Undiscovered Worlds
Real-Time Procedural Generation of Virtual
Three-Dimensional Spaces

Stefan Greuter
Diplom-Informatiker (FH) Digitale Medien
(equivalent to BCompSc(Hons) Digital Media)

Exegesis documentation and Appendices submitted with the Durable Visual Record
of the Project Materials (CD-ROM, Undiscovered Worlds) in fulfilment
of the requirements for the degree

Doctor of Philosophy

School of Creative Media
Portfolio of Design and Social Context
RMIT University
Melbourne, Victoria, Australia.

June 2008

Supervisors
Jeremy Parker, Geoff Leach and Nigel Stewart
Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Stefan Greuter

21.07.2008
Acknowledgements

I would like to thank my senior supervisor, Jeremy Parker, who was always there for me during times of crisis and frustration during the many years of this research project, and without whose guidance and encouragement this exegesis would not be what it is today.

The interdisciplinary nature of a doctorate by project in Animation and Interactive Media led me to seek advice and support in the School of Computer Science at RMIT University. I would like to thank my technical supervisor Nigel Stewart, whose energetic and attentive enthusiasm were invaluable in the complex task of programming the work described herein. I am forever indebted to Geoff Leach and Nigel Stewart for all their help in writing up several research papers, which included an introduction to Latex, numerous revisions, late nights and many long discussions.

For funding, I sincerely thank the German Academic Exchange Service (DAAD) for granting me a scholarship for the first two years of this research project. I also thank the Australian Government as well as RMIT University for granting me an International Postgraduate Research Scholarship (IPRS) to continue my studies at a doctoral level. I would like to thank the School of Creative Media for providing me with funding and for granting me study leave.

Finally, I would like to thank my wonderful wife, Fiona and her father Ian, for the advice, encouragement and help they provided to me during my research.

Without doubt there will be errors, omissions and over-simplifications, for which I take absolute responsibility, as is customary, while hoping that the rest of the material will be enough to stimulate insights and new trains of thought.
# Table of Contents

Declaration ............................................................................................................................................... i  
Acknowledgements ................................................................................................................................. ii  
Table of Contents ................................................................................................................................... iii  
Table of Figures ..................................................................................................................................... xi  
Abstract ................................................................................................................................................... 1  
1  Introduction ..................................................................................................................................... 3  
   1.1  Overview .................................................................................................................................. 3  
   1.2  Components of this PhD ........................................................................................................ . 3  
   1.3  Definition of Terms ................................................................................................................. 3  
      1.3.1  Electronic Games ............................................................................................................ 3  
      1.3.2  Game Worlds and Game Levels...................................................................................... 4  
      1.3.3  Virtual Space ................................................................................................................... 4  
      1.3.4  Repetition ........................................................................................................................ 4  
      1.3.5  Variety ............................................................................................................................. 4  
      1.3.6  Visual Variety ................................................................................................................ . 4  
      1.3.7  Real-time Rendering ....................................................................................................... 5  
   1.4  The Dawn of Interactive Electronic Entertainment ................................................................. 5  
   1.5  Increase in Size and Detail of Game Worlds ............................................................................ 5  
      1.5.1  Increase in Detail............................................................................................................. 6  
      1.5.2  Increase in Game World Size .......................................................................................... 6  
   1.6  Hire More Artists ....................................................................................................................... 7  
   1.7  The Issue of Repetition ........................................................................................................... 7  
   1.8  Current Approaches to Alleviate Repetition .......................................................................... 8  
   1.9  The Impact of Repetition ......................................................................................................... 8
# Table of Content

1.9.1 Immersion .......................................................................................................................... 8
1.9.2 Presence ............................................................................................................................ 9
1.9.3 The Influence of Repetition on Presence ...................................................................... 10
1.10 Novel Approach to Increase the Illusion of Presence ................................................... 11
1.11 Summary ............................................................................................................................. 11
1.12 Exegesis Outline .................................................................................................................. 11

2 Background ............................................................................................................................ 12
2.1 Overview ............................................................................................................................. 12
2.2 The Practice of Game World Creation .............................................................................. 12
2.2.1 3D Modelling Techniques Used in Game Development ........................................... 12
2.2.2 Texturing Techniques Used in Game Development .................................................... 13
2.2.3 Level Editors ............................................................................................................... 14
2.2.4 Monohedral Tilings ....................................................................................................... 14
2.3 Procedural Generation ......................................................................................................... 15
2.3.1 Iterated Function Systems ........................................................................................... 15
2.3.2 Random Number Generators ....................................................................................... 15
2.3.3 Noise ............................................................................................................................. 16
2.4 Formalised Approaches to Procedural Generation ............................................................ 17
2.4.1 Fractals ......................................................................................................................... 17
2.4.2 L-Systems ................................................................................................................... 18
2.5 Related Work ....................................................................................................................... 18
2.5.1 Generative Art .............................................................................................................. 18
2.5.2 Procedural Generation for Time-based Media ............................................................ 19
2.5.3 Procedural Generation Used in Games ....................................................................... 20
2.5.4 Real-time Procedural Generation ............................................................................... 20
2.6 Summary ............................................................................................................................. 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Methodology</td>
</tr>
<tr>
<td>3.1</td>
<td>Motivation</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Procedural Generation</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Objects and Virtual Spaces</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Visual Variety</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Real-time 3D</td>
</tr>
<tr>
<td>3.2</td>
<td>Research Questions</td>
</tr>
<tr>
<td>3.3</td>
<td>Research Through Design</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Iterative Design</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Software Prototypes</td>
</tr>
<tr>
<td>3.4</td>
<td>Data</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Qualitative Data</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Qualitative Data Collection</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Qualitative Data Analysis</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Quantitative Data</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Development and Test Platform</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Quantitative Data Collection</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Quantitative Data Analysis</td>
</tr>
<tr>
<td>3.5</td>
<td>Validation</td>
</tr>
<tr>
<td>3.6</td>
<td>Summary</td>
</tr>
<tr>
<td>4</td>
<td>Prototypes</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of Prototypes</td>
</tr>
<tr>
<td>4.2</td>
<td>Prototype 1: MaxScript</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Crates in Electronic Games</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Crates as Candidates for Real-Time Procedural Generation</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Crate Geometries</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Procedural Crate Generation using Max Script</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Analysis of the Generated Visual Variety</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Analysis of Real-time Performance</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Limitations</td>
</tr>
<tr>
<td>4.2.8</td>
<td>New Questions</td>
</tr>
<tr>
<td>4.2.9</td>
<td>Summary</td>
</tr>
<tr>
<td>4.3</td>
<td>Prototype 2: Lattice Examiner</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Real World Wooden Crates</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Random Texture Coordinate Generation</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Analysis of the Generated Visual Variety</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Analysis of Real-time Performance</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Limitations</td>
</tr>
<tr>
<td>4.3.6</td>
<td>New Questions</td>
</tr>
<tr>
<td>4.3.7</td>
<td>Summary</td>
</tr>
<tr>
<td>4.4</td>
<td>Prototype 3: Crate Examiner</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Real-time Procedural Crate Generation</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Analysis of the Generated Visual Variety</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Analysis of Real-time Performance</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Limitations</td>
</tr>
<tr>
<td>4.4.5</td>
<td>New Questions</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Summary</td>
</tr>
<tr>
<td>4.5</td>
<td>Prototype 4: Pile Examiner</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Generating Crate Piles</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Analysis of the Generated Visual Variety</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Analysis of Real-time Performance</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Limitations</td>
</tr>
<tr>
<td>4.5.5</td>
<td>New Questions</td>
</tr>
</tbody>
</table>
4.5.6 Summary ........................................................................................................................50

4.6 Prototype 5: Warehouse .........................................................................................................51

4.6.1 Generating a Warehouse ................................................................................................51
4.6.2 View Frustum Filling ......................................................................................................51
4.6.3 Dynamic Procedural Generation ....................................................................................52
4.6.4 Hashing .......................................................................................................................53
4.6.5 Theoretical Size of the Virtual Space .............................................................................54
4.6.6 Reduction of Crate Polygons ..........................................................................................54
4.6.7 Analysis of the Generated Visual Variety ......................................................................54
4.6.8 Analysis of Real-time Performance ...............................................................................56
4.6.9 Limitations ...................................................................................................................56
4.6.10 New Questions .............................................................................................................56
4.6.11 Summary ........................................................................................................................56

4.7 Prototype 6: Building Examiner Version 1 ............................................................................58

4.7.1 Generating Buildings ......................................................................................................58
4.7.2 Analysis of the Generated Visual Variety ......................................................................59
4.7.3 Analysis of Real-time Performance ...............................................................................60
4.7.4 Limitations ...................................................................................................................61
4.7.5 New Questions .............................................................................................................61
4.7.6 Summary ........................................................................................................................61

4.8 Prototype 7: Building Examiner Version 2 ............................................................................62

4.8.1 Generating Buildings ......................................................................................................62
4.8.2 Floor Plan Generation ....................................................................................................63
4.8.3 Building Facade Generation ...........................................................................................64
4.8.4 Building Facade Texture Mapping ...............................................................................65
4.8.5 Analysis of the Generated Visual Variety ......................................................................66
4.8.6 Analysis of Real-time Performance ...............................................................................70
4.8.7 Limitations ................................................................................................................... 70
4.8.8 New Questions ............................................................................................................... 70
4.8.9 Summary ........................................................................................................................ 71
4.9 Prototype 8: Undiscovered City ............................................................................................ 72
  4.9.1 Real and Virtual City Spaces ......................................................................................... 72
  4.9.2 Generating Cities ............................................................................................................ 72
  4.9.3 Caching ....................................................................................................................... 73
  4.9.4 Display List Caching ...................................................................................................... 73
  4.9.5 OpenGL Display Lists .................................................................................................... 74
  4.9.6 LRU Cache ..................................................................................................................... 74
  4.9.7 LRU Cache Implementation ........................................................................................... 74
  4.9.8 Display List Cache Implementation ............................................................................... 76
  4.9.9 The Theoretical Size of the Undiscovered City ............................................................. 78
  4.9.10 Analysis of the Generated Visual Variety ................................................................... 78
  4.9.11 Analysis of Real-time Performance ............................................................................. 79
  4.9.12 OpenGL Display Lists and Procedural Building Generation ......................................... 80
  4.9.13 LRU Performance Characteristics ............................................................................ 82
  4.9.14 Undiscovered City Performance ................................................................................. 83
  4.9.15 Limitations ................................................................................................................... 84
  4.9.16 New Questions ............................................................................................................. 84
  4.9.17 Summary ...................................................................................................................... 84
4.10 Prototype 9: Street Grid Examiner ......................................................................................... 86
  4.10.1 Generating Street Grids Based on Integer Grid .............................................................. 86
  4.10.2 Street Grid Generation .................................................................................................. 87
  4.10.3 Node Displacement ...................................................................................................... 88
  4.10.4 Street Geometry Generation ......................................................................................... 89
  4.10.5 Lot Shape Generation .................................................................................................. 93
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10.6</td>
<td>Terrain Generation</td>
<td>95</td>
</tr>
<tr>
<td>4.10.7</td>
<td>Analysis of the Generated Visual Variety</td>
<td>96</td>
</tr>
<tr>
<td>4.10.8</td>
<td>Analysis of Real-time Performance</td>
<td>99</td>
</tr>
<tr>
<td>4.10.9</td>
<td>Limitations</td>
<td>99</td>
</tr>
<tr>
<td>4.10.10</td>
<td>New Questions</td>
<td>99</td>
</tr>
<tr>
<td>4.10.11</td>
<td>Summary</td>
<td>99</td>
</tr>
<tr>
<td>4.11</td>
<td>Prototype 10: Neverland Framework</td>
<td>100</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Framework Structure</td>
<td>100</td>
</tr>
<tr>
<td>4.11.2</td>
<td>Node Based View Frustum Filling</td>
<td>101</td>
</tr>
<tr>
<td>4.11.3</td>
<td>Geometry Generation</td>
<td>102</td>
</tr>
<tr>
<td>4.11.4</td>
<td>Building Geometry Generation Based on Lot Shape</td>
<td>102</td>
</tr>
<tr>
<td>4.11.5</td>
<td>Building Height Distribution</td>
<td>103</td>
</tr>
<tr>
<td>4.11.6</td>
<td>Modes of Control</td>
<td>104</td>
</tr>
<tr>
<td>4.11.7</td>
<td>Analysis of the Generated Visual Variety</td>
<td>105</td>
</tr>
<tr>
<td>4.11.8</td>
<td>Analysis of Real-time Performance</td>
<td>105</td>
</tr>
<tr>
<td>4.11.9</td>
<td>Limitations</td>
<td>106</td>
</tr>
<tr>
<td>4.11.10</td>
<td>Summary</td>
<td>107</td>
</tr>
<tr>
<td>5</td>
<td>Conclusion</td>
<td>108</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>108</td>
</tr>
<tr>
<td>5.2</td>
<td>Research Outcomes</td>
<td>108</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Procedural Generation of Object Geometries in a Visual Variety</td>
<td>109</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Texture</td>
<td>109</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Distribution of Generated Objects in a Visual Variety</td>
<td>110</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Real-time Performance of a Procedurally Generated Virtual Space</td>
<td>111</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Contributions</td>
<td>111</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Impact</td>
<td>112</td>
</tr>
</tbody>
</table>
5.3 Areas and Directions of Future Research ......................................................... 113
5.4 Potential Applications .................................................................................. 115
5.5 Final Conclusions ...................................................................................... 115
6 References .................................................................................................. 117
Appendix: Prototype Documentation ............................................................... 122
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Detail comparison of Doom and Doom 3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>GTA size comparison</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Generated outdoor terrain spaces</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Commercial applications</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Research pathway map</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>Crates in electronic games</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>Sketch of the crate object template</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>Crate geometries and their generation settings</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>Selected generated crate objects</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>Diversely textured box objects</td>
<td>42</td>
</tr>
<tr>
<td>11</td>
<td>Visual variety of frame volume</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>Visual variety in frequency evident as change in formal structure</td>
<td>45</td>
</tr>
<tr>
<td>13</td>
<td>Visual variety in space</td>
<td>45</td>
</tr>
<tr>
<td>14</td>
<td>Visual variety in colour</td>
<td>45</td>
</tr>
<tr>
<td>15</td>
<td>Visual variety of texture</td>
<td>46</td>
</tr>
<tr>
<td>16</td>
<td>Visual variety of crate objects</td>
<td>46</td>
</tr>
<tr>
<td>17</td>
<td>Crate piles in a visual variety of volume, dimension, size, colour, texture and space</td>
<td>49</td>
</tr>
<tr>
<td>18</td>
<td>View frustum culling and view frustum filling compared</td>
<td>52</td>
</tr>
<tr>
<td>19</td>
<td>View frustum visibility testing</td>
<td>53</td>
</tr>
<tr>
<td>20</td>
<td>Reduction of crate polygons</td>
<td>54</td>
</tr>
<tr>
<td>21</td>
<td>Warehouse showing visual variety in from, texture and colour</td>
<td>55</td>
</tr>
<tr>
<td>22</td>
<td>Warehouse showing no visual variety</td>
<td>55</td>
</tr>
<tr>
<td>23</td>
<td>Building template</td>
<td>59</td>
</tr>
<tr>
<td>24</td>
<td>Building objects showing a visual variety of dimension, scale and volume</td>
<td>59</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>25</td>
<td>Visual variety of window textures</td>
<td>60</td>
</tr>
<tr>
<td>26</td>
<td>Building objects showing a visual variety of form</td>
<td>60</td>
</tr>
<tr>
<td>27</td>
<td>Buildings with obvious floor plan shapes</td>
<td>62</td>
</tr>
<tr>
<td>28</td>
<td>Buildings with floor plans that are extruded in height</td>
<td>63</td>
</tr>
<tr>
<td>29</td>
<td>Iterative floor plan generation</td>
<td>64</td>
</tr>
<tr>
<td>30</td>
<td>Building facade generation</td>
<td>65</td>
</tr>
<tr>
<td>31</td>
<td>Building facade texturing</td>
<td>66</td>
</tr>
<tr>
<td>32</td>
<td>Building geometries of various complexity</td>
<td>67</td>
</tr>
<tr>
<td>33</td>
<td>Building geometries with various number of setbacks</td>
<td>68</td>
</tr>
<tr>
<td>34</td>
<td>Building variation with 3 setbacks and a complexity of 4</td>
<td>69</td>
</tr>
<tr>
<td>35</td>
<td>Building variation with 9 setbacks and a complexity of 10</td>
<td>69</td>
</tr>
<tr>
<td>36</td>
<td>Building variation with 19 setbacks and a complexity of 20</td>
<td>69</td>
</tr>
<tr>
<td>37</td>
<td>Buildings with a high number of setbacks and complexity</td>
<td>70</td>
</tr>
<tr>
<td>38</td>
<td>Screen shot of the Undiscovered City prototype from bird’s eye view</td>
<td>73</td>
</tr>
<tr>
<td>39</td>
<td>Diagram of the LRU cache using a doubly linked list and a balanced tree</td>
<td>75</td>
</tr>
<tr>
<td>40</td>
<td>The Undiscovered City viewed at street level</td>
<td>78</td>
</tr>
<tr>
<td>41</td>
<td>Undiscovered City prototype with different textures</td>
<td>79</td>
</tr>
<tr>
<td>42</td>
<td>View with 100, 200, 400 or 800 buildings</td>
<td>80</td>
</tr>
<tr>
<td>43</td>
<td>Generation, display list compilation and drawing</td>
<td>81</td>
</tr>
<tr>
<td>44</td>
<td>Display list compilation overhead in percent</td>
<td>81</td>
</tr>
<tr>
<td>45</td>
<td>Display list drawing speedup factor</td>
<td>82</td>
</tr>
<tr>
<td>46</td>
<td>Display list cache hit</td>
<td>83</td>
</tr>
<tr>
<td>47</td>
<td>Display list cache miss</td>
<td>83</td>
</tr>
<tr>
<td>48</td>
<td>City performance</td>
<td>84</td>
</tr>
<tr>
<td>49</td>
<td>The integer grid and its node id’s</td>
<td>87</td>
</tr>
<tr>
<td>50</td>
<td>Generated street grids</td>
<td>88</td>
</tr>
<tr>
<td>51</td>
<td>Street geometry cases</td>
<td>89</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>52</td>
<td>Curve intersection example from node of interest to first path trace</td>
<td>90</td>
</tr>
<tr>
<td>53</td>
<td>Second path trace and offset vertices</td>
<td>91</td>
</tr>
<tr>
<td>54</td>
<td>Street grid polygonisation</td>
<td>92</td>
</tr>
<tr>
<td>55</td>
<td>Lot tracing</td>
<td>94</td>
</tr>
<tr>
<td>56</td>
<td>Lot tracing continued</td>
<td>94</td>
</tr>
<tr>
<td>57</td>
<td>Generated street grid on terrain</td>
<td>96</td>
</tr>
<tr>
<td>58</td>
<td>Procedural street grids without displacement</td>
<td>97</td>
</tr>
<tr>
<td>59</td>
<td>Satellite image of Melbourne, Australia</td>
<td>97</td>
</tr>
<tr>
<td>60</td>
<td>Procedural street grids with displacement</td>
<td>98</td>
</tr>
<tr>
<td>61</td>
<td>Satellite image of Rome, Italy</td>
<td>98</td>
</tr>
<tr>
<td>62</td>
<td>Diagram of the Neverland Framework</td>
<td>101</td>
</tr>
<tr>
<td>63</td>
<td>The integer grid with view frustum filling shape and visible nodes</td>
<td>102</td>
</tr>
<tr>
<td>64</td>
<td>Building height determined by single height map</td>
<td>103</td>
</tr>
<tr>
<td>65</td>
<td>Modes of control that are integrated into the Neverland Framework</td>
<td>105</td>
</tr>
<tr>
<td>66</td>
<td>Street grid and procedural city</td>
<td>106</td>
</tr>
<tr>
<td>67</td>
<td>Game project</td>
<td>112</td>
</tr>
<tr>
<td>68</td>
<td>The Lost Babylon theatre play</td>
<td>113</td>
</tr>
<tr>
<td>69</td>
<td>User interface, settings and resulting crate object</td>
<td>122</td>
</tr>
<tr>
<td>70</td>
<td>Lattice examiner</td>
<td>124</td>
</tr>
<tr>
<td>71</td>
<td>Crate Examiner</td>
<td>126</td>
</tr>
<tr>
<td>72</td>
<td>Pile Examiner</td>
<td>128</td>
</tr>
<tr>
<td>73</td>
<td>Warehouse</td>
<td>130</td>
</tr>
<tr>
<td>74</td>
<td>Building Examiner</td>
<td>132</td>
</tr>
<tr>
<td>75</td>
<td>The Undiscovered City</td>
<td>134</td>
</tr>
<tr>
<td>76</td>
<td>Street Grid Examiner</td>
<td>136</td>
</tr>
<tr>
<td>77</td>
<td>Neverland Examiner</td>
<td>137</td>
</tr>
<tr>
<td>78</td>
<td>Neverland Viewer</td>
<td>140</td>
</tr>
</tbody>
</table>
Abstract

This study attempted to create a game world of enormous proportions that consisted of a visual variety of procedurally generated and distributed objects. A framework was designed to accommodate several procedural generation approaches for the various components of the virtual space.

Game worlds often contain a large number of objects that make up the environment in which the games are played. The objects are generally created by game artists. However, the manual creation of game objects can be very time consuming, and in order to create especially large game worlds more effectively, it is a common practice to reuse objects several times in a game world to fill it with content.

Game worlds have become increasingly larger and more detailed over the last decade. The approach to manually create all the objects in a game world has become more and more problematic as the increase of workload demands a higher commitment of time and resources and the current solution to “hire more artists” does not scale well with the higher demands of future game worlds.

The other approach to reuse a limited collection of objects to fill a game world with content is also problematic, as this repetition can be recognised by the players. The importance of variation and randomness of objects in a game world grows especially with the ability to render such objects with more detail. As a result, the frequent repetition of geometries and textures becomes more obvious and runs the risk of boring the player. Moreover, such frequent repetition may have a negative effect on the illusion of presence that the player experiences, when playing a game. The experience of the illusion of presence is believed to be an essential factor regarding the usefulness and profitability of computer games.

This research project investigated how three-dimensional virtual spaces such as game worlds can be created more effectively and was guided by two research questions. The first research question asked how three-dimensional objects can be generated in a visual variety using computer software. To this end, this project investigated several procedural generation approaches to generate objects in a visual variety, addressing the problem of repetition in game worlds. In particular, this research project investigated how objects that are very common to many current game worlds such as crates and buildings can be procedurally generated.
The second research question extended the first research question and asked how three-dimensional virtual spaces that consist of such generated objects in a visual variety can be generated in real-time using computer software. To this end the investigation focussed on approaches to distribute generated objects within a virtual space as well as issues of real-time performance and finished with the development of a framework that combined the findings.

