# Evolving 3D Buildings for the Prototype Video Game Subversion

Andrew Martin, Andrew Lim, Simon Colton and Cameron Browne

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

**Abstract.** We investigate user-guided evolution for the development of virtual 3D building structures for the prototype (commercial) game Subversion, which is being developed by Introversion Software Ltd. Buildings are described in a custom plain-text markup language that can be parsed by Subversion's procedural generation engine, which renders the 3D models on-screen. The building descriptions are amenable to random generation, crossover and mutation, which enabled us to implement and test a user-driven evolutionary approach to building generation. We performed some fundamental experimentation with ten participants to determine how visually similar child buildings are to their parents, when generated in differing ways. We hope to demonstrate the potential of user-guided evolution for content generation in games in general, as such tools require very little training, time or effort to be employed effectively.

## 1 Introduction

Creating content for a game world is a notoriously time-consuming process, largely due to the complexities involved in 3D modelling. Of all of the environments that can be created in a 3D world, one of the most difficult to realise is a city, and yet they are becoming increasingly common in modern games. Significant amounts of time and money are required to create such environments, and even for larger companies, this can represent a sizeable financial risk. Furthermore, it is often necessary to make significant concessions in order to achieve a result within an appropriate time frame, such as repeatedly reusing the same assets or settling for an unrealistically small game environment. Subversion is an early game prototype by Introversion Software Ltd. (www.introversion.co.uk), which will be set in an automatically generated city. The aim here was to create a tool that would allow a player to rapidly design buildings for the Subversion game world, to increase their engagement with the game and to make the 3D environment richer and more diverse.

As described in section 2, buildings in Subversion are represented in a custom structural markup language. This representation is amenable to random generation, crossover and mutation, as described in section 3. Hence, we were able to implement a user-directed building creation interface which users/players can employ to evolve a set of satisfactory and unique building models from an initial generation of stock (or randomly generated) models. If this interface is to find its way into the Subversion release, various experiments relating to user satisfaction have to be carried out. In section 4, we describe the most fundamental such experiment. This tested whether the evolutionary operators lead to useful changes, i.e., the user can make satisfying progress using the tool, where child buildings look enough like their parents for there to be a purpose to choosing candidates for evolution, but are not so similar that progress is too slow. As described in section 5, other approaches could be used to solve this problem, such as providing the user with intuitive modelling tools similar to those seen in the popular game Spore. However, the main advantage of the evolutionary approach is that the user/designer need not have a clear image of their desired result in mind before proceeding. Instead, the user is presented with visual cues for the duration of the design process, and simply has to visually assess each building model presented to him/her. This also allows for a satisfyingly minimalist interface, far removed from the rather intimidating interfaces normally found in 3D modelling tools.

## 2 Building Generation

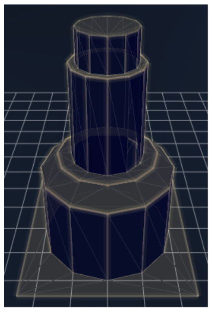
The Subversion procedural content engine is capable of generating and displaying 3D models of building represented as data files that describe their key features as sets of structural commands. The language was custom designed for the description of buildings and the resulting data files are in human readable form.

### 2.1 Command Language

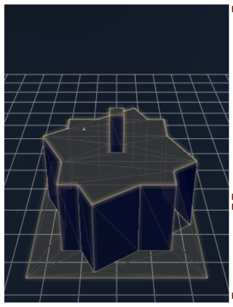
The command language essentially defines each building as a stack of three-dimensional objects, each described as a two-dimensional shape that is vertically extruded. The two-dimensional base shapes include circles, rectangles and squares, and can be subject to various simple operations such as translation, rotation and union. Base shapes may also be contracted, and it is also possible to offset the base of a shape from its initial position using a three-dimensional vector, which can be used in conjunction with a contraction to create slanted edges. Examples of the command code used to describe two buildings are shown in figure 1a, with an illustrative example given below. The Subversion interpreter parses the building description files into internal tree representations, which can then be used to rapidly generate 3D models of the buildings as required. An advantage of internally representing objects in tree form is that this facilitates their manipulation by evolutionary techniques, as described in section 3.

