Evolving Pixel Shaders for the Prototype Video Game Subversion

Andrew Howlett[1], **Simon Colton**[1] **and Cameron Browne**[1]

Abstract. Pixel shaders can be used to create a variety of visual effects in 3D environments, far more efficiently than if produced using the standard graphics pipeline. For such efficiency reasons, pixel shaders are commonly used in video game rendering, to add artistic or other visual effects. We investigate the automated creation of novel shader programs for rendering scenes in the Subversion virtual game world, with a view to providing the player with a visually richer and more diverse 3D environment. We show how shader programs based on the OpenGL shading language may be represented in a hierarchical tree form. This representation admits an evolutionary approach to shader creation, and we show how the application of genetic programming techniques can lead to the evolution of new and interesting shaders. We harness this for an approach where the user supplies details of a fitness function for the overall look of the city environment. We experimented with a number of different fitness function setups in order to produce some preliminary results about this approach. While generally successful in the creation of novel and visually interesting shading effects with little effort, we find some drawbacks to the approach and suggest methods for improvement.

1 INTRODUCTION

Subversion (www.introversion.co.uk/subversion) is a video game prototype being developed by Introversion Software Ltd. that takes place in a virtual 3D world. The environment features procedurally generated landscapes and cityscapes produced by a custom 3D generation engine. The Subversion engine generates terrain and cities on the fly and allows the user to custom-design their own buildings [1] to increase their enjoyment of and engagement with the game, and to make the 3D environment richer and more diverse. The city models created by the software adhere to a single – slightly retro – style by default, as seen in the screenshot of an example city provided in figure 1. Hence, another way in which users may customise the game world is to specify preferred rendering schemes for the procedurally generated world content, in order to enhance the visual presentation of the game world and possibly to make the game experience more immersive. In addition, such rendering schemes may be applied automatically by the game engine in order to reflect certain moods and emotion, for example, forests my be predominantly green in hue, oceans blue, deserts yellow, or action scenes may be tinged with red to foster a sense of menace and tension.

[1] Computational Creativity Group, Department of Computing, Imperial College, London. www.doc.ic.ac.uk/ccg

The elements in a virtual Subversion world are defined by 3D polygons, which are broken down into triangles and passed to the machine's graphics card for rendering. This allows vertex shaders and pixel shaders to provide a convenient way to apply custom rendering schemes on the fly [2]. A vertex shader is a program that operates upon the geometry of the triangle vertices passed to the graphics pipeline, while a pixel shader is a program for performing per-pixel rendering of a scene based upon this underlying geometry. Both are typically optimised for and executed on the machine's graphics processing unit (GPU) to maximise performance. As Subversion already handles the vertex geometry, we were interested in customising the pixel shading program to apply custom rendering schemes to the resulting cities. We aimed to determine a viable and convenient method for users to define custom colouring schemes in which the procedurally generated Subversion game content may be rendered. Ideally, users should be able to define shaders easily and with no technical knowledge of the shading language used.

One approach to enabling users to specify aspects of the visual look and feel of the game is to allow them to specify a fitness function, and evolve pixel shaders in order to produce cityscapes rendered to look approximately like the user specified. For instance, if the user specifies a reddish hue with highly saturated (vibrant) colours, this can be expressed as a fitness function for an evolutionary search for pixel shaders. We investigate the plausibility of such an approach here. In order to do this, we specify a representation of pixel shaders in a hierarchical tree format, as described in the next section. In the following section, we describe how an evolutionary search can be set up, by detailing the evolutionary operators we employ and the nature of fitness functions which drive the search for pixel shaders. In the experimentation section, we describe 21 short sessions with the evolutionary tool, ranging over six different fitness function setups. We demonstrate the feasibility of evolving pixel shaders, and we conclude by putting this work into the context of related work and suggesting future directions.

