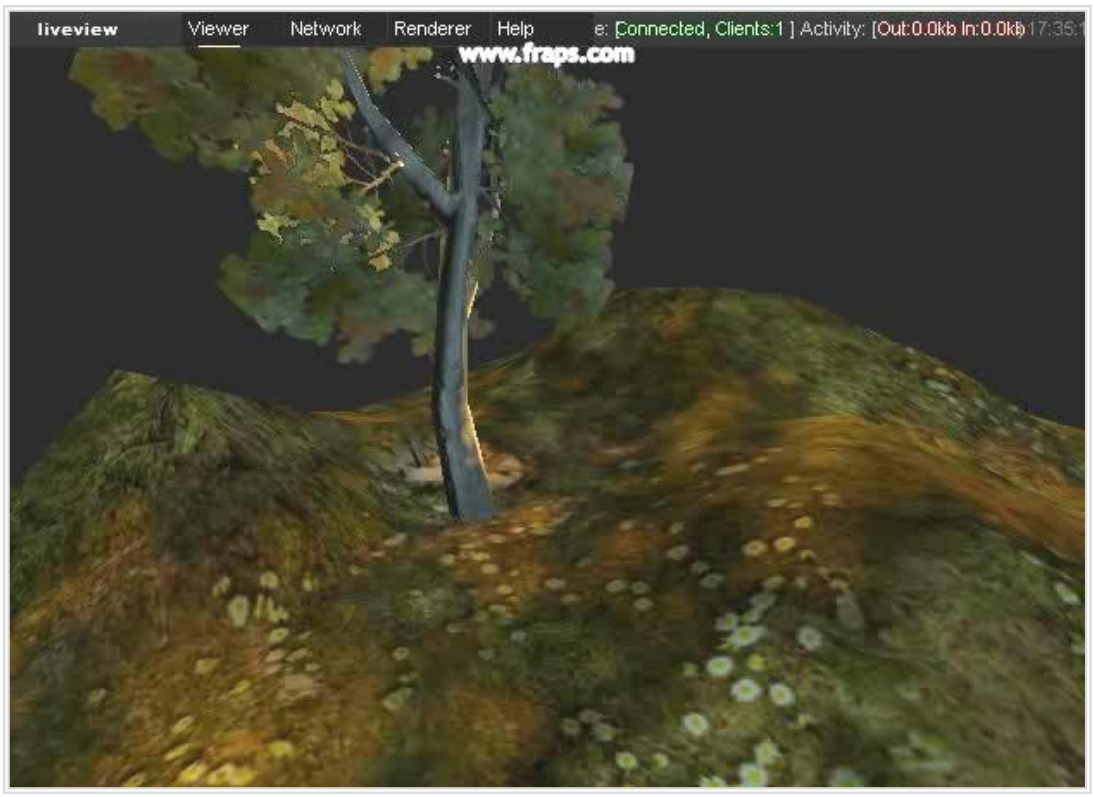
Additional surface detail can be achieved in game scenes via the tool chains procedurally based material composition system.

1		

2		
	Procedural noise is added to the ‘normal channel’ of the ground geometry’s material.	Visual reproduction of the ground surface changed by the influence of the noise function assigned to the ground material’s ‘normal’ channel.
		
	The artist increases the noise function’s ‘frequency’ and ‘depth’ (granularity) parameters. The RTCE detects these changes and immediately transmits the corresponding data from Maya. These interfaces are built in to Maya and reused by the RTCE.	Changes to the noise’s ‘frequency’ and ‘depth’ parameters are reflected by the GRC’s reproduction of the ground material. Note the enhanced appearance of ‘bumps’ across the ground surface, which results from the noted parameters changes.

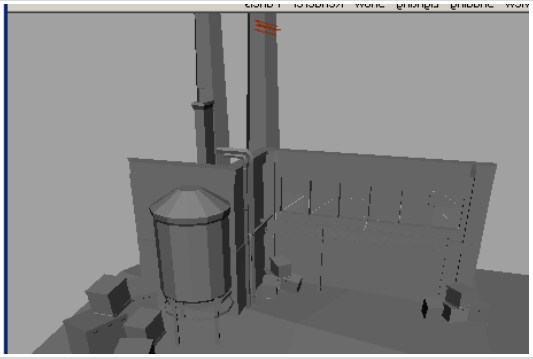
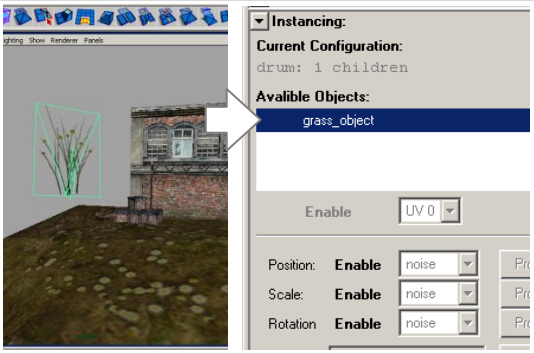
1	2	3
		

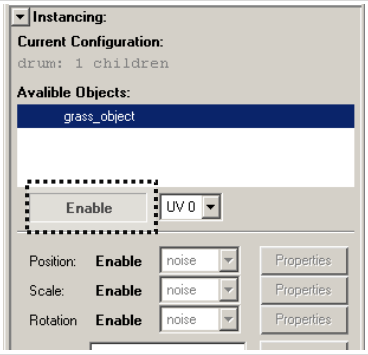
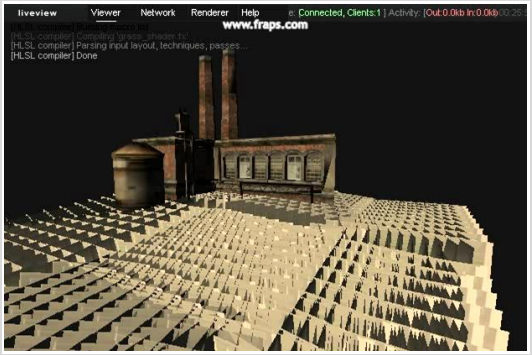
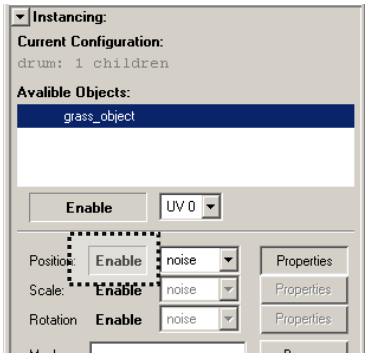
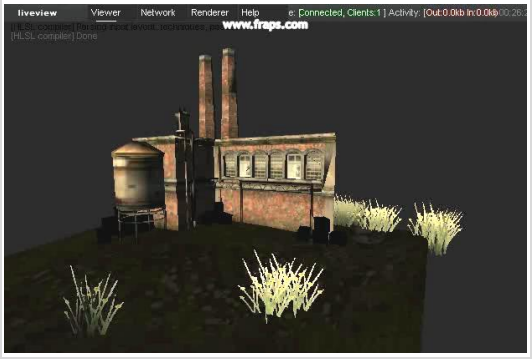
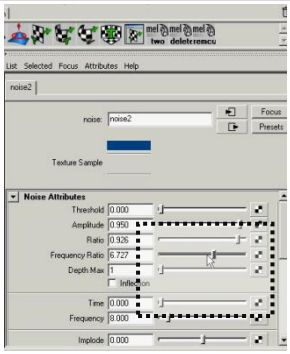
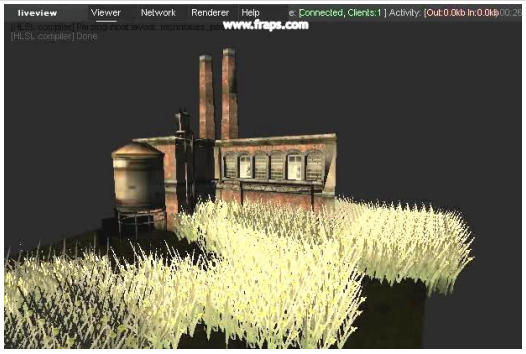
These images show how material composition can be applied to other aspects of the scene. Here the feature is applied to the tree trunk to achieve a more detailed and realistic final appearance in the GRC’s rendered result.

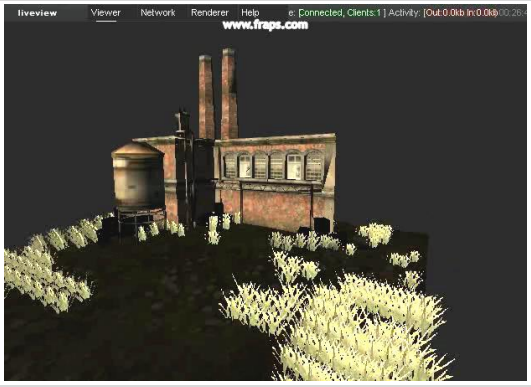

liveview
Viewer
Network
Renderer
Help
e: Connected, Clients:1] Activity: [Out:0.0kb In:0.0kb
www.fraps.com

Real-time generative instancing

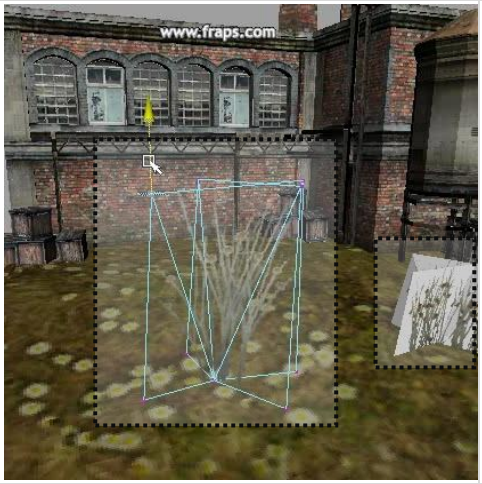
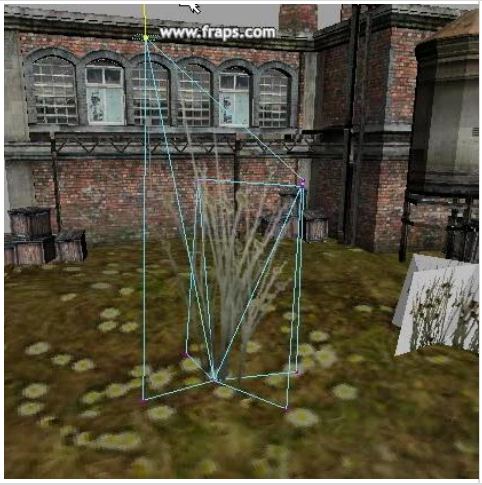
The following images demonstrate the use of the procedurally based, real-time instancing algorithm that was integrated into the tool chain (RTGI). In this demonstration, the algorithm is used to introduce overgrown grass in an industrial setting.

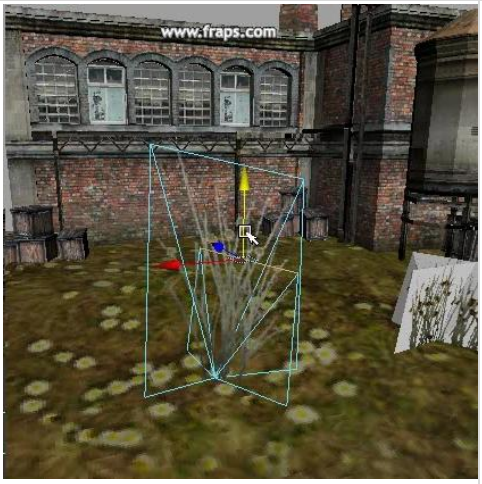
	RTCE (Maya)	GRC (Game renderer)
1		
	An existing, partially constructed scene is loaded into Maya. Elements of this scene are transmitted to the GRC as determined by the artist.	This image shows the first items of the scene that have been sent to the GRC from the RTCE.
2		
	A grass object represents the geometry that will be used in the instancing process. The object is 'assigned' for instancing, via the RTCE's interface. Note that the geometry can exist in the context of the whole Maya scene.	All scene objects have now been sent. The GRC does not render the grass object because it is reserved for instancing.

3		
	The artist ‘enables’ instancing of the grass object via the RTCE.	The GRC immediately responds, distributing the grass object across the targeted ground geometry of the scene.
4	A ‘custom shader’ developed by the artist is assigned to the ‘grass object’ via the RTCE interface. This shader integrates into the tool chain’s shader system. The shader expresses an ‘alpha channel’ in the grass to clip portions of the grass object. The effects of this shader are immediately shown in the GRC.	
5		
	Via the RTCE interface, a noise procedural function is activated (‘Enabled’) for the RTGI’s ‘mask channel’. This controls the placement of grass instances via evaluation of the noise procedural function.	The distribution of grass instances is now ‘irregular’ and has a more ‘natural’ appearance. This distribution corresponds to the noise procedural function that was applied to the ‘mask channel’ of the RTGI’s application.
6		
	As the artist modifies parameters of the mask procedural, these changes are interactively propagated through the tool chain.	Changes to the mask procedural’s parameters are immediately shown in the distribution of grass across the scene’s ground surface.

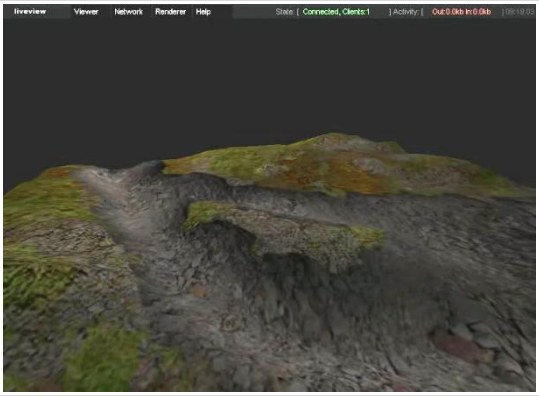
7		
	These images show the instancing result following further refinement to the parameters of underlying procedural functions. In addition, these images show how variety can be achieved via procedural functions that are assigned to the scale and orientation channels of the RTGI algorithm.	