This inquiry used an iterative design approach, which involved the creation of prototypes, which were subsequently, analysed and redesigned to progress knowledge. The prototypes demonstrate and document the results at various stages of this project and are an integral part of this research.

Overall, the study has contributed some weight to the argument that game worlds can be created using a procedural generation approach. Unlike current manual approaches to construct large game worlds which require an increasing number of game artists or the repetitive reuse of objects, this procedural generation approach is potentially faster than the manual creation approach and does not rely on object repetition to fill a virtual space with content. All objects within such a virtual space exist in a visual variety. As such it is believed that especially for large virtual spaces, this approach of procedural generation contributes to a greater illusion of presence experienced by the player.
1 Introduction

1.1 Overview
This chapter provides the context of this research project, describes the problem of repetition in electronic games and outlines the approach taken by this research project. It starts with a description of the components that are submitted as part of this doctoral investigation and a definition of the essential terms that are used within the exegesis. Next follows a brief historical outline about the early days of interactive electronic entertainment, which leads into a short description of the context and the issue of frequent reuse of objects in game worlds. This follows a discussion about the theoretical model of presence and the influence of repetition on presence to elucidate why repetition is a problem in regards to electronic games. A brief description explains the approach taken by this research project, which is believed to achieve a greater illusion of presence through real-time procedural generation. The chapter concludes with a summary and a section outlining the structure of the exegesis.

1.2 Components of this PhD
In accordance with RMIT University guidelines (2002), this research project consists of two parts. The first part is the written exegesis, which describes the purpose, context and theoretical background of the research project as well as the process of knowledge production. In essence, the exegesis answers the questions of how the project has been developed and what has been achieved. The second part is the observable and durable record of the completed project and is represented by the CD that accompanies this exegesis. The CD contains a menu which facilitates the execution of the prototypes that have been developed by this research project and provides access to the source code.

1.3 Definition of Terms
In order to establish a firm conceptual framework for this exegesis, several key terms with varying connotations in the reviewed literature are defined as follows.

1.3.1 Electronic Games
This exegesis uses the umbrella term electronic games to describe the many platforms such as arcade games, video games, computer games, PC games, console games etc.
1.3.2 Game Worlds and Game Levels
Morris and Hartas (2004) use the term “game world” to describe the visual, audible and interactive experiences of an electronic game. Such game worlds can be further subdivided into individual segments which are often called game levels. All game levels usually share the overall style of representation and usually provide similar audible and interactive experiences. Game levels can be two-dimensional or three-dimensional virtual spaces that constitute the environment in which gameplay takes place. The terrain, objects, sounds and music that comprise a game level are often described by the term game assets or game content.

1.3.3 Virtual Space
This research project focuses on the purely visual and inanimate component of a game world, which is defined by the term virtual space. The purely visual and inanimate component of a game world typically consists of textured polygon objects that are distributed on a terrain and remain static while a user is navigating through the space.

1.3.4 Repetition
Repetition has been defined as the arrangement of several objects with identical design features (Leborg, 2006, p. 40). Repetition can be noticed by the viewer in form of single design elements such as line, shape, form, texture and colour (Zelanski & Fisher, 2007a).

1.3.5 Variety
Variety has been described as a companion of repetition and is defined as “change rather than sameness through space and time” (Zelanski & Fisher, 2007a). Variety can also be described as organisational principle, which is characterised through a form of order that can be discovered by the viewer and consists of “parts that are seemingly different from each other, nonetheless have something in common” (Zelanski & Fisher, 2007b).

1.3.6 Visual Variety
As this research project focuses on the purely visual and inanimate component of a game world it was necessary to use the term visual variety to describe the variety that can be experienced by differences of visual design elements such as line, shape, form, texture and colour within a rendered image.
1.3.7 Real-time Rendering

Real-time applications respond to stimuli within a limited response time, typically in the range of milliseconds or microseconds (Howe, 1993). Real-time rendering is described as a process in which a user acts or reacts to an image on the screen, resulting in an appropriate change of the image. The cycle of user interaction and image rendering is measured by the number of displayed frames per second (fps) (Akenine-Möller & Haines, 2002, p. 1). A frame rate of between 15 fps and 72 fps is recommended, to create for the viewer the illusion of a dynamic process rather than a sequence of individual images. Given these limitations, the software and hardware that renders the three-dimensional space needs to perform all calculations in a time frame between 13.8 ms and 66.6 ms. Film and video frame rates are standardised for PAL to 25 fps and NTSC to 30 fps. Electronic real-time 3D games, however, are often experienced at refresh rates of 60 fps or higher.

1.4 The Dawn of Interactive Electronic Entertainment

The era of interactive electronic entertainment began in 1948 with the idea of a Code Ray Tube Amusement device, which was subsequently filed as a patent (Goldsmith Jr. & Mann, 1948). In 1958, William A. Higinbotham created a two player game called “Tennis for Two”, which was displayed on an oscilloscope and controlled by basic push buttons. In 1962, Stephen Russell, Martin Graetz and Wayne Wiitanen, developed the game Spacewar on a DEC PDP-1 computer, which is often regarded as the first computer game. In 1967, Ralph Baer created the first video game called Chase, to be played on a television set. Computer game enthusiasts emerged who could enjoy various forms of game-play inside a virtual play arena. This virtual play area, also often described as game level or game world, is the focus of this research project.

1.5 Increase in Size and Detail of Game Worlds

Will Wright claimed at GDC 2005 that: “modern games demand more and more content” (Kosak, 2005). Game worlds have become increasingly larger and more detailed over the last decade on a par with the increased processing capabilities of hardware and the availability of more effective tools. This trend is likely to continue, as computer hardware still continues to improve. Moore’s Law (Moore, 1965), although criticised but commonly acknowledged predicts that CPU processing speed doubles every two years. It is claimed by nVidia (Kirk, 2002) that GPU processing speeds are increasing even faster, doubling every six months. Software libraries such as DirectX and OpenGL and artist tools such as Autodesk’s 3D Studio Max and Maya have become available and are continuously improved to allow game developers to deal with the challenges of an increasing workload and the increasing expectations of players.
1.5.1 Increase in Detail
The increase in detail is resulting in longer development times. This has become apparent in Id software’s Doom series. Tim Willits, lead designer at Id Software (www.idsoftware.com), states that “it takes nearly 12 times as long to lay out a virtual foot in Doom 3 as it took in the original version of Doom – despite the use of much more powerful tools” (Kent, 2004, p. 117), where “virtual foot” in this context refers to the Imperial distance unit, not the physical foot model of a game character. The first version of Doom was released in 1993 and Doom 3 in 2004. For Id Software this represents a 12 fold increase in workload within 11 years of game development. Figure 1 shows two screen captures taken from Doom and Doom 3 which exemplify the visual difference in detail.

![Doom (1993)](image1) ![Doom 3 (2004)](image2)

Figure 1: Detail comparison of Doom and Doom 3.
Reprinted with permission.

1.5.2 Increase in Game World Size
The increase in game world size can be observed in the game worlds of Rockstar’s Grand Theft Auto (GTA) series. GTA San Andreas was released in 2004 is roughly four times larger than the game world of GTA3, released in 2001. Figure 2 shows the game world maps of GTA 3 (2001), GTA Vice City (2002) and GTA San Andreas (2004) at the same scale in comparison.

The trend of growing game worlds and an increase in detail can also be observed in many other game titles such as Blizzard’s Diablo series and World of Warcraft, Sony’s Everquest series, Id Software’s Quake series and Epic’s Unreal Tournament series. The creation of game worlds today requires a team of specialists and has become an increasingly substantial component of game development (Crawford, 1984). This research project focuses further on the aspect of world size.
1.6 Hire More Artists

As game worlds increase in size and detail, more objects and textures need to be created. The increase in complexity and size of game worlds results in increased demands of time and resources, which has been confirmed by the “hundredfold” increase of game development budgets since 1980 (Crawford, 2003). As Stokes (2005) points out, the approach to “hire more artists” does not scale well with the rising demands of the next generation game worlds.

1.7 The Issue of Repetition

Stokes (2005) explains that the process to fill a large game world with many individual and visually convincing representations of objects is time consuming and very costly, as game artists manually create every object in a game world. He points out that due to the time consuming and costly nature of game world creation most games contain objects that look exactly alike. He states that “artists make one fire hydrant model, and use that same model on every curb; or they make a small number of tree models, and populate an entire forest with just a few trees”.

Figure 2: GTA size comparison
Reprinted with permission.
The repetition of game assets can also be observed in form of entire rooms with identical architecture that the player traverses in a game level. Stokes (2005), comments, that “the larger the game world, the more the practice of reusing a limited collection of models makes that environment look repetitive and unconvincing”. Duvall (2000) also identifies the repetition of objects as a problem in electronic games and states that it “widens the reality gap” if the objects are closely linked in time and space.

1.8 Current Approaches to Alleviate Repetition

Designers have acknowledged the existence of this issue and try to minimise repetition by methods described by Beram (2001) in which objects are combined with other objects to arrive at new appearances. Another approach to address this issue uses manual random scaling and rotation. While such approaches can alleviate the problem, players still notice the repetition of objects and entire rooms in electronic games (Cahil, 2003; Milgate, 2003; Vaxick, 2003).

1.9 The Impact of Repetition

It was stated that the repetition on objects in electronic games ‘looks unconvincing’ (Stokes, 2005) and ‘widens the reality gap’ (Duvall, 2000). While repetition is often used in design to group objects into logical patterns, “precise repetition runs the risk of boring the viewer” (Zelanski & Fisher, 1996, p. 38) as “the effect can be irritating, as if someone is repeatedly poking you into the arm” (Zelanski & Fisher, 2007b, p. 64). In order to understand why the repetition of objects in a game world may irritate users, I will unpack this issue by referring to the theoretical models of immersion, perceptual realism and the concept of presence with a focus on their relevance to 3D real-time virtual spaces such as game worlds.

1.9.1 Immersion

In psychology and the cognitive sciences, perception is the process of acquiring, interpreting, selecting, and organizing sensory information acquired with our eyes, ears, nose and skin or kinaesthetic sense. Heim states that immersion is achieved by “devices that isolate the senses sufficiently” from the real world and provide other information that “make the person feel transported to another place” (Heim, 1998). Electronic games usually only provide visual, audible and kinaesthetic stimulation. The visual stimulation is usually provided by the display device such as a monitor or a television set displaying rapidly changing images. The audible stimulation is delivered via headphones or speakers. Kinaesthetic stimulation in form of vibration and force feedback can be provided by joysticks and console game controllers. Heim (1998) classifies computer simulations, such as computer and arcade games, as perceptive immersions. He explains that in arcade games and similarly
in electronic games “immersion happens through a suspension of selfhood”. In this type of immersion, the player does not only perceive the graphic images but identifies him/herself as a character in the game (Heim, 1998, p. 102).

1.9.2 Presence

Lombard and Ditton (1997) define presence as “the perceptual illusion of non-mediation”. The concept of non-mediation refers to the human experience of interacting with a virtual environment through a medium with normal perceptual, cognitive and affective systems as if the interaction is not transmitted through the medium. In this situation the medium becomes transparent to the user and the interaction with the virtual environment through the medium is experienced as a direct interaction with the virtual environment itself. Similarly, the player described in Heim’s “suspension of selfhood” experiences presence while playing an arcade game in that the player experiences “a sense of being there while using a virtual environment” (Robertson & Oberlander 2002). In the instance of electronic games, the player experiences presence as an “open window”, through which the player can interact with the virtual space (Lombard & Ditton, 1997). The verb “open” in this case describes the audible experience of the player, stimulated through sound and music. “Window” on the other hand describes the visual experience confined within the boundaries of the medium, which in the case of an electronic game is the monitor or the television.

Lombard and Ditton (1997) identify six dimensions of presence: social richness, realism, transportation, immersion, social actor within medium, and medium as social actor, which are briefly outlined below.

*Presence as social richness* describes the social qualities of a medium used to interact with other users. Such social qualities can include descriptions such as warm, sensitive, personal or intimate.

*Presence as realism* “concerns the degree to which a medium can produce seemingly accurate representations of objects, events, and people” and is divided into two areas: perceptual realism and social realism. Perceptual realism refers to the extent to which the visual, acoustic and kinaesthetic accuracy of the mediated world in comparison to the real world. Social realism, on the other hand refers to the degree of which the mediated portrayal of events in the virtual world is believable to the user or comparable to the real world.

*Presence as transportation* describes the degree to which users feel they are taken to another place in space or time; or the degree to which media users feel another world is brought to the user, or the degree to which media users feel they are sharing another space with another user.
Presence as immersion refers to the degree to which a virtual environment masks the perceptual system of the user by input from the virtual world.

Presence as a social actor within a medium refers to the degree a user socially interacts with and overlooks the artificial nature of an artificial entity such as people or computer characters within a medium.

Presence as medium as social actor describes the degree, the user interacts with the medium itself as another social actor.

Lombard and Ditton (1997) further describe that presence either occurs or does not occur during media usage. The perception that users feel a “greater or lesser sense” of presence during a “media-use experience […] is attributable to there being a greater or lesser number of instants” of presence during media usage. In other words the illusion of presence can be interrupted several times during the use of a medium, in which case the medium becomes opaque to the user and presence is not experienced at a deep level.

1.9.3 The Influence of Repetition on Presence

Endless visual variety and infinite visual detail have been reported as natural phenomena (Heim, 1998). As discussed in section 1.9, artists and designers of game worlds often use a relatively small set of objects to fill a virtual space with content. The repetition of objects within a virtual space, however, can be noticed by the player. A space in the real world consists of objects that are not exact copies but appear in a visual variety and are infinitely detailed. In order to make the creation of large virtual spaces by 3D artists and designers possible, the amount visual data of a scene in a game world is reduced using methods of abstraction. The decreased amount of visual detail through the use of repetition, and therefore the decreased amount of data in form of visual variety decreases the quality of visual realism. Visual realism is one component of perceptual realism (Lombard & Ditton, 1997), next to other components such as acoustic and kinaesthetic realism.

The quality of visual input is also linked to immersion, as the images provide the sensory information for the human visual system that is interpreted together with the sensory information that is fed to the other human senses in order to achieve immersion.

Repetition of objects influences two dimensions of the theoretical model of presence as defined by Lombard and Ditton (1997): immersion and perceptual realism. It seems logical to assume that if large virtual spaces are constructed by using objects and textures repetitively, that this repetition can be noticed by the user and subsequently cause “irritation” as described by Zelanski and Fisher (2007b).
The user may also notice repetition in the form of a decreased amount of visual detail. Both factors have a negative effect on the dimensions of immersion and perceptual realism and subsequently interrupt the illusion of presence experienced by the user. The experience of presence in an electronic game, however, is important as it is viewed as a “central” aspect regarding the “use”, “usefulness and profitability” of electronic games and other new technologies (Lombard & Ditton, 1997).

1.10 Novel Approach to Increase the Illusion of Presence

This research project investigated a novel approach of generating large scale virtual spaces using procedural generation in real-time, addressing the negative role of repetition and subsequently contributing to increased presence in electronic games. Unlike current manual approaches to constructing large game worlds which require an increasing number of game artists or the repetitive reuse of objects, this novel approach is potentially faster than the manual creation approach and does not rely on object repetition to fill the game world with content. All objects within such a virtual space exist in a visual variety. As such it is believed that especially for large virtual spaces, this approach to procedural generation contributes to a greater illusion of presence experienced by the player.

1.11 Summary

This section provided the theoretical foundation of this research project. It highlighted the trend that game worlds are becoming increasingly larger and more detailed and consequently require more time and resources to construct them. This led to a discussion of common strategies such as hiring more artists and the repetitive reuse of objects and their limitations. A connection was made between repetition, immersion, perceptual realism and the concept of presence. It was postulated that the repetition of objects might negatively affect the illusion of presence experienced by a player. The chapter concluded with the introduction of the approach taken by this research project to alleviate the negative effect of repetition.

1.12 Exegesis Outline

The structure of the exegesis is as follows. Chapter 2 provides background and an overview about relevant research in the area of real-time procedural content creation. Chapter 3 describes the research methodology and defines the methods and evaluation criteria. Chapter 4 describes the prototypes that were developed as part of this research project and contains for each prototype a presentation of selected results and a discussion. Finally, chapter 5 summarises the outcomes of this research project and suggests directions for future research.
2 Background

2.1 Overview
This chapter documents the literature review that was carried out within the main timeframe of this research project from June 2000 till March 2006. It describes the common modelling and texturing practices used by the game industry to create three-dimensional virtual spaces that are of relevance to this research project. Furthermore it outlines the field of procedural generation and identifies research, relevant to the generation of virtual spaces. The third section focuses on real-time procedural generation of individual objects and entire virtual spaces.

2.2 The Practice of Game World Creation
The appearance of a virtual space is defined first and foremost by the visual style that determines the visual representation of objects within a game world. Game artists create the actual game world based on the defined visual style by using tools to model and texture objects that are then placed within a virtual space by using a level editor.

2.2.1 3D Modelling Techniques Used in Game Development
3D game artists use 3D modelling and animation packages such as Autodesk’s 3D Studio Max, Maya and Pixologic’s Z-Brush often in combination to create objects and entire virtual spaces. The 3D modelling and animation packages provide a range of tools which facilitate the modelling, texturing and animation of three-dimensional objects and virtual three-dimensional spaces. Despite the availability of tools for surface and subdivision modelling, three-dimensional objects for the use in a real-time 3D game levels are predominantly modelled with polygon manipulation tools (Thompson, Berbank-Green, & Cusworth, 2007).

Polygon manipulation tools enable designers to modify the geometry on various levels during the design process. Complex shapes are often broken down into simpler geometric primitives such as cubes, cylinders, spheres, etc. The geometric primitives are then combined to a more complex mesh in a subsequent step involving the movement and merging of vertices. Extrusion of faces and edges is often used to add more detail to an existing shape. Polygons can also be split or subdivided to add detail at particular areas of the mesh. Furthermore, the detail of a mesh can be manipulated at a vertex level where game artists can transform, add or subtract the vertices of a mesh to manipulate the final

Generally, the more polygons used to describe an object, the more detailed is its representation when rendered to the screen. Highly detailed objects, however, require longer rendering times and more memory to store the data, than objects of low detail. As already mentioned in section 1.3.7, the time to render the frames in real-time applications such as electronic games is limited. The objects in electronic games are therefore modelled with the trade off of visual detail and performance in mind and subsequently consist of as few polygons as necessary to describe their visual representation. The number of polygons used for game characters is often referred to as a benchmark to compare the amount of detail in an electronic game. At the time of the investigation character models consisted of approximately 2000 polygons (Franson, 2003, p. 68) while significantly fewer polygons are used in the modelling of the virtual space in which game-play takes place, as shown in Ahern (2002) and Clayton (2003). Characters for today’s electronic games often consist of 8000 polygons (Midgley, 2008).

### 2.2.2 Texturing Techniques Used in Game Development

Texturing is a process that applies an image to the surface of a polygon or polygon mesh (Akenine-Möller & Haines, 2002, p. 118). Instead of precisely modelling the geometry, the texture image usually depicts the necessary detail of the applied material and defines the object’s specific look and feel. Hundreds of textures are used to texture today’s electronic game worlds (Ahern, 2001, p. 18). To avoid extremely large image files, textures for large areas such as the environment’s terrain, large walls or the floor of a space are usually textured by relatively small images that are tiled on the surface. To disguise the otherwise obvious repetition from the tiling, the seams of the textures match with the pattern of the opposite side, effectively creating seamless textures.

Image manipulation software such as Adobe’s Photoshop is typically used to create and manipulate raster images for use as textures. Raster images consist of rectangular arrays of pixels, which create the impression of an image. In the context of electronic game level creation, image manipulation software is used to create textures and convert them to various image file formats which can be utilised by 3D modelling software and level editors.
2.2.3 Level Editors

Level editors bridge the gap between the content creation tools and the final game content. Once a mesh is modelled and textures are created, individual objects can be exported in various file formats which, in turn, can be imported by other programs such as level editors. A level editor such as Epic’s UnrealEd (Busby, Parrish, & Eenwyk, 2005, p. 48) is an interactive game level design tool. Such tools usually facilitate the visual assembly of game levels, which involves the manual placement of objects, application of textures, placement of lights, integration of sounds and music as well as the deployment of scripts. Some level editors such as Epic’s UnrealEd also enable designers to test the game level in real-time from within the editor (Busby et al., 2005).

2.2.4 Monohedral Tilings

Monohedral tiling, as described in Hill (Hill, 2001, pp. 486-488) is a popular method used in board games (Kobbert, 1986; Kramer, 1999; Teuber, 1995) to break down the game board into modular tiles which can be reassembled in various combinations to present the player with new layouts and challenges. The tiles usually consist of regular shapes, such as triangles, squares or hexagons.

Early two-dimensional computer games such as Ozark Softscape’s M.U.L.E. in 1983 or Ludodelire’s Full Metal Planet in 1990 also used a monohedral tiling approach to randomise the layout of the two-dimensional digital game board.

The monohedral tiling approach has also been used in games such as Bullfrog’s Populus in 1989 and Blizzard’s Diablo in 1996. Blizzard’s Diablo used the monohedral tiling approach in combination with three-dimensional isometric perspective. The images used on the tiles can be rearranged in various ways. Blizzard’s Diablo generates a new labyrinth using a random dungeon generator for every level based on a limited set of such tiles.

Monohedral tiling approaches have been used successfully in many games to generate game levels and board game layouts exhibiting a variety of tile distribution. However, the approach draws on a limited number of pre-fabricated tiles, which are reused in the generation of various layouts. The content of the pre-fabricated tiles does therefore not change and subsequently does not solve the problem of object repetition in an electronic game.
2.3 Procedural Generation

Procedural generation focuses on creating media content based on a sequence of instructions, i.e. a procedure. Such procedures are typically implemented in software, that specifies the “characteristic of a computer generated model or effect”. (Ebert, Musgrave, Peachey, Perlin, & Worley, 1998, pp. 1-4).

There are several well understood approaches in computer science and mathematics which deal with procedural generation. Procedural generation often involves the use of techniques such as random number generators, iterated function systems and noise.

2.3.1 Iterated Function Systems

Lecky-Thompson (2001, pp. 203-204) defines an iterated function system (IFS) as an algorithm that generates an output which is then passed back as the input to the next iteration. This concept is often used for pseudo random number generators, fractals and L-systems. Lecky-Thompson, however points out that the generation of IFS-based shapes can be very calculation intensive and may be problematic to implement for a real-time application.