### 2.2 An Illustrative Example

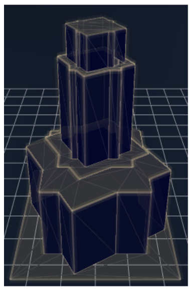
Figure 2 portrays a pair of building files, where the first, `CirclePart`, has another building (`SquarePart`) embedded on top of it. The internal data structure that is used to represent the buildings is a node grid which consists primarily of a list of vertices in addition to fields for vertical extrusions and height. The `Creation`



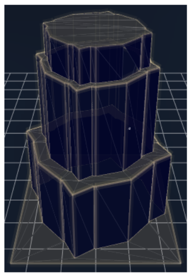
```
BEGIN parent1 1
  BEGIN Creation
    BEGIN Circle
      radius 20
      nodes 12
    END
  END
  BEGIN Sectors
    SetHeight 25
    BEGIN Contract
      amount 5
      BEGIN ContractSuccess
        BEGIN NewSectors
          BEGIN MoveSectors
            offsetY 5
          END
          BEGIN Contract
            amount 4
            BEGIN ContractSuccess
              BEGIN NewSectors
                SetHeight 29
                BEGIN Contract
                  amount 2
                  BEGIN ContractSuccess
                    BEGIN NewSectors
                      SetHeight 10
                    END
                  END
                END
              END
            END
          END
        END
      END
    END
  END
END
```



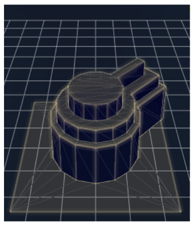
```
BEGIN parent2 1
  BEGIN Creation
    BEGIN Rectangle
      length 32
      width 32
    END
    Addition
    Rotate 60
    BEGIN Square
      length 34
    END
    Solidify
  END
  BEGIN Sectors
    SetHeight 25
    BEGIN CreateChild
      type parent2_1
    END
  END
END
BEGIN parent2_1 1
  BEGIN Creation
    BEGIN Square
      length 4
    END
  END
  BEGIN Sectors
    SetHeight 13
  END
END
```



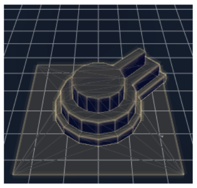
```
BEGIN child1 1
  BEGIN Creation
    BEGIN Rectangle
      length 33
      width 31
    END
    Addition
    Rotate 60
    BEGIN Square
      length 34
    END
    Solidify
  END
  BEGIN Sectors
    SetHeight 25
    BEGIN Contract
      amount 5
      BEGIN ContractSuccess
        BEGIN NewSectors
          BEGIN MoveSectors
            offsetY 5
          END
          BEGIN Contract
            amount 4
            BEGIN ContractSuccess
              BEGIN NewSectors
                SetHeight 29
                BEGIN Contract
                  amount 2
                  BEGIN ContractSuccess
                    BEGIN NewSectors
                      SetHeight 10
                    END
                  END
                END
              END
            END
          END
        END
      END
    END
  END
END
```



```
BEGIN child2 1
  BEGIN Creation
    BEGIN Circle
      radius 20
      nodes 12
    END
    Addition
    Rotate 60
    BEGIN Square
      length 34
    END
    Solidify
  END
  BEGIN Sectors
    SetHeight 25
    BEGIN Contract
      amount 4
      BEGIN ContractSuccess
        BEGIN NewSectors
          SetHeight 29
          BEGIN Contract
            amount 2
            BEGIN ContractSuccess
              BEGIN NewSectors
                SetHeight 10
              END
            END
          END
        END
      END
    END
  END
END
```