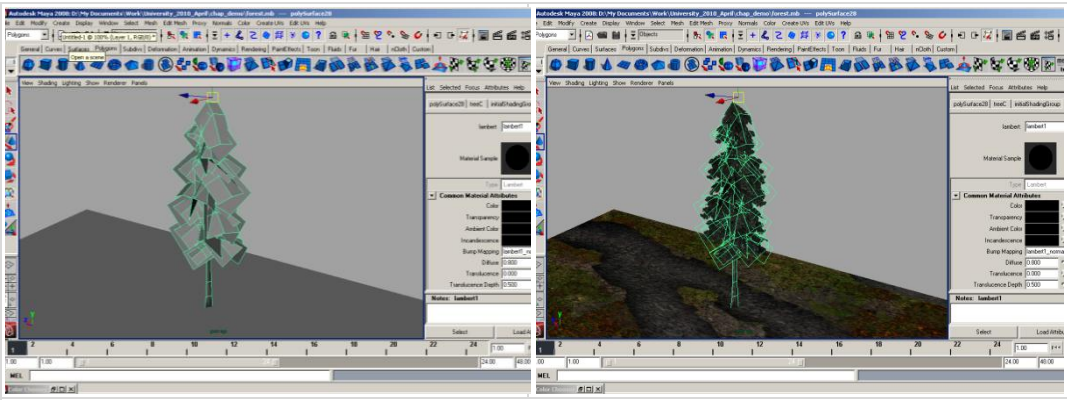
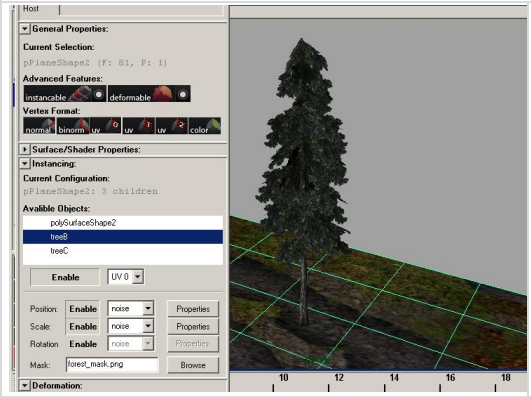
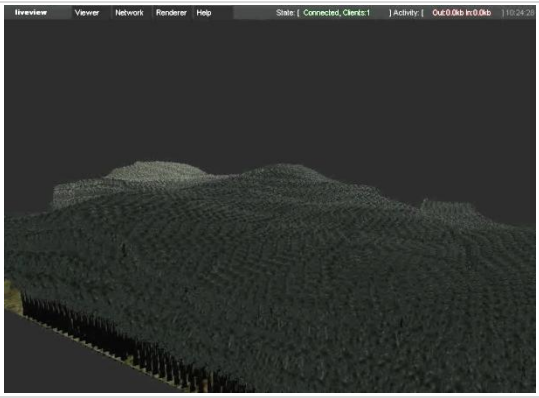
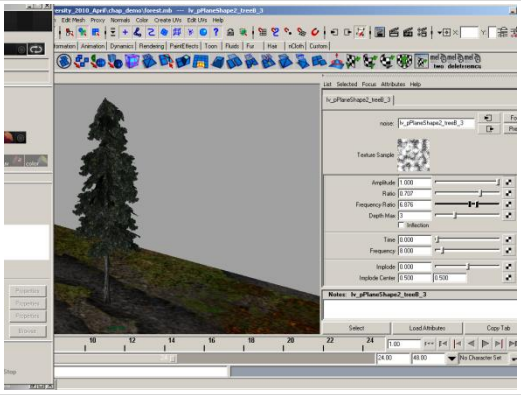
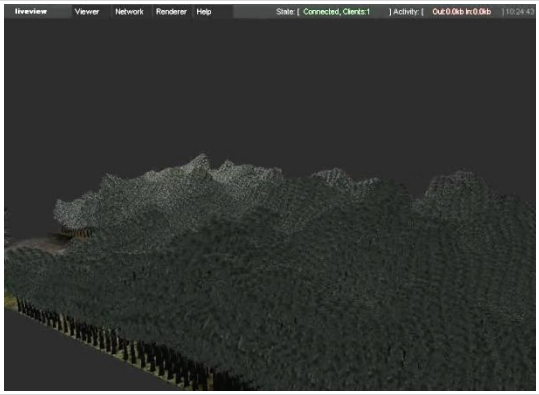
The following example demonstrates the RTGI algorithm's robust and flexible integration into the interactive tool chain. These images show how instanced geometry can be manipulated in the context of the interactive tool chain, providing a powerful content creation mechanism for artists.

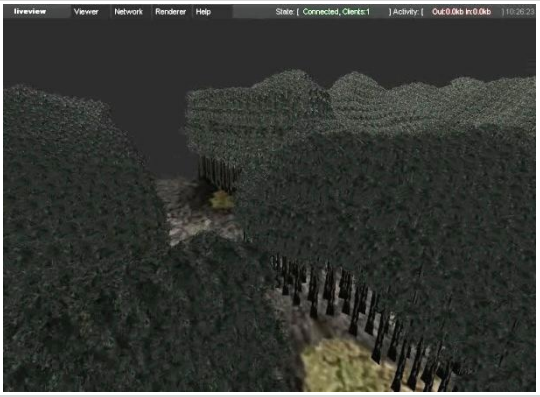
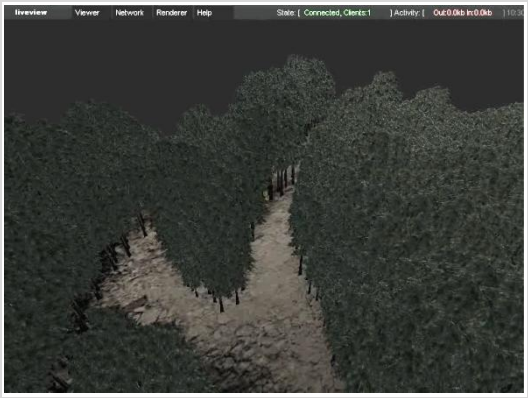
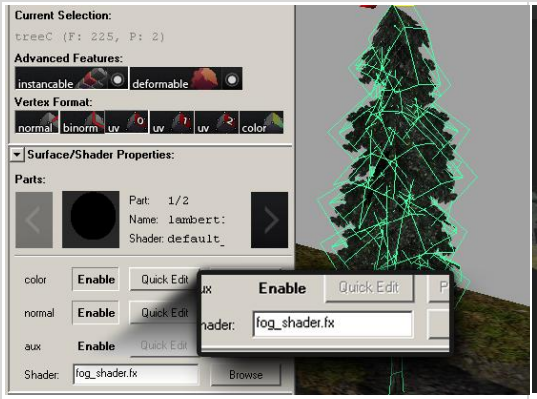
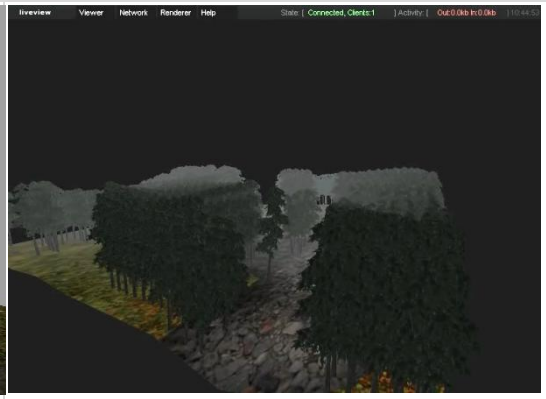
8		
	This image represents the game scene in Maya. Note that both of the highlighted grass objects are now being instanced via the RTGI algorithm.	This shows the GRC's ability to instance multiple object 'types' across a surface. Here the two grass objects selected in Maya, are instanced across the scene's ground surface.
9		
	The RTCE captures and transmits all modification made by the artist to the grass geometry. Here, the grass object's geometry is being modified.	The GRC interactively responds to this modification. The changes are immediately displayed throughout all instances of the grass object. The corresponding grass object is now visibly taller.

10		
	Artist interactions and modifications to the grass objects are consistently transmitted during runtime.	Grass that is instanced by the RTGI algorithm continues to reflect the artist's interactions during runtime.

The following sequence shows how the RTGI's 'cookie cutter' feature can be used to prevent geometry from instantiating at specific regions on the manifold surface. The feature is demonstrated in the context of a 'forest scene'. In this situation, the scene's terrain consists of artist prescribed pathways. The 'cookie cutter' feature is used to prevent trees from 'violating' the scene's pathways.

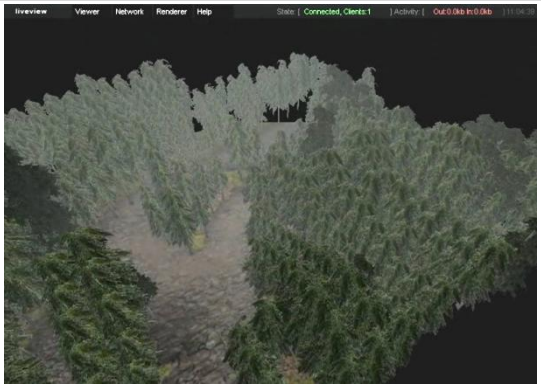
	Maya (RTCE)	GRC (Game renderer)
1		
	Terrain that underlies this scene is created in the context of this research's tool chain (via Maya). The image shown is the texture image which is directly mapped onto the scene's terrain geometry.	This image shows the GRC reproduction of the scene geometry created in Maya.

2		
	Tree objects that will constitute the forest of the game scene, are also created in Maya. These tree objects will be instanced by the RTGI algorithm to generate the forest automatically.	
3		
	Again, the tool chain's RTGI functionality is invoked. The tree object is assigned to the scene's terrain geometry. Note that procedural noise has also been assigned to dictate scale and masking of tree instances.	The GRC immediately renders the corresponding RTGI configuration for instancing. In this instancing example, the density of instancing has been maximized. Note the 'wavy' appearance of instanced trees. This is a manifestation of the noise function that is assigned to 'scale' each tree instance.
4		
	Here, the artist increases the 'frequency' parameter of procedural noise function that is assigned to the RTGI algorithm's 'scale channel'.	The GRC immediately reflects this parameter change. Note that the 'frequency' of scale variation between tree instances is visibly greater.

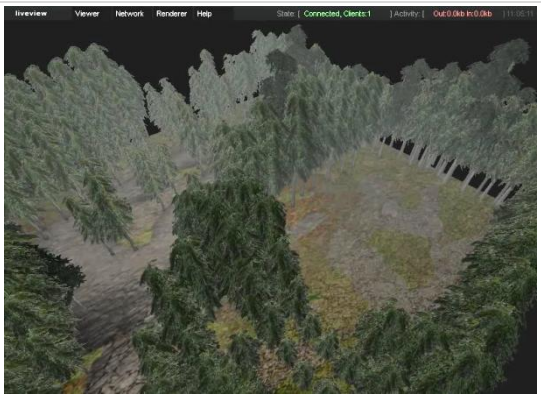
5		
	The RTGI's 'cookie cutter' feature is invoked to prevent trees from instantiating over 'pathways' which exist on the terrain's surface texture. This image has been created and specified by the artist as the 'instancing cookie'. Note that it corresponds to pathways in the original terrain texture.	The GRC immediately responds to the specified cookie data/image. Note that the instancing of tree's now corresponds to the supplied cookie cutter.
		
	These images show the effect of the cookie cutter image in this scene from different vantage points.	
6		
	To add atmosphere to the scene, a customized 'fog shader' (which again uses the tool chain's shader system) is applied to the scene's geometry. Note that this shader is developed/implemented by the artist.	The results of this fog shader are immediately reproduced in the GRC.

The following images provide further illustration of the RTGI algorithm's cookie cutter feature. Two different tree objects/types are instanced across the terrain's surface via the RTGI algorithm. Each 'tree type' is subjected to different procedural function parameter

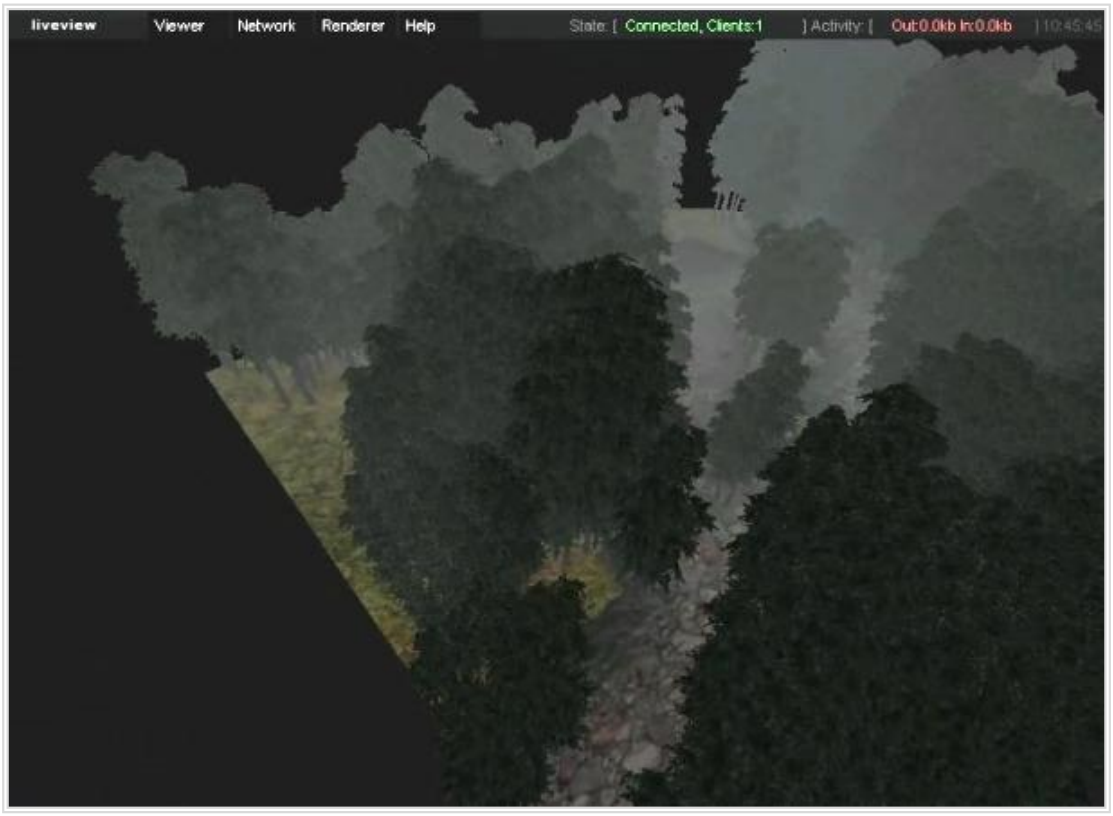
configurations. These images also show how the cookie cutter feature can be applied to different instancing objects.