2.3.2 Random Number Generators

It has been suggested that random number generators that are to be used for the generation of 3D real-time procedural virtual spaces need to be “not random”, “return a sequence of numbers with a good random distribution” and “should be fast” (Macri & Pallister, 2000).

Random number generators can be classified in two categories: true random number generators and pseudo random number generators. True random number generators produce a unique stream of random numbers which cannot be reproduced. In the context of procedural generation of virtual spaces, true random numbers are usually undesirable, because it is not possible to reproduce the identical sequence of numbers for subsequently generated results (Smith, 1984) (Macri & Pallister, 2000).

Tom Duff suggests that pseudo random number generators generally use a small set of rules to generate strings of sufficient length and complexity to simulate true randomness (as cited in Smith, 1984). The generation is typically based on an initial seed value. When initialised with the same seed, the identical sequence of numbers is reproduced.
There are many implementations of random number generators, with varying statistical properties. An explanation of these properties can be found in Lecky-Thompson (2001, p. 14). Generally, however it can be said that pseudo random number generators with good statistical properties are computationally more intense.

This research project used initially pseudo random number generators to add variation to the generation of textures and meshes. The deterministic nature of pseudo random number generators was essential to maintain the temporal consistency frame to frame as well as the consistency of the virtual space to allow users to freely navigate in the virtual world and to return to a previously visited place.

The generation of virtual spaces in particular required the generation of many short sequences of less than 50 random numbers per generated object. As such, good statistical properties were of lesser importance than the generation speed of the pseudo random numbers.

2.3.3 Noise

Peachey (Ebert et al., 1998) defines noise as an irregular primitive function that can be used to break up the monotony of patterns. Peachey (Ebert et al., 1998, p. 64) distinguishes further between white and pink noise. White noise is described as a series of uniformly distributed and uncorrelated random numbers. Such a sequence of random numbers can be produced by a true random number generator. However, as discussed in the previous section, the generation of true random numbers is not desirable.

Pink noise on the other hand is described to exhibit the following six properties (Ebert et al., 1998, pp. 64-66):

- Pink noise is repeatable, that is, a noise function can reproduce a previously created noise sequence given the same input values.

- Pink noise has a known range. Such noise functions only produce pseudo random output values between a minimum and maximum value such as -1.0 to 1.0.

- Pink noise is band limited. This means that all pseudo random outputs that the noise function produces are contained within a finite frequency range, also called band.

- Pink noise does not exhibit obvious periodicities or regular patterns. Pseudo random functions are ultimately always periodic. Peachey (Ebert et al., 1998, p. 65) therefore suggests using noise functions with longer periodicities so that the viewer does not become aware of repetitive artefacts or patterns.
Pink noise is stationary, meaning that the statistical properties of the noise function do not change over time or position.

Pink noise is isotropic. Isotropic noise is independent of direction.

Peachey (Ebert et al., 1998, p. 66) also describes lattice noises as the most popular implementations of noise. In the context of procedural generation of textures, Perlin noise (Perlin, 1985) is probably the most commonly known lattice noise type. Ken Perlin’s noise function returns a pseudo random floating point value between -1.0 and 1.0 given single or multiple input values. A one-dimensional implementation of Perlin noise that is based on a pseudo random number generator is explained in Elias (2000). The same principle has been applied to generate material textures such as wood and marble (Ebert et al., 1998), cloud like textures (Pallister, 2000) and geometries for landscapes (Macri & Pallister, 2000). Perlin noise has also been applied as three-dimensional implementation to generate irregular surfaces as demonstrated by Perlin’s ice cube (Perlin, 2002). This research project uses Perlin noise to generate a terrain and to regulate the height of building objects to shape the overall silhouette of a virtual city space in the Neverland Framework.

2.4 Formalised Approaches to Procedural Generation

There are several formalised approaches to procedural generation. In this context this research project has reviewed Fractals and L-Systems. These formalised approaches depend on the principle of "database amplification". The term has first been coined to describe the property of formal languages, which allows the generation of complex images from small databases (Smith, 1984).

2.4.1 Fractals

A fractal is defined as a shape that exhibits “self similarity” at different scales (Mandelbrot, 1977). A fractal shape generally consists of segments, with each segment consisting of a reduced-size copy of the entire initial shape with all its segments. The creation of a fractal shape is usually based on a set of rules and implemented in some form of iterated function system as explained in section 2.3.1.

Many natural objects in the real world show properties of “self similarity” and repeating patterns that can be seen at every scale. Such natural objects include mountain ranges, cloud formations, and on a smaller scale, the frond of a fern (Mandelbrot, 1982).

While the prospect of generating “self similar” forms was of interest to this research project to generate textures, fractals were not used in this research project because of their generally computation intensive nature.
2.4.2 L-Systems

Lindenmayer (Prusinkiewicz et al., 1990) developed a formal language, called L-systems, to model the cellular development of plants. L-systems consist essentially of a string rewriting mechanism. This mechanism replaces symbols in a given word following a simple set of rules and results in a new word that is generally more complex. The new word can then be used as input to the same string-rewriting mechanism, again using the same production rules and the process is repeated. The final word or string can be parsed by software such as the turtle graphics system to generate a graphical output from the symbols in the word.

Stochastic L-systems add randomness to the production process by introducing several production rules with the same matching conditions. A probability distribution as a parameter of this set of rules is used to determine which rule is applied in the production process. Stochastic L-systems have been used successfully to generate plant and tree models in a visual variety in real-time (nVidia, 1999) (Oppenheimer, 1986).

The stochastic L-systems approach was of interest to this research project because it provided a formal grammar to generate three-dimensional meshes in a variety of form. However, they were not used for two reasons. First, the complexity of the man-made objects in a game level such as crates and buildings was generally quite low at the beginning of this research project. Subsequently, the use of stochastic L-systems that have a lot to offer in terms of generating a detailed model seemed overkill, given the rather unintuitive process of developing a suitable grammar. Second, was the difficulty of intuitively applying the grammar based approach of generating an object to the modelling and texturing approach used by game artists.

2.5 Related Work

Procedural generation is often mentioned in relation to the generation of textures and meshes showing natural phenomena for time-based media such as film, video and TV (Ebert et al., 1998). Procedural generation has, however, also been used successfully in the areas of generative art, electronic games and more recently in middleware.

2.5.1 Generative Art

Generative art is a system oriented approach to art, where practitioners generally use procedures to generate, compose, or construct artwork. Such generated art spaces are generally not imitating the real-world but often challenge the traditional notions of the real in their virtual spaces, which becomes particular evident in the generative artworks of McCormack (2003, p. 5), Dorin (2004), Davies (2004)
and Soddu (2003). While there are games such as Sega’s Rez, Namco’s Katamari Damacy or Introversion Software’s Darwinia that adopt a similar view, the majority of games tends to depict the virtual space with visual reference to the real-world. Similarly, this research project was aimed at the generation of virtual spaces with visual reference to the real-world.

### 2.5.2 Procedural Generation for Time-based Media

Procedural modelling, texturing and animation techniques have been used in many movie productions to generate visual effects, textures and three-dimensional objects.

Perlin (2001) states that one of the main techniques to create the effect of subtle irregularities on computer generated objects is noise-based procedural shading. He notes that noise-based procedural shading has been used in particular in Disney’s The Lion King to create mists and layered atmosphere, and in Warner Brothers’ The Perfect Storm.

Procedural modelling, texturing and animation techniques are often used to generate natural objects such as terrains, trees and foliage. For example, a procedural tree and foliage modelling was used in New Line Cinema’s The Lord of the Rings: The Two Towers, to decorate the computer generated characters with a wide variety of foliage (Aitken & Preston, 2003).

Procedural animation has also been applied to animate virtual crowds to achieve a realistic look used in recent movies such as Lucasfilm’s Star Wars, Universal’s Jurassic Park, New Line Cinema’s The Lord of the Rings and DreamWork’s Shrek 2 (Thalmann et al., 2004).

The procedural generation of cities consisting of a street grid and buildings has been proposed for a system called CityEngine (Parish & Mueller, 2001), which generates a city model consisting of a street grid, building objects and textures based on a small set of statistical and geographical input data that can be rendered with 3D modelling and animation packages. Similarly, the procedural generation of highly complex building facades is the focus of other research such as Wonka et al. (2003), Finkenzeller (2008) and Mueller et al. (2006).

This small overview demonstrates the use of procedural generation approaches to generate objects, textures, natural phenomena and animation with the aim to imitate the randomness and variety that exists in the real world. However, the time to render every frame for time-based media such as film and television is not limited by the target frame rate of the medium. Pixar (2001) states that the time to render a single frame for the 3D animated movie Monsters Inc. took “about 6 hours and that some frames have taken as many as 90 hours”. In real-time applications such as electronic games, however,
every frame needs to be rendered in the range of milliseconds. As a result the scope of procedural generation in electronic games is limited to algorithms and procedures that are fast to calculate and produces content that can be rendered in real-time.

2.5.3 Procedural Generation Used in Games

Procedural generation has been used in many two-dimensional jump and run and platform games as described in Lecky Thompson (2002). Early 3D games such as Braben and Bell’s Elite in 1984 also used procedural generation to generate the vast amount of data that comprised the solar systems in the game. Another recent game demo published in 2004, .theprodukkt's beta version of the first person shooter .kkrieger, features an impressive amount of procedurally generated content such as models and textures, which are generated before the game begins.

The above mentioned games demonstrate that a virtual space can be generated based on the repetition of a small set of generated objects and textures, that are then reused repetitively in the virtual space. None of the above mentioned games, however, generates objects and textures in a visual variety.

2.5.4 Real-time Procedural Generation

Advances in processing speed and graphics hardware are making it possible to use procedural modelling and texturing in a dynamic, on-the-fly manner in real-time on commodity computers. This section provides an overview of relevant research and commercial software that use real-time procedural generation to create objects and virtual spaces.

nVidia’s Grove software (nVidia, 1999) shows a procedurally generated tree in a self-contained environment. The tree’s geometry is entirely generated in real-time. The generation parameters can be manipulated via control sliders and change the tree’s appearance and complexity. According to nVidia’s documentation, the software facilitates the generation of trees with a complexity of up to one million triangles.

Lecky-Thompson (2001) explains the principles of seeded random number generation in relation to procedural content generation of infinite two-dimensional electronic game worlds. He provides several programs that demonstrate principal approaches to generate a pseudo infinite variety of terrains, faces, ferns, names and a game space. The prototypes mainly use ASCII and two-dimensional vector graphics, but these principles can also be applied to three-dimensional real-time generation of virtual spaces.
Pallister (2000) describes the generation of a cloud pattern based on Perlin noise. The implementation uses the graphics card and CPU for smoothing, compositing and blending operations to achieve interactive frame rates. In the demo, the sky texture is mapped onto a curved object which simulates the curvature of a sky. Similar approaches to generate cloud and vapour textures using noise can be found in Elias (2000) and Ebert (1998).

Intel’s procedural world software (Macri & Pallister, 2000) shows procedurally generated terrain with trees and a two-dimensional cloud layer based on Perlin noise (Perlin, 1985). The terrain can be freely explored on the horizontal plane in real-time and expands around the user’s point of view. A similar approach for landscapes has been outlined by Maurice Danaher (2002). Figure 3(a) shows a screen shot of Intel’s procedural world demo.

Bionatics (2004) offers several programs based on a library for generating trees, plants and gardens procedurally. NatFx is designed for level designers and provides plug-ins for Autodesk’s 3D Studio Max and Alias’ Maya, which generate plants and trees with a customisable level of detail. RealNat generates trees and plants for 3D real-time environments such as flight simulators and virtual reality applications.

Interactive Data Visualization (2004) developed an Advanced Programming Interface (API) called SpeedTreeRT, which supports generation of textured tree geometries for interactive, real-time applications. SpeedTreeRT can render tree models in user configurable levels of detail, and uses smooth transitions at runtime to optimise performance. To present the variety of tree objects that can be generated, IDV’s Huge Forest demo distributes the trees on a terrain. A screen shot of the Huge Forest demo is given in figure 3(b).
Virtual Gardening (JFP, 2004) is a self-contained game, which features the interactive design of a virtual garden environment. The program simulates the growth of plants and trees over time, influenced by changing seasons. Figure 4(a) shows a screen shot.

Binary Worlds (2004) developed a commercial real-time procedural world generation middleware called Descensor engine. Binary Worlds’ Descensor engine shows several similarities to the Neverland Framework developed by this research project. Descensor and Neverland are both capable of procedurally generating a terrain and cities with streets and buildings, based on parameters that can be customised via an editor. Descensor and Neverland also, both generate only the geometry that is visible to the virtual camera and stores already generated objects in memory. Figure 4(b) shows a screen shot of the Descensor Engine Demo.

![JFP's Virtual Gardening](image1.png) ![Binary World's Descensor Engine](image2.png)

**Figure 4: Commercial applications**

While there are similarities between Descensor and Neverland, there are also some differences. Descensor improves performance and stability of frame rates based on impostor and occlusion culling techniques, whereas Neverland uses a LRU caching approach, described in sections 4.9.3 – 4.9.8. Also the generation of the building geometries is different, as Descensor uses encoded architectural construction procedures which describe the building features, room distribution and other details, while the Neverland Framework uses an approach to generate floor plans from merged shape primitives, as described in section 4.8.2. Street grids in Descensor are based on a connectivity graph, whereas Neverland uses a node based system, that can be evaluated as needed.

According to Pablo Puente, Binary Worlds first offered the product in late 2003, approximately 6 months after this research had published an approach to generate buildings and a framework based approach to generate virtual spaces (Greuter, Parker, Stewart, & Leach, 2003a, 2003b). In contrast to this research project, Binary Worlds has not published any details regarding the implementation details.
of Descensor. Furthermore Binary Worlds has not specified the generation of object or spaces in a visual variety on their website as one of their aims.

Spore by Maxis is still in development at the time of writing this exegesis. It was first introduced at the Game Developers Conference (GDC) 2005 in San Francisco. The game allows players to guide the evolution of a new species from its early stages in a primordial soup into a civilization with cities and spaceships. The game uses procedural generation to fill the game world around the player with content, while the player can freely and seamlessly zoom in and out from a cellular level on the surface on a planet to the level of a procedural galaxy consisting of solar systems with several hundred thousand planetary bodies that are all procedurally generated, and which can all be visited and colonised. The game features procedurally generated eco systems complete with procedurally generated and procedurally animated animal life, procedural texturing and procedural building generation. An integrated building editor allows players to generate their own content procedurally.

2.6 Summary
This chapter described the manual approach of modelling, texturing and level creation that is commonly used by the games industry. This followed an overview of procedural generation techniques in order to outline the differences between both manual and procedural content creation approaches. Within the context of procedural modelling and texturing, relevant research was identified in the area of time-based media, electronic games and real-time applications.
3 Methodology

This chapter describes the methodology of this research project. As such it clarifies the influences that motivated the focus on real-time procedural generation approaches to generate a visual variety of objects and virtual spaces. Next follow the two research questions that were investigated by this inquiry. The section on research through design describes how the investigation was conducted in order to answer the research questions. This includes a description on how data was collected, analysed, evaluated and validated.

3.1 Motivation

This research dealt with the issue of visual variety of objects within large real-time virtual spaces. As discussed in chapter one, large real-time virtual spaces are often constructed by frequently reusing manually created objects and textures. In section 1.9.3 the point was made that the precise repetition of objects could potentially cause irritation, which could negatively influence visual realism, immersion and subsequently the illusion of presence. This research project postulates that if virtual spaces consist of visually varied objects – as opposed to objects which are exact copies of another object – the irritating effect caused by repetition can be alleviated, which has a direct positive influence on visual realism, immersion and subsequently the illusion of presence experienced by the player.

As discussed in chapter one, it is possible to manually create objects in a visual variety. The manual creation approach, however, becomes increasingly more time-consuming as game objects become more detailed, as explained in section 1.5.1. Moreover, game worlds also increased in size, as explained in section 1.5.2. As a result an increasingly larger number of objects need to be created to fill the larger virtual spaces of such game worlds with content. This trend has subsequently lead to larger team sizes, higher production costs (Crawford, 2003), which does not scale well with the rising demands in terms of game world size and object detail of next generation game worlds (Stokes, 2005).

This research project was motivated to investigate procedural generation approaches to generate three-dimensional objects in a visual variety and to distribute such objects to generate a virtual space in real-time to alleviate the increasing demand of objects for next generation game worlds, while not relying on object repetition. As such, the formulation of the motivation contained four aspects that are described in more detail in the following sections:
3.1.1 Procedural Generation

Procedural generation was selected for this research project for two reasons:

1. Procedures can be used to encode the characteristics of three-dimensional objects

2. Procedures can be controlled via parameters

This research project was motivated to use a procedural generation approach to characterise and encode in software certain objects types that can be used to fill a virtual space. It was envisaged that certain object characteristics could be parameterised and that random numbers could be used to generate such objects in a visual variety.

3.1.2 Objects and Virtual Spaces

As the problem was located within the real-time 3D games culture, it was decided from the outset that the generated objects and virtual spaces need to be similar in visual style and complexity to current electronic games that were available at the time of investigation. As such, this research project endeavoured to generate objects that were similar in detail and polygon count to objects in games such as Microsoft’s Flight Simulator 2004 and Rockstar’s GTA 3, which were both considered cutting edge electronic games at the time of investigation. As a result, the investigation focussed on approaches to generate polygon geometries that consist of as few polygons as necessary and small image based textures to achieve real-time frame rates.

A significant amount of research on procedural techniques to generate natural phenomena was already available at the start of this research project. However, only a limited amount of research on man-made objects or virtual spaces that consist of man-made objects could be located in the literature.

Moreover, the available research on natural and man-made phenomena, however, was predominantly directed to the use in time-based media, such as film and video. No research on real-time procedural generation of man-made objects could be located in the literature. This research project focussed on the development of procedures to generate objects such as crates, buildings, terrains and street grids and spaces such as a warehouse and a virtual city because of this gap in the body of knowledge and the cultural significance of man-made objects in electronic games.
3.1.3 Visual Variety
Heim (Heim, 1998, p. 156) uses the term “infinity” to describe the endless visual variety of colours and shapes that can be observed in nature. This research project aimed to increase the illusion of presence by increasing the visual variety to objects and virtual spaces to alleviate the problem of increased workload for artists as described in section 1.6 or object repetition as described in section 1.7.

It was envisaged that the generated objects show a visual variation in comparison to other objects of the same type. It was also envisaged that the visual variation can be identified, described and evaluated based on design elements that are defined in a visual grammar (Leborg, 2006).

3.1.4 Real-time 3D
Computer graphics is an active applied area of computer science research and concerned with digitally synthesizing and manipulating visual content. Three-dimensional computer graphics is a sub-field of computer graphics concerned with the calculations and rendering of a three-dimensional representation of geometric data into two-dimensional images that can be displayed on a screen. Real-time rendering is one specific application of 3D computer graphics used in systems that need to interactively generate visuals based on the viewer’s actions. The most common application of real-time rendered 3D computer graphics is currently electronic games.

To stay within the culture of real-time 3D electronic games, this research project required the investigation of methods to dynamically generate virtual spaces at frame rates between 15 fps and 72 fps. It was envisaged that such frame rates can only be achieved if the project was implemented with an industry standard programming language such as C++ and a graphics library such as OpenGL.

3.2 Research Questions
In order to carry out this project, two main research questions directed this research project. The first question concerned the generation of objects and second question concerned the real-time generation of a virtual space consisting of generated objects.

1. How can three-dimensional objects be generated in a visual variety using computer software?

2. How can three-dimensional virtual spaces that consist of objects of a visual variety be generated in real-time using computer software?
3.3 **Research Through Design**

This research project used research through design as central research method to advance knowledge in the area of game world design, within the broader context of electronic game development. Research through design is a component of what is often labelled as design research (Frayling, 1993). Simon (1969, pp. 55-56) states that design devises courses of action aimed at changing existing situations into preferred ones. Research through design is described as a systematic inquiry, which generates communicable knowledge through the practical act of designing (Archer, 1995; Downton, 2003). Design research follows the constructivist learning theory, which regards learning as an active process of creating, rather than acquiring, knowledge. As such a learner constructs meaning through the experience of interacting with an environment and the subsequent reflection as a reflective conversation with the design and as a reflection on the design process (Schön, 1995). The design process “follows an iterative decision sequence of problem, analysis, synthesis and evaluation” (Lawson, 1997).

### 3.3.1 Iterative Design

There are many similarities between research through design and Archer’s research through practice. Archer describes research through practice as a form of action research (Archer, 1995). Action research is characterised by a cyclic process which includes planning, acting, observing and reflecting to progress knowledge (Susman, 1983). It has been stated that the cyclical and emergent process of action research is similar to the problem, analysis, synthesis, and evaluation cycle of design through research (Stapleton, 2005).

Similarly, this research project used an iterative approach based on the design process, which involves the creation of prototypes that were tested, analysed and redesigned to progress knowledge. Subsequent iterations were built on the insights gained from previous iterations. This approach to iterative design has also been successfully used to develop the software of several other game projects that are described in Burdick (Laurel, 2003, p. 82), Zimmerman (Laurel, 2003, p. 176) and Westecott (Laurel, 2003, p. 130).

### 3.3.2 Software Prototypes

The creation of prototypes is in itself a potential generator of knowledge an essential component of research through design. The insights gained through the development of the prototype and the results of the subsequent evaluation are fed back through the exegesis.
The software prototypes that were developed as part of this research project connect between fields of knowledge and served as platforms, through which ideas could be explored, hypotheses could be tested and new questions could be developed.

This research project was motivated by the overall goal to generate objects in a visual variety and to distribute them in a virtual space as expressed in the two research questions. Each prototype dealt with issues of visualisation but also technical issues related to its implementation in software and the technical requirement to achieve real-time performance.

The actual number of prototypes was not defined at the start. Each prototype shows the developmental stages of this research project and the exegesis reports the goals, problems and thinking processes of each iterative development and resulted in the order illustrated in figure 5.

### 3.4 Data

This research project followed an iterative design research approach based on the development and evaluation of a progression of prototypes. The inquiry was driven by three goals: (i) to generate objects in a visual variety, (ii) to generate virtual spaces that consist of objects in a visual variety; and a technical requirement: (iii) realise the generation in real time. Consequently quantitative and qualitative data was collected and evaluated in order to measure the quality of the inquiry.

#### 3.4.1 Qualitative Data

This research project investigated how three-dimensional polygon objects can be generated in a visual variety and subsequently how virtual spaces that consist of such objects can be generated.

While the performance of the program code can be measured and clear assessments can be made if the developed procedures are suitable for real-time generation, it is difficult to objectively measure the experience of visual variety. Qualitative data was collected that could be analysed in order to evaluate ideas and develop new hypotheses that were directed towards the development of subsequent prototypes.

