Algorithmic Design of Graphical Resources for Games Using Genetic Algorithms

Daniel Eriksson

Tutor, Erik Berglund
Examinator, Anders Fröberg
Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Daniel Eriksson
Algorithmic Design of Graphical Resources for Games Using Genetic Algorithms

Daniel Eriksson
Linköping University
Linköping, Sweden
daner045@student.liu.se

ABSTRACT
Producing many varying instances of the same type of graphical resource for games can be of interest, such as trees or foliage. But when randomly generating graphical resources, you can often end up with many similar looking results or perhaps results that doesn't look like what it is meant to look like. This work investigates whether genetic algorithms can be applied to produce greater varying results when generating graphical resources by basing the fitness of each individual for each genetic generation on how similar the graphical resource is to previously generated resources. This work concludes from the limited work that was performed that while it seems possible that the use of genetic algorithms might be able to produce visually different graphical resources, Blender currently doesn't seem to be able to produce enough results in a reasonable time frame for this to be usable on a large scale.

INTRODUCTION
This section introduces why this work was performed and what questions the work aim to answer.

Motivation
Graphical resources can be created by hand using visual tools or by algorithms. There are advantages to using algorithms when generating a huge set of slightly different assets representing things such as nature or creatures looking distinctly different based on different abilities whilst still resembling the same base object. This can be useful since by simply seeing a large set of the same type of object generated differently there is suddenly many different results that can be hand-picked to be used since they look good or interesting, a large set of generated graphical resources could also contain results unique in a way not previously considered to be created to begin with due to the wide variety it could potentially produce.

But none of this is useful if the algorithm generates assets that look wrong or simply just gives repeated results, a vital part of generating assets this way would be to optimize the process so that it actually produces results that can be recognized and pleasing to human eyes whilst still having enough variation in the large set of generated graphical resources.

Purpose
The purpose of this work is to investigate how an algorithm that generates a certain graphical asset, a tree, can be optimized to produce better results. This optimization is based on genetic algorithms which will identify results that are deemed unacceptable or not different enough and exclude them from the selection process between each generation of produced results, hopefully eventually resulting in avoiding similar unwanted output in future generations.

Research questions
1. Can a wide variety of graphical resources that look acceptable and varied enough be generated when optimizing using genetic algorithms?
In this case acceptable results are determined by the user, judging if a graphical resource resembles what it is intended to be.
2. How feasible is it to use Blender to implement a generative model and run a genetic algorithm to produce a huge set of results?

Limitations
Due to time constraints the work will be limited to a simple generative model and the evaluation of the genetic algorithm may be limited by the amount of time required to perform rigorous amount of testing to produce results to evaluate.

THEORY
This section describes topics and terminology relevant to the work as well as earlier similar work that has been performed.

Genetic Algorithms
Genetic algorithms were first conceived by Holland[1] in 1975, since then there’s been further work within genetic algorithms based on Holland’s work performed by Grefenstette[2], Davis[3,4], Golberg[5] and Michalewicz[6] amongst others. Beasley et al.[7] summarized the fundamentals of genetic algorithms: A genetic algorithm is based on natural behavior. There is a population of individuals that a variant of natural selection can be applied thanks to being able to calculating the fitness
of an individual. By selecting the most fit individuals in a population, these individuals can cross breed and mutate to potentially produce offspring that are even more fit. This can be analogous to a search problem where a more fit individual is closing in on a desirable solution.

For a problem to be solvable with a genetic algorithm it requires individuals to be represented by some sort of properties like a genome, by changing a gene you change the individual itself and over the course of several generations a genetic algorithm will have potentially changed the object to something closer to the desired result.

**DEAP**

DEAP\(^1\) is a Python framework for evolutionary computation. It supports genetic algorithms along with other types evolutionary computation.

**Blender**

Blender\(^2\) is a 3D computer graphics program that will be used for this work, 3D modeling is among one of its features. The program is built in C, C++ and Python, the user interface itself is made using Python and the majority of the actions that can be performed within it is part of Blender’s Python API as Blender operations which allows for easy scripting and automation of the program. Blender scripts can be packaged as Blender add-ons and be integrated into the user interface itself to handle and use custom-made scripts more conveniently.

![Figure 1. Primitive 3D-meshes available in Blender.](image)

One of the features of Blender that will be primarily used are *primitives*\(^5\), which are preset 3D meshes that are a good starting point to model from. Blender also features *modifiers*\(^6\) which are tools for modifying a 3D mesh in a variety of ways using parameters. The original mesh is retained while using modifiers until the modifier itself is applied to the mesh which makes it very simple to experiment with different parameters before finally applying the changes.