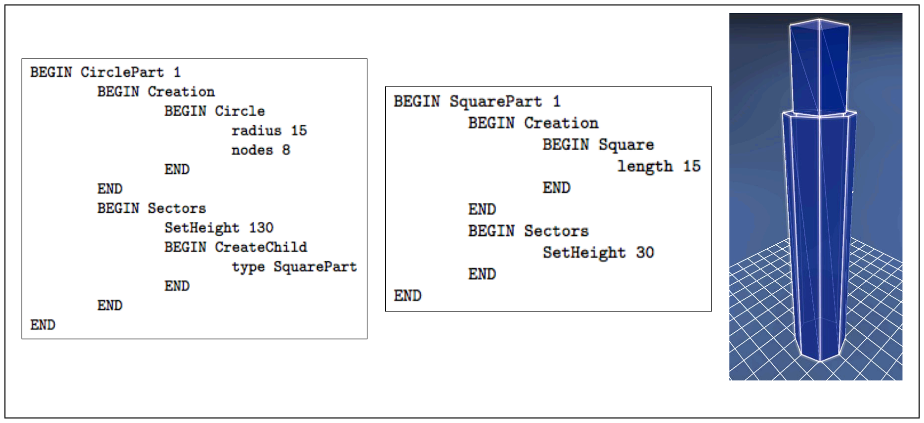


```
BEGIN StructMutant1 1
  BEGIN Creation
    BEGIN Circle
      radius 15
      nodes 15
    END
    BEGIN Translate
      offsetX 10
      offsetY -10
    END
    Addition
    Rotate 45
    BEGIN Rectangle
      length 15
      width 30
    END
    Solidify
  END
  BEGIN Sectors
    SetHeight 15
    BEGIN Contract
      amount 2
      BEGIN ContractSuccess
        BEGIN NewSectors
          SetHeight 5
          BEGIN Contract
            amount 4
            BEGIN ContractSuccess
              SetHeight 6
            END
          END
        END
      END
    END
  END
END
```



```
BEGIN ParamMutant1 1
  BEGIN Creation
    BEGIN Circle
      radius 15
      nodes 15
    END
    BEGIN Translate
      offsetX 10
      offsetY -10
    END
    Addition
    Rotate 45
    BEGIN Rectangle
      length 15
      width 30
    END
    Solidify
  END
  BEGIN Sectors
    SetHeight 3
    BEGIN Contract
      amount 2
      BEGIN ContractSuccess
        BEGIN NewSectors
          SetHeight 5
          BEGIN Contract
            amount 4
            BEGIN ContractSuccess
              SetHeight 6
            END
          END
        END
      END
    END
  END
END
```

**Fig. 1.** (a) Two example buildings and their Subversion command code descriptions. (b) Two children produced by crossing over the parents above. (c) A structural mutant [left] and a parametric mutant [right].



```
BEGIN CirclePart 1
        BEGIN Creation
                BEGIN Circle
                        radius 15
                        nodes 8
                END
        END
        BEGIN Sectors
                SetHeight 130
                BEGIN CreateChild
                        type SquarePart
                END
        END
END
```

```
BEGIN SquarePart 1
        BEGIN Creation
                BEGIN Square
                        length 15
                END
        END
        BEGIN Sectors
                SetHeight 30
        END
END
```

**Fig. 2.** A simple building represented as two building files, and the resulting node grid.

section of each file in figure 2 expresses what the base shape for the building is going to be. It allows for basic shapes like circles, rectangles and squares as well as more complex shapes specified by vertex lists, which can also be subject to various simple operations like rotation and union. The `Sectors` parts of building files define transformations to be applied using the shape defined in `Creation`, and they also allow for the embedding of other buildings by the `CreateChild` command. The transformations include: rotation (which is applied to the entire base shape as opposed to the per part rotation done in `Creation`), scaling and contraction (which is the result of moving all the edges along their inner normal at a constant rate, as shown in figure 2 by the protruding tower).