#### 3.4.2 Qualitative Data Collection

In line with approaches to qualitative data collection that are described in Creswell (2007, p. 130), I have collected qualitative data in form of notes and visual materials at various stages throughout this research project. The notes and visual materials documented observations conducted by myself as participant using the prototypes. I have also gathered notes on observations conducted by my
supervisors using the prototype. The notes were accompanied by visual materials such as screen captures and parameter data that were used to generate objects and virtual spaces.

It needs to be noted that the prototypes that were created as part of this inquiry generated an enormous amount of visual material. It was not practical and also not necessary to capture all the visual data that was generated by the prototypes in this exegesis as the reader can vicariously experience the incredible amount of generated data through the prototypes available as part of this research project. In order to illustrate the discussion of the various prototypes, a representative sample of image material was collected as screen captures and included into the exegesis.

### 3.4.3 Qualitative Data Analysis

The qualitative data analysis documented and evaluated the generated objects and virtual spaces for their visual variety. Each prototype was developed to improve certain aspects regarding the generation of a visual variety of objects and subsequently virtual spaces. The generated results of each prototype were evaluated based on observations by myself and my supervisors for design elements that showed a visual variety or a visual repetition. The design elements that showed a visual variety or a visual repetition were identified based on a visual grammar consisting of elements, patterns, processes and relations between image elements (Leborg, 2006). As a result of using such a visual grammar, I was able to examine, describe and interpret the generated visual results and subsequently direct the development of new questions, ideas and hypotheses.

### 3.4.4 Quantitative Data

While the generation of objects and virtual spaces in a variety of form and detail was central to this investigation, real-time performance was a requirement for the development of prototypes by this research project. It was therefore necessary to conduct performance measurements of critical components. This was achieved via measurements of code execution times and frame rates of prototypes.

### 3.4.5 Development and Test Platform

The test platform consisted of the following hardware: ASUS P4B533 Motherboard, 2 GHz Intel Pentium 4, 512 MB RAM, GeForce 4 TI 4600 GPU with 128MB memory. The development and test platform is dated in comparison to the hardware available today, however the above mentioned hardware configuration was regarded as a high end computer system during the prototype development phase of this research.
3.4.6 Quantitative Data Collection

In order to test the performance of prototypes, frame rates and code execution times were measured at various stages during the research project. In order to conduct frame rate measurements, I inserted a counter into the program code. The counter provided a reliable measurement of the number of frames rendered for every second of execution time. This allowed the measurement of frame rates with an increasing number of objects in a virtual space.

In addition to the frame rate, it was also necessary to measure the execution speed of various aspects of the LRU caching data structure. The code execution speed was generally measured by recording the system start time and the system end time of a particular operation. In order to record accurate measurements, all graphics related commands such as the rendering the visuals on screen were commented out. The results for the measurements described in sections 4.9.12 – 4.9.14 were recorded in a text file and subsequently plotted as a graph to analyse the results.

All measurements can be reproduced via the prototypes and the source code available on the CD that accompanies this exegesis. Please note that the performance results may vary because of differences in hardware, operating systems, installed software and drivers.

3.4.7 Quantitative Data Analysis

Performance is a key criterion for real-time applications. In section 1.3.7 it was recommended to render the images at frame rates of at least 15 fps to create the illusion of a dynamic process rather than a sequence of individual images (Akenine-Möller & Haines, 2002). Subsequently all calculations needed to be performed at speeds that are faster than 66.6 ms on the test platform.

The frame rate for virtual spaces generally varied inversely with the number and complexity of the generated objects as would be expected. The more objects generated or the higher the complexity of the generated objects, the lower the frame rate. Measurements were taken to ascertain the number of objects that could be displayed at frame rates greater than 15 fps. The sufficiency of the number of displayed objects could not be measured and required a qualitative data analysis.

3.5 Validation

Westecott describes the production and communication of new knowledge via prototypes, academic papers and documented methodologies as common practice in design research (Laurel, 2003, p. 129).

This research project used the institutional method of peer review through publication and subject expert feedback to validate its findings (Creswell, 2007, p. 208). As such this research project has
published four papers that presented several of the key findings at international conferences in order to find confirmation from colleagues and industry experts. The findings have also been presented at various stages to Melbourne game companies to obtain expert feedback. One of the prototypes was also commissioned as interactive background artwork for the 2005 Adelaide Fringe Festival theatre play “The Lost Babylon”.

3.6 Summary

This chapter explained that this research project was motivated to investigate how procedural generation can be used to generate three-dimensional objects in a visual variety and to distribute such objects to generate a virtual space in real-time. As such the research project aimed to alleviate the increasing demand of objects for next generation game worlds, while not relying on object repetition.

As such the inquiry was driven by the goals to generate objects in a visual variety and to generate virtual spaces that consisted of objects in a visual variety. The inquiry was also motivated to realise the generation in real time. To achieve this aim, the research project followed an iterative design research approach based on the development and subsequent evaluation of several prototypes. The evaluation of each prototype was based on qualitative data in form of notes and screen captures and quantitative data in form of performance measurements. The research validation followed the institutional method of peer review through publication and subject expert feedback.
4 Prototypes

This chapter contains the description, analysis and a discussion of the results for each of the prototypes that were developed to answer the research questions. Within the timeframe of five years, this research project has produced ten prototypes. The exact number and form of prototypes as well as the sequence in which they were developed was not planned from the beginning of this research project and is the result of the iterative design process that was used to progress knowledge. As such, the problems of generating three-dimensional objects with a visual variety and the generation of virtual spaces were broken down into smaller sub-problems that fall into the categories: texture, geometry and distribution. Each prototype provided an answer to a specific set of problems that contributed to the larger research questions that were investigated by this research project. The last prototype combined the findings of this research project within the structure of a software framework.

4.1 Overview of Prototypes

Prototype number one was a proof of concept and dealt with the problem of generating a visual variety of crate objects through variation of volume, dimension, form, size, frequency and space. The generation was based on a parameterised template encoded in MaxScript. The prototype allowed the generation of a visual variety of crate geometries. The template approach, however, limited the generated visual variety to the formal structure that was encoded to resemble the characteristics of only one type of crate object.

The focus of the second prototype, called “Lattice Examiner” was the generation of a visual variety of texture and colour that could not be generated in the previous prototype. The visual variety of texture was generated by sampling random rectangular texture snippets from a larger material texture. The visual variety in colour was generated by randomising the hue, tone and saturation of the texture.

The third prototype, called “Crate Examiner” combined the findings of the first two prototypes and generated crate objects in a visual variety of volume, dimension, form, size, colour, texture, frequency and space in real-time. The generation, however, was limited to only one crate object at a time.

To examine the generated visual variety of crate objects and examine the performance within a virtual space two more prototypes were developed that focussed on problems related to the distribution of crate objects within the logical unit of a pile structure and the distribution of such pile structures within a virtual space.
The fourth prototype called “Pile Examiner” subsequently generated crate objects arranged in a pile structure. The prototype generated pile structures in a visual variety of volume, dimension, size, colour, texture and space.

The “Warehouse” was the fifth prototype that was produced by this research project and combined the findings of the “Crate Examiner” and the “Pile Examiner”. It was the first prototype that generated a virtual space that consisted of generated objects in a visual variety in response to the second research question. The approach to generate a virtual space, however, was too limited to fully answer the second research question, but provided the foundation to investigate more complex object types such as buildings and other virtual spaces such as cities.

The sixth prototype was called “Building Examiner 1” and focused on the generation of building objects in a visual variety of volume, dimension, size and texture in real-time. The prototype also used a template approach to generate a visual variety of building geometries. While the template approach worked well for the crate objects, the visual variety in form was too limited for the building objects generated by the Building Examiner 1.

The “Building Examiner 2” was the seventh prototype and investigated an approach to generate a visual variety of form in response to the limitation experienced in the Building Examiner 1 prototype. This prototype generated emergent forms of ground floor shapes that were subsequently extruded to building facade geometries. The texturing of the building facade geometries was based on the texture snippet approach developed for the Lattice Examiner. The Building Examiner 2 generated building objects in a visual variety of volume, dimension, form, size, and texture in real-time.

The eighth prototype was called “Undiscovered City”. It generated a virtual city space by combining the dynamically generation approach implemented in the Warehouse, with the approach to generate building objects in the Building Examiner 2 prototype. The prototype generated a virtual space that consisted of a visual variety of building objects. The formal structure of the street grid which subdivided the virtual space into square lot shapes, however, reintroduced an element of repetition to the virtual space. Performance problems with the new virtual space required the development of a procedure to limit the generation of objects and the development of a caching data structure to avoid the calculation intensive re-generation of objects for every frame.

The ninth prototype was called “Street Grid Examiner”. In response to the limitation experienced in the Undiscovered City prototype, this prototype generated a street grid that was characterised by an informal structure to increase the visual variety of the distribution of objects in subsequent prototypes.
The generation of a virtual space was dependant on several approaches to generate the geometry and textures of objects. The use of a caching data structure and view frustum filling approach allowed the display of such generated virtual spaces in real-time. The last prototype, called “Neverland Framework” combined several of the developed approaches to procedural generation to increase the usefulness and applicability of this research project.

The “Neverland Framework” managed the dynamic procedural generation and performance of procedurally generated virtual spaces. The framework was designed with a flexible middleware structure that consisted of an interface, which facilitated the communication with other software. The structure of the framework also allowed the integration of object generators other than crates, buildings and street grids to facilitate the generation of other environments.

To answer the research questions, this research project has produced ten prototypes. The overall research path is illustrated in figure 5. A more detailed description of each prototype as well as selected generated visuals and a discussion of the results are integrated into the following sections. Information about the usage of each prototype and their parameters is provided in the Appendix.
Prototype 1: MaxScript
Templated Crate Geometries

Prototype 2: Lattice Examiner
Randomised Texture Coordinates
Real-time Generation

Prototype 3: Crate Examiner
Templated Crate Objects
Randomised Texture Coordinates
Real-time Generation

Prototype 4: Pile Examiner
Templated Crate Objects
Randomised Texture Coordinates
Real-time Generation
Pile Distribution

Prototype 5: Warehouse
Templated Crate Objects
Randomised Texture Coordinates
Real-time Generation
Pile Distribution
Formal Structure Distribution
View Frustum Filling Based on Hashing

Prototype 6: Building Examiner 1
Templated Building Geometries
Randomised Texture Coordinates
Real-time Generation

Prototype 7: Building Examiner 2
Emergent Building Geometries
Randomised Texture Coordinates
Real-time Generation

Prototype 8: Undiscovered City
Emergent Building Geometries
Randomised Texture Coordinates
Real-time Generation
Formal Structure Distribution
View Frustum Filling Based on Hashing
LRU Caching

Prototype 9: Street Grid Examiner
Path Displacement
Street Geometry Generation

Prototype 10: Neverland Framework
Emergent Building Geometries
Randomised Texture Coordinates
Real-time Generation
Path Displacement
Improved View Frustum Filling Based on Hashing
LRU Caching
Street Geometry Generation
Framework Structure with Examiner and Viewer

Figure 5: Research pathway map
4.2 Prototype 1: MaxScript

The first prototype was a preliminary experiment to explore how a procedure could be used to generate a visual variety of objects. As test subject I decided to generate a visual variety of crate geometries. The preliminary experiment was realised in MaxScript and visualised with Autodesk’s 3D Studio Max 4.

4.2.1 Crates in Electronic Games

Crate objects are a “standard” object used in many electronic games (Franson, 2003, p. 66) as containers for items or just as “extra models” that are used to fill a game level with more detail (Co, 2006, pp. 255-256). In electronic games, crates usually consist of textured box geometries in various sizes. The number of crate textures used in an electronic game is usually limited to only a few textures. Such crate textures often depict different crate types that consist of various materials. The texture for a wooden crate object usually depicts the frame of the crate and several wooden laths. Once a crate box object has been created and textured, the object is commonly placed many times within a game world. Figure 6 shows an example of crate objects used in electronic games.

![Figure 6: Crates in electronic games](image_url)

Image provided by Todd Picken (www.garagegames.com). Reprinted with permission.
4.2.2 Crates as Candidates for Real-time Procedural Generation
Crates are very common in game worlds. Based on my own experience as a gamer and through targeted observations of electronic games, I identified crate objects in games as one object type that introduces an obvious element of repetition. The simplistic design of a crate object in the real world and in electronic games made it an ideal candidate as a starting point for my research on real-time procedural generation.

4.2.3 Crate Geometries
In the real world, crates can be constructed in a variety of ways and exist in many different sizes and shapes. Wooden crates usually consist of a frame and several wooden laths that are nailed to the frame to create a holding space for goods. In order to generate crate objects procedurally, I used the geometrical makeup of a real-world crate to construct a template, which subsequently allowed the development of a procedure to generate crate objects. A sketch of the crate object template is presented in figure 7.

Figure 7: Sketch of the crate object template
4.2.4 Procedural Crate Generation using Max Script

Based on the template it was possible to identify six visual characteristics of the crate object which could be parameterised and allowed the generation of a visual variety of crate geometries. The six crate characteristics included:

- Crate object width
- Crate object height
- Crate object depth
- Rectangular frame profile dimensions in width
- Rectangular frame profile dimensions in height
- Lath thickness
- Number of laths
- Gap between the laths

To generate the crate objects, the procedure first generated the crate frame and subsequently the lath objects for each side of the crate. The crate frame was generated based on the parameters determining the crate dimensions and the profile dimensions. The lath objects are generated based on the parameters for the dimensions, rectangular frame profile dimensions in width and height, lath count and variable gap between each lath, as well as the thickness of each lath. The MaxScript code to generate the crate objects is included on the CD that accompanies this exegesis. Instructions on how to run the script to generate a crate object using Autodesk’s 3D Studio Max 4 as well as a description of the parameters are provided in the Appendix. Figure 8 shows two different crate geometries and their generation values.

Figure 8: Crate geometries and their generation settings
Figure 9 shows a visual variety of procedurally generated crate geometries that were rendered using Autodesk’s 3D Studio Max 4.

### 4.2.5 Analysis of the Generated Visual Variety

To evaluate the generated visual variety I generated several crate objects in 3D Studio Max and collected qualitative data in form of notes and screen captures. The visual results were compared and subsequently analysed.

The generated results showed a visual variety of volume, dimension, form, frequency and size and space. The visual variety of volume, dimension and size is achieved through the variation of the height, length and depth parameters that control the dimensions of the crate object. The change in frame thickness parameter generated a visual variation in form. A change in lath count generated a visual variety in frequency. A change in distance between laths generated a visual variety in space, which was defined as gaps of dense and open areas. A more detailed discussion of the results is available in section 4.4.2.
4.2.6 Analysis of Real-time Performance
As explained in section 2.2.1, 3D Studio Max 4 is a modelling and animation package and not a real-time rendering environment. It was therefore not possible to conduct real-time performance tests directly from within the modelling and animation package. The testing of the real-time performance of a scene consisting of generated crate objects required the creation of another prototype.

4.2.7 Limitations
While the prototype successfully facilitated the generation of crate objects in a visual variety of volume, dimension, form, size, frequency and space, it did not generate a visual variety in texture. While Autodesk’s 3D Studio Max 4 provides \textit{uv} texturing tools, I was unable to generate random \textit{uv} texture positions to map the individual box components of the crate object using Max Script.

The user interface facilitated the manipulation of the generation parameters, but it was not possible to observe how the changes of the generation parameters affect the generated geometry until after the object was generated. Crate geometries that did not match the desired visual properties needed to be manually deleted and regenerated in a tedious trial and error process.

4.2.8 New Question
- How can a visual variety of texture and colour be generated in real-time to texture the frame and wooden lath elements of the generated crate object?

4.2.9 Summary
This prototype procedurally generated crate objects in a visual variety of volume, dimension, form, size, frequency and space based on a template encoded in MaxScript. The generated crate objects, however, were not textured and the geometry was not rendered in real-time. The real-time generation of textures was the focus of the next prototype.
4.3 Prototype 2: Lattice Examiner

The previous prototype procedurally generated crate objects that consisted of flat shaded box objects without textures in a visual variety of volume, dimension, form, size and space. With this prototype the research project investigated how a visual variety in texture can be generated in real-time. To this end, the investigation focussed on the texturing of simple building blocks such as wooden box objects to imitate the appearance of individual timber laths that show a variation of wood grain and colour.

4.3.1 Real World Wooden Crates

Real world wooden crates are often constructed of several wooden laths. Every wooden lath naturally shows a unique wood grain pattern and the colour of the wooden material can vary. Real world crates are often manufactured in exactly the same way. The wooden material, however, provides even crates that are identical in geometrical makeup with a unique individual appearance. The generation of an individual wood grain pattern for each individual box object was the focus of this investigation. It was envisaged that such box objects can be used in the construction of more complex objects such as crates.

4.3.2 Random Texture Coordinate Generation

Elias (2000) provides algorithms based on Perlin noise to generate three-dimensional procedural wood textures. The generation of such three-dimensional textures, however, was calculation intensive and did not seem feasible in real-time given the number of objects that need texturing.

The approach taken by this prototype generated random texture coordinates to create texture snippets from a large material texture. The texture snippets were subsequently mapped onto a box object to imitate the appearance of a uniquely textured wooden lath object. Figure 10(a) shows one of the tileable base textures used by this prototype. All generated texture snippets were rectangular and proportional in size to box object’s dimensions. All textures used in this experiment were tile-able and consisted of 256 x 256 pixels. To generate a visual variation in colour between the generated box objects, the prototype randomised the material’s diffuse colour within a user definable range.

The prototype used a pseudo random number generator to generate an uncorrelated sequence of $uv$ texture coordinates and colour values for each box object’s material diffuse colour, which subsequently provided each generated box object with its individual appearance.
4.3.3 Analysis of the Generated Visual Variety

To facilitate the analysis and subsequent evaluation, of the generated visual variety, the prototype displayed several textured box objects in a user definable number of rows and columns on the screen. This juxtaposition of box objects allowed a direct comparison between the various generated results. As the objects were generated in real-time, generation settings could be changed, which subsequently altered the visual result and allowed the analysis and evaluation of objects in a visual variation of volume, dimension and size, as well as a visual variation in colour and texture. The analysis was based on qualitative data in form of notes and screen captures.

The Lattice Examiner generated box objects in a visual variety of texture and colour. The texture sampling technique created the effect of a unique wood grain pattern for every box object. The randomisation of the material’s diffuse colour value intensified the visual variety of the textured box objects, as each wood texture showed in addition to the variation of texture also a visual variation in diffuse colour. Figure 10(b) shows the generated visual variety of textured box objects. The prototype was also able to generate lattice patterns. Figure 10(c) shows a generated wooden floor texture which resulted from the lattice pattern.

![Figure 10: Diversely textured box objects](image)

4.3.4 Analysis of Real-time Performance

The random generation of the texture coordinates only required the calculation of 4 $uv$ coordinates per face; a total of 24 $uv$ coordinates per box object. Similarly, the randomisation of the textures diffuse values only required 3 random values per box object. The generation of only 27 random number values for a single box object did not seem to pose a significant processing workload that could impact on the overall frame rate. The real-time performance was therefore not measured at this stage of the project.
4.3.5 **Limitations**
While the generated wood texture generated a visual variety in texture, the pattern was still based on an image texture and was therefore not unique, unlike the wood pattern of lath objects in the real-world. A close inspection also revealed that the wood grain did not naturally continue from one face to a neighbouring face. Moreover, the prototype only generated simple textured box objects. The generation of objects other than crates required the generation of base primitives in different shapes and with different materials and textures that were not covered by this prototype.

4.3.6 **New Questions**
- Is it possible to generate a wooden crate object that consists of such textured box objects in real-time?
- Can a visual variety of crate objects be generated based on the random texture snippet and material diffuse colour approach?

4.3.7 **Summary**
This prototype demonstrated that box objects can be generated in a visual variety of texture and colour in real-time. The visual variation in texture was achieved through the application of texture snippets that were sampled from a larger material texture and mapped onto a basic box object. The visual variation in colour was realised through a randomisation of the material’s diffuse colour. The prototype demonstrated an elementary approach to procedural generation that allowed the construction of more complicated objects. The construction of a crate object that consisted of such generated objects was the focus of the next prototype.
4.4 Prototype 3: Crate Examiner

The Crate Examiner prototype was developed to generate the first real-time prototype that generated a crate object in a visual variety. As such it combined the procedures that were used to generate crate geometries and textured box objects in real-time.

4.4.1 Real-time Procedural Crate Generation

The Crate Examiner prototype reused the procedure developed in the first prototype to generate the crate geometries in real-time. The generated crate objects consisted of several of box objects that were arranged to create a frame and several lath objects. The generation of the geometry was controlled by the same set of parameters that was identified in section 4.2.4. To generate crate objects in a visual variety of texture and colour, the prototype also reused the procedures developed in the previous prototype. Subsequently, the box object’s that made up the structure of the crate object were individually textured and the material’s diffuse colour was randomised.

4.4.2 Analysis of the Generated Visual Variety

The evaluation of the visual variety was based on a juxtaposition of screen captures that show how certain elements of the crate object are influenced by single generation parameter. The juxtaposition allowed a direct comparison between the various generated results. A selected range of visual results is presented in Figures 11 – 16 to illustrate the discussion of the findings.

Figure 11 shows crate frames with various frame profile dimensions. As evident by the images, the change in frame profile results in a visual variety of frame volume.
Figure 12 shows a visual variety in frequency, which is evident by the change in formal structure through the even distribution of the changing number of wooden laths that make up the side panels of the crate object.

![Figure 12: Visual variety in frequency evident as change in formal structure](image)

A visual variety in space, achieved by an increasing gap between the five wooden lath objects on each side of the crate is illustrated in figure 13.

![Figure 13: Visual variety in space](image)

Figure 14 shows four crate objects that are identical in geometry and texture. As evident in these images, the change in diffuse colour for every box object creates a visual variation in colour between the otherwise identical objects.

![Figure 14: Visual variety in colour](image)
Figure 15 shows four crate objects that are identical in geometry, but display a visual variety in texture. While the image used to texture the crate was the same for all crate objects, a visual variety in texture was generated by the texture snippet procedure that applied a difference in the pigmentation of the wood grain pattern for every wooden lath object.

The analysis of the generated crate objects showed that crate objects can be generated in a visual variety of volume, space, form, texture and colour. The selected crate objects in figure 16 show a visual variation in dimension, form, size, volume, texture, colour, frequency and space. The visual variation in form displayed in figure 16 becomes evident through the various shapes of the crate objects.