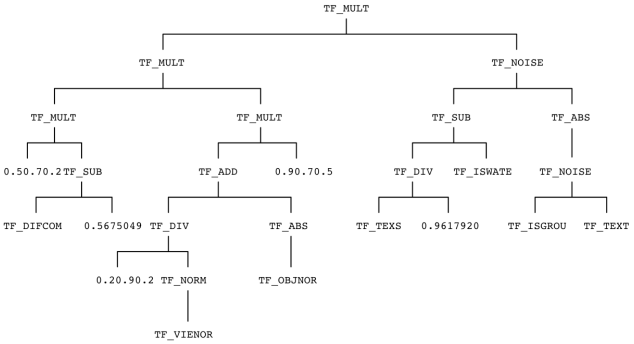
Figure 1. Example city generated by a standard Subversion rendering (taken from www.introversion.co.uk/subversion).

2 REPRESENTING PIXEL SHADERS

We chose to implement our pixel shaders in the OpenGL Shading Language (GLSL). GLSL is a high-level shading language based on the C programming language, and is probably the most prevalent shading language for these purposes [3]. This, together with the fact that Subversion is optimised for OpenGL, made GLSL an obvious choice. GLSL applications do not operate in isolation but in conjunction with OpenGL, which provides an API extension for pixel shaders. Each custom shader program is represented as a tree of operators, with child nodes acting as operands of their parent nodes. The output from each tree after normalisation is an RGBA colour value which describes how each pixel in the user's viewport is to be rendered. Each leaf node represents an attribute of the pixel to be shaded, or a constant 4-vector. The attributes can either refer to a 3D co-ordinate (in which case the fourth vector value is zero), a normal to a plane or a lighting component of the pixel. Whenever a floating point number (attribute or constant) is required to be a vector, it is converted by duplicating the floating point in all four components (with Booleans similarly handled as duplicated ones or zeros). All possible pixel attributes are described in table 1. Note that TEXS and TEXT together define the approximate position on a wall, roof etc., that the pixel corresponds to. Note also that the NodeGrid is an internal data structure for representing buildings used by Subversion (see [1] for details).

Leaf node value	Type	Description
ISSTRE	Boolean	Checks whether the pixel is part of a road
ISBUIL	Boolean	Checks whether the pixel is part of a building wall
ISROOF	Boolean	Checks whether the pixel is part of a building roof
ISGROU	Boolean	Checks whether the pixel is part of the ground
ISWATE	Boolean	Checks whether the pixel is part of a river
OBJPOS	Co-ordinate	The 3D world (city) co-ordinates that correspond to the pixel
VIENOR	Direction	The normal of the polygonal face to which this pixel belongs, in user view space
OBJNOR	Direction	The normal of the polygonal face to which this pixel belongs, in world (city) space
SPECOM	Lighting component	The pixel's specular lighting component
DIFCOM	Lighting component	The pixel's diffuse lighting component
TEXS	Co-ordinate	The s-texture co-ordinate for the pixel
TEXT	Co-ordinate	The t-texture co-ordinate for the pixel
NGRAND	Float vector	Per-NodeGrid random value for the pixel
PROPOS	Co-ordinate	The pixel's on-screen position
VIEPOS	Co-ordinate	The pixel's position in user view space

Table 1. Pixel attributes, each of which is a four-dimensional vector. Boolean vectors are either entirely ones or zeros, and are generated by a conditional acting on the pixel attributes. Co-ordinate vectors describe 3D co-ordinates of the pixel either in the world view, or the user view, and can be relative to building walls, etc. Direction vectors represent normals to planes in the world view. Lighting vectors represent specular or diffuse lighting components for rendering.




```
(((vec4(0.549183,0.743805,0.162998,0.174860))*((difcom)-(0.567505)))*
((((vec4(0.152538,0.881744,0.169250,0.803552))/(normalize(vec4(vienor,1.0))))+
(abs(vec4(objnor,1.0))))*(vec4(0.902191,0.735901,0.488169,0.825391))))*
(texture2D(Noise,vec2(length(((gl_TexCoord[0].s)/(0.961792))-
((gl_TexCoord[0].p>0.975&&gl_TexCoord[0].p<0.985?0.5:0.0))),
length(abs(texture2D(Noise,vec2(length((gl_TexCoord[0].p>0.985&&
gl_TexCoord[0].p<0.995?0.5:0.0)),length(gl_TexCoord[0].t))))))))
```