When parsed, the building file is split into two separate sections called `Creation` and `Sectors`. As mentioned previously, when the generator is called by the game engine, it uses the information in the file to generate a 3D node grid, represented with a vertex list, height and child information. In our example, the node grid generated by `CirclePart` after Creation is:

$$\text{vertexlist} = \{(15, 0, 0), (10.6, 0\ ,10.6), (0, 0, 15), (\text{-}10.6, 0, 10.6), (\text{-}15, 0, 0), \\ (\text{-}10.6, 0, \text{-}10.6), (0, 0, \text{-}15), (10.6, 0, \text{-}10.6)\} \\ \text{height} = 0, \text{child} = \text{null}.$$

In this example, immediately after the base has been generated, the `Sectors` commands will be run and for the `CirclePart` node grid, the vertexlist would remain unchanged, but the height would become 30 and the child field would point to a new node grid, that eventually becomes the node grid for the `SquarePart`:

$$\text{vertexlist} = \{(7.5, 0, 7.5),(\text{-}7.5, 0, 7.5),(\text{-}7.5, 0, \text{-}7.5),(7.5, 0, \text{-}7.5)\} \\ \text{height} = 30\ \text{child} = \text{null}$$

After a building has been generated into a node grid, the grid is triangulated and passed to the renderer. The buildings themselves are then simply represented as an array of triangles. The colour and alpha values of the triangles to be rendered are stored elsewhere in the engine, where the RGBA values of everything that is to be rendered inside the Subversion city are stored.

## 3 Evolving Buildings

Our building evolution tool maintains a population of building models and operates using a standard evolutionary approach. The population is initially seeded with the randomly generated building descriptions, and those buildings which are selected by the user constitute the breeding pool used to create the next generation. If the breeding pool is empty when the user requests a new generation, then more random buildings are produced for the next generation. If not, then random pairs of parents are selected from the breeding pool, crossed over to create children and then mutated, based on the likelihoods specified through the user interface (see subsection 3.3 below). This process is repeated until enough children have been created to fill the next generation.

### 3.1 Crossover

Crossover is performed as per standard Genetic Programming techniques. Each branch of the parent trees is assigned a tag denoting that branch's role and what other branches it is compatible with. For example, branches tagged `Circle` and `Square` both denote two-dimensional base shapes, hence are functionally equivalent within the language and therefore compatible. Similarly, a branch tagged as a command to define extrusion length may be crossed over with other similarly tagged branches. Each command within the subset of the language that we use has a corresponding tag.

After the trees are tagged, both parents are cloned and a branch is randomly selected from one of the clones. All branches in the other clone with matching tags are then located and one is randomly selected; these two compatible branches are then swapped. This process of swapping branches may then be performed multiple times, depending on the user-specified crossover strength. Once the swapping is completed, one of the modified clones will be deleted, and the other saved as the child. Two examples of crossover can be found in figure 1b. For example, the building depicted in figure 1b [left] is a child produced by single-swap crossover, taking buildings A and B depicted in figure 1a respectively as its two parents. The child has taken A as its base, and has had its `Creation` branch swapped for the `Creation` branch of B. The building depicted in figure 1b (right) is a child produced by four-swap crossover between A and B. The child has taken A as its base again, but this time the `Circle` subtree was swapped for the `Rectangle` subtree of B, and then the entire `Creation` subtree was swapped for the (now modified) `Creation` subtree of B, bringing the `Circle` subtree back into A. Similarly, the entire `Contraction` subtree was first swapped out for the `CreateChild` subtree of B, but then the `CreateChild` subtree was swapped back out for a new `Contraction` subtree beneath the original Contraction subtree. As we can see from figures 1a and 1b, the two children are clearly different from both parents, but have inherited aspects of both.

## 3.2 Mutation

After crossover, the following two kinds of mutation may be applied to the resulting offspring, depending on the user-specified strengths.

• **Structural Mutations.** These apply to the structure of the tree itself, so that some subtrees are completely replaced with randomly generated subtrees (which are functionally equivalent, as explained in section 3.1). In order to structurally mutate a tree, another tree is randomly generated with the constraint that it must contain the same number of sections as the first tree. Swapping is then performed between the trees as per crossover. As before, the strength of the structural mutation is defined as the number of swaps performed between the randomly generated tree and the original tree. Figure 1c [left] shows a structural mutation of a building. A random tree was generated containing a `Contract` subtree, which was then swapped into the original building in place of the `CreateChild` subtree.