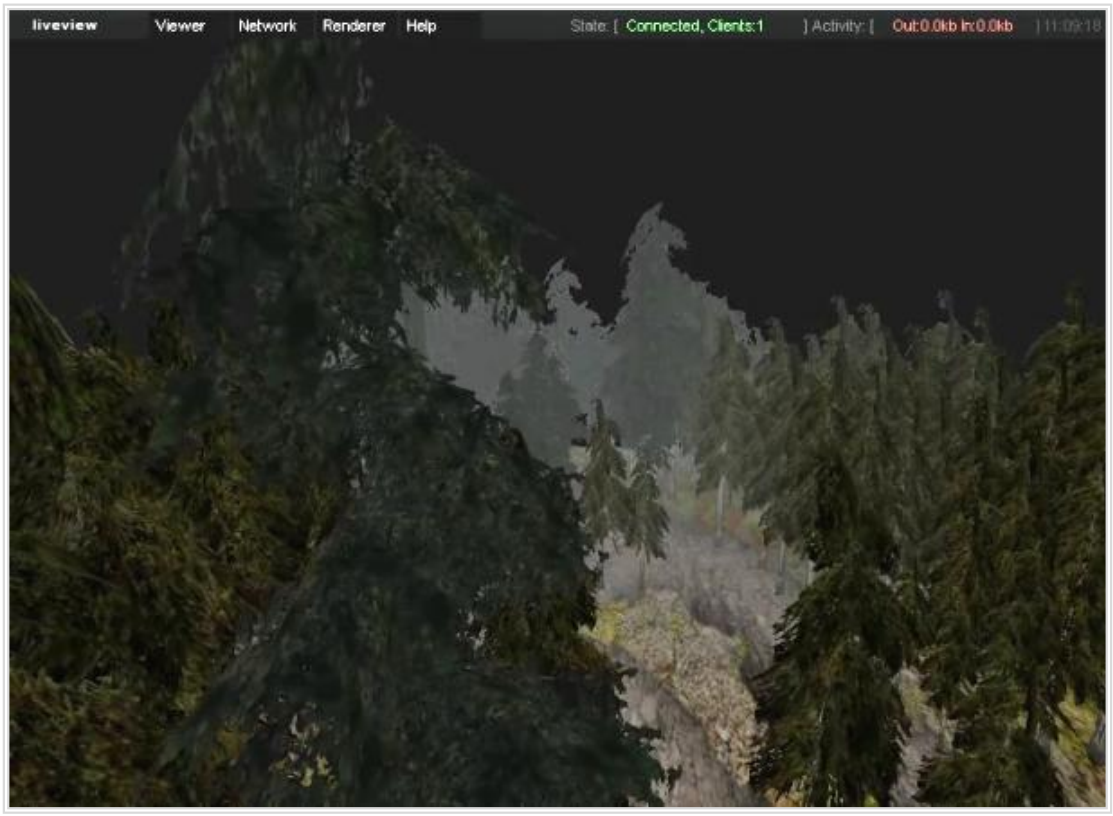
Two tree types are instanced via the RTGI algorithm. Note that each tree type is instanced by distinct ‘procedural parameters’. This causes each tree type to be uniquely distributed.

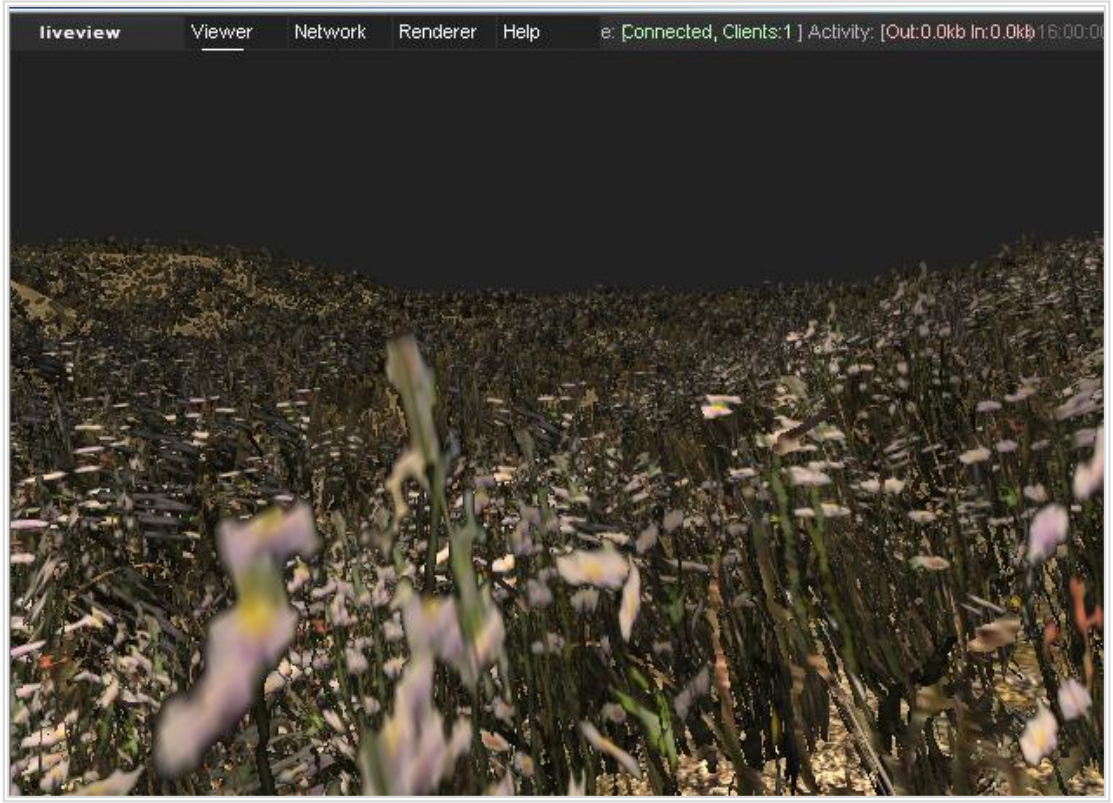


Note that the rectangular feature of the cookie cutter image is manifest in the GRC’s rendered result. Furthermore, both tree types are subjected to the cookie cutter image, as determined by the artist.


liveview
Viewer
Network
Renderer
Help
State: [Connected, Clients:1] Activity: [Out:0.0kb In:0.0kb] 10:45:45

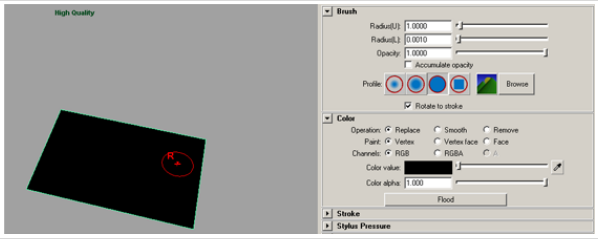
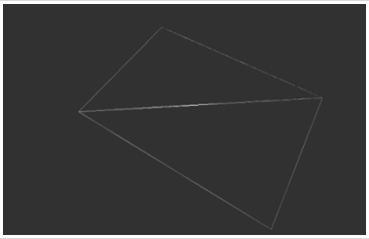
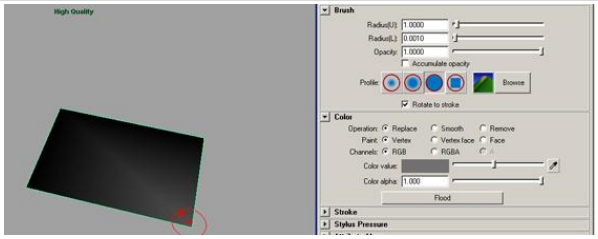
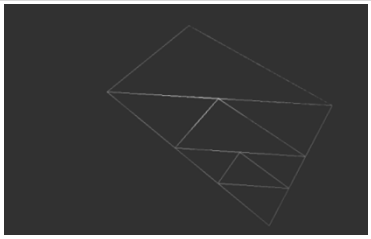
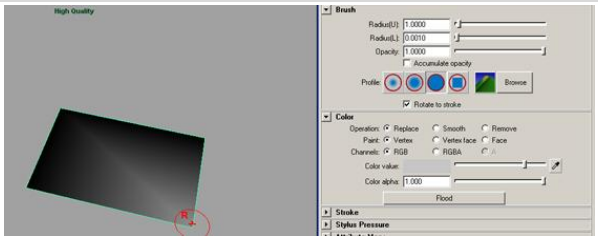
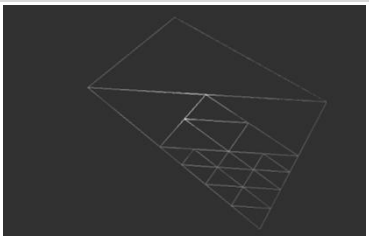
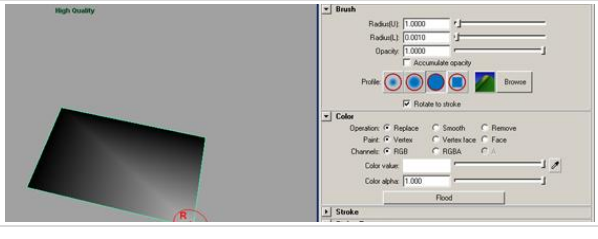
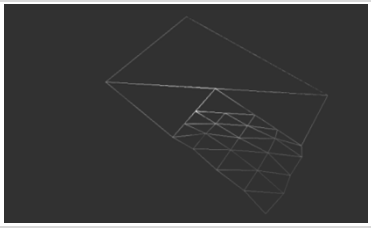

liveview
Viewer
Network
Renderer
Help
State: [Connected, Clients:1] Activity: [Out:0.0kb In:0.0kb] 10:45:59


liveview
Viewer
Network
Renderer
Help
State: [Connected, Clients:1] Activity: [Out:0.0kb In:0.0kb

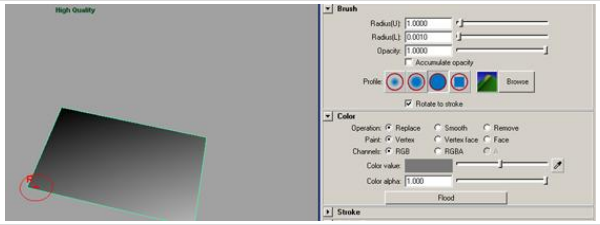
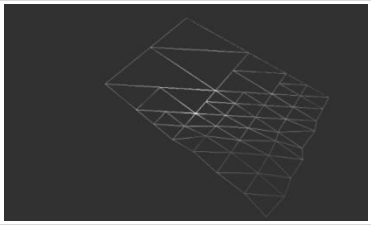
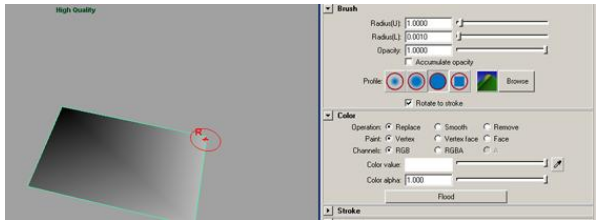
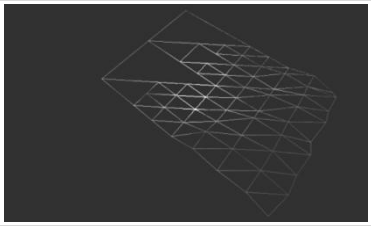
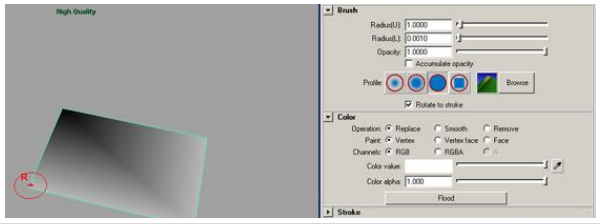
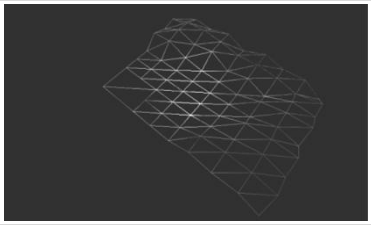
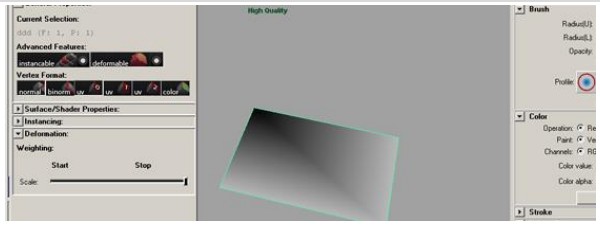
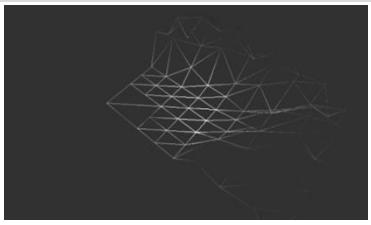
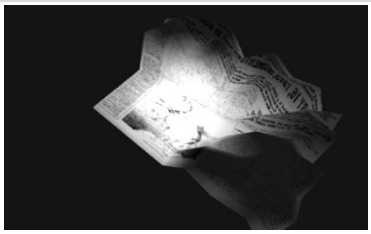

liveview
Viewer
Network
Renderer
Help
e: Connected, Clients:1] Activity: [Out:0.0kb In:0.0kb

NPD algorithm demonstrations

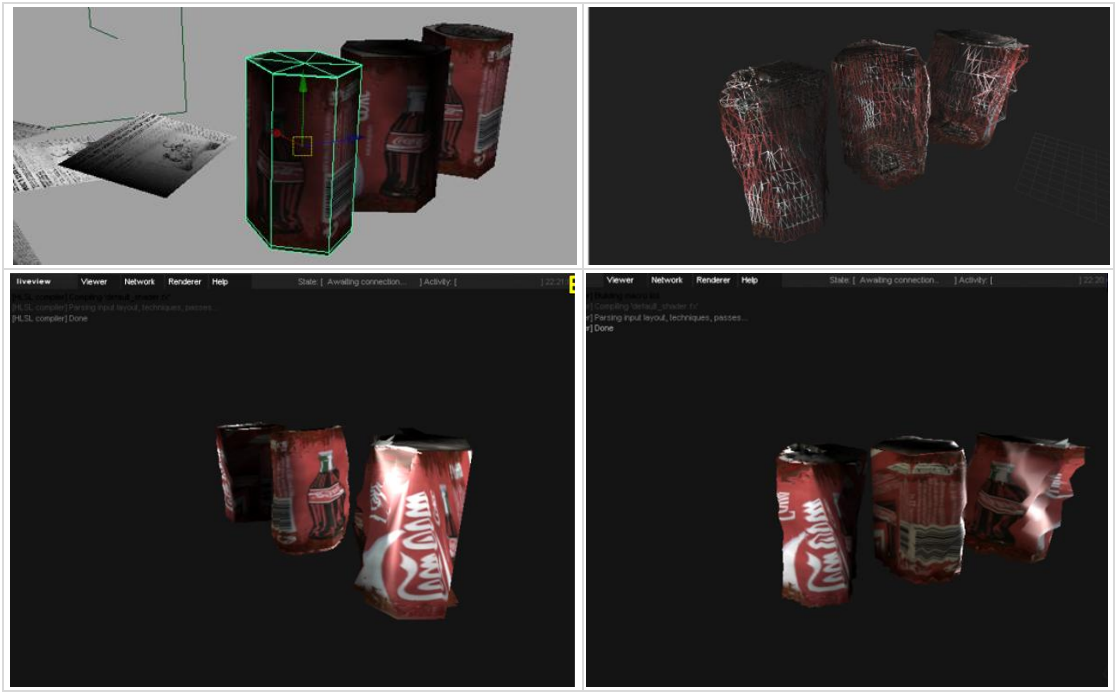
The following images show the sequence of interactions that are required in order for artists' to take advantage of the tool chain's procedurally driven, geometric deformation algorithm (NPD). The initial images show how NPD can be used to algorithmically generate variety amongst props that could be used to add detail to a game scene (i.e. rubbish).