4.4.3 Analysis of Real-time Performance
A detailed measurement of the real-time performance of the generated and textured crate objects was not conducted, as the prototype only generated one crate object. A rough measurement of the frame rate revealed that the generation of a single crate object could be displayed at the maximum display frame rate of 100 fps. The performance test of several real-time generated crate objects in a virtual space required the development of a new prototype.
4.4.4 Limitations
The prototype could only generate one crate object at a time. It was therefore not possible to compare the generated crate objects side by side in an environment with other crate objects embedded within a space and subsequently analyse the visual variety of a virtual space. Without the generation of a virtual space filled with generated crate objects it was also not possible to ascertain how many crate objects could be generated at real-time frame rates and how many crates need to be generated to create a believable virtual space.

4.4.5 New Questions
- How can a virtual space be generated that consists of real-time generated crate objects?
- What is the impact of procedural generation on the frame rate?
- How many objects need to be generated to create a believable virtual space?

4.4.6 Summary
As a partial answer to the first research question, this prototype demonstrated how objects, such as crates, can be generated in a visual variety of volume, dimension, form, size, frequency, space, texture and colour in real-time. The prototype, however, only generated one crate object at a time. The generation of a virtual space consisting of a visual variety of crate objects was therefore the objective of the following two prototypes.
4.5 Prototype 4: Pile Examiner

A crate warehouse was the ideal test environment to examine the virtual variety of the generated crate objects. One of the inspirational sources regarding the construction and the overall layout of the virtual space came from the government warehouse scene of the movie: Indiana Jones and the Raiders of the Lost Ark (Spielberg, 1981). However, before it was possible to generate an environment which consists of crate objects it was necessary to investigate an approach to distribute several crate objects to form a pile structure.

4.5.1 Generating Crate Piles

The government warehouse in the movie Indiana Jones and the Raiders of the Lost Ark is a very large and dark storage warehouse filled with many crates. All the crates that are stored in the warehouse seem to consist of the same kind of timber but are of different sizes and arranged in a semi-organised but still somewhat random way. Images that show the government warehouse could not be included for copyright reasons, but can be found in Wikipedia (2006).

This prototype was developed to examine the distribution of several crate objects within a pile structure. At the beginning of this investigation I tried to adapt an approach that was originally developed for the procedural modelling of brick wall textures (Miyata, 1990). I envisaged that the appearance of the stacked crate objects in the warehouse could be achieved with a regular arrangement that is randomly disturbed through changes in crate size, random translations and rotations. However, I found it was too difficult to adapt Miyata’s two dimensional approach to the problem of stacking crate objects in a three-dimensional space.

This prototype generated a simplistic pile structure that consisted of geometrically identical crate objects in a visual variety of texture and colour. To generate a visual variation in form of the pile structure, the individual crate objects were randomly rotated around the vertical axis. The pile structure consisted of 24 crate objects that were divided into eight columns with three crate objects for each column around a pillar.

4.5.2 Analysis of the Generated Visual Variety

The Pile Examiner displayed one pile structure on the screen. Generation parameters to vary the pile structure’s dimension, the gap between the pile columns and the random horizontal angle range allowed the generation of pile structures in a visual variety of volume, dimension, size, colour, texture and space. Figure 17 shows a visual variety of pile structures.
4.5.3 Analysis of Real-time Performance

A detailed measurement of the real-time performance of the generated pile object was not conducted. A rough measurement of the frame rate revealed that the generation of a single pile with 24 crates could be displayed with the maximum display frame rate of 100 fps. The performance test of several real-time generated pile objects in a virtual space required the development of a new prototype.
4.5.4 Limitations
The prototype generated a single pile object that showed a variable number or crate objects. All crate objects however were of the same size and geometry. While the pile itself could be generated in a visual variation of volume, dimension, form and size, the prototype did not facilitate the generation of crate objects in a visual variety other than texture and colour. As the prototype only generated one pile object, it was not possible to analyse the visual variation of an environment filled with crate piles.

4.5.5 New Questions
- How can a space that consists of crate pile structures be generated in real-time?
- How does the visual variety of the generated crate objects affect a virtual space?
- What is the real-time performance of a virtual space that consists of generated crate objects?

4.5.6 Summary
The Pile Examiner generated crate objects that were organised in a box shaped pile structure at real-time frame rates. The pile objects could be generated in a visual variety of volume, dimension, size, colour, texture and space. Unfortunately it was not possible to generate a visual variety of objects within a pile structure. The generation of a virtual space, which consisted of such pile structures was the objective of the Warehouse prototype.
4.6 Prototype 5: Warehouse

The Warehouse was the first prototype of a procedurally generated virtual space, which consisted of generated objects in a visual variety. The generated virtual space consisted of crate objects that were organised in several pile structures, which in turn were distributed within the virtual space. The Warehouse prototype facilitated the examination and analysis of the visual variety of the generated crate objects in comparison to each other. Moreover it was possible to measure the effect of the real-time procedural generation on the real-time performance of the space.

4.6.1 Generating a Warehouse

Before it was possible to generate a virtual space in real-time with objects that did not exist until they were generated and subsequently distributed, it was necessary to develop several new technical approaches: view frustum filling was an approach to limit the generation of objects, dynamic procedural generation facilitated the generation of objects as needed and a hashing approach was used to maintain the persistence of the virtual space.

4.6.2 View Frustum Filling

Electronic real-time 3D games often use a view frustum culling approach to constrain rendering to just the geometry visible from a particular viewpoint. As such, game engines either load the entire game level or continuously load parts of a game level into the computer’s memory. In most cases, the frustum culling approach is used to cull the level’s geometry to the portion that is visible by the player (Watt & Policarpo, 2001). Figure 18(a) illustrates the approach to view frustum culling. The conventional culling approach starts with existing level geometries illustrated by black squares. The level data outside the viewing area is culled in a subsequent processing step.

The virtual space of this prototype was initially empty. Only objects visible to the user’s viewing area were procedurally generated, textured and positioned in space. In other words, the view frustum is filled with geometry and does not cull hidden, but existing geometry. The term view frustum filling was used to describe the restriction of procedural generation to parts of the virtual world that are located within the camera’s view. Figure 18(b) illustrates the approach to view frustum filling. Before the view frustum could be filled with geometry the procedure needed to determine which objects were visible before they were generated.
The approach to view frustum filling that was implemented in this prototype, partitioned the flat terrain into square cells. The cells were arranged in loops around the camera’s position. Each cell represented a proxy for its procedurally generated content and its centre position was evaluated for visibility. For every frame, cells were tested for potential visibility before their content was generated and drawn. The potential visibility of a cell was determined by the angle between the cell and viewing direction, as well as the distance from the camera. The Warehouse prototype displayed only the content of cells that were located within a 60° viewing angle within a user definable distance.

Figure 19 shows an illustration of all cells that are subsequently tested for visibility from a bird’s eye view. The viewing direction of the viewing area is indicated by a green arrow. Only cells with centres located within 30° on either side of the viewing direction are visible. Visible cells are depicted with a black stroke and invisible cells are depicted with a gray stroke.

### 4.6.3 Dynamic Procedural Generation

To be able to navigate through the procedurally generated virtual space, it was necessary to limit the amount of procedural generation. View frustum filling was used to determine which objects are within the visible viewing area. The prototype only dynamically generated objects that were located in the users view frustum for every frame.
To prevent confusion and disorientation the virtual space needed to be visually coherent because users were only able to navigate through a virtual space or revisit a particular location if the prototype could regenerate the same object geometries and textures for a particular cell for every frame. To achieve this, each cell in the virtual space was identified by a unique 32 bit *id* value.

The *id* value was determined by hashing the *x* and *y* integer coordinates of the cell’s centre position with a global *worldId* value, which identified the number of the virtual space. The hashing approach used the following procedure:

\[
\text{cellId} = \text{hash}(x^{\text{hash}(y^{\text{worldId}}))};
\]

The hashing procedure used Thomas Wang’s 32 bit mix function (Wang, 2002) which was based on a sequence of bitwise operations and returned a 32 bit integer value for a given integer ‘key’ value. The procedure provided a good distribution of *id* values.

The generated *id* values were used to generate parameter values that controlled the construction and texturing of the objects within the virtual space. In this prototype a cell contained either a floor path tile, a pile of crates around a pillar on a floor tile or a floor tile and a pillar segment without crate objects. The *id* value for the cells containing the crate piles were used to generate values for the
random rotation of the crate objects within a pile as well as values that controlled the texturing and
colour of the individual crate objects within a pile object. Neither, the floor texture tiling nor the pillar
objects were affected by the generated \(id\) values.

4.6.5 Theoretical Size of the Virtual Space
The use of a 32 bit integer for \(x\) and \(y\) limited the extent of the virtual world to \(2^{32}\) proxy cells in each
dimension. At a speed of two cells per second, a journey in a straight line from one end of the virtual
space to the other would take about 68 years — approximately one human lifetime.

4.6.6 Reduction of Crate Polygons
In the first version of the Crate Examiner the crate geometry consisted of only box objects with eight
vertices for every box object. After using the crate object in the Warehouse prototype the frame rate
dropped to 15 fps when displaying approximately 200 crate objects. Consequently, the geometry of the
crate template was simplified to increase performance. This simplification was achieved by replacing
the wooden lath box objects with textured rectangles. Figure 20 shows two crate objects that were
generated based on the original and the simplified template. The reduced crate geometry resulted in a
slightly better frame rate of 20 fps when displaying approximately 200 crate objects.

![Figure 20: Reduction of crate polygons](image)

(a) box objects as wooden laths  (b) rectangles as wooden laths

4.6.7 Analysis of the Generated Visual Variety
The Warehouse prototype generated a virtual space filled with crate objects that were organised in a
pile structure. The crate objects were identical in geometry, but showed a visual variety in texture and
colour. The crate objects were arranged in a pile structure which consisted of columns made up of
stacked crate objects. Each crate object within the pile structure was randomly rotated which generated a visual variety in form. Figure 21 shows a screenshot of a virtual space consisting of approximately 3000 generated crate objects.

To analyse the generated visual variety I rendered the same scene without the variation in form, texture and colour so that both images can be compared and the visual variety can be identified by the reader. Figure 22 shows the generated result; a scene that consists of pile objects that are constructed of just one crate object.
4.6.8  Analysis of Real-time Performance

The implementation of view frustum filling enabled the real-time display of a scene filled with generated objects at interactive frame rates below 30 fps. However, all objects in the Warehouse prototype were regenerated for every frame, which was very calculation intensive. Approximately 300 crate objects could be rendered at a low, but still interactive frame rate of 15fps. In order to increase the performance and subsequently display a larger number of objects, a more efficient solution, than the regeneration of objects and texture coordinates afresh for every frame, needed to be investigated.

4.6.9  Limitations

The prototype demonstrated the generation of a virtual space that consisted of objects in visual variety in form, texture and colour. However, the geometry of every crate, in each pile, was identical and reintroduced an element of repetition. Similarly, the approach to generate the crate pile structure reintroduced another element of repetition, as all crate structures were generated with identical volume, dimension and size. Also the regular distribution of the pile structures within the virtual space created another element of repetition.

4.6.10  New Questions

- Is it possible to use the approach to generate objects, textures and virtual spaces to generate other object types and virtual spaces?

- How can the performance of real-time procedural generation be improved?

4.6.11  Summary

The Warehouse prototype generated a large scale virtual space that was filled with crate objects. In order to keep the coherence of the virtual space, the prototype divided the virtual space into cells that were identified by their \( xy \) position. A hashing approach was used to generate the values for the generation of the objects based on the cell positions. To facilitate real-time navigation, a view frustum filling approach was used minimise the procedural generation of objects in combination with a dynamic procedural generation approach.

The Warehouse prototype was the first prototype that provided an opportunity to analyse the visual variation of the generated objects in a virtual space and the performance. The prototype demonstrated a visual variety in form, texture and colour, however, it was not possible to analyse the visual variety in dimension, volume, scale, space, frequency and form of the crate objects that could be generated by
the Crate Examiner due to limitations of the procedural stacking approach that was developed in the Pile Examiner.

The Warehouse prototype represented the end of the first series of prototypes. To explore if the developed approaches to generate a visual variety of objects could be applied to another type of environment I considered a new and more ambitious theme. The subsequent investigation focused on the generation of a visual variety of buildings and subsequently virtual spaces such as cities.
4.7 Prototype 6: Building Examiner Version 1

The Building Examiner was the first iteration of prototypes that generated building objects. The prototype reused the template approach to generate the facade geometries of the buildings. The building geometries were subsequently textured by the texture snippet approach developed in the Lattice Examiner.

4.7.1 Generating Buildings

Real world cities can be seen as a conglomeration of buildings that are located within the confines of building lots. The building lots are the result of a street grid, which subdivides the terrain. The buildings within each lot shape are often unique in geometry and texture. The visual style of the building objects generated by the Building Examiner Version 1 was influenced by the low polygon style of contemporary electronic games, such as Microsoft’s Flight Simulator 2004, but also drew on impressions and observations of primarily the city of Melbourne, Australia.

In order to generate building objects procedurally, I examined several images of real-world buildings as well as building objects that were used in electronic games. I limited the investigation at this stage to only buildings that consisted of rectangular floor plans that were subsequently extruded in height. Similar to the approach taken in the first prototype, I identified the following five visual characteristics:

- Building width
- Building height
- Building length
- Ceiling thickness
- Ceiling overlap

To generate the building objects the procedure used a rectangular floor plan defined by the width and length parameters as a basis. The rectangular floor plan was subsequently extruded to create the floor and ceiling geometries for every floor.

Every floor was defined by a ceiling box object of a certain height that could extend over the rectangular floor plan base. How far the ceiling object could extend outside the basic floor plan was defined by the overlap parameter. The number of floors that was generated for every building was defined by the height parameter. Figure 23 shows the template drawing.
4.7.2 Analysis of the Generated Visual Variety

To evaluate the visual variety of the building objects I generated a range of building objects and collected qualitative data in form of notes and screen captures and analysed their visual variety. The Building Examiner Version 1 prototype generated very basic, box shaped office buildings in a visual variety of dimensions, scale, size, volume which can be seen in the selected screen captures in figure 24.
The same procedure that generated random \( uv \) coordinates, that was used to generate the random distribution of the wood texture in the second prototype was also used by this prototype. The Building Examiner Version 1 prototype used the procedure to generate different window states for every window based on a variety of curtain positions in the image texture. A comparison of selected screen captures illustrates the visual variety of texture in figure 25.

![Figure 25: Visual variety of window textures](image)

The building objects that could be generated were quite bland. Column objects and ceiling overlaps were added to increase the visual variety of form. The subsequent screen captures in figure 26 show the visual variety in form that could be generated.

![Figure 26: Building objects showing a visual variety of form](image)

### 4.7.3 Analysis of Real-time Performance

A detailed measurement of the real-time performance of the generated building objects was not conducted, as the Building Examiner Version 1 was only a preliminary experiment. A rough measurement of the frame rate, however, revealed that the generation of a single building could be displayed with the maximum frame rate of 100 fps.
4.7.4 Limitations
The main body of the building, as well as the ground shape of the balconies were always of a rectangular shape. As a consequence, the prototype generated building shapes that resulted in a very predictable representation. Furthermore, the buildings did not show visually different sections for the ground floor, mid section and rooftop section, which would have added more variety. While the window textures that were applied with different window states on the building added a visual variety of texture, I felt that the visual difference between buildings of the same geometric makeup was very subtle.

4.7.5 New Questions
- How can building objects be generated that consist of more complex floor plans?
- How can building objects that consist of several sections be generated?

4.7.6 Summary
The approach to generate building objects followed the same approach that was used in the Crate Examiner prototype, and subsequently generated buildings in a variable but predictable form. As such, the Building Examiner Version 1 prototype generated building objects that show primarily a visual variety of dimension, scale, size, volume and texture. The introduction of parameters to change the overlap of floor segments and column objects further enhanced the visual variety of the generated building objects in form, as the added geometry masked the box shaped character of the generated building objects. This observation motivated the development of an approach to generate building objects based on a visual variety of floor plans and a visual variety of form based on building sections.
4.8 Prototype 7: Building Examiner Version 2

The Building Examiner Version 2 focused on the generation of a variety of floor plan shapes based on polygon primitives such as squares, rectangles, and n-sided polygons. The generated floor plans were subsequently extruded to generate building facades in a visual variety of form that could not be achieved in the Building Examiner 1. To increase the visual variety of form, the building facades were subdivided into sections to mimic the appearance of setbacks.

4.8.1 Generating Buildings

This investigation on building objects was based on observations about buildings in Melbourne. An examination of buildings in Melbourne showed that floor plan shapes of buildings can often be subdivided into primitive geometric shapes such as squares, rectangles, and n-sided polygons. Figure 27 shows some selected examples of buildings in Melbourne.

![Figure 27: Buildings with obvious floor plan shapes](image)

High rise buildings in Melbourne often consist of several sections. Sections can often be identified by a change of the facade design or a change in the floor plan shape, resulting in a setback. Within each section, the floor plan shape is usually repeated. Figure 28 shows images of buildings in Melbourne consisting of several sections.

Based on the observations, the prototype generated a floor plan for each section of building object. In order to generate the facade of the building, the section floor plans were extruded in height.
4.8.2 Floor Plan Generation

The floor plans that were generated by this prototype were two-dimensional polygon shapes. Floor plans were generated for each section of the building object from the top level to the ground level. The floor plan shapes were constructed of randomly selected and merged regular polygons such as squares, rectangles, and n-sided polygons.

The generation of the floor plans of every section of a building object was realised by an iterative process, which is described on the following example illustrated in figure 29. The first iteration generated a random base polygon which served as the top floor plan in this example. The result is illustrated in figure 29(a). Figures 29(b) – (e) show subsequent iterations, where a new floor plan is generated for every building section. Every iteration generated a new random polygon and combined it with the floor plan polygon of the previous iteration. The combination of floor plans was realised by a boolean union operation. The prototype used the General Polygon Clipping Library (Murta, 2000) to
combine the two two-dimensional shapes. Figure 29(f) illustrates the final combined floor plan of this example generation.

![Figure 29: Iterative floor plan generation](image)

The resulting polygon geometry was scaled to fit into a unit square at the end of the floor plan generation and surface normals were calculated. The final floor plan was stored in a `std::vector` data structure. The pseudo-code for the procedure is provided in algorithm 1.

**Algorithm 1: Floor plan generation**

```plaintext
src ← random polygon

for every building section do
    tmp ← random polygon
    rotate tmp randomly about y axis
    translate tmp to random vertex in src
    src ← src union tmp
```

4.8.3 Building Facade Generation

The shape of the building facade in the Building Examiner Version 2 prototype was defined by the section floor plans and the number of sections or setbacks in the building object. The generation of the building facade is explained on the example illustrated in figure 30. The figure shows the generated floor plans for each section and the correspondingly generated facade geometry.

As described in section 4.8.2, floor plans are generated for each building section and the building facade is generated from the top of the building to the bottom of the building. Figure 30(a) shows that the first generated floor plan shape is not used to extrude a building section in order to avoid roof tops that consist of just a single basic shape such as a rectangle. The extruded building section illustrated in figure 30(b) shows that at least 2 primitive shapes need to be combined to extrude a building section such as the roof top. Figures 30(b-f) show building sections of random height that are each attached to the bottom of the preceding section and that the building subsequently increases in height.
4.8.4 Building Facade Texture Mapping

Texture coordinates were generated for every floor of the building. The texture library consisted of several textures for the ground floor, mid sections, roof section and roof top. Textures for the ground floor and mid sections of the building consisted of four rows and contained four windows in each row.

To generate the texture coordinates of a single floor face in the building, the procedure calculated the horizontal width of the floor face and divided it by the width of a single window. The rounded result provided the number of windows that needed to be mapped onto the face. To generate the effect of random window states, the building generator randomly selected an image texture from the image texture library for the entire building and randomly selected a row within the texture. For every floor, the texture generator randomly selected the mapping coordinates of a window within the texture and from there maps the required number of windows onto the face. This process was repeated for every floor face in the building and subsequently led to a visual variety of texture.

Roof tops, however, were textured with a single roof texture that was applied onto the entire rooftop face. Figure 31 shows the wire frame of a building and the selection of textures used to create the textured version of the same building. The selection of slices in the textures and the application on the building is highlighted in blue, red, yellow and green.
4.8.5 Analysis of the Generated Visual Variety

The evaluation of the visual variety was based on a juxtaposition of screen captures that show how certain elements of a building object are influenced by single generation parameter changes. The juxtaposition allowed a direct comparison between the various generated results. A selected range of visual results is presented in figures 32 - 37 to illustrate the discussion of the findings.

Figure 32 shows building geometries with varying complexities in the range from 2 to 30 primitive shapes. The more primitive shapes were added to the floor plan, the more complex became the building geometry. If, however, the number of primitive shapes that was added to the floor plan exceeded a certain amount, the generated buildings lost some of the characteristics of a building. The last geometry’s floor plan consisted of 30 combined primitive shapes and shows only a few textured windows because the faces were too small for a window texture. Faces that are too small for a window texture were textured with a concrete texture.
Figure 32: Building geometries of various complexity
Setbacks affected the volume and horizontal silhouette of a building. Figure 33 shows building geometries with a varying number of setbacks. The generation of setbacks depended on the number of shapes that were integrated into the floor plan. A floor plan always needed to consist of at least one more shape than the number of setbacks that was applied to the building.

Figure 33: Building geometries with various number of setbacks
Figures 34-37 show building objects of a visual variety in volume, dimension, form, size and texture. All buildings were generated with a complexity parameter lower than 20 and a setback parameter of up to 19. The generated buildings of each category are similar in vertex count.

Id’s: (a) 159  (b) 197  (c) 244  (d) 353  (e) 136
Figure 34: Building variation with 3 setbacks and a complexity of 4

Id’s: (f) 269  (g) 1936  (h) 386  (i) 6206  (j) 261
Figure 35: Building variation with 9 setbacks and a complexity of 10

Id’s: (k) 8570  (l) 2410  (m) 10536  (n) 2542  (o) 6828
Figure 36: Building variation with 19 setbacks and a complexity of 20
4.8.6 Analysis of Real-time Performance

A detailed measurement of the real-time performance of the generated building objects was not conducted. A rough measurement of the frame rate revealed that the generation of a single building object could be displayed with the maximum display frame rate of 100 fps. Further performance tests were intended for the next prototype.

4.8.7 Limitations

Compared to the Building Examiner Version 1, the shape of the floor plans that are subsequently used to generate building geometries is less predictable and generates a much larger and more intricate visual variety in form. However, the final geometry of the generated buildings is largely unknown before its generation and the shape cannot be influenced by the user. This could be problematic for applications where the shape of building sections need to be defined by the artist.