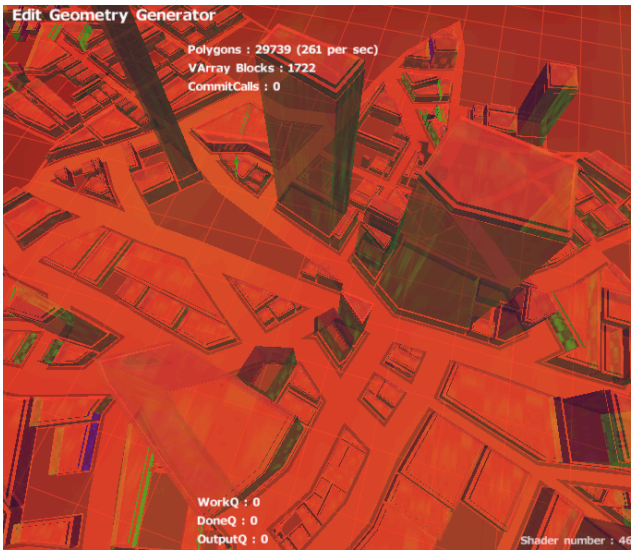


Figure 2. An example shader tree, compiled script and resulting rendering by the pixel shader. Note that some of the numerical details are missing from the tree.

Branch nodes are mostly arithmetical functions, including ABS, MULTIPLY, DIVIDE, ADD and SUBTRACT, and act on the vectors in the obvious ways. There is also a NOISE node, which produces its output using its inputs as coordinates to lookup an area of a texture. The texture was manually created in Photoshop, using the clouds filter. Each tree is compiled to a pixel shader script or program in GLSL format that is wrapped between appropriate header and footer text, then passed to the graphics pipeline to direct the per-pixel aspects of the rendering of the scene at runtime. As an example, the tree shown in figure 1 generates the GLSL script shown below it in the figure. The image at the bottom of the figure shows the final result of rendering a scene from the Subversion world with this shader program.

3 SHADER EVOLUTION

Genetic programming (GP) involves the evolutionary optimisation/generation of computer programs [4] via the simulation of genetic operators such as crossover and mutation, and the simulation of genetic drivers, such as survival of the fittest, tournament selection, etc. We use a standard GP approach for evolving shader programs. The process starts with a pool of N candidate shader trees that are generated randomly in a straightforward way by iteratively growing the trees. Then a particular scene is rendered with each shader, so that the shader can be evaluated according to a user-defined fitness function, as described below. Individuals are then chosen for an intermediate population using roulette selection, hence potentially copied a number of times according to their relative fitness. Pairs of (possibly identical) parents are then randomly selected from this intermediate population and either copied, crossed over or mutated, as described below. Each such mating produces two children that are added to the next generation, until the required number of children are created. The process then repeats. In the experiments described below, the evolutionary process runs for a total of 10 generations by default, and each population comprises 60 individuals by default.

Standard one-point crossover is performed between each pair of parents, in which a subtree is randomly chosen from each parent and the two subtrees swapped. We also implemented an additional form of crossover in which a new tree is created with a single 'if' node at its root and the two parents attached as the true and false branches to create a new child program. This clearly has the potential to lead to program bloat, and we need to perform further analysis to determine the extent of this problem. Crossover is made with a 15% chance per node. We implemented two forms of mutation: subtree and data mutation, each with a 70% chance per node. In subtree mutation, a subtree of the parent is randomly selected and replaced with a randomly generated tree with depth proportional to the depth of the subtree being deleted. This is essentially a crossover with a randomly generated subtree. In data mutation, floats and vectors may be assigned random values, vectors may have their components swapped or zeroed, or randomly selected variable nodes may be replaced with function nodes (for which the original variable node is attached as an operand). It was found that the last step was useful for encouraging shaders to produce noticeably different results in the presence of certain variables. For example, the object normal (OBJNOR) would typically produce results from a very limited palette unless this step was performed at least some of the time. This tree modification process was also applied a number of times to all trees in the initial generation, for the same reason – random tree node generation tended to construct trees that would appear similar if they used a similar set of variables.