• **Parametric Mutations.** These apply to those commands with numerical parameters, such as the sizes of shapes, rotations, translations, extrusions and contractions. The process is straightforward: numerical parameters are selected at random from the tree and replaced with new values randomly generated within each parameter's specified range. For example, a value specifying the degree of rotation might be mutated to any value between 0 and 360. The strength of the parametric mutation dictates the number of parameters that will be mutated. Figure 1c [right] shows a typical parametric mutation. Note that significant changes may result from the parametric mutation of buildings with complex Creation subtrees, but as we see from the experiments below, weak parametric mutations will probably not have a significant effect.

## 3.3 User Interface

Upon starting a session with the building evolution tool, the user is shown an initial generation of randomly generated data files, each describing a building, for which the corresponding 3D models are generated and displayed as a grid of tiles. The user can control a camera in virtual 3D space to view the buildings from any desired angle; every building is viewed from the same angle to avoid the tedium of manipulating a separate camera for each. Clicking on a tile expands that building to fill the screen, and clicking again reverts the interface back to the tile view. Each tile has a button to indicate its status as an individual of interest. The interface also provides a number of sliders to control (a) the structural mutation strength and likelihood (b) the parametric mutation strength and likelihood; and (c) the crossover strength. When the user has finished selecting buildings of interest and configuring the sliders, they click on the *Next Generation* button to create a new generation of buildings from those selected. The tile grid is then cleared and repopulated with the new generation; previous generations can be revisited using tabs along the top of the grid.

## 4 Experimental Survey and Results

With user-driven evolutionary systems such as those found in evolutionary art, the first question to be asked is whether the evolution of artefacts provides a satisfying experience. To be satisfying, the user must feel like (a) their choices make a difference, i.e., the children in each subsequent generation bear an adequate resemblance to the parents that the user chose and (b) evolution is not proceeding too slowly, i.e., the children in subsequent generations should not look too much like their parents. Especially when evolving complex objects such as building designs which are marked up, we should not take such a satisfying search for granted, as it is possible that the objects are so complex that even minor changes at genome level make huge changes at the artefact level.

For a fundamental experiment to address this question, and to test whether the different generation techniques produce children as expected, ten participants were asked to complete a survey where two parent buildings and 16 offspring buildings were presented for twenty pairs of parents. Figure 3 shows an example of the questionnaire sheet. We see that the participant is asked to rate each child as being (a) too similar (b) too dissimilar or (c) neither: okay to their parents. Designs marked as 'too different' often included those that were too outlandish or physically infeasible to make pleasing buildings, while those that showed trivial variation from either parent and therefore offered little genetic diversity were often marked as too similar. The nature of the way in which the 16 children were generated was varied across the twenty questionnaire sheets. In particular, the following six generation methods were exhibited, with the sheets taken from the 1st, 6th and 14th populations, to produce 18 sheets:

1. Strong crossover (20 swaps per child).
2. Weak crossover (5 swaps per child).
3. Strong structural mutation (20 structural mutations per child).
4. Weak structural mutation (5 structural mutations per child).
5. Strong parametric mutation (20 parametric mutations per child).
6. Weak parametric mutation (5 parametric mutations per child).

Two sheets with randomly generated children were also used as controls.

### 4.1 Results

The results of the surveys are shown in table 1. The values shown indicate the percentages of those designs marked too similar, too different and OK respectively. The results were largely as to be expected. In particular, it is clear that the random generations bear little resemblance to the supposed parents (on average only 0.7% of the children were deemed too similar, with 67.4% of the children deemed too different). As expected, the randomly generated control set performed worse than the controlled parameter sets, with children being marked as OK much less often on average than any other generation method. It is interesting to note, however, that 31.9% of randomly generated children