4.8.8 New Questions

- How can a virtual city space be generated?
- How many buildings need to be generated in order to obtain the impression of a virtual city?
- How will the real-time generation of a visual variety of buildings impact on performance?
4.8.9 Summary

The Building Examiner Version 2 generated floor plan shapes by combining shape primitives at random. The generated floor plans were subsequently extruded into sections to generate the facade shape of building objects. The building facades were textured with a modified version of the texture snippet procedure developed in the Lattice Examiner prototype. Compared to the Building Examiner Version 1, the shape of the floor plans that are subsequently used to generate building geometries is less predictable and generates a much larger and more intricate visual variety in form. The shape of the generated floor plans, however, cannot be controlled by a user. The division of a building into sections generated building objects in a visual variety of form. The progression to generate a virtual city consisting of such buildings was pursued in the following prototype.
4.9 Prototype 8: Undiscovered City

The Undiscovered City prototype showcased a computer generated virtual city that consisted of dynamic procedural generated building objects. The virtual city facilitated real-time interactive exploration from a first person perspective. The building geometries within the virtual space were generated using the approach discussed in the previous prototype.

4.9.1 Real and Virtual City Spaces

Cities are very common in many electronic games (Morris & Hartas, 2004). The cities depicted in games are often fictional but there are games that use city environments that are modelled on well known cities. Microsoft’s Flight Simulator 2004 uses satellite images of real-world cities to texture the terrain and uses three-dimensional models of landmark buildings and structures to recreate the city in the virtual space. In Activision’s True Crime series, the game worlds are detailed three-dimensional recreations of New York and Los Angeles. Rockstar’s GTA series uses fictional three-dimensional cities based on real world cities, such as New York, Miami, San Francisco and Los Angeles. In Electronic Art’s Sim City 4, players create their own fictional cities. The cities in game worlds, however, often consist of objects that show a naturalistic representation in dimensions, volume, scale, colour, form and texture. The dynamic procedural generation of a virtual city seemed to be a suitable subject on which to apply the methods so far developed in this research.

4.9.2 Generating Cities

The Undiscovered City prototype combined the findings of the Warehouse and Building Examiner Version 2 prototypes. Similar to the Warehouse prototype the Undiscovered City prototype used a deterministic approach to generate the shape of a building object based on its location in the virtual space, which made it possible to regenerate the same buildings in the case of a user returning to an already visited location.

The Undiscovered City prototype also used the view frustum filling approach that was used in the Warehouse prototype as well as the approaches to dynamic procedural generation and distribution of objects in a virtual space. The frame rate of the Warehouse prototype as well as the number of objects that could be displayed in real time, however, was despite the implementation of view frustum filling not satisfactory for a real-time application.

To facilitate the dynamic procedural generation at real-time frame rates, a display list cache with a least recently used (LRU) replacement policy was implemented to improve the performance. A screenshot of the Undiscovered City prototype is illustrated in figure 38.
4.9.3 Caching
The LRU display list cache stored generated objects temporarily in memory from where they could be redrawn for subsequent frames. While a user was navigating through the virtual space, only a few new objects appeared on the horizon and needed to be generated. The geometry of most other objects was generated and remained the same for subsequent frames. Once objects were generated and stored in a cache, they could be redrawn without being regenerated. Cached object remained in memory until they were no longer needed. The LRU display list cache kept track of all objects and removed the least recently used objects from memory. Objects that were revisited but were no longer available in the cache needed to be regenerated.

4.9.4 Display List Caching
Display list caching was based on the following assumptions:

- It is substantially faster to render a procedural shape than to actually generate it
- Geometry drawn in the current frame is likely to be redrawn in subsequent frames (temporal coherence)
- The granularity of procedural geometry is fine enough to allow dynamic procedural generation interleaved with display
- There is sufficient memory to store procedural geometry for reuse in subsequent frames.

The Undiscovered City prototype used an OpenGL display list cache with least recently used replacement policy for the cache management. The cache was responsible for maintaining a set of recently used buildings and street geometries, reusing older display lists for new buildings, compiling new display lists and deleting old display lists as necessary.
4.9.5 OpenGL Display Lists

Display lists are often used in situations where the same geometry needs to be redrawn multiple times (Wright & Sweet, 2000). An OpenGL display list captures a sequence of OpenGL instructions, which is then compiled and the same sequence of instructions is performed whenever the display list is used (ARB, Shreiner, Woo, Neider, & Davis, 2004, pp. 271-294). Depending on the OpenGL implementation, calling a display list is usually more efficient than reissuing the same instruction stream to OpenGL commands. However, display lists cannot be modified once compiled, so they may prove less suitable for geometry which continuously changes over time.

Display lists have several advantages in the context of procedurally generated geometry. Nearly all of the available OpenGL functionality is captured without the need for complicated data structures. This allows ease and flexibility of immediate mode procedural generation and rendering for the developer, combined with the performance of compiled display lists in subsequent frames.

The performance advantage using display lists varies between OpenGL implementations, with the details of the display list compilation being hidden from the developer.

4.9.6 LRU Cache

The total memory requirement and generation time can be substantial in a pseudo infinite procedural world. During navigation in this prototype only a fraction of the world was visible, and it was only this part that needed to be generated. A lazy evaluation approach allowed procedural generation of geometry only as needed. Making use of a cache with a LRU replacement policy ensured that OpenGL display list resources were used efficiently, with the reuse of display lists least likely to be used in subsequent frames.

The implemented LRU cache performed three main tasks: determination if a particular item was in the cache; insertion of a new item into the cache; location of a particular item in the cache. These queries needed to be efficiently handled in order to prevent a performance bottleneck.

4.9.7 LRU Cache Implementation

The lru container was implemented in C++ using a doubly linked list container (std::list) and a balanced tree (std::map) container from the C++ standard library as illustrated in figure 39. These two data structures in combination allow tracking of the order of access and efficient queries.
The list was sorted by order of access; items were moved to the front whenever they are queried, ensuring that less recently used items drift towards the end of the list. The \texttt{std::map} provided a fast index into the linked list. Querying a balanced tree required $O(\log n)$ time, whereas $O(n)$ time is required for a linked list.

The \texttt{lru} container stored pairs of keys and values. The key was the integer id of the originating cell, obtained by the hashing function and was used for querying the cache. The value was a collection of data including the OpenGL display list identifier and a time stamp corresponding to the time of last access. The \texttt{std::map} was used to lookup list entries based on the key. The C++ \texttt{lru} container used in this prototype was templated in order to support arbitrary keys and values for flexibility:
template<class Key,class Value> class lru
{
  typedef std::list<std::pair<Key,Value>> List;
  typedef std::map<Key,List::iterator> Index;

public:
  const uint32 size() const;
  const Value &front() const; // Most recent
  const Value &back() const;  // Least recent
  const Value pop_front();   // Most recent
  const Value pop_back();    // Least recent

  Value *find(const Key &);
  Value &insert(const Key &,const Value &);
  Value &insert(const Key &);   // Recycle LRU item

private:
  List list;
  Index index;
};

4.9.8 Display List Cache Implementation

The display list caching procedure is described in algorithm 2. The behaviour of the cache was configured via three parameters: the minimum age of an item before it is reused (minAge); the maximum age of an item before it is removed (maxAge); and the capacity of the cache (capacity). The parameter minAge was implemented to store new items in the cache for a short time. The parameter maxAge allowed the cache to shrink over time and to return system resources. The capacity parameter established an upper limit for the cache size to avoid excessive consumption of memory for caching.
Algorithm 2: Display list cache draw

// Release very old items

while time stamp of LRU item > maxAge do
    remove LRU item
    delete display list

// Cache lookup

query cache for requested item

// Draw Item

if item already exists then
    move item to front of list
    update time stamp
    draw display list
else
    \[ a \leftarrow \text{age of LRU item} > \text{minAge} \]
    \[ b \leftarrow \text{age of LRU item} > 1 \text{ AND cache capacity exceeded} \]
    if \( a \) OR \( b \) then
        reuse LRU item
    else
        insert new item
        update time stamp
        compile and draw new display list

Each item in the cache was time stamped with the frame number in which it was last used. Items aged when they were not being used and were available for reuse once they reached \( \text{minAge} \). Items may have been reused earlier if the cache capacity was exceeded. Items that reached \( \text{maxAge} \) were always removed to keep the number of objects stored in the cache to a minimum, and to provide space for newly generated objects, without the need to delete an object at the same time, thus balancing the workload of the processor. Note that the capacity of the cache was not strictly enforced. The cache would always retain items from the previous frame whether or not the desired capacity was maintained.
Buildings close to the left and right edge of the view frustum were only displayed for a few frames before they left the view frustum. To avoid the regeneration of objects in case of a left or right turn a \( \text{minAge} \) value of 100 frames was used. Buildings that reached the \( \text{maxAge} \) of 1000 frames were considered outside the visible view frustum even in the event of a 180° turn. The view frustum filling procedure limited the number of objects that were displayed in a frame to roughly 100 buildings at a viewing angle of 35°. To allow the storage of all objects for the case that a user turned 360°, a \( \text{capacity} \) value of 1500 items was used.

### 4.9.9 The Theoretical Size of the Undiscovered City

In theory, the virtual city was extremely large and consisted of geometrically different buildings equivalent to about 600,000,000 times the world’s current human population. Hardware limitations, however, only allowed a limited number of buildings to be displayed in each frame. The virtual city, however, could be freely explored from a first person perspective. Figure 40 shows a screen shot from street level.

![Figure 40: The Undiscovered City viewed at street level](image)

### 4.9.10 Analysis of the Generated Visual Variety

The Undiscovered City prototype generated a virtual space filled with building objects. As such, the virtual city consisted of buildings of a visual variety in volume, dimension, form, size and texture. The geometry of the buildings and the street grid was procedurally generated. The texturing, however, was based on image maps. A change of the image map textures resulted in a virtual city with a different look and feel. Figure 41 shows a screen shot of the Undiscovered City prototype using a different texture set.
Figure 41 illustrates screen captures that show a juxtaposition of virtual city spaces with various building count. The analysis of the number of buildings that were generated highlighted that the visual difference between 100 and 200 buildings that were rendered on the screen was noticeable. Also the difference between 200 buildings and 400 buildings could still be noticed. The visual difference between 400 and 800 rendered buildings, however, seemed less significant. The images show that at a viewing angle of 35°, 400 generated buildings were sufficient to create the impression of a city, as the space between the buildings was filled with other buildings in the background and did not show the sky texture. As the space between buildings became less obvious, the generation and subsequent display of new objects at the outer rim of the viewing area was less noticeable.

4.9.11 Analysis of Real-time Performance

I conducted several performance tests to assess the performance improvement of using the LRU display list cache. The following sections discuss the results of the measurements regarding the OpenGL display list caching, LRU performance and the overall performance of the Undiscovered City prototype. All tests were conducted on the test platform described in section 3.4.5 running RedHat Linux 8.0 using an 800x600 pixel OpenGL window in 32 bit colour.
4.9.12 OpenGL Display Lists and Procedural Building Generation

OpenGL display list compilation added an additional overhead to the time needed to procedurally generate an object. This series of tests was used to determine the speedup factor of using OpenGL display lists in comparison to the previously used approach of regenerating all visible objects for every frame.

In order to test the suitability of OpenGL display lists in regards to the dynamic procedural generation I used the building generation procedure developed for the Building Examiner Version 2 prototype to generate the content for all the tests in this series. The tested building objects consisted of at least 300 and up to 2000 vertices per building depending on the building complexity.

The measurement of the time for the generation of a building, the display list compilation, and the drawing, was performed on a range of building complexities. The building complexity combined the degree of floor plan complexity with the maximum number of extrusion steps and served as an indicator for the highest polygon count of a building (worst case). A building complexity of 10, for example, described a building generated from a floor plan with 10 iterations that consisted of 10 floor shapes and 9 extrusion steps. The sample population consisted of one thousand random buildings for each measurement and a range of building complexities from 2 to 20. The generated buildings were positioned to fill the viewport and the tests were conducted with the depth buffer disabled.
Figure 43 shows a comparison graph of time for building generation, display list compilation, and drawing from a pre-compiled display list for various building complexities. The display list compilation time increased slightly with growing building complexity, but remained substantially less than the generation time.

![Graph showing comparison of generation, display list compilation, and drawing time](image)

**Figure 43: Generation, display list compilation and drawing**

Figure 44 shows the display list compilation overhead in relation to the total display list compilation and drawing time. The overhead increased slowly with building complexity to around 40% for a highly complex building.

![Graph showing display list compilation overhead in percent](image)

**Figure 44: Display list compilation overhead in percent**
The result of using display lists is plotted in Figure 45 and indicated a speedup factor of up to 8 times, depending on the complexity of the building.

![Figure 45: Display list drawing speedup factor](image)

The performance measurements results of the OpenGL display lists indicated an overall increase in performance despite the overhead time to compile a display list. Buildings were typically redrawn hundreds of times without being regenerated. The longer display list compilation time was therefore more than compensated for by a speedup in subsequent frames.

### 4.9.13 LRU Performance Characteristics

Tests to measure the overhead of the cache management were also conducted. In this test, the CPU time required by the lru container was measured in two separate scenarios: the time taken to look up an item already cached (cache hit) and the time taken to determine the LRU item and reuse it (cache miss). This involved the measurement of times for a range of cache sizes up to one million, which involved the timing of two different index data structures: std::map, and the hash-table based std::hash map included with GCC Version 3.2.

Figures 46 and 47 show that the time required by a cache query is several orders of magnitude less than the time required to display the actual item. Depending on the frame coherence of the path that the viewer took through the virtual space, cache misses were relatively infrequent and procedural building generation and display dominated the performance.
The results showed that the \texttt{std::hash} map container had a performance advantage over \texttt{std::map} for extremely large caches, however both containers scaled up to around ten thousand items gracefully. Overall, the caching scheme imposed little CPU load in context with the dynamic procedural generation of a virtual city. The use of a \texttt{std::hash} map was preferable to a \texttt{std::map}, but neither could be expected to impact noticeably on the final frame rate.

### 4.9.14 Undiscovered City Performance

The performance measurement of the Undiscovered City prototype was based on frame rate. Figure 48 shows the performance in fps with respect to the number of buildings displayed. The graph shows that the frame rate varies inversely with the number of buildings as was expected. The screen captures in
figure 42 show that for this prototype roughly 200 – 400 visible building objects are a desirable number. The graph in figure 48 shows that 200 buildings could be displayed at approximately 50 fps and 400 buildings could be displayed at a frame rate of approximately 30 fps. The desirable amount of buildings could therefore be displayed at real-time frame rates.

4.9.15 Limitations
All buildings that are generated show a visual variety of volume, dimension, form, size and texture. However, the building objects were positioned on square lot shapes that were embedded in a street grid. The street grid was based on a formal structure that was visible and reintroduced an element of repetition. The limited set of textures that were used on the facade geometry also reintroduced another element of repetition that became apparent during navigation through the virtual space.

4.9.16 New Questions
- How can a street grid be generated that does not show a repetitive formal structure?
- How can the visual variety of texture be improved?

4.9.17 Summary
This prototype displayed a virtual city space which consisted of generated building objects that were placed on a regular street grid. The prototype used approaches of view frustum filling, dynamic procedural generation and LRU display list caching to improve the real-time performance of the virtual space.
Performance measurements showed that it was significantly faster to redraw a previously generated building object from a display list than to regenerate the same object for every frame. In comparison to the Warehouse prototype, the implementation of the LRU display list allowed the display of a larger number of objects with greater complexity at a higher frame rate.

While the buildings in the virtual city were generated with a variety of form, dimensions, volume, formal structure and texture, the street grid and the limited set of textures reintroduced elements of repetition.
4.10 Prototype 9: Street Grid Examiner

The Warehouse and Undiscovered City prototypes highlighted the importance of how the generated objects were distributed within the virtual space. Despite the generation of objects in a visual variation, both prototypes reintroduced an element of repetition by basing the distribution of the generated objects on a regular and visible formal structure.

The Street Grid Examiner prototype was the result of an investigation on how street grids with an informal structure could be generated with the aim to facilitate the distribution of generated objects within the virtual space. It was envisaged that a more irregular street grid could alleviate the noticeable repetition in distribution within the virtual space that was exhibited in the Warehouse and Undiscovered City prototypes.

4.10.1 Generating Street Grids Based on Integer Grid

Unlike the approach taken in the Warehouse prototype where cells were used as proxies for generated objects, this prototype used an integer grid, which subdivided the terrain into a grid with $x$ and $y$ components. Each intersection on the grid represented a node that could be uniquely identified by an integer $x$ and $y$ coordinate. Figure 49(a) illustrates a small section of a terrain from an orthogonal top view which shows one round node at each grid intersection.

Algorithm 3 was used to convert the $x$ and $y$ integer coordinates of a node into a single integer. Hashing each node in this manner resulted in a ‘pseudo-random’ value per node which was used for the subsequent generation of the street grid and later for the generation of the buildings within lot shapes. A manually configured integer $worldId$ value was also incorporated into the per-node hash to facilitate the generation of street grids for entirely different worlds. Bitwise exclusive-or is denoted as $\oplus$. The illustration in figure 49(b) shows the pseudo random integer values that are associated with each node on the integer grid.

Algorithm 3: Node Hash

Require: $hash$ is an integer hash function

Require: $x$ and $y$ is the integer node position

Require: $worldId$ is the world id integer

return $\text{hash}(x \oplus \text{hash}(worldId \oplus y))$
4.10.2 Street Grid Generation

To generate a street grid, the procedure evaluated nodes for their connections to directly neighbouring nodes to the east and the south. A node could be connected to a neighbour to the east and south, to just one of the neighbours to the east or south, or not be connected to a neighbouring node at all. Nodes could receive connections from the north and the east.

Algorithm 4 determined if a pair of nodes was connected or not connected. The node hashes from Algorithm 3 for every node were combined and hashed into a floating point value between 0.0 and 1.0. The value was compared to a threshold fraction \( w \) that was manually configured to control the probability of node connections. A lower threshold value resulted in sparse node connections, a higher threshold value resulted in dense node connections. The threshold for horizontal and vertical connections could be configured separately.

Algorithm 4: Node Connection Test

Require: \( a \) and \( b \) is the pair of nodes

Require: \( h \) is Algorithm 3

Require: \( w \in [0.0, 1.0] \) is the threshold fraction

if floatHash\((h(a) \otimes h(b)) < w\) then \( a \) and \( b \) are connected
Apart from the hash-based node connections, there were also periodic arterial roads that were always connected to either the south or the east, or to the south and the east nodes. The arterials ensured that there was a basic street structure and that building lot shapes that were generated in a later processing step did not get too large and complicated. Figure 50(a) illustrates a street layout with hash-based connections as thin lines as well as arterial connections as thick lines.

4.10.3 Node Displacement

The generated street grid displayed in figure 50(a) showed similarity to the formal structure encountered in the Warehouse and Undiscovered City prototypes due to the right angled node connections. To arrive at a more informal structure, the node points were randomly translated on the $xy$ plane. A similar node translation approach was used for the generation of stone wall patterns (Miyata, 1990).

Each node was randomly translated by a displacement procedure, which generated a displacement vector in the range of -0.5 and 0.5 for every node based on the node’s integer $xy$ position and a common worldId value. The resulting displacement vector was subsequently used to translate a node in the random direction of the displacement vector on the $xy$ plane. Nodes located on an arterial were not displaced. The randomness of the street grid was controlled via a global scale factor $s$ that scaled the displacement vectors. Figure 50(b) shows a street grid with nodes that are randomly translated.

![Street grid with node connections](image1)

![Street grid with node displacement](image2)

Figure 50: Generated street grids
4.10.4 Street Geometry Generation

The street grid of this prototype consisted of nodes which could be connected to a maximum of four neighbouring nodes. To generate polygons for shading and texturing purposes, paths needed to be traced to expand the connected nodes to street polygons. For path tracing three cases were distinguished: nodes with two, three and four connections. Street polygons for nodes with only one connection (dead end streets) were currently ignored in this prototype. Figure 51 shows the corresponding three polygons for the possible intersection cases.

![Figure 51: Street geometry cases](image)

The approach to generate street geometry used in this prototype produced straight road segments and intersections for every visible node in the viewing area. The street polygon generation for a node started with the identification of directly neighbouring nodes and subsequently stored the directly neighbouring node positions in a `std::vector` data structure. A procedure then traced several paths to facilitate the generation of street polygons. To ensure that the generated intersection street geometry of a node connected seamlessly with the street geometry of neighbouring nodes, the connectivity of neighbouring nodes also needed to be considered.

The street geometry generation process is explained on the following example of a curved segment. The curved segment consists of three nodes that are connected by two straight connections. The originating node \((x_0,y_0)\) is connected to a node in the north \((x_0,y_1)\), and a node to the east \((x_1,y_0)\). The connections between the nodes are illustrated in figure 52(a) as lines in a dark shade. The nodes \((x_0,y_1)\) and \((x_1,y_0)\) are also connected to other nodes and are illustrated as lines in a lighter shade in figure 52(a).
To generate the street polygons for the curved intersection at node \((x_0, y_0)\), the procedure traces two paths. The first path trace starts from the node in the north \((x_0, y_1)\) to the node in the east \((x_1, y_0)\). As mentioned above, the path needs to be extended by two nodes to seamlessly connect the geometry of the intersections in the street network. To create the first extension for the current path the node in the north is evaluated for another connection to another neighbouring node using a clockwise winding rule starting west. This evaluation produces a connection to node \((x_{-1}, y_1)\). The position of \((x_{-1}, y_1)\) is then calculated and stored in a `std::vector` data structure called `path0`. After this step, the procedure adds the position of the nodes \((x_0, y_1), (x_0, y_0)\) and \((x_1, y_0)\) to the `path0` vector. To create the second extension for the node in the east \((x_1, y_0)\) the procedure uses a counter clockwise winding rule, which produces node \((x_1, y_{-1})\). The position of node \((x_1, y_{-1})\) is calculated and added to the `path0` vector. Figure 52(b) shows the resulting trace of `path0` in green.

The second path trace starting from the opposite direction, starting with the node in the east \((x_1, y_0)\) progressing to the node in the north \((x_0, y_1)\). Also this tracing path needs to be extended by two more nodes. The node in the east is therefore evaluated for another connection to a neighbouring node using a clockwise winding rule, which produces node \((x_2, y_0)\), as shown in figure 53(a). The position of the node is calculated and stored in a vector named `path1`. After this step the procedure stores the position of the node in the east \((x_1, y_0)\), the originating node \((x_0, y_0)\) and the node in the north \((x_0, y_1)\) in the `path1` vector. This follows an evaluation of the node in the north \((x_0, y_1)\) for a connection to another node using a counter-clockwise winding rule. This evaluation reveals node \((x_1, y_1)\), which follows the calculation of the nodes position and its storage in the `path1` vector. Figure 53(a) shows the resulting trace of `path1`. In the case that no connection
can be found for the extension nodes on either of the ends of the street segment, the procedure generates and stores an artificial connection to a virtual neighbouring node.