	RTCE (Maya)	GRC (Game renderer)
1		
	The RTCE's deformation functionality has been activated. Maya enters into the 'vertex painting' mode. Note that the assignment of 'black' (or 'zero') deformation to the geometry, suppresses tessellation during the NPD process.	This shows the corresponding geometry in the GRC. The GRC's renderer has been set to 'wireframe mode' in order to show the geometric tessellation which results from the NPD process.
2		
	The artist assigns a dark shade of grey to a vertex on the geometry. As discussed, the brightness of colouration dictates the level of tessellation. As the following images show, tessellation in the GRC's reproduction of the geometry, corresponds to the brightness and distribution of 'deformation weightings' (colour) in Maya's view of the geometry.	The GRC's reproduction of the geometry algorithmically 'tessellates' geometry about the vertex that was 'coloured'. Note the non-uniform distribution of tessellation in the geometry.
3		
4		

At this stage, the algorithm detects a high enough deformation weighting to permit actual deformation to the tessellated geometry. Tessellated portions of geometry in the GRC are visibly offset from their original positions.

5		
6		
7		
8		
	The RTCE's interface allows the artist to specify the amplitude of geometric deformation. For demonstration purposes, the amplitude has been maximized.	The GRC interactively reproduces this change, causing the geometric deformation to increase.
9		

Under normal circumstances, the GRC renders geometry in 'non-wireframe' mode. This image shows how procedurally deformed rubbish/newspaper would appear. Note that the NPD algorithm introduces additional geometry (tessellation) to the prop's original geometry (shown in Maya). The following images show how the NPD algorithm was applied to other 'rubbish' props. Note that variation between instances of the rubbish is evident, despite the original 'base geometry' being identical.



This example demonstrates the integration of NPD into a game scene, via props and scene elements. Images 10 to 14 of this sequence show stages in the construction of a game setting/ context which apply NPD.

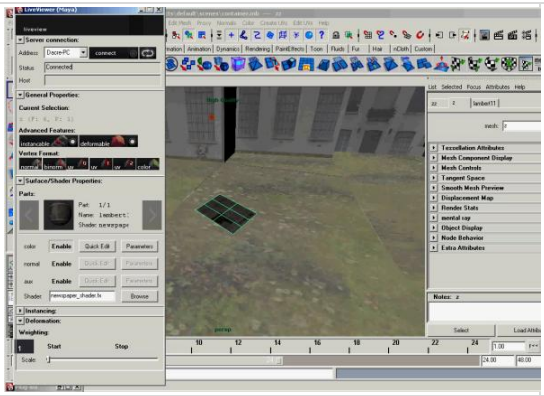
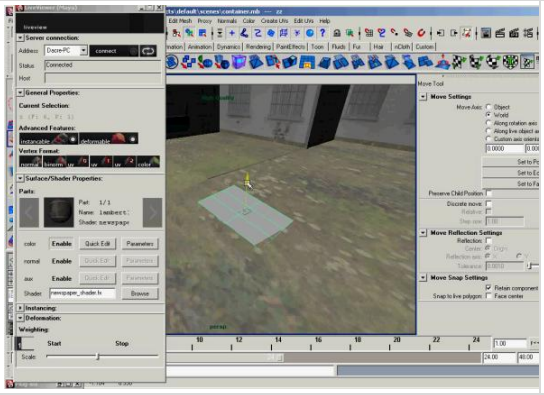
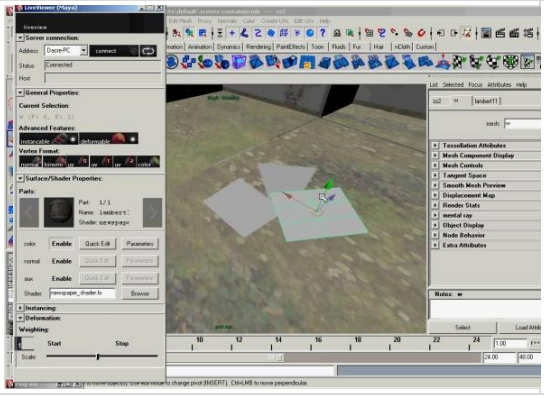
10

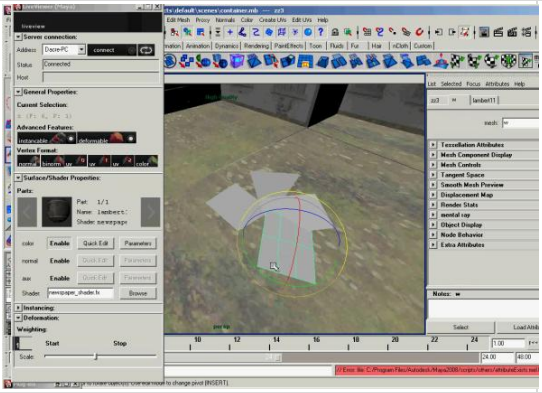
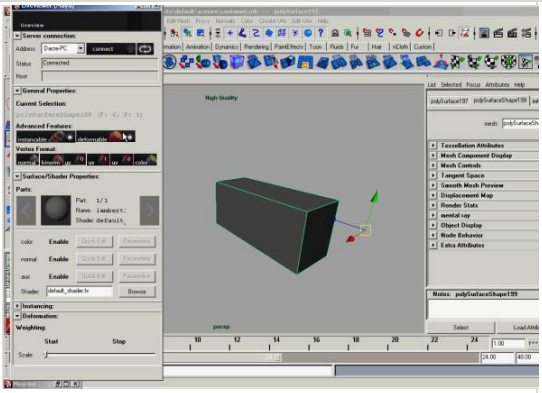
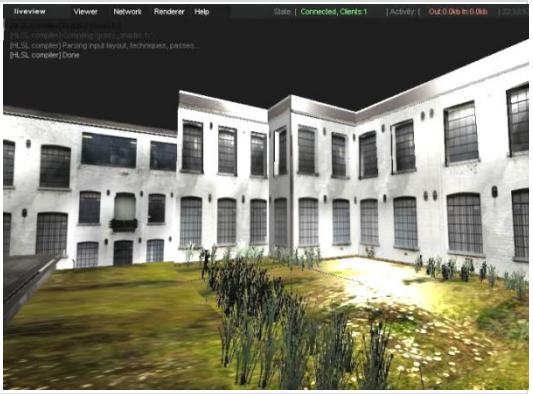
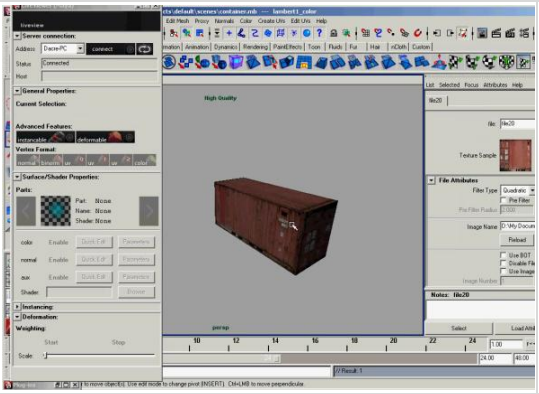
11

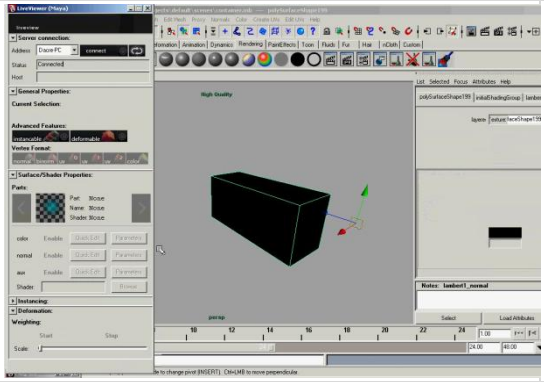
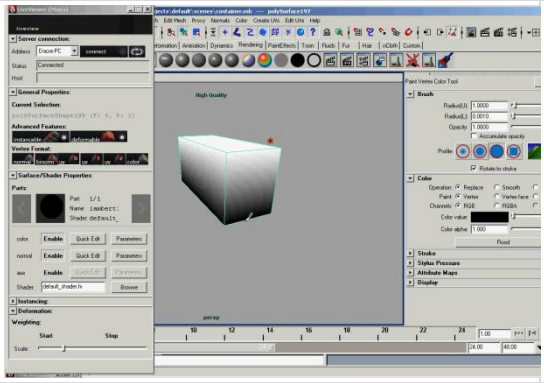
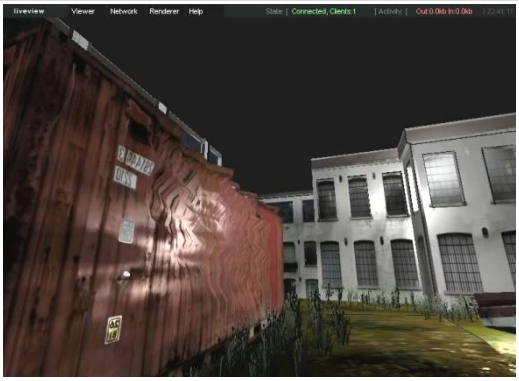
12

14

Image 14 represents the final game context.

15		
	The newspaper/rubbish prop that was previously developed is inserted into Maya's instance of a game scene.	The tool chain responds to this interaction, immediately inserting a corresponding 'rubbish object' into the GRC.
16		
	Deformation weightings (colour) are painted uniformly across the newspaper prop via Maya/RTCE functionality.	The tool chain interactively displays the effect of the deformation weightings. The newspaper now exhibits geometric variation.
17		

18		
	Images now show the same rubbish prop being duplicated throughout Maya's instance of the game scene.	The GRC interactively responds to this, inserting game objects that correspond to the interactions in Maya. Note that each instance of the newspaper in the GRC exhibits unique geometric variation, which is driven by the NPD algorithm.
18		
19		
	A 'shipping container' prop is constructed and introduced into the game scene. This prop serves as a good candidate for the NPD algorithm as 'geometric damage' can be algorithmically simulated across the object.	

20		
	The RTCE's deformation painting functionality is again activated to apply algorithmic deformation to the 'container prop'. The process begins with no deformation being assigned to the object (i.e. the 'black' overlay).	The GRC's reproduction of the shipping container corresponds to the low deformation weightings assigned in the RTCE. Thus, the original 'form' of the prop is preserved.
21		
	The artist 'digitally paints' high deformation weightings to the upper portion of the shipping container, via the RTCE/Maya.	The tool chain responds to the artist's interactions in real-time. Deformation is interactivity introduced to the shipping container. Note that the distribution of geometric deformation corresponds to the distribution of deformation weightings in the RTCE.


liveview
Viewer
Network
Renderer
Help
State: [Connected, Clients:1] Activity: [Out:0.0kb In:0.0kb] 22:41:06


liveview
Viewer
Network
Renderer
Help
State: [Connected, Clients:1] Activity: [Out:0.0kb In:0.0kb] 20:25:28
[HLSL compiler] Compiling 'default_shader.fx'
[HLSL compiler] Parsing input layout, techniques, passes...
[HLSL compiler] Done

Chapter 6: Conclusion

This research aimed to contribute towards the trend of increased quality and realism in computer game experiences via content creation strategies based on procedural methods (PM). The introduction illustrated a number of emerging constraints currently limiting improvement in game graphics. In particular, the space complexity constraint was noted for becoming increasingly important due to significant growth in the console gaming market. This is because consoles have static hardware specifications and thus, technological limitations become significant especially towards the end of a consoles lifetime.

As identified in the literature review, a correlation between improvements in game graphics and increased volumes of underlying graphics data is evident. This indicates a requirement for larger volumes of graphics data to improve the visual quality of games. The literature review discussed how increased data is not only difficult due to capacity constraints of gaming hardware, but also showed how it tends to significantly increase the game artists production workload.

If the trend for increased graphics quality/data extrapolates, production milestones will inevitably exceed feasible workloads for standard production timelines. The implications of this are longer production cycles for games and/or larger game development teams; both of which are economically unfavourable for developers.