**3D and Blender terminology**

A 3D-mesh\(^5\) consists of points called vertices (singular: vertex), edges that connect two different vertices with a line and faces which is the inner area of a cyclic loop of edges. The vertices and edges make up the tree-structure that is a 3D-mesh.

An *edge loop*\(^6\) is a set of lines that form a loop, models are often built around edge loops to easily keep and modify the shape of the mesh.

A *loop cut*\(^7\) is when a new edge loop is inserted around already existing edge loops.

Boolean operations\(^8\) can be performed in Blender on meshes and produces resulting meshes in the same way boolean algebra works.

*FreeStyle*\(^9\) is an non-photorealistic rendering engine usable in Blender, it can be used to draw lines around meshes in a variety of ways when rendering a scene.

*Bool-Tool*\(^10\) is a Blender add-on that comes packaged with Blender 2.78 and provides functionality to perform more advanced boolean operations than the standard boolean modifier in Blender.

**Earlier works**

There has been work about how different Python libraries supporting genetic algorithms perform. Garrett\(^[8]\) performed a study about what capabilities different Python libraries have for evolutionary computation, the author compare Pyevolve, DEAP and the author's own library Inspyred. The results show that all three libraries are quite capable for working with genetic libraries and that DEAP has an extensive set of features for working with different types of evolutionary computation.

There are several studies in generating 3D graphical assets. Bao et al.\(^[9]\) performed a study about generating different building facades in 3D. The facade layouts were generated by placing different building elements together based on a

\(^1\) https://github.com/DEAP/deap
\(^2\) https://www.blender.org
\(^3\) https://docs.blender.org/manual/en/dev/modeling/meshes/primitives.html
\(^4\) https://docs.blender.org/manual/en/dev/modeling/modifiers/index.html
\(^5\) https://docs.blender.org/manual/en/dev/modeling/meshes/structure.html#structure
\(^6\) https://docs.blender.org/manual/en/dev/modeling/meshes/structure.html#edge-loops
\(^7\) https://docs.blender.org/manual/en/dev/modeling/meshes/editing/subdividing/loop_subdivide.html
\(^8\) https://docs.blender.org/manual/ja/dev/modeling/modifiers/generate/booleans.html
\(^9\) https://docs.blender.org/manual/en/dev/render/free_style/introduction.html#what-is-freestyle
\(^10\) https://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Object/BoolTool
set of given constraints. The results of this study shows that
great variation could be produced, but the system requires
the users input so whether the system generates good
looking buildings was not taken in account. Merelli and
Manocha[10] presented a system where a user supplies an
example model and constraints to procedurally generate
more advanced models based on the given model. Boolean
expressions are used to properly “synthesize” together
different parts of the given model for the generated model
which results in a great variation for the output.

While the previously mentioned works produce acceptable
varied output, the algorithms are based on parameters
being set by user input. There are studies in using genetic
algorithms instead to produce varied and acceptable output.
Gravina and Loiacono[11] performed a study using single
and multi-objective genetic algorithms to produce balanced properties for weapons in *Unreal Tournament III*. Results from matches with bots (computer players) in the
game were used to decide the selection in the algorithm. A
user test showed that the algorithm ended up generating
weapons that were interesting and fun for players.
Pirovano et al.[12] created a tool that used a genetic
algorithm to generate different swords in 3D where the
user controlled the selection process. A survey indicates
that the algorithm ended up mostly generating swords that
appealed to the users.

Previous work shows that genetic algorithms has been able
to be used to produce varied acceptable optimized output.

**METHOD**

This section describes how the different part of this work
was set up and what was intended to be gotten from the
results of performing it.

The work consists of three parts. First the rules that
generate a specific type of object had to be established and
implemented, then a means of comparing the similarity of
a generated object in Blender and finally the tests were
implemented into Blender using the DEAP evolutionary
library together with the generative model and comparison
functionality.

**Generative Model**

A generative model was created to support generating
objects to make it resemble a certain object whilst still
being able to be visually distinct from each other, a list of
parameters acting as genes were used for this. In this case a
generative model was built to resemble a simple tree.

**Comparison**

The genetic aspect of this work requires the objects that are
generated from the generative model to be comparable to
be able to calculate the fitness of an individual since it is
based on previously generated objects. Two given genomes
generates the objects to be compared.