The path tracing as discussed above generates two paths consisting of nodes. To generate street polygons, the two paths need to be offset from their current position by half the road width in opposite directions. The offset procedure uses ray casting to calculate the intersection of connection edges between nodes to calculate new offset vertex positions. Implementation details on ray casting algorithms can be found in Dunn and Parberry (2002). The procedure returns three vertices derived from the node positions in each trace path. Figure 53(b) shows the six new offset vertices derived from the two trace paths.

![Figure 53: Second path trace and offset vertices](image)

The offset vertices that are generated for the extension nodes are discarded as they were only needed to calculate the offset vertices for the nodes \((x_0, y_1)\) and \((x_1, y_0)\). The street polygon generation uses the two offset path vertices to create two street polygons consisting of two quadrilateral shapes. To create the first quadrilateral shape, the procedure uses the vertices 0, 1, 4 and 5. The second shape uses the vertices 1, 2, 3 and 4 to form the outline of the second quadrilateral. Figure 54(a) shows the vertices of the two offset paths numbered in a sequential order.

Texture coordinates for each polygon are generated by determining the intersection type and the shape as well as the orientation of each polygon. The intersection type is determined by the number of node connections. The number of vertices determines the street segment type and its orientation. In the cases of straight road segments and bends the two quadrilateral polygons can either be horizontal or vertical. The direction of each quadrilateral road segment is determined when the street polygon is created. The procedure scales the proportions of a street polygon to unit size and uses the scaled vertex coordinates as mapping coordinates for the texture. The visual result of the textured street polygons is presented in figure 54(b). The path tracing approach is given in algorithm 5.
Algorithm 5: Path Tracing

Require: 2 neighbouring nodes

evaluate first neighbouring node
search clockwise for connection

if connection found then
    store found node in path

if not connection then
    create artificial node
    store artificial node in path

store first neighbouring node
store origin node
store second neighbouring node
evaluate second neighbouring node
search counter clockwise for connection

if connection found then
    store found node in path

if not connection then
    create artificial node
    store artificial node in path
4.10.5 Lot Shape Generation

In this research project, a lot shape was defined as an area that was surrounded by streets on a street grid. Lot shapes were generated recursively in this prototype. The procedure that traced the lot shapes evaluated nodes with connections to other nodes for a valid trace start position. Valid start positions were defined as nodes with a connection to a node in the east and south. The tracing procedure assumed the starting node was the west-most and north-most ‘corner’ of the lot shape. If there were several candidates in the west, the north-most position was used. Figure 55(a) shows the appropriate tracing start position for the given lot polygon with a red circle.

The procedure used two tests when evaluating a node for connections to neighbouring nodes to ensure that the lot polygons were traced from the west-most and north-most starting position so that lot polygons were not retraced from another node. The first test examined each node during the tracing process for its location. If a node was located above the tracing start position, the node was not located in the north-most and west-most position of the lot shape, and the tracing process was terminated. In other words, if the following condition was true, the tracing process was terminated:

\[ x == startX \text{ and } y > startY \]

The second test examined each node during the tracing process for its location to the west of the starting position. In the case that a node point used to start the tracing was not the west-most node in the lot polygon, the tracing process was terminated. In other words, if the following condition was true, the tracing process was terminated:

\[ x < startX \]

To trace the lot shape every node was evaluated for connections to other nodes following a counterclockwise winding rule. The traced nodes were temporarily stored in a stack. Figure 55(b) shows a traced pathway indicated by white arrows ending in a dead end.
A dead end was defined as a connection from one node to another node that terminated inside a larger lot shape. If the procedure encountered a dead end, the last node was removed from the stack and the previous node was re-evaluated for further connections following the counter clockwise winding rule. Figure 56(a) shows the removal of the last trace node and the re-evaluation of the previous node for other connections. This procedure continued until either a new connection was found or all nodes in the trace path were removed. In any case, the search finished when the trace path returned to the initial starting position. Figure 56(b) shows the finished trace and the lot polygon as shaded square. Algorithm 6 describes the lot tracing procedure.
Algorithm 6: Trace Lot Shapes

Require: valid trace start position

\[ \text{status} \leftarrow \text{incomplete} \]

if valid polygon found then

\[ \text{status} \leftarrow \text{polygonfound} \]

if current node pos further north than start then

\[ \text{status} \leftarrow \text{wrongstartpos} \]

if current node pos further west than start then

\[ \text{status} \leftarrow \text{wrongstartpos} \]

while not connection and connections left to search do

if connection found then

\[ \text{stack} \leftarrow \text{push(nodeposition)} \]

\[ \text{status} \leftarrow \text{trace(newnode)} \]

if status is polygonfound or wrongstartpos then

return status

if status is incomplete then

pop stack

return status

4.10.6 Terrain Generation

Procedural terrain generation is well researched in computer graphics and approaches include implementations based on Fractals (Laeuchli, 2001), Height Fields (Ebert et al., 1998, pp. 325 - 340) and Height Maps (Polak, 2003). The terrain generated for the street grid and subsequently the generation of a virtual city was based on a height field, which displaced the nodes on the \( z \)-axis to create a landscape with hills and valleys. The height field was generated by a Perlin noise (Perlin, 1985) based procedure, which generated values between \(-1.0\) and \(1.0\), where negative values indicate a distance below the ground level and positive values a distance above the ground level. The values generated by the noise function were multiplied by a scale factor for the maximum height of the terrain specified by the terrain height parameter to determine the position of the node on the \( z \)-axis.
The calculation of height values was evaluated for each individual node. Figure 57 shows a screen capture of a street grid with nodes that are displaced by the terrain procedure.

**Figure 57: Generated street grid on terrain**

### 4.10.7 Analysis of the Generated Visual Variety

The evaluation of the visual variety was based on a juxtaposition of screen captures that show a generated range of street grids that were generated with different parameter settings for connection threshold and scale value. The juxtaposition allowed a direct comparison between the various generated results. A selected range of visual results is presented in figures 58 and 60 to illustrate the discussion of the findings.

The Street Grid Examiner prototype generated street grids that were characterised by an informal structure. The informal structure of the street grid was made possible by random node connections and the displacement of nodes, which in turn created a visual variety.

As described in section 4.10.2, the threshold for horizontal and vertical connections was configured by the parameters for \( w_v \), \( w_h \) and a scale value \( s \). The three images in figure 58 were generated with different settings for \( w_v \) and \( w_h \) and a scale value \( s \) of zero. Figure 58(a) shows a generated street grid based on low threshold values for \( w_v \) and a high threshold value for \( w_h \). Figure 58(b) shows the result a low threshold value for \( w_h \) and a high threshold value for \( w_v \). Both figures show reasonable road connectivity, however the lot shapes show little visual variation and an extreme bias. Figure 58(c) shows a street grid generated with high threshold values that are similar for \( w_v \), \( w_h \), resulting in a dense, balanced street grid with a good road connectivity and good variation of lot shapes. The size
and distribution of lot shapes are similar to the street grid of many modern cities central business districts such as Melbourne. A satellite image of Melbourne is provided in figure 59 for comparison.

Figure 58: Procedural street grids without displacement

Figure 59: Satellite image of Melbourne, Australia
Street grids that were generated with random node displacement produced visually more varied results as shown in figure 60. The lot shapes of an extremely regular street grid changed substantially and provided every lot with a unique shape if generated using random node displacement as illustrated in figure 60(a). Also the monotony of a biased street grid could be broken up through the use of the node displacement technique as demonstrated in figure 60(b). Figure 60(c) shows that the character of balanced street grids with good road connectivity and good variation of lot shapes can be further altered by randomising the position of nodes. These settings allowed the generation of street grids with layout characteristics similar to old cities, such as Rome. A satellite image of Rome is provided in figure 61 for comparison.

Figure 60: Procedural street grids with displacement

Figure 61: Satellite image of Rome, Italy
4.10.8 Analysis of Real-time Performance
The approach to individually evaluate nodes for their connections to other nodes and the subsequent generation of street geometry and the lot shape was effective as it allowed the generation of a street grid in a designated area anywhere within the virtual space. The view frustum filling procedure used in the Warehouse and Undiscovered City prototype was not implemented as the generated street grid was based on nodes and not proxies as in the previous prototypes. As a result, performance tests were not conducted at this stage of the project.

4.10.9 Limitations
A general limitation of the Street Grid Examiner was its restriction to only street grids and empty lot shapes without buildings. In addition, the prototype was only able to generate street networks that were suitable for city environments that consisted of a grid with high road connectivity. Road networks for areas outside cities could not be generated by this approach.

4.10.10 New Questions
- How can frustum filling be used with a node based street grid?
- How can buildings be generated based on the available lot shapes?

4.10.11 Summary
This prototype demonstrated an approach to generate a textured street grid on a terrain. The generated street grid showed a visual variety which was characterised by an informal structure created by random node connections the subsequent displacement of nodes. The discussed street grid generation approach facilitated the creation of street grids with perpendicular road intersections that showed similarity to the central business district street grid of Melbourne, Australia but also facilitated street grids with layout characteristics similar to old cities, such as Rome. The individual node based approach used in this prototype allowed the dynamic generation of a street grid of any size and for any position within the virtual space in real-time.
4.11 Prototype 10: Neverland Framework

The previous prototypes demonstrated approaches that focussed on various aspects of dynamic procedural generation. This prototype provided a framework, which combined several of the findings of the previous prototypes and managed the generation of the virtual space. The prototype was called Neverland Framework. The framework also facilitated a node based approach for view frustum filling and a node based adaptation of the LRU display list cache discussed in the Undiscovered City prototype.

The Neverland Framework was designed as middleware to generate a three-dimensional virtual space. Other software that provided the narrative, game-play, artificial intelligence and music of an electronic game could interface with the framework. The Neverland Framework could be modified in the source code but certain virtual world generation parameters could also be controlled in real-time by the Neverland Examiner interface. To explore the virtual city generated by the framework, a dedicated program called Neverland Viewer facilitated the full screen free exploration of the virtual space via the mouse and the keyboard.

4.11.1 Framework Structure

The framework combined various components of procedural generation into a system that facilitated the cohesive generation of a virtual space in real-time. Virtual spaces, such as cities, are complex systems with objects that are influenced by other objects. In the Neverland Framework, the terrain of the virtual space influenced by the shape of the street grid, the street grid influenced the shape of lot shapes and each lot shape influenced the geometry of a generated building. To generate a cohesive space some of the information about the space needed to be centralised and the generation managed.

The proposed framework consisted of three major components: view frustum filling, LRU display list caching and geometry generation. A diagram of the framework is illustrated in figure 62. The operation of the framework is explained on the following example of a flight simulator.

A flight simulator, provides the view position and view direction as indicated by step 1. With the view position and view direction, the framework can calculate the visible nodes on an integer grid. The visible nodes on the grid can then be evaluated for already generated content in step 2 by the display list cache. If the geometry for this position has already been generated, it can be redrawn directly from the display list cache as indicated in step 4 thus omitting step 3. If, however, the geometry does not exist in the cache it needs to be generated by the geometry generator shown in step 3. The geometry generator decides which object types need to be generated based on their position on the integer grid.
The generation process included the generation of the terrain, street grid, lot shape and all objects contained within the lot shape in this prototype. During the generation process the objects were drawn on screen and the drawing instructions were stored in a display list, which was managed by the display list cache.

Figure 62: Diagram of the Neverland Framework

4.11.2 Node Based View Frustum Filling

Section 4.6.2. described the approach of view frustum filling used in the Warehouse prototype, which was based on square proxy shapes arranged in loops around the camera’s position. The version of the view frustum filling procedure used in the Neverland Framework, determined visible nodes based on a convex two-dimensional polygon that represented the visible viewing area on the \( xy \) plane. The visible viewing area was limited in size to avoid the rendering of too many objects in the distance, which in turn allowed the display of objects at real-time frame rates and thus alleviated the limitation in scalability that was experienced in the Warehouse and Undiscovered City prototypes. The array of visible nodes was calculated by a scan conversion algorithm based on Heckbert’s Generic Convex Polygon Scan Conversion and Clipping Algorithm (Glassner, 1990, pp. 84-86). An integer grid with a view frustum shape that encapsulated the visible nodes is displayed in figure 63.
4.11.3 Geometry Generation

The geometry generator managed the procedural generation of objects for the virtual space. It was called if the LRU display list cache did not contain a display list for a requested nodeId. The nodeId was then passed on to the geometry generator, which in turn generated the geometry for a lot shape. The geometry generator called on a set of object type generators that used object specific generation rules which were optimised for the generation of only certain object types. The building generator, for example, generated only a visual variety of buildings and could not be used to generate the terrain of the virtual space. While the type of object was constrained by the object type, the final form, dimensions, volume, scale and texture of the object however could not be specified and was not known before the object was actually generated and displayed. Once the object was generated, it was stored as a new display list in the display list cache and could be redrawn from the display list for subsequent frames through calling the nodeId from the LRU cache.

4.11.4 Building Geometry Generation Based on Lot Shape

The street grid generated lot shapes in a visual variety of form. The building generator that was developed for the Undiscovered City prototype, however, could not be used with the lot shapes generated by this prototype, as the generation of the buildings was optimised for a rectangular lot shape. The building objects in this prototype consisted of extruded lot shapes that were textured, by the texturing approach discussed in the Building Examiner prototype.
4.11.5 Building Height Distribution

The height of a building in the Undiscovered City prototype was determined by the building’s id value, which produced buildings in an uncorrelated visual variation of height. The uncorrelated height of the buildings, however, made it impossible to identify areas such as city centres or residential areas. The centre of a city in real-world cities can often be identified by a conglomeration of high-rise buildings that are concentrated on a small area in comparison to a residential area that can be identified by many low-rise buildings that stretch out over a large area on the terrain.

Initial experiments with a single height map that controlled the height of buildings on a terrain produced an active structure that reflected the shape of the height map and could be recognised as such. As a result, building objects and their direct neighbours were too similar in height and the relationship between the buildings was too strong. Figure 64 shows a screen capture of buildings objects with heights determined by a single height map.

![Figure 64: Building height determined by single height map](image)

To increase the visual variety in dimension in regard to the building height between the buildings within the virtual space a procedure was used to divide the buildings into categories before their generation, with the aim to associate different maximum building heights values with each category. The Neverland Framework distinguished between the following three categories: commercial, residential and industrial. The height values associated to the categories were defined as tall for commercial buildings, medium for industrial buildings and small for residential buildings. Maxis’ Sim City series used a similar distinction of categories to determine zones, which ultimately influenced the appearance and the dimensions of a building.
The probability that a building of a particular category was generated could be controlled by weight variables with values in the range of $[0.0, 1.0]$ for the residential, commercial and industrial categories. A low weight value for a category resulted in a low probability that a building of this category will be generated. The pseudo code in algorithm 7 elucidates how the buildings are divided into categories.

Algorithm 7: Determining building category

\[
\text{scalefactor} \leftarrow \frac{1.0}{\text{residentialWeight} + \text{commercialWeight} + \text{industrialWeight}}
\]

\[
\text{lotvalue} \leftarrow \text{float value based on nodeid}
\]

\[
\text{industrialThreshold} \leftarrow (\text{residentialWeight} + \text{commercialWeight}) \times \text{scalefactor}
\]

\[
\text{commercialThreshold} \leftarrow \text{residentialWeight} \times \text{scalefactor}
\]

\[
\text{if } \text{lotvalue} > \text{industrialThreshold} \text{ then}
\]

lot is industrial

\[
\text{else if } \text{lottype} > \text{commercialThreshold} \text{ then}
\]

lot is commercial

\[
\text{else } \text{lot is residential}
\]

The height calculation amongst buildings was based on two height maps that were initialised with different input parameters, which produced values between $[0.0, 1.0]$. The resulting height map value for a building was subsequently multiplied by the maximum building height depending on the building’s category.

4.11.6 Modes of Control

The generation of a virtual city space with the Neverland Framework could be tested within a viewer and an editor that were part of the framework. The Neverland Examiner, provided a user interface for the Neverland Framework, which facilitated the manual manipulation of generation parameters in real-time. Figure 65(a) shows a screen capture of the Neverland Examiner. The Neverland Viewer displayed the virtual city space that was generated by the Neverland Framework in full screen view at real-time frame rates. The movement through the space could be controlled by a user using the mouse and the keyboard, adjusting speed, direction and height. Figure 65(b) shows a screen capture of the Neverland Viewer.
4.11.7 Analysis of the Generated Visual Variety

The Neverland Framework combined the findings of the previous prototypes and the visual variety that could be generated by the street generation approach carried forward. The procedure to categorise building objects and randomise their height added a visual variety of form to the distribution of the building objects within the virtual space as illustrated in figure 65. Figure 65 also shows the visual variety of volume, dimension, form size and texture, generated by the extruded lot shape buildings.

4.11.8 Analysis of Real-time Performance

The framework improved the performance of real-time rendering by implementing a new view frustum filling procedure that was more scalable and efficient than the previous approach used in the Undiscovered City prototype. The framework also made use of the LRU display list cache to reuse already generated objects, which included the generation of the terrain, street grid and building geometries.

A fully textured street network that consisted of about 500 nodes was generated at speeds of 30 fps and 1000 nodes at speeds of 15 fps. The street grid is illustrated in figure 66(a) and 66(c) for 500 and 1000 nodes respectively. The procedural virtual city using these procedural street grids is illustrated in Figure 66(b) and 66(d) for 500 and 1000 nodes respectively.
4.11.9 Limitations

The generated city generated a street grid and buildings in a visual variety. The buildings, however, were not as intricate as the buildings generated in the Undiscovered City prototype. The building generator developed for the Building Examiner could not be used for this prototype because the buildings could only be generated on rectangular lot shapes. The lot shapes generated by the Neverland Framework, however, were not always rectangular. Subsequently, the generated buildings in the Neverland Framework showed less variety in form. Billboards and roof geometry were added to alleviate this problem.

The virtual city currently consists of only street grids and buildings on a terrain. The visual variety of such a city space could be improved by adding object type generators for objects such as cars, pedestrians, mail boxes and fire hydrants, for example. The framework could also be used to generate other environments such as a tropical archipelago, for example, if object type generators were added to generate water and plants that are combined with the generated terrain.

On the technical side there are also some problems regarding the placement of the buildings on the terrain. On very steep hills, for example, the building’s ground floor section was not high enough to cover the inclination of the street and the street level cuts into the mid section of the building.
Moreover, the Neverland Viewer was not implemented with collision detection to prevent the user from navigating through a building. The framework also did not track building changes such as damage or aging.

4.11.10 Summary
This research produced a general purpose framework which was called the Neverland Framework that can be used for the generation of pseudo infinite and varied infinite worlds. The framework demonstrated how three-dimensional virtual spaces that consist of objects that are generated in a visual variety can be generated in real-time using computer software and subsequently answered the second research question. The generation of the virtual space was demonstrated on the example of a virtual city environment. The structure of the framework, however, is flexible and extensible and may be adapted to generate virtual spaces other than cities.

The Neverland Framework prototype combined several of the findings of the previous prototypes. As such, it combined approaches to view frustum filling and LRU display list caching to manage the dynamic procedural generation of a virtual space in real-time. The space demonstrated in this prototype consisted of objects such as a terrain, a street grid, street geometry and buildings that were generated in relationship to each other. The terrain, for example, affected the generation of the street grid. The street grid in turn affected the shape of the areas on which building objects can generated, which in turn influenced the shape of the generated buildings. The division of building objects into categories facilitated the definition of relationships of height between the buildings and allowed the generation of densely populated city centres with a conglomeration of commercial hi-rise buildings or areas of many small residential buildings that stretch out over a large area. More importantly though, all objects were generated in a visual variety, which created a visually rich virtual environment, that did not rely on repetition or the meticulous manual creation by a game artist. The Neverland Framework provided two modes of control. The Neverland Examiner provided a user interface, that allowed users to manipulate the virtual city and observe the changes in real-time and with the Neverland Viewer, users could interactively explore a virtual world that has truly not been discovered.
5 Conclusion

5.1 Introduction

This chapter summarises the research outcomes and contributions of this research project in relation to the research objectives. As a consequence of the findings and experiences collected in this project, recommendations for future research are provided.

5.2 Research Outcomes

This work highlighted that game worlds are increasing in size and complexity. As a result, the construction of such game worlds requires an increasing amount of content. This research work identified and discussed current approaches to create the content for game worlds such as the manual creation process that is generally very time and labour intensive and the approach to fill a game world by reusing objects and textures in a game world, which introduces an element of repetition. Both approaches are currently occurring but have their limitations. It was stated that the approach to hire more artists to compensate for the increase in workload to create the game worlds of future games does not scale well. The repetition of objects within a game world, on the other hand, is often noticeable and may interrupt the illusion of presence experienced by the player.

This research investigated several approaches to dynamically generate game world objects in a visual variety for two reasons. First, to alleviate the negative effect of repetition, that can be experienced if the same objects are encountered in a game world. Second, to reduce the high demands of time and labour involved in the construction of game world objects. To achieve this, the investigation was led by two research questions.

The first research question asked how three-dimensional objects can be generated in a visual variety using computer software. In order to answer the first research question current approaches to game object creation as well as procedural generation approaches were investigated and described in the background chapter. The investigation informed the development of procedures and prototypes that demonstrated the findings on how objects such as crates and buildings can be generated in a visual variety. To this end the investigation focussed on the procedural generation of geometry and texture.
5.2.1 Procedural Generation of Object Geometries in a Visual Variety

The findings of the investigation on how procedural generation can be used to generate object geometries in a visual variety indicated that:

- User parameters should only describe generic attributes of an object such as the dimensions, other attributes controlling the shape and texture should be randomised within clearly defined boundaries.

- Procedural generation based on an object template produces predictable visual results but the visual variation in form is limited to the template.

- Iterative procedural generation based on basic shape elements produces an emergent visual variety in form. The final result cannot be predicted, but controlled on a more abstract level by using the generation parameters.

- Every object type requires a customised generation procedure and a customised set of parameters that control the attributes of the object and allow the manipulation of the object’s shape.

5.2.2 Texture

The findings of the investigation on how procedural generation can be used to generate object textures in a visual variety indicated that:

- Procedural texturing based on a larger tile able material texture from which random snippets are extracted and mapped onto object components, is a less calculation intensive approach to generate textures in a visual variety than the procedural generation of material textures.

- The randomisation of a material’s diffuse colour value for entire object components, such as the wooden laths that make up a crate, creates a visual variety of colour. The visual variety of colour of object components provides objects that are of identical geometrical makeup with a unique character.

- The change of textures in combination with a change in generation settings produces an entirely different new virtual space.
The second research question asked how three-dimensional virtual spaces that consist of objects of a visual variety can be generated in real-time using computer software. To this end the investigation focussed on the distribution of generated objects as well as the real-time performance and developed a framework with a novel dynamic procedural generation approach.