An ‘artist centric’ content creation workflow was developed during this research, which incorporated algorithms and concepts tailored to counteract these emerging constraints. To achieve this, the tool chain integrated procedural functionality which played a central role in the delivery of the ‘tool chain’s’ results, namely ‘real-time generative instancing’ (RTGI) and ‘non-uniform procedural deformation’ (NPD). As discussed in the literature review, procedural functions have a number of attractive qualities, such as a low memory to data output ratio. In addition, PM’s are capable of delivering a diverse range of characteristics in generated data, via parameterization.

The literature review chapter illustrated how this is achieved, by investigating the implementation of ‘Perlin’ noise; a prominent procedural function which algorithmically generates data with ‘seemingly natural’ characteristics. The ‘functional’ nature of PM’s was also noted. This provided a basis of understanding, explaining the suitability of PM’s in this research, which is based on the parallelized GPU architecture.

Because the implemented algorithms (i.e. RTGI and NPD) operate on the GPU in real-time, any changes that are made to parameters of underlying procedural functions can be immediately reflected in accordance with artist interaction. Through this, the tool chain essentially achieves 'real-time interaction'. Thus, artists can interactively refine procedural parameters to alter the manifestation of procedurally generated data in these algorithms. This avoids 'overhead penalties' during the process of 'refinement', which is characteristic of 'traditional' content creation workflows. Thus, the amalgamation of GPU technology with PM's not only aligns with the constraints/themes of this research, but also compliments the level of interaction/feedback sought in an artist's tool chain.

GPUs represent a massive processing resource, which is well established and 'local' to all modern gaming hardware. Thus, it was appropriate to utilize this processing bandwidth by implementing the algorithms of the tool chain in this technological context.

The literature review also identified qualitative characteristics of the interactive tool chain paradigm, which are beneficial towards artists' workflow, through 'context relevant' visual feedback.

As indicated, traditional workflows are based on the 'manual' propagation of games content/data through a series of content authoring tool(s), to the target game/engine. This hinders both the production process, as well as the extent to which 'visual feedback' is provided to artists during content production. In these workflows, interactive visual feedback is limited to that provided by the content creation tool (i.e. Maya's 'modelling viewport').

In response to this, a core premise of the research was derived; that artist centric, 'interactive content workflows' can improve the efficiency, capacity and quality of an artist's content production via real-time flow of content data between authoring tool(s) and game rendering technology.

Such a workflow allows artists to view the content in a 'technologically relevant context' (i.e. the game renderer). Furthermore, the low penalty of viewing content modifications in a relevant rendering context via this workflow encourages refinement to games content during creation, while also promoting 'content prototyping'. These characteristics therefore make the integration of procedurally driven content creation strategies more feasible in an artists' workflow. Procedural parameters can be interactively altered, thus yielding real-time feedback in the results of corresponding algorithms of the tool chain.

Autodesk's 'Maya' was selected as the authoring tool on which to base the 'artist interface' of this research's tool chain. This was motivated by a number of factors, namely Maya's mainstream use in the games industry (Autodesk: Autodesk In Games, 2010).

Many prominent game studios have integrated Maya into their development processes, due to the software's vast array of functionality and "*wide variety of features*" (Price, 2008) (Insomniac Games, 2008) (Terminal Reality, 2009). Maya also maintains a highly extensible developer interface, making it possible to integrate functionality specific to this research.

The interactive tool chain chapter section illustrated how Maya's development interfaces were used to implement this research's custom tool chain functionality into the modelling package. As demonstrated, this effectively 'bridged' Maya's advanced modelling and content authoring capabilities, to an external 'game rendering context'.
This research also extended Maya's standard functionality, by providing artists with 'interactive' access to algorithms/functionality specific to this research. Emphasis was placed on the 'reuse' of interface and interaction conventions native to Maya, to maintain transparency and familiarity for experienced Maya users.
Not only did the practical outcome of this provide a basis for experimentation and testing through development, it also demonstrated how the ideas proposed in this research can integrate with 'industry standard' content authoring tools.

The developed tool chain was based on the 'connection' model, consisting of two core software components. This model was selected because of its flexibility, permitting different workflow configurations for individual and/or collaborative production by artists, in an interactive content authoring context. In addition, the connection model can facilitate interactive content authoring across different 'architectures/platforms', making it attractive for cross platform game development projects. Selecting this model therefore demonstrates how algorithms and outcomes of this research could benefit a wider range of game development situations. In addition, the interactive tool chain section established that this model is suited for integration with other content authoring tools in addition to Maya.
Content data is associated and synchronized between the model's two software components via a real-time communications link (TPC/IP based), through which 'interactivity' and responsiveness is delivered.

The tool chain's first software component is the RTCE which as mentioned, integrates into the content authoring tool (Maya). The RTCE provides artists' with access to parameters/controls specific to the tool chain's functionality, namely the NPD and RTGI algorithms. Furthermore, the RTCE integrates with Maya's built-in user interfaces, particularly those for creating and specifying procedural functions and materials.

The RTCE internally associates Maya's built-in controls with corresponding parameters of the NPD and RTCE algorithms. This association is made via Maya's 'event mechanism', in conjunction with the tool chain's 'network interface'. Through this, artists harness the 'procedural authoring' functionality provided by Maya, to interactively control and refine game content/scenes that employ the tool chain's RTGI, NPD and material composition algorithms. As a result, PM's can be explicitly used by artists, through this interactive tool chain, for a variety of content authoring tasks.

The

event mechanism section explained how the RTCE binds to Maya's event system, allowing the tool chain to immediately respond to and process relevant artist interactions. Thus, when a 'registered event' took place in Maya, corresponding RTCE functionality was invoked which typically resulted in scene data being transmitted to the tool chain's 'rendering component' (GRC). The functional side effects of this responsiveness were illustrated in the sequential diagrams of demonstrations chapter. Thus, simple editing interactions by the artist in Maya are interactively reproduced in real-time, in the GRC.

As discussed, the RTCE's design adopts data representations that are native to Maya. This influenced the way 'procedural material composition' and parameters of the NPD and RTGI algorithms were exposed in the RTCE's user interface. Thus, conventions such as 'material channels' and 'vertex structure' were integrated with the RTCE's interface and functional implementation. Note that these 'data conventions' are common to most modelling packages.
The interactive tool chain section showed how the RTCE's interface underwent an iterative development process, which aimed to expose all required parameters/controls, to artists. Recall that a major outcome in the RTCE's final design iteration was the integration of Maya's own internal material structures, which provided a basis for the RTCE's material composition feature.
This RTCE implementation therefore, demonstrated that a correspondence between Maya's data conventions/representations could be maintained when 'reproducing' data in the 'rendering component' of the tool chain. If for example, an artist assigned a procedural function to the 'colour channel' of a material structure in Maya, the materials reproduction in the GRC would be rendered in a consistent fashion.
The RTCE's direct integration into Maya therefore, exposed the tool chain's unique functionality (RTGI, NPD, material composition) to artists, encouraging the use and resulting benefits of these algorithms in the content authoring process.

The second software component that was developed for this tool chain was the ‘game rendering context’ (GRC). Recall that a custom renderer was built as a substitute for the game/engine renderer that would ideally be used in an interactive tool chain. The primary reason for this ‘customized’ renderer was the project’s requirement for GPU based procedural functionality, in conjunction with ‘low level’ Direct3D 10 API access. A survey of open source/usable rendering engines showed that none fully satisfied these requirements (see Appendix A). Thus, the GRC was programmed directly on Microsoft’s Direct3D 10 rendering API.

As illustrated in the game rendering context (GRC) section, Microsoft’s Direct3D is the predominant API used by the games industry for accelerated graphics rendering. To maintain relevance to current technology, Direct3D 10 was selected as the GRC’s interface to graphics hardware. Direct3D 10 also provided access to modern GPU/shader functionality, which was required for this research’s ‘GPU based’ RTGI and NPD implementations.

As mentioned in the implementation chapter, these algorithms use a number of hardware acceleration features available on Direct3D 10 certified graphics hardware, namely ‘geometry shaders’, ‘data output-streaming’ and ‘hardware instancing’. Aside from the assumed performance benefits, this GPU based ‘implementation pathway’ was also motivated by an interest in the features and characteristics of the GPU architecture.

The custom framework that was developed provided a high level of flexibility for the development process and was well suited to the iterative and experimental nature of these framework elements. This ‘framework’ served as an ideal platform for experimental/conceptual ideas that constituted the research’s RTGI, NPD and material composition algorithms.

In keeping with the tool chain’s interactive nature, the GRC application was designed to immediately respond to network traffic, transmitted from connected RTCE ‘instances’. Because the GRC is a real-time rendering application, ‘response’ to network traffic was handled in the GRC’s application loop. Responding to network traffic at each ‘loop interval’ delivered the necessary level of ‘responsiveness’ to artist interaction/network traffic in the GRC.

Recall that most of the functionality that drives algorithms in this tool chain is ‘shader/GPU based’. The implication of this was the need for a software structure that would support initialization and execution of these shader based algorithms. Most of this shader support structure existed in the GRC’s ‘renderer’ and ‘materials’ modules. These modules also

integrated functionality capable of dynamically responding to network traffic which originates from RTCE instances.

A key feature of the GRC's rendering module was its integration of a flexible and customizable shader framework. This shader framework was developed to support dynamic shader generation/assembly in response to artists' interaction with the tool chain at runtime. If for example, the artist made changes to the 'vertex structure' of a game object from the RTCE (Maya), the rendering module would respond by recompiling the game object's shader to facilitate the new geometry/vertex format.

In addition, the rendering module rebuilds shaders associated with game objects when the composition of an object's material structure is modified by the artist, or when shader based RTGI/NPD functionality is requested for a shader, via the shader assembly mechanism. For example, if an artist invokes such functionality from the RTCE, the GRC responds by embedding corresponding shader functionality into shaders of the respective game objects.

The shader system that was implemented into the GRC, demonstrated dynamic shader assembly via the use of pre-processor directives (available in Microsoft's HLSL shader language). As shown, this strategy provided a robust mechanism for dynamically building 'adaptive' and arbitrary shaders.

It was established however, that the shader recompilation process tended to incur a brief 'delay' in the fluidity of the GRC's interaction, particularly when NPD or RTGI functionality were embedded into a shader. Although acceptable, possible solutions to minimize the duration of shader compilation are explored in the following section.

The shader system however, successfully offloaded development overhead on artists, that typically results from arbitrary 'vertex formats' in game geometry, via the use of pre-processor directives and specialized shader structure. In addition, extensions were implemented in the tool chain's shader system which allowed 'technical artists' to provide custom shader code. The system not only manages the association of custom shader functionality with arbitrary 'geometry formats', but its design also makes procedural material composition functionality available for use in custom shaders. This therefore demonstrated the integration of a shader system that merged automated, adaptive shader assembly with customization and procedural elements, in the context of an interactive tool chain.

The instancing algorithm section described how shader based functionality was developed to generate 'instance data' across an arbitrary 'manifold surface' specified by the artist. The algorithm combines this data with GPU based 'hardware instancing', to efficiently render

many ‘instances’ or copies of a specified object. Due to the algorithms GPU based implementation RTGI could harness the parallel processing capabilities of the architecture, to deliver instance generation interactively and in real-time.

Procedural functionality was incorporated into the RTGI algorithm to dictate the distribution of generated instances and/or their unique parameters. Through this, the algorithm enabled parameterization of rotation and scale for instanced geometry across a manifold surface, based on the evaluation of associated procedural functions. In addition, the RTGI algorithm allowed a procedural function to be supplied at runtime, to control the distribution of instances across the manifold surface.

Following investigation into RTGI based instancing strategies for games and discussion with professionals in the games industry, it was clear that explicit control over instancing would be required in some applications. Thus, provision for a ‘cookie cutter’ image mask was integrated into the tool chain and RTGI algorithm, to provide ‘hybrid’ object instancing that merged control (offered by conventional image based art forms) with procedural functionality.

This implementation of RTGI was designed to take advantage of the interactive tool chain context and thus, was largely based on parameterized, procedural functionality. The algorithm is parameterized in real-time, enabling it to immediately respond to artists’ instigated parameter changes. Thus, changes to the algorithms visual result, which correspond to artist interaction, are shown by the tool chain’s game renderer.

Because the level of control offered by RTGI’s procedural functionality corresponds to that provided by Maya’s built-in content authoring interfaces, this tool chain algorithm delivers a high level of configuration and control over the distribution of instanced objects. This therefore, improves upon ‘static’ (un-parameterized) implementations of procedural object placement, which are typical of many ‘game environment’ authoring tools.

The final major component of this research was the NPD algorithm. The NPD algorithm was designed to provide a procedurally based strategy for unique geometric variation in objects. The motivation for this was to better facilitate the delivery of geometric variety between similar objects, while minimizing both artist workload and the game’s overall data size.

Game objects/props are often reused in game scenes as a method for increasing detail and/or reflecting characteristics of game environments. Unfortunately, visual repetition can have a negative effect on the overall realism of game scenes. The NPD algorithm was therefore developed to counteract this.

By interpreting evaluations of procedural functionality as deformation, the NPD algorithm achieved geometric variety between instances of the same 'base geometry'.

The NPD implementation maximizes artist control over the presence of procedurally driven variation in base geometry, permitting 'non-uniform' distributions of geometric deformation. Non-uniform deformation was based on a per-vertex 'weighting' scheme. From the artist's working environment, the application of variation to base geometry was achieved by the 'painting' metaphor, which allowed intuitive and fast application of 'deformation weightings' to base geometry.

The NPD algorithm also demonstrates the use of triangular tessellation through which better deformation/variation results could be manifest in simple base geometry. Because deformation weightings can be arbitrarily distributed across base geometry, NPD tessellation is therefore adaptive, which avoids unnecessary tessellation of the base geometry. In addition, the variable nature of weights was used to allow different levels of tessellation and variation in NPD processed based geometry.

Similarly to the RTGI algorithm, the NPD system operates in real-time and thus, coincides with the interactive tool chain paradigm. Due to the NPD's implementation in the context of this research's interactive authoring environment, artists' can easily harness procedural functionality to specify object variation via fluid and real-time visual feedback.

This research has demonstrated a series of novel and compelling applications of procedural methods, for content creation in games. The motivation for this was to deliver strategies which promote further improvement in the detail and complexity of games graphics thus, increasing the realism of visual experiences in games.

Concepts and algorithms which build upon this objective have been integrated into an artist centric, interactive content creation workflow, to take advantage of the interactive paradigm's benefits.