Figure 3 shows the screenshot of an evolutionary session in action. The user may stop the evolutionary process at any point, and may view and/or export any tree, corresponding shader program or rendered scene throughout the process. In the figure, the user has chosen to view the rendered output from four candidates of each generation as a way of keeping an eye on progress. Each generation can be saved for later reload.

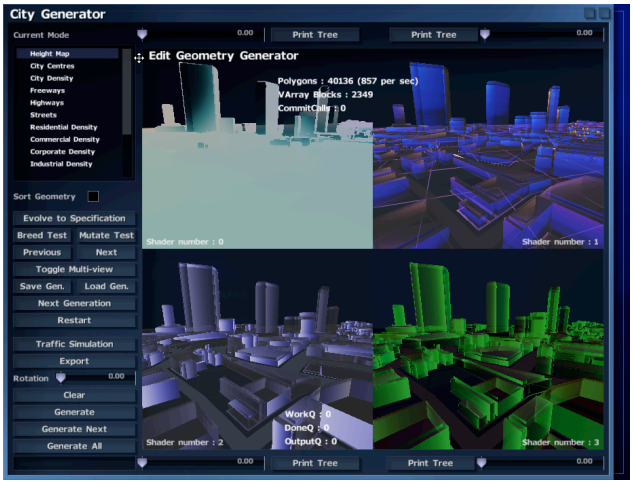


Figure 3: Screenshot of a session with the evolutionary system.

The user specifies a preferred fitness function using the parameters shown in figure 4. The main controls are three sliders indicating the user's preferred hue, saturation and luminance in the HSL colour space [5] and the relative importance of each component, all in the range 0 to 1. The user may also choose to either penalise or reward the presence of noise and textures in the shading algorithm (although we do not use these parameters in the experiments described below).

The fitness function may be applied in two ways: per-pixel or by blocks. In per-pixel mode, one pixel in four of the rendered image is sampled and the average hue, saturation and luminance values found over all sampled pixels is calculated. Then, the shader's fitness is defined as the maximum discrepancy between any of these three average values and the corresponding values specified by the user, modulated by importance. In block mode, a similar measurement is made but instead of measuring individual pixels, the image is segmented into blocks and the hue, saturation and luminance values most characteristic of each block are averaged. The hue sliders may be adjusted at run time throughout the evolutionary process, allowing the user to interact with the system and drive the creative process more directly. If the user does not adjust the sliders, then the process continues automatically to seek shaders that most closely resemble the user's specified hue requirements.

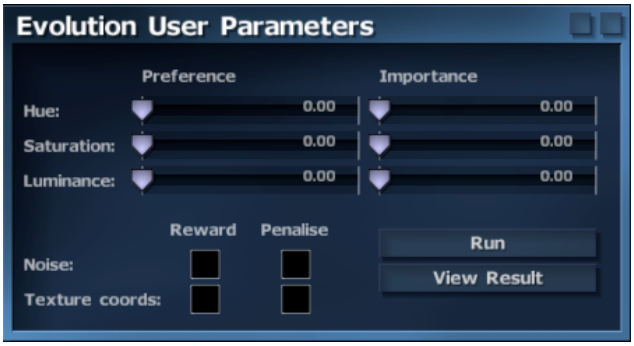


Figure 4. User interface for specifying the fitness function.