To find a reasonable method of comparing two generated
objects, different ways of comparing the overall size of the
object in Blender was investigated and evaluated.

**Tests in Blender**

To be able to evaluate if the genetic algorithms would be of
any use when working with the established generative
model and the chosen method of comparing different
instances of said generative model, three tests were created
as Python scripts in Blender using DEAP to implement the
genetic aspect. A typical but simple genetic algorithm was
implemented that uses the generative model and
comparison method to perform these tests.

The first test was performed to determine how many
generations of trees were necessary to see a significant
enough difference in the trees from crossing and mutating
the genomes of the trees across generations, this was done
by running several tests that applied the genetic algorithm
to produce one tree over the course of 100 generations and
checking at which generations the fitness improved.

The second test was run to produce 50 trees as a result.
This test kept the comparison pool constant in size by
adding the newest best individuals and removing the oldest
for each set of trees that were run through the genetic
algorithm and produce one tree as a result. This test was
run to determine if we could keep getting different trees by
constantly replacing the comparison pool and as such not
affect the complexity of genetic algorithm by constantly
growing the comparison pool by one for each result
generated.

The final test was performed to investigate a performance
issue that appeared in the second test. To determine
whether the issue was due to poor implementation of the
generative model, the DEAP library or Blender a test was
performed where a significant amount of Blender operators
were used and timestamped.

**RESULTS**

This section presents the results from the work that was
described in the method section.

**Generative Model**

The generative model was created by writing a class in
Python that has a genome array as a member and a
`generate_model` function that actually constructs and
modifies the model based on its genome, the following text
details how the generative model for the tree constructs
itself:

The tree consists of one cylinder primitive for the trunk of
the tree and four cube primitives resembling groups of
leaves. These primitives are deformed using Blender
operators along with random values from a range of values
acting as a constraints for each operation, these constraints
are predefined to ensure that each primitive is modified
within a reasonable fashion to not produce crazy results.
The trunk is deformed by performing horizontal loop cuts along the cylinder primitive to give more edge loops to modify, these edge loops are rotated, moved and resized to make the trunk look like it bends and gets smaller the further up it goes. The top-most face of the cylinder primitive is then moved, rotated and resized as well since it is effectively the same as the top-most edge loop.

Each group of leaves gets set to a random position near the trunk, then a subsurface subdivision-modifier is applied to give more geometry to work with. Then it is deformed in a similar fashion to the trunk, loop cuts are performed, the top face is resized to give a more round shape at the top. Then the whole object gets resized so that each leaf-group may differ in size from each other.

Smooth shading is applied and the appropriate material is set to each object. Freestyle Lines are also applied to draw an outline around each object, which helps define the general shape of the whole model as well as help cover up some visual artifacts from intersecting geometry.

To support the genetic aspect of this work, every time the generative model uses a random value within given constraints that value is saved to the generative model-instance as an array of values. When the generative process is finished this array contains all the parameters that modified the object in some way and acts as the generated objects own set of genes, a genome. This genome can be supplied when creating a new instance of the generative model to generate the exact same object again.

Before the boolean operation could be performed, the generated objects needed to be fully joined into one closed mesh instead of consisting of several intersecting meshes, this was performed with a boolean union.

Boolean subtraction was the first attempt at comparing two objects by subtracting one object from the other and then calculating and returning the total surface area of the remaining mesh. The boolean subtraction gave varying results when comparing different objects depending on their size and shape. If the object that gets subtracted doesn't completely cover the other object, the resulting object gets an inset which depending on its size can return a larger total surface area than the original object. This issue is illustrated in figure 3.

Boolean intersection was the second attempt at comparing two objects, boolean intersection returns a mesh of the parts of the two models that overlap each other, then the total surface area of this mesh is calculated and returned.

The comparison using boolean intersection should theoretically always result in a lower or equal total surface area compared to the original objects unless they have a gap in between areas where they intersect. This was decided to be the less error-prone approach and as such was decided to be the preferred method of comparing how closely an object resembles to another. A function was written for the generative model that takes two generative
models initialized with a genome, generates both objects and compares them with boolean intersection and then returns the percentage (valued from 0.0 to 1.0) of the surface area that the result has compared to the surface area of one of the original objects.