This research has therefore successfully demonstrated the integration of procedural methods into relevant content creation processes and algorithms, which enhance the prospects of quality, detail and realism in games content and graphics.

Future work

The following section outlines some possible avenues for future work in aspects of this research.

Material composition system

As discussed, the material composition system is primarily implemented on the GPU and thus, is written in HLSL (High Level Shader Language). Recall that shaders of this tool chain which integrate 'material composition' functionality require recompilation when the artist alters the material's composition. Shader recompilation however, incurs a short 'delay' during the tool chain's otherwise seamless responsiveness during runtime.

An approach to reducing and/or eliminating interaction delays might involve the use of 'Dynamic Shader Linking'; a feature of 'Shader Model 5.0' which is available in Direct3D11 (Direct3D 11 Features, 2010) Note that implementations of Direct3D11 for graphics hardware started to emerge during this research. 'Dynamic Shader Linking' appears to provide functionality similar to the pre-processor based shader assembly feature that underlies this research's material composition and shader system. Because 'Dynamic Shader Linking' is native to Direct3D11 however, it is likely that a composition system based on this feature would deliver more rapid response.

The current implementation of the material system achieves composition by 'averaging' the contribution of active procedural functions in the material. Although this 'procedural combination' computation is sufficient for many situations, it would be useful if the system allowed artists to 'combine' procedural functions via other combination operations; for example 'multiplication' or 'difference'. From an artists' perspective, this would offer more flexibility in the tool chain as a wider range of composition results could be achieved. This would require the integration of respective 'combination operations' in both the tool chain interface (RTCE), as well as the material system's shader code.

Real-time generative instancing (RTGI)

The RTGI concept is expressed on the GPU architecture in order to achieve high runtime performance. As discussed, the GPU implementation is limited by some aspects of the current GPU architecture, as well as current graphics API's.

The GPU based implementation of RTGI uses 'data streaming' and 'GPU instancing' and thus, must be expressed as a multi-stage rendering algorithm. The consequence of this, is that

intermediate storage of instance data is required between stages, limiting the number of instances to the amount of memory allocated for these buffers (or the memory available on the hosting system), as well as data bandwidth between the GPU/host system processor.
Ideally, future graphics API's would allow this algorithm to be expressed in a single pass, avoiding the need for intermediate data storage. This could enable 'uncapped' volumes of procedurally driven geometry instancing in games, with instancing data only existing 'on-the-fly' during rendering.

Other improvement could be made to the RTGI implementation's runtime efficiency. An extra layer of processing could be added to the algorithm's 'instance generation' shader; 'frustum culling'. Frustum culling would integrate into the instance generation stage, preventing the generation of instances that fall outside of the camera's 'field of view' or 'view frustum'.
Frustum culling is an optimization technique that 'culls' non-visible geometry/objects prior to the rendering process (Bourke, 2000). The process begins by computing a 'view volume' (i.e. view perspective) which corresponds to a volume that encapsulates all visible portions of the 3D scene (Bourke, 2000). The culling process involves 'view planes' being extracted from the view volume (Hartmann & Gribb, 2010). If an 'object being rendered' falls on the outer side of a view plane, that object is culled from the subsequent rendering process. The view volume is a product of 'view' and 'projection' transformations, that represent the scene's view perspective (Hartmann & Gribb, 2010). Because these transformations are available in the context of the RTGI's instancing shader, integrating this form of culling into the instance generation shader, to avoid the generation of unnecessary/non-visible instances, would be feasible.

Furthermore, improvements could be made to the integration of the RTGI algorithm's 'cookie cutter' feature. Recall that the 'cookie cutter' is an auxiliary 'image mask' which maps over manifold geometry to explicitly control areas where instancing can occur. As discussed, this offers artists a high level of control over the behaviour/results of instancing. This control however, comes at the expense of undesirable memory overhead. Currently, this RTGI implementation uses an uncompressed bitmap image to deliver 'cookie cutter' data to the shader. The image uses a four channelled colour format at 32-bits per pixel. Thus, the opportunity exists for reducing memory overhead by simplifying the cookie image's data precision to a 'single bit' per pixel. This precision would be sufficient, given that the cookie data is interpreted as a Boolean value by the instancing shader. Other memory conservation

strategies could include compression schemes such as ‘run length encoding’, to efficiently represent the cookie image in system/GPU memory. This would require that a GPU based decompression/decoding operation be integrated into the RTGI’s instancing shader.

Non-uniform procedural deformation (NPD)

The NPD implementation allows artists’ to specify ‘unique’ procedurally driven geometric variation across many instances of the same ‘base’ geometry. Currently, only a single ‘procedural noise’ function can be used to deliver unique geometric variation between objects. Although this single function has proven to be sufficient for many situations, it may be useful if the artist could select different procedural functions to drive geometric deformation. Furthermore, these developments could integrate with the ‘procedural composition’ mechanism that was used in the research’s material system, to deliver more flexibility and control over the procedural deformation result.

Another avenue for improvement would be to support more extensive variation between objects. As discussed, geometric variation is limited by the ‘form’ of the base geometry. Geometric variation is achieved by modifying the position of vertices in the base geometry via the procedurally based, ‘vertex offset’ mechanism. Thus, the NPD algorithm could be extended to allow variation between objects, where portions of geometric structure/form of a processed base object (geometry) are omitted from the rendered result; this providing more ‘substantial’ variation.

References

3ds Max 2011 SDK. (2010, Febuary 5). Retrieved May 16, 2010, from Autodesk, Inc: www.autodesk.com/3dsmax-sdk-docs

3DVision. (2010, March 1). Retrieved April 17, 2010, from Crytek To Demo CryENGINE 3 in Stereoscopic 3D at GDC 2010: http://3dvision-blog.com/tag/livecreate/

Accardo, S. (2007, August 7). *id Software's New Title: Rage*. Retrieved April 3, 2010, from gamespy: http://au.pc.gamespy.com/pc/id-tech-5-project/810525p1.html

Adams, B. (2009, December 15). *Bethesda To Publish RAGE*. Retrieved June 2010, 3, from EBA: http://epicbattleaxe.com/wp-content/uploads/2009/12/RAGE_screen1.jpg

Age of Empires. (2010, April 23). Retrieved April 28, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Age_of_Empires

Age of Empires,Features. (1998). Retrieved June 3, 2010, from Microsoft: http://www.microsoft.com/games/aoeexpansion/features.htm

Ahearn, L. (2006). Mr. In L. Ahearn, *3D Game Textures: Create Profession Game Art Using Photoshop* (p. 103). Oxford, UK: Focal Press.

Allegorithmic, About. (2010). Retrieved April 23, 2010, from Allegorithmic.: http://www.allegorithmic.com/?PAGE=ABOUT

Allegorithmic, Products. (2010). Retrieved April 13, 2010, from Allegorithmic: http://www.allegorithmic.com/?PAGE=PRODUCTS

Anderson, N. (2007, August 30). *Video gaming to be twice as big as music by 2011*. Retrieved May 11, 2010, from ars technica: http://arstechnica.com/gaming/news/2007/08/gaming-to-surge-50-percent-in-four-years-possibly.ars

Ashrafi, R. (2008, May 1). *GTA IV Budget Revealed*. Retrieved May 19, 2010, from Digtial Battle: http://www.digitalbattle.com/2008/05/01/gta-iv-budget-revealed/

ATI Technologies. (2010, May 12). Retrieved May 16, 2010, from Wikipedia: http://en.wikipedia.org/wiki/ATI_Technologies

Autodesk. (2010). Retrieved May 16, 2010, from Autodesk, Inc: http://usa.autodesk.com/industries/media-entertainment/games

Autodesk. (2010). Retrieved May 16, 2010, from Autodesk, Inc: http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13567410

Autodesk. (2010). Retrieved May 13, 2010, from Autodesk, Inc: http://images.autodesk.com/adsk/files/fy11_games_brochure_us.pdf

Autodesk Maya. (2010). Retrieved May 16, 2010, from Autodesk, Inc: http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13583239#channels_Pipeline Integration

Banks, M. (2009, May 3). *CoD4 Patch Bends Cheaters Over a Table*. Retrieved May 27, 2010, from Loot Ninja: http://loot-ninja.com/wp-content/uploads/2008/03/map3.jpg

Bantick, M. (2009, August 31). *Uncharted 2 could not be achieved on Xbox 360*. Retrieved March 13, 2010, from iTWire: http://www.itwire.com/your-it-news/entertainment/27331-uncharted-2-could-not-be-achieved-on-xbox-360

Bathroom, Allegorithmic. (2010). Retrieved March 15, 2010, from Allegorithmic: http://www.allegorithmic.com/?PAGE=GALLERY.demos

Blythe, D. (2006). The Direct3D 10 System.

Blythe, D. (2006). The Direct3D 10 System. (p. 2). Microsoft Corporation.

Booth, M. (2009, October 27). *The AI Systems of Left 4 Dead.* Retrieved March 22, 2010, from Valve Software: http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf

Bourke, P. (2000, November 18). *Frustum Culling.* Retrieved May 18, 2010, from Western Australian Supercomputer Program (WASP): http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/frustum/

Breckon, N. (2008, May 28). *FarCry 2 Preview*. Retrieved March 24, 2010, from ShackNews: http://www.shacknews.com/featuredarticle.x?id=880

Breckon, N. (2008, September 16). *id: Rage Content Cut due to Xbox 360 Size Limit*. Retrieved March 22, 2010, from ShackNews: http://www.shacknews.com/onearticle.x/54778

Bright, P. (2010, March 11). *OpenGL 4 spec arrives with Direct3D 11 feature parity*. Retrieved May 16, 2010, from Ars Technica: http://arstechnica.com/software/news/2010/03/opengl-4-spec-arrives-with-direct3d-11-feature-aars

Brooks, T. (2010, March 10). *Sentinel level*. Retrieved May 24, 2010, from Wikipedia: http://upload.wikimedia.org/wikipedia/en/9/9a/Sentinel_level.png

Brown, P. (2010, 3 23). *ARB_gpu_shader_fp64.* Retrieved March 18, 2010, from OpenGL: http://www.opengl.org/registry/specs/ARB/gpu_shader_fp64.txt

Brown, P. (2009, December 14). *EXT_geometry_shader4.* Retrieved May 12, 2010, from OpenGL: http://www.opengl.org/registry/specs/EXT/geometry_shader4.txt

Brudvig, E. (2007, December 7). *RoboBlitz Review*. Retrieved March 25, 2010, from IGN AU Edition: http://au.xboxlive.ign.com/articles/749/749898p1.html

Business Wire. (2007, August 20). *The Global Entertainment Industry is Expected to Show an Annual Growth of 10% in the Next Four Years and That Growth Will Be Driven by China*. Retrieved May 12, 2010, from BNet: http://findarticles.com/p/articles/mi_m0EIN/is_2007_August_20/ai_n19452832/

Carmack, J. (2010). *Rage PC Interview - Carmack on Rage, Doom, and Gaming Development*. Retrieved June 4, 2010, from GameSpy: http://au.pc.gamespy.com/dor/objects/926419/id-tech-5-project/videos/carmack_spy_080408_part1.html;jsessionid=4pdlp0f6uebq3

Champandard, A. J. (2009, November 6). *Procedural Level Geoemtry from Left 4 Dead 2: Spying on the AI Director*. Retrieved March 25, 2010, from aigamedev.com: http://aigamedev.com/open/discussion/procedural-level-geometry/

Cifaldi, F. (2006, September 1). *The Gamasutra Quantum Leap Awards: First-Person Shooters*. Retrieved April 4, 2010, from Gamasutra: http://www.gamasutra.com/features/20060901/quantum_01.shtml

Crossley, R. (2009, May 26). *Interview: Krome's Robert Walsh*. Retrieved May 19, 2010, from develop: Intent Media ©: http://www.develop-online.net/news/33625/Study-Average-dev-cost-as-high-as-28m

CryEngine. (2010, Feburary 26). Retrieved April 12, 2010, from Wikipedia: http://en.wikipedia.org/wiki/CryENGINE

CryEngine 3.0. (2010, April 20). Retrieved May 1, 2010, from Wikipedia: http://en.wikipedia.org/wiki/CryEngine_3

CryEngine 3.0. (2010, April 20). Retrieved May 1, 2010, from Wikipedia: http://en.wikipedia.org/wiki/CryEngine_3#cite_note-4

CryENGINE® 2 Specifications. (2010). Retrieved May 28, 2010, from CRYTEK©: http://www.crytek.com/technology/cryengine-2/specifications/

Current Technology - Unreal Engine 3. (2010). Retrieved May 1, 2010, from UnrealTechnology: http://www.unrealtechnology.com/technology.php

D3DXCreateEffectFromFile Function. (2010, April 7). Retrieved May 21, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb172768(v=VS.85).aspx

Delta3D - Features. (n.d.). Retrieved May 1, 2010, from Delta3D: http://www.delta3d.org/article.php?story=20051209133127695&topic=docs

DeMaria, R., & Wilson, J. L. (2003). *High Score!: The Illustrated History of Electronic Games, Second Edition.* McGraw-Hill.