5.2.3 Distribution of Generated Objects in a Visual Variety

The findings of the investigation on how procedural generation can be used to distribute generated objects of a visual variety within a virtual space indicated that:

- All procedures involved in the procedural generation of virtual spaces need to be deterministic. Deterministic procedures can be used to regenerate a space with all its objects and textures. This allows a user to explore a virtual space and return to an already visited location.

- Dynamic procedural generation of objects and textures as they are needed within the confines of a visible view frustum allows the generation of virtual spaces that are of enormous proportions in real-time.

- The layout of a structure that influences the distribution of objects within a virtual space can reintroduce an element of repetition that is characterised by a formal structure. This form of repetition can be avoided by the procedures investigated in this research project, that break up the formal structure or procedures that generate informal structures.

- Relationships between objects within a virtual space can be used to control the visual variety in form of the virtual space itself. Objects can influence the shape and dimensions of other objects in a feed forward process. Objects can also be divided into categories that share certain attributes that can help to break up the linear flow of overarching procedures that control the terrain or the overall height distribution of objects.
5.2.4 Real-time Performance of a Procedurally Generated Virtual Space

The findings of the investigation on how the performance of procedurally generated virtual spaces can be improved and managed indicated that:

- Procedural generation of virtual spaces can be quite calculation intensive. It is therefore recommended to program the dynamic real-time generation of a virtual space with a high performance programming language such as C++ in combination with a graphics library such as OpenGL.

- The real-time procedural generation of a virtual space requires a structured framework approach which manages the generation of objects and provides procedures to control the performance by minimising the generation of objects for subsequent frames.

- In contrast to conventional approaches, the generation of objects that make up a virtual space can be limited to the area that is visible by the viewer. This limitation reduces the number of objects that need to be generated in real-time and makes dynamic real-time generation possible.

- The reuse of already generated and subsequently cached objects is generally faster than regenerating objects for every frame. The use of data structures such as the LRU Display List Cache reduces the high demand of processing and frees up performance to allow real-time frame rates.

5.2.5 Contributions

This research by project endeavoured to investigate how virtual spaces that consist of objects in a visual variety can be procedurally generated in real-time. The main contribution of this research project included:

- A critical review of current game world creation practices.

- An overview of relevant real-time procedural generation approaches.

- An experimental approach to potentially increase the illusion of presence by introducing visual variation of objects to game worlds compared to current game worlds that reuse objects to fill the game world with content.
• An innovative real-time procedural generation approach to generate objects and textures in a visual variety of volume, dimension, form, size, colour and texture.

• An innovative real-time procedural generation approach to generate street grids on a terrain.

• A novel view frustum filling approach to increase the performance of procedural generation by limiting the generation of objects to the area visible by the user.

• A novel LRU Display List Caching data structure to increase the performance of procedural generation by managing the generation, storage and retrieval of display lists to minimise the regeneration of objects.

• An extensible framework structure to manage and control real-time procedural generation of virtual spaces that are extremely large in size and that consist of objects in a visual variety.

5.2.6 Impact
As part of a project in the real-time animation and 3D games programming course at RMIT University, Melbourne, Australia in 2003, David Carlin programmed a game based on the Neverland Framework. The game used the generated city as a location to chase robot drones through the streets with a car. David was able to program his game using the Neverland Framework interface with a few modifications that allowed collision detection based on the road network. The game was exhibited at the Australian Game Developers Conference (AGDC) in 2003 in Melbourne, Australia. Figure 67 shows three screen captures of the game.

![Figure 67: Game project](image)

The Undiscovered City played a crucial role in a theatre production called Lost Babylon, directed by Russell Fewster that was performed for the Adelaide 2006 fringe festival. The play was based on a story by Takeshi Kawamura and was a black comedy/melodrama, exploring film noir, action movie, anime and video game genres. It was set in a theme park where customers play virtual killing games with real targets.
The director, Russell Fewster, used the Undiscovered City as an animated backdrop that interactively responded to the actor’s movements such as walking, running, chasing and hunting. “The Undiscovered City offered a compelling realisation of the virtual world imagined by the playwright. The City in its brilliance and luminescence certainly demonstrated affective qualities of light, colour and shape that initially stunned the actors in rehearsal and later audiences and reviewers in performance.” (Fewster, 2006).

Figure 68: The Lost Babylon theatre play.

5.3 Areas and Directions of Future Research

This subchapter offers recommendations for future researchers who are interested in further investigating procedural generation of virtual spaces.

This research project investigated the generation of crates and buildings in general. The real-world, however, consists of many more object types that appear frequently and exist in a visual variety. The availability of more object generators or the availability of tools that allow artists to manually create objects and use procedural approaches to generate a visual variety could be useful to create virtual worlds that are not only visually more diverse but also help to reduce the time required to create them.
The approach of randomly sampling snippets from a much larger texture produced a visual variety in texture for objects that consist of the same geometry. The texture sampling approach, however still relied on a manually created texture with a relatively large memory footprint. Large material textures and even a visual variety of window states could be generated from smaller textures that are randomly tiled following a Wang tile approach as described in Cohen et al. (2003).

The Undiscovered City prototype highlighted the importance of an object distribution approach that is not based on a formal structure. The Street Grid generation approach that was developed for the Neverland Framework alleviated this problem but was limited to city street grids with high road connectivity. Other approaches to city street generation, such as Parish and Muellers CityEngine (2001) allow the generation of intricate lot shapes and city infrastructures, which subsequently would increase the visual variety of the virtual space.

The performance of the virtual spaces could be further enhanced by using occlusion culling techniques. The frustum filling procedure used in the Neverland Framework could be extended to perform occlusion-based prioritization. Buildings need not be drawn if they are occluded by nearer ones. The temporal coherence of the city could be used to accelerate computation in subsequent frames over and above the use of display list caching.

Real world environments such as cities change over the years. Game worlds usually depict a game world at a certain state over a short period of time that is usually not dependant on environmental effects such as weathering. The visual variety of objects, however is often affected by environmental influences. Approaches to mimic weathering that change the appearance of objects such as (Dorsey, Pedersen, & Hanrahan, 1996) could help to increase the visual variety of objects within a virtual space.

One of the main problems with programming a custom solution is the high amount of time spent on development and the technical knowledge required about the intricacies of a programming language and the libraries. The time to develop a procedure to generate an object such as a building in particular, can require substantial more time than the development of the same object in a 3D modelling and animation package. New approaches need to be investigated on how game artists can control and integrate procedural generation into their modelling workflow, so that a visual variety can be generated from objects that are largely created by the game artist.
5.4 Potential Applications

It was stated at GDC 2005 that: “People love to make their own content.” (Kosak, 2005). The approach to generate objects and virtual spaces procedurally in real-time allows users without modelling and texturing knowledge to generate their own virtual spaces by changing generation parameters. As described in section 5.2.6, Russell Fewster was able to generate a virtual city with the Undiscovered City prototype, without any in depth 3D modelling or texturing knowledge for his theatre production. Similarly the prototypes can be used by other artists and users to change an object though the modification of generation parameters and observe immediate effects on the geometry and texture in real-time as generation parameters are manipulated and subsequently immerse themselves into the virtual space that they have generated.

Procedural art is also an interesting pathway in which this technology can be used. It is precisely at this point of the research project, that the potential of such a dynamic real-time procedural generation system can really be explored.

The general procedural generation approach investigated by this research project could potentially offer an alternative to the need to meticulously hand create the artwork for massively large game worlds. With dynamic procedural generation, players can also generate game levels manually or by tweaking generation parameters. Such generated game levels could provide a new level layout filled with objects that are consistent with the visual style of the game, but that vary visually in geometry and texture.

New generation hardware now allows approaches such as dynamic procedural generation in real-time. Dynamic procedural generation in real-time provides further technical and artistic opportunities for exploration in the context of education, architecture, simulation, entertainment and art.

5.5 Final Conclusions

In summary, this exegesis compiled a snapshot of the current status quo of game world creation within the game development industry. As a result of a critical review of current game world creation practices, a real-time procedural generation approach was investigated. This inquiry used an iterative design approach, which involved the creation of ten prototypes, which were subsequently, analysed and redesigned to progress knowledge. This approach was finally implemented in a software framework that facilitated the real-time procedural generation of virtual spaces.

This research project contributed some weight to the argument, that extremely large game worlds can be created without the need to repeat or meticulously construct every single object, to fill the world
with content. To date, there has been limited research which examined the role of real-time procedural generation of objects in a visual variety or the generation of game worlds that consist of objects in a visual variety in real-time. However, an increasing number of games are currently in developed that use procedural generation approaches for the creation of virtual spaces.

Variation and randomness are key attributes of the real world. New generation of games are produced with an increasingly higher degree of visual realism. A true to life replication of the real-world, however, cannot be achieved without the endless visual variety of objects that can be observed in the real world and without it, the virtual space will always stay a visual representation that is strongly influenced by the designer’s idea of reality.

I hope that the results of this research project serve as starting point for future research in this exciting area and are useful for the creation of next generation three-dimensional virtual worlds, that are not only visually more stimulating but also increase the illusion of presence.
Due to the nature of this project, several of the resources that were drawn on were only available online. When publicly available web pages were used, their URLs are provided. It is often the case that these kinds of references tend to disappear after some time. Nonetheless, it might still be possible to retrieve the reference through a website named the “Internet Archive” (http://www.archive.org).

Ahearn, L. (2001). *3D Game Art f/x and Design*: The Coriolis Group, LLC.


Appendix: Prototype Documentation

MAXScript 3D Crate
The MAXScript 3D Crate script generates a variety of simple crate objects using the scripting language of Autodesk’s 3D Studio Max 4. The script uses box objects to generate a variety of crate geometries and uses arithmetical calculations to determine the placement and the size of each box object. Figure 69 shows an illustration of the user interface with settings and the resulting crate object.

![3D Crate Interface](image)

**Properties**
- **Length**: Length of the crate object
- **Width**: Width of the crate object
- **Height**: Height of the crate object
- **Frame Width**: Width of frame
- **Indentation**: Indentation of laths in relation to frame
- **Lath Count**: Number of laths on each side of the crate
- **Lath Gap**: Gap between each lath
- **Lath Thickness**: Thickness of the lath objects
Appendix: Prototype Documentation

Buttons
Generate Button .......................Generate a crate with the parameter values
Close Button..........................Close the dialog box

Instructions to use the 3D Crate script in 3D Studio Max
1. Start 3D Studio Max
2. Click on the Utilities tab and select MaxScript
3. Click on the run Script Button
4. Browse to the file 3dCrate.ms in the MaxScript folder within the Prototypes folder on the CD
5. Select 3D Crate from the Utilities drop down box
6. Enter values into the parameter dialog boxes
7. Click on the Generate button
8. Click the Close button when finished
**Lattice Examiner**

The Lattice Examiner prototype generates box objects in a visual variety of texture and colour. The prototype provides an interface with generation parameters. Each box object features its own individual wood grain texture. The box objects are intended as building blocks to construct more complicated objects. Figure 70 shows 2 screen shots of the Lattice Examiner with different arrangements of the box objects.

![Figure 70: Lattice examiner](image-url)
Properties
Width ......................................... Width of a single box object
Height ........................................ Height of a single box object
Length ....................................... Length of a single box object
Rows .......................................... Number of rows of box objects
Columns .................................... Number of columns of box objects
Gap .......................................... Gap between each box object
Offset ....................................... Offset of each box object in relation to the neighbouring rows
Tone variance ............................ Difference range in materials diffuse value for each box object
ID .............................................. Different id values change wood grain pattern of all box objects
Randomise Lattice button............. Randomises width, length and id values

Texture options
Wood texture 1 ....................... Rough wood texture
Wood texture 2 ...................... Smooth wood texture

Material properties
By reducing the amount of red, green or blue, the wood texture can be darkened and tinted in a
different colour to simulate the real world varieties of wood and metal.

Diffuse red ......................... Amount of red in the texture
Diffuse green ..................... Amount of green in the texture
Diffuse blue ....................... Amount of blue in the texture
Randomise Material button....... Randomises the RGB values to simulate different materials.

Options
Grid ........................................... Toggles grid
Axes .......................................... Toggles x,y,z axis

Randomise all button............. Randomises properties of box objects and colours
Quit button ............................. Closes the application
Crate Examiner

The Crate Examiner is a prototype to explore crate geometries with individual wood or metal textures. The software can generate crate objects in different sizes and facilitates parameters to change the thickness of laths, the number of laths and uses a procedural texture coordinate generator that applies a unique texture to each lath. Figure 71 shows a screen shot of the Crate Examiner.

Figure 71: Crate Examiner

Properties
- Width......................................... Width of the crate object
- Height  ....................................... Height of the of the crate object
- Length ....................................... Length of the crate object
- Frame thickness ......................... Thickness of the rectangular frame of the box object
- Lath count .................................. Number of laths on each side of the crate
- Gap ............................................ Gap between each lath
- Tone variance ............................ Difference range in tone for each lath and frame wood object
- Crate ID ..................................... ID or number of the crate
- Randomise Crate button ............ Randomises property values to generate a variety of crate geometries
**Texture options**
Wood texture 1 ......................... Rough wood texture
Wood texture 2 ......................... Smooth wood texture
Metal texture 1 .......................... Stained metal texture
Metal texture 2 .......................... Rusty metal texture

**Material properties**
By reducing the amount of red, green or blue, the wood texture can be darkened and tinted in a different colour to simulate the real world varieties of wood and metal.

Diffuse red .................................. Amount of red in the texture
Diffuse green .............................. Amount of green in the texture
Diffuse blue ............................... Amount of blue in the texture
Randomise Material button ........... Randomises the diffuse colours

**Options**
Grid ......................................... Toggles grid
Axes ......................................... Toggles x,y,z axis

Randomise all button ................. Randomises all values except texture options and lights
Quit button ............................. Closes the application
Pile Examiner

The Pile Examiner generates a crate pile that allows the manipulation of parameters to change the form of the pile. Figure 72 shows a screen shot of the Pile Examiner.

**Properties**
- **Pile Width**: Width of the pile
- **Pile Height**: Height of the pile
- **Pile Length**: Length of the pile
- **Object Gap**: Gap between each pile column
- **Width Objects**: Number of crate objects in width
- **Height Objects**: Number of crate objects in height
- **Length Objects**: Number of crate objects in length
- **Angle Jitter**: Range by which each object can be randomly rotated
- **Render Button**: Apply settings to pile object
Options
Environment............................Shows other piles located within the view frustum

Textured ...............................Textures all visible piles

Quit button ............................Closes the application
**Warehouse**

The Warehouse prototype shows a virtual space filled with crate objects of the same geometry. The crate geometries are arranged on top of each other to create piles. The Warehouse can be freely explored by using the mouse. Figure 73 shows a screen shot of the Warehouse.

![Figure 73: Warehouse](image)

**Warehouse navigation**

- **Left Mouse Button** ............... Move forward
- **Right Mouse Button** ............... Move backward
- **Centre Mouse Button** ............. Rotate without movement
- **Left Mouse Movement** ............. Move left if one of the mouse buttons is pressed
- **Left Mouse Movement** ............. Move right if one of the mouse buttons is pressed

**Warehouse keyboard short cuts**

- **m** ................................ Uniform / Variation toggle
- **l** ................................ Toggle light
- **n** ................................ Decrease loops
- **N** ................................ Increase loops
- **s** ................................ Solid / wireframe toggle
Warehouse keyboard short cuts continued
f ................................................. Fog toggle

t .................................................. Texture toggle

TAB ........................................... Toggle full screen view

F9 .............................................. Save screen shot in png format in the same directory as the prototype

ESC or q ............................... Close the prototype
Building Examiner

The Building Examiner is a prototype to browse building geometries used in the Undiscovered City prototype. The prototype facilitates parameters to change the building’s dimensions and complexity of geometry. Several building geometries can be generated by changing the building’s $id$ value. Four textures can be selected manually or at random. The texture coordinates are automatically generated to cover various building dimensions. Figure 74 shows a screen shot of the Building Examiner.

![Building Examiner](image)

**Figure 74: Building Examiner**

**Building Options**
- **Width**: Width of the building
- **Height**: Height of the building
- **Depth**: Depth of the building
- **Floor Height**: Floor height of a single floor in the building’s mid section
- **Setbacks**: Number of steps or indentations in the building
- **Complexity**: Number of basic shapes combined to generate the final floor plan
- **ID**: ID number used to generate random number to generate the building
Building Position
X-Axis ................................. Numerical x value to determine x-axis position
Y-Axis ................................. Numerical y value to determine y-axis position
Z-Axis ................................. Numerical z value to determine z-axis position
X-Rotation .......................... Numerical rotation value around x-axis
Y-Rotation .......................... Numerical rotation value around y-axis

Texture Sets
Random Texture ................... Generator selects texture for every generated building
Texture Set 1 ....................... First set of building textures
Texture Set 2 ....................... Second set of building textures
Texture Set 3 ....................... Third set of building textures
Texture Set 4 ....................... Fourth set of building textures

Options
Grid ................................. Toggles grid
Axes ................................. Toggles x, y, z axis
Quit button ....................... Closes the application
Undiscovered City

The Undiscovered City prototype is a dynamically procedurally generated virtual city which can be interactively explored from a first person perspective in real-time. All geometrical components of the city are dynamically generated as they are encountered by the user. The shape of a building is determined by its location in the virtual space. If the user returns to a particular location, the same buildings will be regenerated. Buildings and streets which are located in the view frustum are generated and stored in memory to avoid unnecessary regeneration. Buildings that drop out of the view frustum are deleted and the memory is reclaimed. The amount of information stored in memory remains roughly constant, even though the virtual city has no apparent A screen shot of the Undiscovered City is illustrated in figure 75.

![Figure 75: The Undiscovered City](image)

Undiscovered City navigation

- Left Mouse Button .................... Move forward
- Right Mouse Button .................. Move backward
- Page up key ............................... Move up
- Page down key ............................ Move down
**Undiscovered City keyboard short cuts**

L ................................. Increase number of buildings
l ................................. Decrease number of buildings

W ................................. Increase street width
w ................................. Decrease street width

C ................................. Increase city ID value
c ................................. Decrease city ID value

s .................................. Toggle solid and wireframe view
f .................................. Toggle fog
t .................................. Toggle texturing

TAB .............................. Toggle full screen view

F9 .................................. Save screen shot in png format in the same directory as the prototype

ESC .............................. Close the application
Street Grid Examiner

The Street Grid Examiner showcases the flexibility and variety of a dynamically generated city street grid which can be examined via a user interface in real-time. The street grid can be generated for any position in the virtual space. Figure 76 shows a screen capture of the Street Grid Examiner.

Figure 76: Street Grid Examiner

Street Grid Properties

ID ..............................................ID value of the street grid
Attraction Horizontal ................Horizontal attraction value
Attraction Vertical .....................Vertical attraction value
ArterialSize ...............................Size of a street grid quadrant enclosed by arterials
Displacement Jitter....................Random displacement of node points on the x,y plane
Terrain Height .........................Maximum height of terrain

Navigation Options

x -Position .............................Centre street grid position on x-axis
y-Position ...............................Centre street grid position on y-axis
Width......................................Width of examined street grid
Depth......................................Depth of examined street grid

Quit button .............................Closes the application
Neverland Examiner

The Neverland Examiner provides a user interface to manipulate generation parameters used by the Neverland framework to generate the city street grid, the terrain and the buildings in real-time. The prototype displays the street grid and buildings within the confines of the view frustum. Figure 77 shows a screen shot of the Neverland Examiner.

Figure 77: Neverland Examiner

Neverland Settings
World ID .............................. ID value of the Neverland virtual space
Cell Width .......................... Width of a lot made up of four node segments
Road Width .......................... Width of all road segments
Min Building Height .............. Minimum height of all buildings
Max Building Height .............. Maximum height of all buildings
Max Residential Bld Height ...... Maximum height of residential building as fraction
Max Commercial Bld Height ...... Maximum height of commercial building as fraction
Neverland Examiner settings continued
Max Industrial Bld Height......Maximum height of industrial building as fraction
Residential Bld Weight.........Chance of occurring residential building
Commercial Bld Weight.........Chance of occurring commercial building
Industrial Bld Weight..........Chance of occurring industrial building

Terrain Properties
Terrain Height......................Maximum height of terrain

Road Properties
Horizontal..........................Horizontal attraction between nodes
Vertical..............................Vertical attraction between nodes
Arterial Size.......................Number of nodes that are enclosed by arterials
Displacement Jitter...............Random displacement of node points on the x,y plane

FOV Properties
x -Position .........................Centre street grid position on x-axis
y-Position ..........................Centre street grid position on y-axis
Arc angle ............................Angle of viewing area
Direction angle.....................Direction angle of viewing area
Viewing distance ..................Size of viewing area
FOV segments......................Smoothness of field of view

Options
Grid ..................................Toggle for grid
Axes ..................................Toggle for axes
Lot outlines..........................Toggle for lot shape outlines
Buildings ............................Toggle for buildings
Street Path ..........................Toggle for street path
Streets ..............................Toggle for textured street polygons
**Neverland Examiner options continued**

SkySphere ................................. Toggle for sky sphere

Building Types ......................... Toggle for building types

Display List Cache ...................... Toggle for display list cache

Flush Cache Button ..................... Toggle for empty the cache

Quit button .............................. Closes the application
Neverland Viewer

This prototype shows the virtual space generated by the Neverland framework in full screen view. The virtual city is pseudo infinite and can be freely explored from a first person perspective using the mouse and the keyboard. Figure 78 shows a screen shot of the Neverland Viewer.

Figure 78: Neverland Viewer

Neverland Viewer Navigation

At the beginning of the prototype the viewer is in cruise mode following one direction at a set altitude. To take control, press the SPACEBAR. Increase velocity with the ‘w’ key and decrease velocity with the ‘s’ key. View other options by pressing the ‘h’ key.

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPACEBAR</td>
<td>Toggles cruise mode</td>
</tr>
<tr>
<td>Left Mouse</td>
<td>Move forward</td>
</tr>
<tr>
<td>Right Mouse Button</td>
<td>Stop</td>
</tr>
<tr>
<td>Move mouse</td>
<td>Move up, down, left and right</td>
</tr>
<tr>
<td>w</td>
<td>Increase velocity</td>
</tr>
<tr>
<td>s</td>
<td>Decrease velocity</td>
</tr>
</tbody>
</table>
Neverland Viewer navigation continued

1................................. Toggle building lots

f ................................. Turns fog off

F .............................................. Turns fog on

r ................................. Toggle roads

b ...................................... Toggle buildings

v .......................................... Switch views

TAB.................................... Toggle full screen view

F9 ........................................ Save screen shot in png format in the same directory as the prototype

ESC or q ............................... Close the application