**Genetic Algorithm**
The resulting genetic algorithm that ended up being created for the tests involving genetics was the following for:

1. A comparison pool is created an initialized with 4 random individuals (randomly generated genomes).
2. An initial population of 10 individuals are created and evaluated using the decided upon comparison method of comparing generative models.
3. The next generation of individuals is selected from the current population using a tournament selection function in DEAP which gathers 3 fit individuals.
4. The genomes of the offspring are crossed using a crossover function in DEAP.
5. There is a 25% chance that an offspring may mutate, in which there is a 25% chance for each gene in the genome of the offspring to be randomly set.
6. The offspring replaces the old population and their fitness are evaluated using the same fitness method.
7. Step 3-6 is repeated until the decided amount of generations are run, then the most fit individual is added to the comparison pool and results. The algorithm repeats from step 2 until enough results have been produced.

**Test 1**
The first test to see how many generations are needed before we get significant details was set up to run for 100 generations to produce a single tree. This was run 4 different times.

As can be seen in figure 5, in the first run of the test, a pretty significant improvement was reached at generation 24. The second run showed an improvement from 0.822 to 0.699 over 40 generations but had reached a fitness score of 0.719 already at generation 4, the third run showed an improvement from 0.846 to 0.698 over 22 generations and the fourth and final run showed an improvement from 0.840 to 0.724 in 15 generations.

<table>
<thead>
<tr>
<th>Generation</th>
<th>Best Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.883824041623195</td>
</tr>
<tr>
<td>2</td>
<td>0.856939044184079</td>
</tr>
<tr>
<td>11</td>
<td>0.852721577034374</td>
</tr>
<tr>
<td>12</td>
<td>0.8503461776282387</td>
</tr>
<tr>
<td>14</td>
<td>0.82661308674098</td>
</tr>
<tr>
<td>16</td>
<td>0.828411475463644</td>
</tr>
<tr>
<td>18</td>
<td>0.8183982599363222</td>
</tr>
<tr>
<td>24</td>
<td>0.7384396394603843</td>
</tr>
<tr>
<td>31</td>
<td>0.7358031377284615</td>
</tr>
<tr>
<td>34</td>
<td>0.7384396394603843</td>
</tr>
<tr>
<td>61</td>
<td>0.720357066686235</td>
</tr>
<tr>
<td>62</td>
<td>0.7203521254995015</td>
</tr>
<tr>
<td>78</td>
<td>0.7093573050900687</td>
</tr>
<tr>
<td>85</td>
<td>0.7091100760121383</td>
</tr>
<tr>
<td>87</td>
<td>0.7088320155047498</td>
</tr>
<tr>
<td>88</td>
<td>0.708351173399739</td>
</tr>
</tbody>
</table>

Figure 5. This shows the generation at which the best fitness was improved upon during the first test run.

With a small set of trees to compare against, it seems that a somewhat optimized result can be achieved rather quickly at around 20-25 generations, at least when the comparison pool is small.

With this test the genetic algorithm will be tested if it actually produces visually different graphical resources. Based on the results of the previous test, the amount of generations were set to 20, populations were set to contain 6 individuals, the amount of trees to generate to 50 and the comparison pool was changed to only contain the 4 latest generated trees.
Upon reviewing the resulting trees after running the genetic algorithm, I would conclude that the results are visually different enough from each other. An excerpt of the result can be seen in figure 6.

<table>
<thead>
<tr>
<th>Tree</th>
<th>Time elapsed (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53.39658188919985</td>
</tr>
<tr>
<td>2</td>
<td>135.26505875587463</td>
</tr>
<tr>
<td>3</td>
<td>244.8665006160736</td>
</tr>
<tr>
<td>4</td>
<td>392.5973255634308</td>
</tr>
<tr>
<td>5</td>
<td>559.2993023395538</td>
</tr>
<tr>
<td>6</td>
<td>763.9194166660309</td>
</tr>
<tr>
<td>7</td>
<td>980.6423494815826</td>
</tr>
<tr>
<td>8</td>
<td>1268.3748981952667</td>
</tr>
</tbody>
</table>

Figure 7. Time elapsed when the genetic algorithm was done with each tree.

There was however a noticeable increase in the time elapsed when each of the new trees that were optimized by the genetic algorithm were outputted, as can be seen in figure 7, the first tree was optimized in 53 seconds and the last tree was optimized in 27.1 minutes, with the whole process taking 11.5 hours to complete.

