Algorithmically Generating Bugs

Theodor Bladh
Christoffer Stridh

Tutor: Erik Berglund
Examiner: Anders Fröberg
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enskilda kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

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/.

© Theodor Bladh, Christoffer Stridh
ABSTRACT
A method to quickly produce large sets of different textures of insects (bugs) was implemented. The textures were procedurally generated in a 3D-modelling tool and then rendered for use in a 2D-motion-based game. The resulting textures were evaluated regarding quality and distinguishability. No particular features were found to be favoured over others, except some features being better than no features at all. Distinguishing between generated textures was found to be a trivial task for most users.

INTRODUCTION
This project was performed to generate textures for an existing research game developed by Erik and Aseel Berglund at LiU-IDA.

Objective
The goal of this project was to algorithmically generate textures. These textures were then evaluated based on their distinguishability and relative quality among themselves to ensure they can be used as replacements for the existing graphics in a game.

Purpose
Algorithmically generating textures for sprites reduces the time required by a proficient artist working on a project. Being able to deliver a varied graphical environment without excessive development costs or losing quality is important.

Research Question
Our main question was, what makes bug-textures good and liked by users, specifically in the style we used?

• How do we make sure that all textures are varied enough to be distinguishable?

A requirement for the project was for textures to be individually identifiable since various textures represent enemies with varying gameplay behaviour and if users could not easily perceive the difference then it could ruin the experience.

• What features do users perceive as the most defining for a texture?

To be able to make our textures distinguishable, we first have to know in what ways we can alter them so that users can quickly notice the difference. If we make an alteration and it is not detected, then it was pointless to do it in the first place.

• What features do users like best?

If there are certain features that users like then using those in the game will make it look better. Cohesiveness with the existing graphics of the game was not extensively considered as the style may be replaced in the future. However we still attempted to follow a certain graphical style at the request of the games stakeholders.

Limitations
The largest limitation in our project was that the textures will be generated before the game is shipped and will be evaluated by the games developers before they are included. This means that they will receive some quality assurance and much less constraints on time and processing resources. Because of this we focused much less on efficiency and optimising algorithmic complexity as time taken was not an issue.

Because of time constraints and there only being two of us, we limited the user studies to a small set of people. We assumed that the collected data would not show very large variance so the limitation was not expected to be major. Particularly the culling performed before the final study could be affected by the different age-groups having varying opinions, and only the children taking part. We also did not have access to any hand-drawn bugs that we could compare our textures with to evaluate their quality.

The bugs are restricted by both the original bug shape being manually designed and then modified based on subjective opinions about how a bug is supposed to look. The lack of an objective definition of the best visual...
representation for a bug made this unavoidable but it is difficult to evaluate the negative effects of our influence on the bugs if any.

The textures we generated are of comparatively low-resolution bugs. As such our work may not apply to higher resolutions or for textures representing anything other than bugs.

**BACKGROUND**

We decided to create textures using Blender¹ (version 2.69) since it is easily scriptable² using Python³, allowing us to generate and modify shape and colour of objects and export animated textures. It also came more intuitively to manipulate actual objects rather than textures directly and manipulate lighting conditions to affect the objects with less effort than using a 2D-tool. Creating the textures based on 3D-objects also has the benefit that they can be rendered from any angle.

Other possible options would have been to use another 3D-modelling tool such as Autodesk Maya⁴ or directly using 2D-tools like ImageMagick⁵ and/or Photoshop⁶, all of which allow for some amount of scriptability. We chose Blender because of some earlier experience and initial tests showing that the integration with Python worked very well. We decided to use 3D over 2D for the initial generation because of the reasons mentioned above. A minimal amount of ImageMagick is still in use once the 2D-images are rendered, to prepare them for the sprite atlas used in the game.

**Procedural Generation**

We consider the definition for procedural content generation given by Togelius et al.⁹ to be suitable for our project. We don't consider ourselves as users when we define the base-bug or the algorithm that modify it but the developers of the game that will be using our project to create content for their game can be considered users. The selection process of which bugs to eventually use is clearly a form of user-input and this is why we made this distinction. Should they later create another version of the base-bug that would also be considered user-input.