4 INITIAL EXPERIMENTAL RESULTS

In order to begin to determine the plausibility of evolving pixel shaders to the user's specification of a fitness function, we have undertaken some preliminary experiments. Bearing in mind that the long-term application of this approach will be to allow players of Subversion to describe the look and feel of the cities that they play the game in, we note that long evolutionary times are not appropriate. Hence, we restricted our sessions with the evolutionary system to produce just 10 evolved generations, in addition to the initial randomly generated one. The first question to ask with evolutionary approaches is whether the evolution actually has an impact, i.e., we wanted to assess whether any improvement over random generation of shaders is possible in such short evolutionary sessions. Moreover, to restrict the experimental conditions, our experiments here concentrate on achieving particular colours, specified by the Hue, Saturation and Luminance of the colour (HSL). The first three fitness function setups were therefore aimed at producing particularly coloured pixel shaders, and were varied by changing the weighting of the H, S and L values, as per table 2. We speculate that the Hue value is less likely to be used in practice than the saturation and luminance values. This is because changes in saturation and brightness of scenes is more likely to affect a change in mood than the hue of the scene. For instance, a dark scene (whether dark blue, red or any other colour) is likely to produce a more sinister mood than a light scene with highly saturated colours, which is more likely to produce a carnival feel to the city. With this speculation in mind, for fitness setups 4, 5 and 6 in table 2, we fixed the hue weighting to zero, so that it did not guide the search.

S	Hue (v/w)		Saturation (v/w)		Luminance (v/w)	
1	30.21	0.50	0.81	0.50	0.40	0.50
2	30.21	0.55	0.81	0.22	0.40	0.22
3	210.0	0.55	0.75	0.22	0.75	0.22
4	0.00	0.00	0.65	0.50	0.82	0.50
5	0.00	0.00	0.85	0.50	0.20	0.50
6	0.00	0.00	0.20	0.50	0.50	0.50

Table 2. Fitness function (S)etups for evolutionary sessions. Both the (v)alue and the (w)eighting are shown.

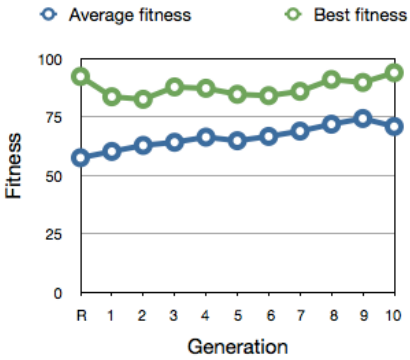


Figure 5. Average and best fitness of individuals in the populations, starting with the randomly generated population (R), and for 10 successively evolved generations. This figure depicts run Q from table 2, with fitness setup 5 from table 1.

In total, we ran 21 sessions to evolve pixel shaders according to the six fitness function prescribed in table 2. For each session, we fixed the evolutionary parameters. In particular, we specified a session of 10 generations, with a population size of 60 individuals. The crossover and mutation rates were as described in the previous section. In figure 5, we portray how the average and the best fitness of individuals in each population changed from generation to generation for a representative session. The results from the 21 sessions are presented in table 3, with summary statistics corresponding to the six fitness functions presented in table 4. We analyse these results in the conclusions section.

Run	S	Rand	Av. last	Av. bests	Best (gen)
A	1	67.0	79.2	90.7	93.5 (9)
B	1	79.1	76.8	92.2	93.7 (6)
C	1	62.8	72.3	87.1	88.8 (6)
D	1	72.3	69.6	85.8	87.5 (0)
E	1	68.4	72.5	87.7	89.3 (5)
F	1	60.7	71.6	84.5	87.5 (0)
G	2	63.8	74.0	88.7	93.6 (1)
H	2	74.1	71.6	85.8	88.6 (6)
I	2	71.4	69.6	81.9	84.2 (0)
J	3	65.7	70.8	85.3	87.5 (4)
K	3	68.3	71.3	88.2	93.6 (7)
L	4	62.6	76.2	90.4	94.7 (7)
M	4	77.9	71.8	92.6	96.2 (3)
N	4	71.3	78.4	91.8	94.5 (9)
O	4	77.6	78.1	89.5	92.9 (0)
P	4	65.1	71.0	87.4	91.6 (0)
Q	5	57.5	70.9	87.5	93.9 (10)
R	5	60.7	72.8	90.9	93.7 (5)
S	6	74.5	80.0	93.1	95.3 (7)
T	6	79.8	82.8	94.8	95.9 (1)
U	6	82.8	78.3	95.1	96.8 (3)
Av.		**69.7**	**74.3**	**89.1**	**92.0 (4.2)**

Table 3. Average fitness of individuals in the initial random generation (Rand); average fitness of individuals in the final generation (Av. last); average of the fitnesses of the best individuals in each generation (Av. bests); fittest individual in any generation, and the generation where it was seen (Best (gen)) for six different fitness functions (S)etups over 21 runs of the evolutionary system.