Blender Performance Test

<table>
<thead>
<tr>
<th>Cubes</th>
<th>Time between</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>2.153114080429077</td>
</tr>
<tr>
<td>10000</td>
<td>3.3455841541290283</td>
</tr>
<tr>
<td>15000</td>
<td>4.527255058288574</td>
</tr>
<tr>
<td>20000</td>
<td>5.881573670762988</td>
</tr>
<tr>
<td>25000</td>
<td>7.498967170715332</td>
</tr>
<tr>
<td>30000</td>
<td>9.442362308502197</td>
</tr>
</tbody>
</table>

Figure 8. Time taken between each set of 5000 cubes.

To test if Blender was the cause of the slowdown, a test was set up where a cube was created and then cleaned up 30000 times. The time between each set of 30000 cubes were logged and as can be seen in figure 8, there was a significant decrease of the performance of Blender operators the more they are used within one run of a script.

DISCUSSION

In this section the results and methods used within this work is discussed.

Result

Creating a generative model with a genome to support genetic evolution worked out well which was expected from previous work of generating a 3D-object from a set of parameters. This is something that could be reused to build upon this work, this implementation is however quite simple since all different parts of the tree are all independent of each other, if a more complex generative model were to be implemented such as that a group of leaves needs to be attached to a tree branch, then the structure behind the generative model would need to become a bit more complex to support parameters depending on each other when crossing individuals, since that process mixes the parameters in the individuals genome at will currently.

While the chosen method of comparison using boolean intersection is far more reliable than using boolean subtraction, it does still have some drawbacks as described in the results however. I feel that this part of the work has a lot of potential to be improved since while boolean operations work very well, it doesn't seem to be 100% accurate, just very accurate in general which can affect some edge cases. I think that this process might become less reliable the more complex the resulting 3D object generated by the generative model becomes, I think the simplified nature of this work is a reason why it seems to have worked as well as it has. This part of the work was not based upon any of the previous work since the described previous work have either used direct statistics or user input as feedback to use with the fitness function, in this work I attempted to compare how visually distinctive objects are from each other using mesh data in Blender to automate the process rather than require user input to decide.

The results from the second test that generated 50 trees seems to generate decent enough results for the genetic algorithm to converge towards generating trees that look somewhat different, however the test was performed by only comparing against the 4 latest generated trees, it feels like a promising start but more testing is required to say for sure. The amount of time it took to generate 50 optimized trees in one run calls into question if using genetic algorithms would be worth it compared to simply just randomly generating a huge amount of trees in a smaller time frame in terms of usability for a real application of this work.

It would've been interesting to perform a test to determine how well the genetic algorithm would perform with a constantly growing pool of results to compare against, for each set of trees that are optimized over the course of several generations, the best individual was added to the
set of trees that were compared against to see if these optimized trees would also seem different from each other or if the system would get “stuck” when trying to optimize trees to not look the same. This test was not performed because of the huge amount of time the second test already took to produce 50 trees, if the comparison pool would grow by one for each of those tests the overall time to produce even half the amount of trees would skyrocket because of the issue with running a huge amount of Blender operations in one run of the test script.

As seen in the performance test, the slowdown over time grows pretty significantly, it’s a shame that something like this impeded the progress of this work. There may be a way around this issue but I didn't find any towards the end of the work, the script could be run to only perform one part of the evolution at a time while storing the progress to a file between runs, but I could not find a way to automate running the script. This work requires extensive use of Blender operations and they require a UI context to be initialized before they can be used, and that happens after Blender runs scripts from the command-line. I don't doubt that this can be solved but in the limited time for this work I did not find a solution.

Method
The overall method of this work has been about finding a way to represent generated objects in a way that supports genetics, finding a way to compare these objects and implementing a genetic algorithm to test the viability of using genetic algorithm to generate more visually distinct results. As such the approach to each of these tasks have been to simply try things to see how well they worked as I got to them, since there wasn’t always a clear path ahead on how to solve a problem I find it likely that the methodology of this work could be improved upon, especially in the parts of the work that was performed differently from the previous work that I found and as such were not based on any previous results.

CONCLUSIONS
Producing different looking graphical resources based on a generative model with a genetic algorithm to optimize the output for the results to be more distinct from each other compared to random generation seems very feasible, however more research is needed since this work was very simplified and cut short because of the performance issue over time.

Using Blender to continuously produce graphical resources in one go with a script doesn't seem feasible in this case due to the amount of Blender operators used to generate and compare models which caused a significant slowdown over time especially since there is so much repeated work by Blender during each generation, but if the cause for this slowdown could be avoided then it could be feasible to use Blender for this purpose.

REFERENCES