Development of Spore. (2010, May 1). Retrieved June 3, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Development_of_Spore

DeWolf, I. (2000). *OCTAVE SUMMING*. Retrieved May 24, 2010, from Martian Labs: http://martian-labs.com/martiantoolz/htmldocs/algorithm/fourier_analysis/Octave_Summing/octave_sum.html

Diffuse minus Specular. (2009, Jun 16). Retrieved June 1, 2010, from open salon: http://static.open.salon.com/files/picture_21245167244.png

Direct3D 11 Features. (2010, April 5). Retrieved May 8, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/ff476342(VS.85).aspx

Direct3D 9 to Direct3D 10 Considerations. (2010, April 5). Retrieved May 17, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb205073(VS.85).aspx#Removal_of_Fixed_Function

Doc:Manual/Materials/Vertex Paint. (2009, September 25). Retrieved March 13, 2010, from Blender: http://wiki.blender.org/index.php/Doc:Manual/Materials/Vertex_Paint

Documentation, MapZone. (2010). Retrieved March 12, 2010, from MapZone: http://www.mapzoneeditor.com/index.php?PAGE=DOCUMENTATION.IvyLeaf

DrawAuto Method. (2010, April 5). Retrieved May 16, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb173564(VS.85).aspx

Effect System Interfaces. (2010, April 5). Retrieved May 19, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb205110(v=VS.85).aspx

Effect Technique Syntax. (2010, April 5). Retrieved May 16, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb205053(v=VS.85).aspx

Efficiently Drawing Multiple Instances of Geometry. (2010, April 7). Retrieved June 1, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb173349(v=VS.85).aspx

Elias, H. (1998, March 1). *Bump Mapping*. Retrieved May 22, 2010, from http://freespace.virgin.net/hugo.elias/graphics/x_polybm.htm

Everitt, C., & Kilgard, M. J. (2002, March 12). *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering.* Retrieved May 12, 2010, from Nvidia: http://developer.nvidia.com/attach/6831

Far Cry 2. (2009, July 18). Retrieved March 24, 2010, from Procedural Content Generation: http://pcg.wikidot.com/pcg-games:far-cry-2

Far Cry 2. (2009, July 18). Retrieved March 23, 2010, from Procedural Content Generation: http://pcg.wikidot.com/pcg-games:far-cry-2

Far Cry 2. (2010, April 28). Retrieved May 2010, 2, from Wikipedia: http://en.wikipedia.org/wiki/Far_Cry_2

Far Cry. (2010, April 2). Retrieved April 22, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Far_Cry

Features, Ogre3D. (2009). Retrieved May 1, 2010, from Ogre3D: http://www.ogre3d.org/about/features

Fryazinov, O., & Pasko, A. (2008). Interactive ray shading of FRep objects. *International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision* (p. 5). Bory: WSCG.

Function Declaration Syntax. (2010, April 5). Retrieved May 13, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb509607(VS.85).aspx

Gebhardt, N. (2009). *Features*. Retrieved May 1, 2010, from Irrlict - An open source 3d engine: http://irrlicht.sourceforge.net/features.html

GeForce 256. (2010). Retrieved May 24, 2010, from NVidia: http://www.nvidia.com/page/geforce256.html

Geometry-Shader Object. (2010, April 5). Retrieved March 23, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb509609(VS.85).aspx

Geometry-Shader Object. (2010, April 5). Retrieved March 23, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb509609(VS.85).aspx

Get Windows Vista: System requirements. (2010). Retrieved May 31, 2010, from Microsoft©: http://www.microsoft.com/windows/windows-vista/get/system-requirements.aspx

Golding, J., & Nalezynski, R. (2010). *Unreal Physics Asset Tool (PhAT) User Guide*. Retrieved April 21, 2010, from Unreal Developer Network: http://udn.epicgames.com/Three/PhATUserGuide.html

Gowers, T. (2004). *Real numbers as infinit decimals*. Retrieved April 2, 2010, from Department of Pure Mathematics and Mathematical Statistics - University of Cambridge: http://www.dpmms.cam.ac.uk/~wtg10/decimals.html

GPU. (2010). Retrieved May 24, 2010, from NVidia: http://www.nvidia.com/object/gpu.html

Grand Theft Auto 1. (2010). Retrieved May 3, 2010, from wikia: http://gta.wikia.com/Grand_Theft_Auto_1

Green, S., & Cebenoyan, C. (2004). *High Dynamic Range Rendering.* Retrieved May 27, 2010, from NVidia: http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_HDR.pdf

Gregor vom Scheidt. (2005). Retrieved May 19, 2010, from Xing: http://www.xing.com/profile/Gregor_vomScheidt

Hagedoorn, H. (2007, July 27). *Radeon HD 2400 XT and 2600 XT review*. Retrieved June 1, 2010, from Guru3D: http://www.guru3d.com/article/radeon-hd-2400-xt-and-2600-xt-review/16

Haines, E. (1994). Point in Polygon Stratergies. In P. Heckbert, *Graphics Gems IV* (pp. 24-46). Academic Press.

Hartmann, K., & Gribb, G. (2010, June 6). *Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix.* Retrieved May 18, 2010, from Raven Software: http://www2.ravensoft.com/users/ggribb/plane%20extraction.pdf

Hecker, C. (2009, May 8). *My Liner Notes for Spore*. Retrieved June 3, 2010, from ChrisHecker: http://chrishecker.com/My_Liner_Notes_for_Spore

HLSL. (2010, 4 5). Retrieved March 15, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx

HLSL:Sample. (2010, April 5). Retrieved May 3, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb509695(VS.85).aspx

Hwu, W.-m. (2009, August 3). *Many-core Computing.* Retrieved May 27, 2010, from Gigascale Systems Research Centre: http://www.mpsoc-forum.org/2009/slides/Hwu.pdf

ICE Attribute Reference. (2009, July 15). Retrieved March 13, 2010, from The Softimage Wiki: http://softimage.wiki.softimage.com/index.php/ICE_Attribute_Reference

id Software. (2010, May 16). Retrieved May 19, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Id_Software

Id Software. (n.d). Retrieved May 16, 2010, from Id Software: http://www.idsoftware.com/business/jobs/index.php

Id Software: Final Doom. (2001). Retrieved May 31, 2010, from Id Software: http://www.idsoftware.com/games/doom/doom-final/

Id Software: Quake. (2001). Retrieved May 31, 2010, from Id Software: http://www.idsoftware.com/games/quake/quake/

id Software: Return to Castle Wolfenstein. (2001). Retrieved May 19, 2010, from id Software: http://www.idsoftware.com/games/wolfenstein/rtcw/images/full08.jpg

id Software: Wolfenstien 3D and Spear of Destiny. (2001). Retrieved May 19, 2010, from id Software: http://www.idsoftware.com/games/wolfenstein/wolf3d/images/full01.jpg

ID3D10Asynchronous Interface. (2010, 4 5). Retrieved April 3, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb173500(v=VS.85).aspx

ID3D10Query Interface. (2010, April 5). Retrieved May 19, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb173823(VS.85).aspx

IGN: Crysis Screenshots. (2007, October 9). Retrieved June 1, 2010, from IGN: http://pcmedia.ign.com/pc/image/article/825/825955/crysis-20071009011856623_640w.jpg

Ingham, T. (2010, April 23). *Xbox 360 sales storm past 40m*. Retrieved May 10, 2010, from CVG: http://www.computerandvideogames.com/article.php?id=243971

Input-Assembler Stage. (2010, April 5). Retrieved May 5, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb205117(v=VS.85).aspx

Insomniac Games. (2008). Retrieved May 14, 2010, from Autodesk, Inc: http://images.autodesk.com/adsk/files/insomniacgames_customerstory_maya.pdf

Instanced Geometry. (2010, April 7). Retrieved May 9, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb173349(VS.85).aspx

Ivan, T. (2010, January 6). *Blu-Ray Storage Capacity Increased*. Retrieved March 23, 2010, from Edge - Online: http://www.edge-online.com/news/blu-ray-storage-capacity-increased

Jube. (2009, August 11). *CryEngine LiveCreate Showcase at GDC Europe*. Retrieved April 14, 2010, from Voodoo Extreme: http://ve3d.ign.com/articles/news/49541/CryEngine-LiveCreate-Showcase-GDC-Europe

Kessenich, J., Baldwin, D., & Rost, R. (2010, March 10). *The OpenGL® Shading Language.* (J. Kessenich, Ed.) Retrieved May 21, 2010, from OpenGL: http://www.opengl.org/registry/doc/GLSLangSpec.4.00.8.clean.pdf

Kilgard, M. J. (2000, July 10). *All About OpenGL Extensions*. Retrieved May 16, 2010, from OpenGL: http://www.opengl.org/resources/features/OGLextensions/

Kilgard, M. J. (2000, July 10). *All About OpenGL Extensions*. Retrieved May 16, 2010, from OpenGL: http://www.opengl.org/resources/features/OGLextensions/

Kudler, A. (2007). *Timeline: Video Games (1985 - 1995)*. Retrieved November 12, 2009, from infoplease: http://www.infoplease.com/spot/gamestimeline3.html

Lagae, A. (2009, May 5). *Procedural Noise using Sparse Gabor Convolution*. Retrieved May 24, 2010, from YouTube: http://www.youtube.com/watch?v=1_Ss2dUvaW8

Lagae, A., Lefebvre, S., Drettakis, G., & Dutr', P. (2009). Procedural Noise using Sparse Gabor Convolution. *ACM SIGGRAPH* , 2.

Lagae, A., Lefebvre, S., Drettakis, G., & Dutr', P. (2009). Procedural Noise using Sparse Gabor Convolution. *ACM SIGGRAPH* , 3.

Lagae, A., Lefebvre, S., Drettakis, G., & Dutr', P. (2009). Procedural Noise using Sparse Gabor Convolution. *ACM SIGGRAPH* , 5.

Lagae, A., Lefebvre, S., Drettakis, G., & Dutr', P. (2009). Procedural Noise using Sparse Gabor Convolution. *ACM SIGGRAPH* , 7.

layeredTexture node. (2010). Retrieved April 30, 2010, from Autodesk: http://download.autodesk.com/us/maya/2010help/Nodes/layeredTexture.html

Left 4 Dead. (2009, October 18). Retrieved Febuary 22, 2010, from Procedrual Content Generation: http://pcg.wikidot.com/pcg-games:left4dead

Left 4 Dead. (2009). Retrieved November 22, 2009, from L4D: http://www.l4d.com/game.html

Lichtenbelt, B., Brown, P., & Werness, E. (2008, August 17). *NV_transform_feedback*. Retrieved March 26, 2010, from OpenGL: http://www.opengl.org/registry/specs/NV/transform_feedback.txt

Lidwell, W., Holden, K., & Butler, J. (2003). *Universal Principles of Design.* Beverly: Rockport Publishers.

Lilly, P. (2010). *Voodoo to GeForce: The Awesome History of 3D Graphics*. Retrieved Feburary 10, 2010, from MaximumPC: http://www.maximumpc.com/article/features/graphics_extravaganza_ultimate_gpu_retrospective

Lionhead Studios. (2010). Retrieved May 28, 2010, from Lionhead Studios: http://lionhead.com/ContactUs.aspx

L-system. (2010, May 29). Retrieved May 30, 2010, from Wikipedia: http://en.wikipedia.org/wiki/L-system

Mackie, D. (2008, October 29). *Blink, Pinky, Inky and Clyde: Smater Than You Think*. Retrieved June 2, 2010, from back of the cereal box: http://2.bp.blogspot.com/_1I7KiCuAU4k/SQVqoynvQJI/AAAAAAAACLs/csfFdlJjMhQ/s400/pac-man_ghosts_blinky_inky.jpg

Making Far Cry 2's Africa. (2008, May 28). Retrieved March 22, 2010, from Kotaku: http://kotaku.com/5011462/making-far-cry-2s-africa

Matossian, M. (2001). 3DS Max for Windows. In *3DS Max for Windows* (p. 393). Berkeley: Peachpit Press.

Matossian, M. (2001). Ms. In M. Matossian, *3DS Max for Windows* (pp. 382 - 386). Berkeley, Califronia: Peachpit Press.

Maya. (2010). Retrieved May 16, 2010, from Autodesk, Inc: http://usa.autodesk.com/adsk/servlet/pc/index?id=13577897&siteID=123112

Maya. (2010, April 26). Retrieved April 30, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Autodesk_Maya

McLean-Foreman, J. (2001, April 6). *An Interview with Epic Games' Tim Sweeney*. Retrieved May 18, 2010, from Gamasutra: http://www.gamasutra.com/view/feature/3093/an_interview_with_epic_games_tim_.php

MFn Class Reference. (2010). Retrieved May 17, 2010, from Autodesk, Inc: http://download.autodesk.com/us/maya/2010help/API/class_m_fn.html

MFnDagNode Class Reference. (2010). Retrieved May 17, 2010, from Autodesk, Inc: http://download.autodesk.com/us/maya/2010help/API/class_m_fn_dag_node.html

Microsoft Age of Empires. (1998). Retrieved April 22, 2010, from Microsoft: http://www.microsoft.com/games/empires/features_more_features.htm

Misc Perlin Noise. (1999). Retrieved March 15, 2010, from GRAFNET: http://www.grafnet.com.pl/photoshop-filters-description.php?kolekcja=filtry/Filter_Forge&filtr=Misc_Perlin_Noise.jpg

Mittring, M. (2006). Triangle Mesh Tangent Space Calculation. In W. Engel, *Shader X4* (pp. 77 - 89). Hingham: Charles River Media, Inc.

Motostorm. (2007, December). Retrieved July 1, 2010, from Videogameblogger.com: http://www.videogamesblogger.com/wp-content/uploads/2007/12/motorstorm-ps3-screenshot-big.jpg

Murdock, K. L. (2010). *Case Study: Resistance 2 Graphics Tools and Pipeline*. Retrieved May 16, 2010, from gamedev.net: http://www.gamedev.net/reference/art/features/resistance2case/

MViewportRenderer Class Reference. (2010). Retrieved May 16, 2010, from Autodesk, Inc: http://download.autodesk.com/us/maya/2010help/API/class_m_viewport_renderer.html

Myers, K. (2007). Using D3D10 Now. *Game Developers Confrence*, (pp. 66-67). San Francisco.

Neider, J., Davis, T., & Woo, M. (1994). *OpenGL Programming Guide.* Reading, Massachusetts: Addison-Wesley Publishing Company.

Nintendo. (2010, March 31). *Consolidated Financial Highlights*. Retrieved May 10, 2010, from Nintendo: http://www.nintendo.co.jp/ir/pdf/2010/100506e.pdf#page=23

Normal Map. (2010, May 4). Retrieved April 15, 2010, from Polycount: http://wiki.polycount.net/Normal_Map

NVidia. (2010). *About Us*. Retrieved May 16, 2010, from NVIDIA: http://www.nvidia.com/page/companyinfo.html

NVIDIA Corporation. (2010). *High-Precison Effects*. Retrieved April 8, 2010, from Nvidia: http://www.nvidia.com/object/feature_HPeffects.html

NVidia: GeForce 8800. (2010). Retrieved June 1, 2010, from Nvidia: http://www.nvidia.com/object/8800_faq.html

OpenGL. (2010, March 16). Retrieved April 20, 2010, from GLSL Sampler: http://www.opengl.org/wiki/GLSL_Sampler

Optical disc. (2010, April 26). Retrieved May 1, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Optical_disc

O'Rourke, J. (1994). Computational geometry in C. In J. O'Rourke, *Computational geometry in C* (pp. 32-44). New York: Cambridge University Press.