S	Rand	Av. last	Av. bests	Best (gen)
1	68.4	73.7	88.0	90.1 (4.3)
2	69.8	71.7	85.5	88.8 (2.3)
3	67.0	71.1	86.8	90.6 (5.5)
4	70.9	75.1	90.3	94.0 (3.8)
5	59.1	71.9	89.2	93.8 (7.5)
6	79.0	80.4	94.3	96.0 (3.7)
1+2+3	68.5	72.7	87.1	89.8 (4)
4+5+6	70.98	76.03	91.31	94.6 (4.5)
All	69.7	74.3	89.1	92.0 (4.2)

Table 4. Summary statistics for each of the fitness function (S)etups (and collections of setups), averaged over all the runs for each setup (or collection).

5 ILLUSTRATIVE PIXEL SHADERS

Figure 6 portrays the rendering of cities afforded by four example pixel shaders that were evolved during the sessions described above. They are provided as an indication of the kinds of outputs achievable by the system.

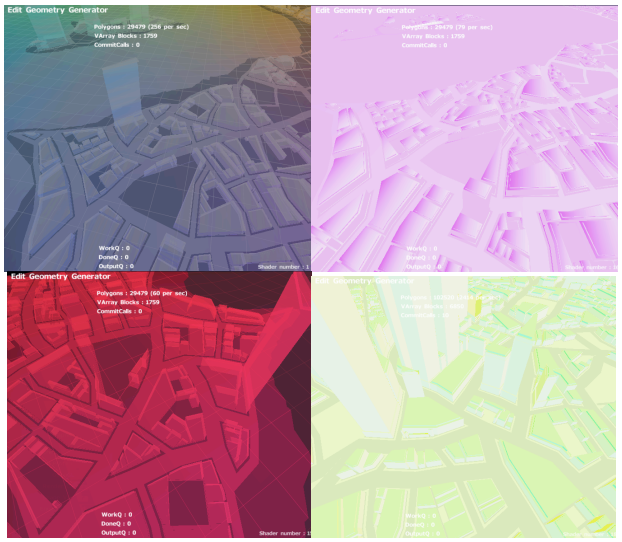


Figure 6. Four examples of evolved pixel shaders.

6 RELATED WORK

With the exception of games based on notions from A-life, notably games in the so-called 'God-genre' such as Black and White, evolutionary methods are rarely employed in commercial video games. This may be because of the long time usually required by the evolutionary processes (and a general tendency in the industry to avoid techniques that may be unpredictable). Within the realm of AI research, evolutionary techniques have been experimented with for the following purposes:

(a) Novel content generation. For instance, in [1], the authors describe a user-driven approach to generating buildings (also for the Subversion environment). A genetic algorithms approach to conceptual building design is presented in [6], where the emphasis is on decision support systems to assist designers. Moreover, in [7], the authors evolve weapons for the Galactic Arms Race game, and in [8], entire levels are evolved for the game Mario.

(b) Novel presentation of existing content. These applications are the most relevant to the work presented here. The idea to use genetic programming to evolve shader programs was initially suggested by [9] and has been successfully demonstrated by various authors. For instance, the approach described in [10] is user-driven, i.e., the user acts as a fitness function during the session by choosing the pixel shaders they prefer, rather than by specifying a fitness function in advance. Our approach also differs from their work in terms of the hierarchical tree-based representation of shader programs that we use. The shader representation they use comprises a (possibly variable length) set of integers which are mapped via a lookup table to commands for NVIDIA's high level Cg shader language, and arguments for the commands. Earlier work on evolving shaders include [11], which used a fixed length representation of shaders which mapped to the assembly language of the hardware they ran the system on. We are not aware of any other approach where the primary target of GPU-based pixel shader evolution has been a video game world.