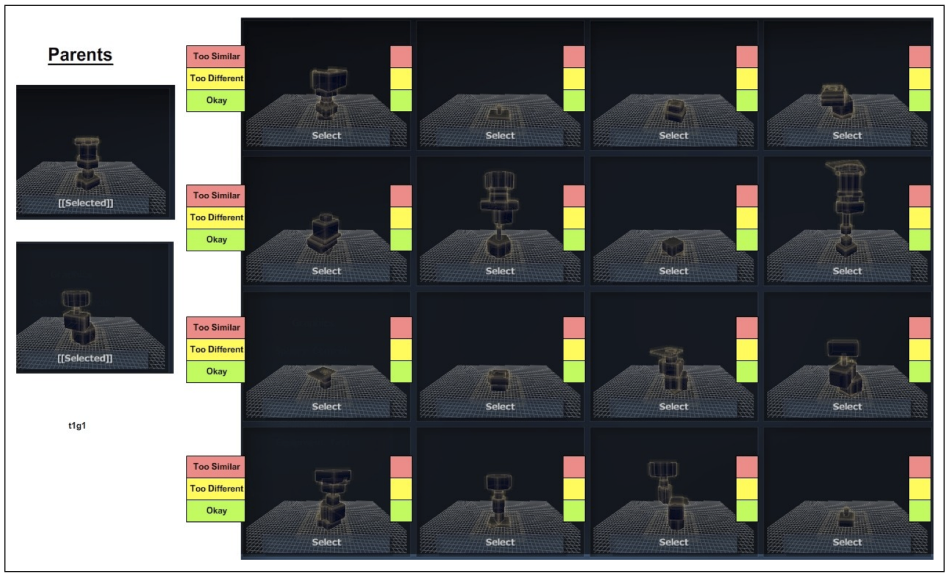


**Fig. 3.** Example questionnaire sheet.

| Construction Method | Generation Number | Participant Evaluation (%) Too similar | Too different | OK |
|---|---|---|---|---|
| Strong crossover | 1 | 1.9 | 52.8 | 45.3 |
| | 6 | 9.4 | 43.1 | 47.5 |
| | 14 | 4.4 | 47.5 | 48.1 |
| | average | 5.2 | 47.8 | 47.0 |
| Weak crossover | 1 | 47.2 | 5.0 | 47.8 |
| | 6 | 49.4 | 4.4 | 46.2 |
| | 14 | 35.6 | 14.4 | 50.0 |
| | average | 44.1 | 7.9 | 48.0 |
| Strong structural | 1 | 1.3 | 51.9 | 46.8 |
| | 6 | 6.3 | 43.7 | 50.0 |
| | 14 | 4.4 | 70.0 | 25.6 |
| | average | 4.0 | 55.2 | 40.8 |
| Weak structural | 1 | 51.6 | 12.6 | 35.8 |
| | 6 | 11.9 | 29.4 | 58.7 |
| | 14 | 17.5 | 8.7 | 73.8 |
| | average | 37.0 | 16.9 | 56.1 |
| Strong parametric | 1 | 1.3 | 56.9 | 41.8 |
| | 6 | 3.8 | 39.4 | 56.8 |
| | 14 | 0.6 | 49.4 | 50.0 |
| | average | 1.9 | 48.6 | 49.5 |
| Weak parametric | 1 | 23.3 | 25.8 | 50.9 |
| | 6 | 28.7 | 11.3 | 60.0 |
| | 14 | 35.6 | 1.3 | 63.1 |
| | average | 29.2 | 12.8 | 58.0 |
| Random | 1 | 0.0 | 56.2 | 43.8 |
| | 14 | 1.3 | 78.7 | 20.0 |
| | average | 0.7 | 67.4 | 31.9 |

**Table 1.** Building designs by their construction method, with evaluation percentages and average evaluation percentages per construction method indicated.

were deemed to be neither too similar nor too dissimilar (OK), which suggests that some random introduction of building designs might be appreciated by the user, and might enable them to drive the search in new directions. In overview, it is encouraging that the non-random generation methods achieve a hit rate of around 50% for OK children, although this could be improved upon.

As expected, we see that weak versions of crossover, structural mutation and parametric mutation all produced children which were assessed far more often as too similar to the parents than the strong versions of these generation methods. Perhaps less expected is the observation that the weak form of each parameter setting led to more child designs being marked by the user as OK than the corresponding strong form, and we plan further experimentation to determine why this might be. Furthermore, we can also see that the weak structural mutation and weak parametric mutation settings led to more children being marked as OK than any other settings, although this improvement is not marked. The only obvious correlation with the generation number is exhibited by the weak parametric generation method, where the percentage of children marked as too different clearly decreases as the generation number increases. This can be explained by the size of the building design tree growing as the session progresses, and because weak parametric mutation has little effect on larger designs.