Orry, J. (2005, April 7). *Team Ninja chief concerned by Xbox 360 DVD media capacity.* Retrieved May 31, 2010, from videogamer.com: http://www.videogamer.com/news/team_ninja_chief_concerned_by_xbox_360_dvd_media_capacity.html

Owen, S. G. (1999, September 6). *Phong Model for Specular Reflection*. Retrieved May 20, 2010, from ACM SIGGRAPH: http://www.siggraph.org/education/materials/HyperGraph/illumin/specular_highlights/phong_model_specular_reflection.htm

Owens, J. (2007). GPU Architecture Overview. *Siggraph*, (p. 11). San Deigo.

Perlin, K. (1999). *Algorithm.* Retrieved April 23, 2010, from Noise Machine: http://www.noisemachine.com/talk1/15.html

Perlin, K. (1999). *Band limited repeatable 'random' function.* Retrieved April 21, 2010, from Noise Machine: http://www.noisemachine.com/talk1/14.html

Perlin, K. (1999). *Controlled Random Primitive.* Retrieved April 14, 2010, from Noise Machine: http://www.noisemachine.com/talk1/6.html

Perlin, K. (1999). *Making Noise*. Retrieved April 5, 2010, from NoiseMachine: http://www.noisemachine.com/talk1/12.html

Perlin, K. (2009). *Noise and Turbluence.* Retrieved April 22, 2010, from NYU Media Research Lab: http://mrl.nyu.edu/~perlin/doc/oscar.html#noise

Perlin, K. (1999). *Perlin Noise.* Retrieved April 25, 2010, from Noise Machine: http://www.noisemachine.com/talk1/12.html

PIX. (n.d.). Retrieved May 15, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/ee417062(v=VS.85).aspx

Postmortem: Naked Sky Entertainment's RoboBlitz. (2007, January 17). Retrieved March 26, 2010, from Gamasutra: http://www.gamasutra.com/view/feature/1737/postmortem_naked_sky_.php

Preprocessor Directives, DirectX HLSL. (2010, April 5). Retrieved May 21, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb943993(VS.85).aspx

Price, T. (2008). Insomniac Games Uses Autodesk Maya and Autodesk MotionBuilder to Re-imagine Recent American History in Resistance 2. (I. Autodesk, Interviewer)

PS3Focus. (2005, May 20). *Xbox360 vs PS3: System specs comparison*. Retrieved March 20, 2010, from Wikipedia: http://www.ps3focus.com/archives/40

QuadTerrain LOD: Finished. (2008, June 7). Retrieved June 2, 2010, from XNA Creators Club Online: http://forums.xna.com/forums/t/15397.aspx

Quake 3 Arena Screenshots. (2006, May 11). Retrieved May 24, 2010, from Softpedia®: http://linux.softpedia.com/screenshots/Quake-3-Arena_1.jpg

Quake III Arena. (2002, October 7). Retrieved May 24, 2010, from id Software: http://www.idsoftware.com/games/quake/quake3-arena/

Rasterisation. (2009). Retrieved May 21, 2010, from Graphics Acedemy: http://www.graphicsacademy.com/what_rasterisation.php

Rasterisation. (2010, March 8). Retrieved March 23, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Rasterisation

recv Function. (2010, May 14). Retrieved May 16, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/ms740121.aspx

Resource Types. (2010, April 5). Retrieved May 5, 2010, from MSND: http://msdn.microsoft.com/en-us/library/bb205133(v=VS.85).aspx#Buffer_Resources

Riley, D. M. (2008, January 31). *2007 U.S. Video Game And PC Game Sales Exceed $18.8 Billion Marking Third Consecutive Year Of Record-Breaking Sales*. Retrieved May 11, 2010, from The NPD Group: http://www.npd.com/press/releases/press_080131b.html

RoboBlitz. (2010, March 19). Retrieved March 24, 2010, from Wikipedia: http://en.wikipedia.org/wiki/RoboBlitz

RoboBlitz. (2010, March 19). Retrieved March 22, 2010, from Wikipedia: http://en.wikipedia.org/wiki/RoboBlitz

RoboBlitz. (2007). *The Game*. Retrieved March 26, 2010, from RoboBlitz: http://www.roboblitz.com/HTML_SITE/game/game.shtml

RoboBlitz, Gallery. (2006). Retrieved June 3, 2010, from RoboBlitz: http://www.roboblitz.com/HTML_SITE/gallery/Game%20Screens/tn_RoboBlitz01.jpg

Rosado, G. (2008). *Chapter 27. Motion Blur as a Post-Processing Effect.* Retrieved May 27, 2010, from NVidia: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch27.html

Rosen, D. (2010, January 8). *Why You Should Use OpenGL And Not DirectX*. Retrieved Maya 16, 2010, from Gamasutra: http://www.gamasutra.com/blogs/DavidRosen/20100108/4051/Why_You_Should_Use_OpenGL_And_Not_DirectX.php

Rossignol, J. (2008, May 29). *Technical Demonstration: Far Cry 2*. Retrieved May 3, 2010, from RockPaperShotgun: http://www.rockpapershotgun.com/2008/05/29/technical-demonstration-far-cry-2/

Runtime Random Level Generation. (2009, July 21). Retrieved March 24, 2010, from Procedural Content Generation: http://pcg.wikidot.com/pcg-algorithm:runtime-random-level-generation

Samyn, K. (2009, May 8). *DirectX9 Rendertarget*. Retrieved June 1, 2010, from knol: http://knol.google.com/k/directx9-rendertarget

SCEI. (2010). *UNIT SALES OF HARDWARE (SINCE APRIL 2006)*. Retrieved May 11, 2010, from Sony Computer Entertainment Inc.: http://www.scei.co.jp/corporate/data/bizdataps3_sale_e.html

Scheidt, G. v. (2005, March 2). *Avid to Showcase Game Content Creation and Asset Management Solutions at GDC 2005*. Retrieved May 19, 2010, from The3DStudio.com: http://www.the3dstudio.com/product_details.aspx?id_product=4161

Segal, M., & Akeley, K. (2010, March 11). *The OpenGL Graphics System: A Specification (Version 4.0 (Core Profile))*. Retrieved May 16, 2010, from OpenGL: http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf

Seyringer, D. H. (2003, December 11). *Lighting and Materials* . Retrieved June 1, 2010, from NatureWizard: http://www.naturewizard.com/Tutorials/Tutorial01/images/image013.jpg

Shader. (2010, March 4). Retrieved May 20, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Shader

Shader Model 4. (2010, April 5). Retrieved May 20, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb509657(VS.85).aspx

Shader Stages. (2010, April 5). Retrieved April 23, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb205146(VS.85).aspx

Shader Stages. (2010, April 5). Retrieved May 12, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb205146(VS.85).aspx

Sidhe. (2010, April 17). Retrieved March 12, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Sidhe_(game_developer)

Sidhe Interactive Online. (2009). Retrieved May 16, 2010, from Sidhe Interactive: http://www.sidhe.co.nz/091126.htm

Simmons, B. (2008, July 29). *Degenerate*. Retrieved May 25, 2010, from Mathwords: http://www.mathwords.com/d/degenerate.htm

Simpson, D. (2009, September 8). *Multiplatform Games vs. Exclusives: What's the Difference Between Me and You?* Retrieved May 18, 2010, from The Gamer Access: http://www.thegameraccess.com/articles/multiplatform/multiplatform-games-vs-exclusives-whats-the-difference-between-me-and-you

Sondergaard, H., & Sestoft, P. (1990). *Referential transparency, definiteness and unfoldability.* Retrieved April 17, 2010, from The University of Copenhagan, Copenhagan, Denmark: http://www.itu.dk/people/sestoft/papers/SondergaardSestoft1990.pdf

Source (game engine). (2010, April 29). Retrieved May 1, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Source_(game_engine)

Stream-Output Object. (2010, April 7). Retrieved June 1, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb509661(v=VS.85).aspx

Stream-Output Object. (2010, 4 5). Retrieved March 24, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb509661(VS.85).aspx

Stream-Output Stage. (2010, 4 5). Retrieved 3 13, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/bb205121(v=VS.85).aspx

Sweeney, T. (2009, May 26). Epic's Sweeney: Games Are 'Factor Of 1000' Off From Graphical Realism. (B. Edwards, Interviewer) Gamasutra.

Tangent space. (2010, May 6). Retrieved April 14, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Tangent_space

Tech ARP. (2010). *Desktop Graphics Card Comparision Guide*. Retrieved April 13, 2010, from Tech ARP: http://www.techarp.com/showarticle.aspx?artno=88&pgno=7

Terminal Reality. (2009). Retrieved May 14, 2010, from Autodesk, Inc: http://images.autodesk.com/adsk/files/customerstory_ghostbusters_3ds_max_maya.pdf

The Financial Express. (2007, December 16). *Media and entertainment industry growth to double soon: study*. Retrieved May 12, 2010, from The Financial Express: http://www.financialexpress.com/news/media-and-entertainment-industry-growth-to-double-soon-study/252134/

The Home Video Game Console. (n.d). Retrieved May 27, 2010, from thegameconsole.com: http://www.thegameconsole.com/

The Sentinel . (2010). Retrieved April 12, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Procedural_generation

tr_noise.c. (2005). *Quake III Arena GPL source release* , Lines 25- 96. Dallas, Texas, United States: id Software.

tr_shade_calc.c. (2010, May 24). *Quake III Arena GPL source release* . Dallas, Texas, United States: id Software.

UE2:UnrealEd 3. (2008, November 22). Retrieved April 23, 2010, from Unreal Wiki: http://wiki.beyondunreal.com/UE2:UnrealEd_3

Unreal. (2010, April 24). Retrieved April 22, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Unreal

UnrealEd. (2010). Retrieved April 13, 2010, from Unreal Engine: http://www.unrealtechnology.com/features.php?ref=editor

Video game console. (2010, May 16). Retrieved May 18, 2010, from Wikipedia: http://en.wikipedia.org/wiki/Video_game_console

Wagner, A. (n.d). *barrels in videogames*. Retrieved June 2, 2010, from Armin Wagner: http://www.arminbwagner.com/crates_and_barrels/barrel_ep1barrel.jpg

Warman, P. (2010, March 5). *Newzoo Games MarketReport.* Retrieved May 18, 2010, from Newzoo: http://corporate.newzoo.com/press/GamesMarketReport_FREE_030510.pdf

Weisstein, E. (2010). *Affine Transformation.* Retrieved March 25, 2010, from MathWorld: http://mathworld.wolfram.com/AffineTransformation.html

Weisstein, E. (2010). *Binormal Vector*. Retrieved April 12, 2010, from Mathworld: http://mathworld.wolfram.com/BinormalVector.html

What is GPU Computing? (2010). Retrieved May 27, 2010, from NVidia: http://www.nvidia.com/object/GPU_Computing.html

What is MaPZone? (2010). Retrieved May 31, 2010, from Allegorithmic: http://www.mapzoneeditor.com/?PAGE=FEATURES

Whiting, M. (2007, January 16). *Brian Eno Doing Spore 'Soundtrack'*. Retrieved June 3, 2010, from 1UP: http://www.1up.com/do/newsStory?cId=3156407

Windows Sockets 2. (2010, May 13). Retrieved May 16, 2010, from MSDN: http://msdn.microsoft.com/en-us/library/ms740673.aspx

Wolfenstein | Media. (2009). Retrieved May 19, 2010, from Wolfenstein: http://cdn.wolfenstein.com/wolfenstein/media/50/image_003_lrg.jpg

Woody. (2010). *Woodland.* Retrieved May 28, 2010, from Lionhead Studios: http://lionhead.com/media/p/3462182.aspx

Appendix A

	Name	Support	Citation	Notes
Open Source	Delta3D	OpenGL 2.0	(Delta3D - Features)	
	Irrlicht	Direct3D 8.1 Direct3D 9.0 OpenGL 1.2 – 3.x	(Gebhardt, 2009)	
	Ogre3D	Direct3D OpenGL	(Features, Ogre3D, 2009)	Latest API versions require custom implementation
Commercial	Source Engine (Valve)	Direct3D OpenGL OpenGL ES 2.0	(Source (game engine), 2010)	
	CryEngine (3.0)	Direct3D 9.0 Direct3D 10 Direct3D 11	(CryEngine 3.0, 2010)	Free version announced April 12, 2010 (CryEngine 3.0, 2010)
	Unreal Engine 3 (Free version announced late 2009)	Direct3D 9.0 Direct3D 10 Direct3D 11	(Current Technology - Unreal Engine 3, 2010)	

Appendix B

	Xbox 360	Playstation 3	PC
Memory	512 MB (GDDR3)	256 MB (XDR) 256 MB (GDDR3)	Assuming a mid-range, modern PC is running Windows Vista, deduce PC system memory to be greater than or equal to 512mb (Microsoft Windows Vista (Basic) requires 512 MB of system memory (Get Windows Vista: System requirements, 2010)
Storage capacity	20-250GB	None – 250GB	20GB required for Windows Vista. Storage capacity averages > 500GB

Appendix C

The following pseudo code extracts tangent space from a triangle with texture coordinates. This code is adapted from material on page 82 of 'Shader X[4]: Advanced Rendering Techniques' (ISBN 1-58450-425-0):

- The triangle is represented by three 3D points/vertices:
  p_A, p_B, p_C
- 2D texture coordinates at each triangle vertex are represented by:
  uv_A, uv_B, uv_C
- 3D vectors that define the triangle's tangent space are represented by:
  T, B, N

$v_A = p_B - p_A$

$v_B = p_C - p_A$

$d_{U1} = uv_{Bx} - uv_{Ax}$

$d_{U2} = uv_{Cx} - uv_{Ax}$

$d_{V1} = uv_{By} - uv_{Ay}$

$d_{V2} = uv_{Cy} - uv_{Ay}$

$div = (d_{U1} \times d_{V2} - d_{U2} \times d_{V1})$

```
if(div != 0.0)  {
      a = dV2/div
      b = -dV1/div
      c = -dU2/div
      d = dU1/div
      T = normalize( vA * a + vB * b)
      B = normalize( vA * c + vB * d)
      N = cross(T, B)
}
```