(c) Controlling of avatars for 1-player modes. For instance, in [12], the authors evolved behaviour trees to control an avatar that played the nuclear war simulation game DEFCON, and in [13], the authors similarly evolved players for Quake 3.

(d) Controlling non-player characters (NPCs). For instance, in [14], the authors evolve psychosocial behaviours for NPCs, and in [15], the authors use evolutionary learning for adaptive game intelligence.

Given the processing power of Graphical Processing Units, various researchers have attempted to use them for efficiently evolving software. For instance, in [16], the authors evolve a vision system for object recognition on a GPU. There is an extensive bibliography on genetic programming for GPUs at www.gpgpgpu.com, and in [17], the authors give a good overview of the efficient evolution of programs on GPUs. In [18], the authors describe a parallel evolutionary implementation on a consumer graphics card, and show its value in achieving up to five times speed up with tests using vertex shaders.

7 CONCLUSIONS AND FUTURE WORK

The experimentation presented above is quite preliminary in nature, and more work is needed for us to be able to draw any general conclusions about the plausibility of evolving pixel shaders to user-required fitness considerations. The graph in figure 5 is typical of the progress of the evolutionary approach that we observed over the sessions – slow but steady nearly-linear increase in average fitness over the populations. From table 3, we see that the average fitness of the individuals in the tenth and final population was higher than the randomly generated population in 16 of the 21 sessions, with average fitness increase during a session of 4.6. This is encouraging, but it is surprising that in five sessions, ten generations of evolution did not produce a better than random population. These results are indicative of a general trend observed over most runs that the best fitness scores tend to improve slightly as the number of generations increases, which suggests that the approach is basically valid. Having said this, the results were sometimes visually disappointing, although this may be partly due to the limited support for shaders currently implemented in the Subversion rendering engine.

While the most fit individual was often found in the earlier generations (sometimes the first generation), this is not a particular worry, as the user would want to be presented with multiple pixel shaders to choose from at the end of the session, hence it is more important that the overall fitness of individuals

increases, as we see in the majority of the sessions. Moreover, recalling the long term goal of enabling Subversion players to prepare their own cities, one mode of supplying pixel shaders during the (possibly lengthy) evolutionary process would be to present them with the best individual from each generation. As we see from table 3, this would ensure that they see more fit shaders than randomly generated ones, with an average fitness of 89.1, rather than 69.7 for the average random generation. In table 4, we see that for each of the six setups, on average a session would show an increase in average population fitness from start to finish, which is again encouraging. We also note that it was more difficult to maximise the fitness functions from setups 1, 2 and 3 than from setups 4, 5 and 6. This was expected, as the latter fitness functions had no hue requirements to satisfy.

In addition to the automated approach described above, we have performed some initial experimentation with a user-driven approach, where the user chooses their preferred shaders for crossover and mutation in successive generations, thus acting as the fitness function on a local level. While the software had a ludic quality, from initial testing, it was our overriding feeling that the process was more successful in finding visually interesting shaders when automated than when user-driven. In particular, the user feels a tangible lack of control during evolution, possibly because the results of crossover and mutation are not always very intuitive. This suggests that we need to experiment with the settings for crossover and mutation, as they are too destructive from generation to generation.

While successful as a preliminary study, the interface we developed needs significant work if it is to be used commercially. For example, it would be useful to compare two arbitrary shaders from the population (this facility is not currently implemented), allow the user to take a more active role in the evolutionary process and override the fitness function to specifically nominate candidates that they find appealing, and perhaps even allow specific mutation requests such as inserting preferred colour values or requesting different shading algorithms for different parts of the image, e.g. walls as opposed to roof areas. Further, the evaluation method might be improved both in speed and accuracy by counting only those regions of interest deemed to be foreground (city) objects rather than sampling over the entire scene. Also, the tree operations that we define only exercise a small fraction of the functionality available in the OpenGL Shading Language. It would be interesting to implement a more complete set of operations and to determine which operations are more likely to be useful, so that tree growth may be biased accordingly and the software may be directed towards more fruitful areas of the search space.