## 5 Conclusions and Future Work

A good survey of procedural techniques for city generation is supplied in [3]. The most well known example of procedural building generation is Pascal Mueller et. al's work [4], which forms part of the city generation procedures [5], which has resulted in the commercial CityEngine package (www.procedural.com). Mueller's approach is different from the one presented here, because it employs a shape grammar rather than a markup language, and is discussed primarily in the context of heavily automated generation. While shape grammars have been evolved, for instance in [9], we do not know of any application of evolutionary techniques to Mueller's approach. A genetic algorithms approach to conceptual building design is presented in [8], where the emphasis is on decision support systems to assist designers. Addressing fundamental questions of whether the evolutionary techniques involved in evolutionary arts projects (in particular, image filter evolution) are better than random generation is the topic of [1], where the authors asked 30 participants to interact with a very similar interface (without the 3D elements) to the one described here. They showed that intelligent generation of image filters is preferred by users up to statistical significance, and the evolutionary techniques outperformed image retrieval and database techniques. We plan to compare and contrast our approach with other evolutionary approaches to content generation for games, such as in [2].

We have demonstrated that a user-directed evolutionary process can be employed for generating custom building designs for game content in a city world for a commercial video game. We have further shown that the search process the user undertakes is satisfying because it neither produces too many overly similar nor overly dissimilar offspring buildings. As suggested by an anonymous reviewer, we need to investigate how disruptive the crossover mechanism is that we employ, and consider experimenting with homologous crossover methods [6] [7]. Moreover, while our preliminary experiments have served an experimental design purpose here, we need to gather more data in order to draw statistically significant conclusions about the nature of user interaction with the system. In future, we plan to develop a more sophisticated user interface for the evolution and selection process, by adding more constraints to both the command language and the generation process. This will hopefully reduce the number of outlandish and infeasible shapes that are invariably rejected by the user, which could be further enhanced by the automated detection of such degenerate cases. Another useful addition to the tool would be the inclusion of constraints for the generation of themed content for situations in which particular building types are required. Examples of such themes might include 'corporate skyscraper', 'residential house', and so on. Also, more sophisticated experiments comparing combinations of parameter settings rather than each parameter setting in isolation could reveal further insights into harmonious parameter settings for the purpose of automated evolution.

## Acknowledgements

We would like to thank the anonymous reviewers for their very useful comments. This work was supported by a Technology Strategy Board grant.

## References

1. S Colton, J Gow, P Torres, and P Cairns. Experiments in objet trouvé browsing. In *Proceedings of the 1st Int. Joint Conference on Computational Creativity*, 2010.
2. 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.
3. G Kelly and H McCabe. A survey of procedural techniques for city generation. *Institute of Technology Blanchardstown Journal*, 14, 2006.
4. P Mueller, P Wonka, S Haegler, A Ulmer and L Van Gool. Procedural modelling of buildings. *ACM Transactions on Graphics*, 25(3):614–623, 2006.
5. Y Parish and P Mueller. Procedural modelling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 2001.
6. J Park, J Park, C Lee, and M Han. Robust and efficient genetic crossover operator: Homologous recombination. In *Proc. Int. Joint Conf. on Neural Networks*, 1993.
7. R Poli, C Stephens, C Nucleares, A Wright and J Rowe. On the search biases of homologous crossover in linear genetic programming and variable-length genetic algorithms. In *Proc. Genetic and Evolutionary Computation Conference*, 2002.
8. Y Rafiq, J Mathews, and G Bullock. Conceptual building design-evolutionary approach. *Journal of Computing in Civil Engineering*, 17(3):150–158, 2003.
9. R Saunders and K Grace. Extending context free to teach interactive evolutionary design systems. In *Proceedings of the EvoMUSART workshop*, 2009.