Working with the Subversion city generation engine, in addition to the work presented here, we have also investigated the evolutionary generation of buildings [1], in addition to user-guided citywide look-and-feel (e.g., ratio of commercial and residential building types, etc.) and fitness function guided evolution of traffic schemes, where the fitness functions encourage city design for fun driving experiences, rather than best traffic flow or other utilitarian purposes. We hope to show that using evolutionary techniques both at game design time and for users to generate content at run time has much potential to increase efficiency of game design, and increase user enjoyment.

ACKNOWLEDGEMENTS

This project was funded by a grant from the UK Technology Strategy Board. We would like to thank the staff of Introversion Software Ltd. for access to their code and for their very valuable input into the project. In particular, we would like to thank Andrew Lim for his help with the Subversion city generation engine. We would also like to thank the anonymous reviewers for their constructive and useful comments.

REFERENCES

[1] A. Martin, A. Lim, S. Colton and C. Browne. Evolving 3D Buildings for the Prototype Video Game Subversion. In the Proceedings of the EvoGames workshop, 2010.

[2] W. Engel. Programming Vertex & Pixel Shaders, Charles River Media, 2004.

[3] R. Rost, B. Licea-Kane, D. Ginsburg, J. Kessenich, B. Lichtenbelt, H. Malan and M. Weiblen. OpenGL Shading Language, 3rd Edition, Addison Wesley, 2009.

[4] J. Koza. Genetic programming. On the programming of Computers by Means of Natural Selection, MIT Press, 1992.

[5] J. Foley, A. van Dam, S. Feiner and J. Hugues. Computer Graphics: Principles and Practice, 2nd Edition, Addision Wesley, 1990.

[6] Y. Rafiq, J. Mathews, and G. Bullock. Conceptual building design-evolutionary approach. Journal of Computing in Civil Engineering, 17(3), 2003.

[7] E. Hastings, K. Guha, and K. Stanley. Evolving content in the galactic arms race video game. In Proceedings of the IEEE Symposium on Computational Intelligence and Games, 2009.

[8] N. Sorenson and P. Pasquire. The Evolution of Fun: Automatic Level Design for 2D Platformers. In Proceedings of the 1st International Conference on Computational Creativity, 2010.

[9] F. Musgrave. Genetic Textures. In D. Ebert et al. (Eds.), Texture & Modeling, A Procedural Approach, 1998.

[10] M. Ebner, M. Reinhardt and J. Albert. Evolution of Vertex and Pixel Shaders, M. Keijzer et al. (Eds.), In Proceedings of EuroGP, 2005.

[11] J. Loviscach and J. Meyer-Spradow. Genetic programming of vertex shaders. In Proceedings of EuroMedia, 2003.

[12] C. Lim, R. Baumgarten and S. Colton. Evolving Behaviour Trees for the Commercial Game DEFCON. In the Proceedings of the EvoGames workshop, 2010.

[13] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels. Evolution of human-competitive agents in modern computer games. In Proceedings of the IEEE Congress on Evolutionary Computation, 2006.

[14] C. Bailey and M. Katchabaw. An emergent framework for realistic psychosocial behaviour in non player characters. In Proceedings of the 2008 Conference on Future Play: Research, Play, Share, 2008.

[15] Ponsen, M., & Spronck, P. (2004). Improving adaptive game AI with evolutionary learning. In Proceedings of Computer Games: Artificial Intelligence, Design and Education, 2004.

[16] M. Ebner. An Adaptive On-Line Evolutionary Visual System. In Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, 2008.

[17] S. Harding and W. Banzhaf. Fast Genetic Programming on GPUs. In Proceedings of the EuroGP, 2007.

[18] K. Fok, T. Wong and M. Wong. Evolutionary Computing on Consumer Graphics Hardware. IEEE Intelligent Systems 22(2), 2007.