**3D-modelling**

In 3D-modelling the concepts of *vertex*, *edge* and *face* are used to represent objects.

- A *vertex* is a point in space.
- An *edge* is a straight line connecting 2 vertices.
- A *face* is a surface area outlined by 3 or more edges.
- A *mesh* is a construct of multiple vertices, edges and faces creating the 3D-model.

Blender is a free open-source 3D-modelling program with a massive amount of features ranging from simple 3D-modelling to various physics simulators including liquids and even a fully fledged game engine. To us the 3D modelling and Python interface were especially interesting as it allowed us to create varied shapes for the bugs along with the textures in an automated way. Using python code to manipulate models it is possible to do nearly everything that can be done manually; create new shapes as well as rotate, scale, translate as well as add and remove vertices. It is also possible to add *materials* to different objects or faces altering their interactions with lighting; making them reflective, transparent or change colour. Blender makes it easy to animate objects by setting *keyframes* specifying location, rotation and/or scale of an object in a specific point in time. Blender interpolates the necessary steps for frames in-between.

**Earlier Works**

Significant research has been performed in the area of procedural model and texture generation. Like Roden et. al.¹¹ we believe that the future lies in procedural aided or fully procedural generation of models and especially textures. If the trend of increasing detail in computer graphics continues then it will become increasingly tedious and expensive to develop every aspect by hand. Being able to automate at least some of these steps could be of great benefit to anyone utilising computer graphics.

---

1. https://www.blender.org/features/
3. https://www.python.org/
Initially we looked at the concepts of evolutionary algorithms for generating computer graphics used by Sims [2] and Reynolds [6] as well as L-systems as described by Hidalgo et al. [3]. We lacked an automated way to determine which bugs generate textures of sufficient quality as our goal was to identify which features would rate highly, and had to do so by hand, inspecting the textures post-generation. Without a good way of choosing which bugs to use as progenitors, an evolutionary algorithm was deemed unsuitable.

One issue we realised early was how to ensure that no seams were visible at the edges of the model, as traditional texture generation does not account for varied surface shapes or wrapping around a model. Turk [4] and Zhang et al. [7] tackled these challenges in their papers, we chose to instead avoid the problem by not applying textures and instead assigning materials to faces to create similar results. However, Blender can only apply a single material per face, this limits our methods level of detail based on the number of faces in the rendered object.

**METHOD**

To create textures in a varied way we chose to use Blender, as it allowed us to generate not only the colouring for the bugs but also new shapes by using Blender's 3D-modelling features.

Blender records all actions the user takes by hand in the graphical user interface (GUI) in a log listing the equivalent Python-functions to call to replicate the behaviour. In most cases it was then simply a matter of copying these function-calls to the appropriate place in the code. In some situations it was more complicated as certain commands requires Blender to be set in certain modes ('Object'/Edit') etc. in which case you have to ensure that the code executes in the correct mode, a very small subset of operations are very difficult to do through code such as the knife tool, as it is dependent on the 3D-view.

Saving actions as mentioned made it easy to prepare a handmade bug and recreate it for every new iteration, which we then modified by moving some vertices and introducing new ones based on a random seed and user assigned parameters. Colouring was done by applying a base material and modifying some of its properties and/or changing the material of some faces.

**Preparing Evaluations**

To evaluate our systems ability to reach the goals specified in our research questions, we performed three separate studies. At first we performed a small (10 subjects) study about midway through the project, to evaluate our progress and collect valuable information. This helped us determine where to focus our efforts in improving the bug generations qualities. The second study was performed as a preparation for the final study, to cull out which of our bugs to examine more thoroughly.

**Midway Evaluation**

Three weeks into the project a small user study with ten participants, all of which were students, was performed. The goal was to ensure that development so far had been focused towards the correct goals and to create guidelines for future work. We explained that we were generating graphics for a game and that we would appreciate data on what features make graphics distinguishable and likeable. We instructed them to look at five 5x5 grids, in sequence, of randomly generated bugs. They were allowed a maximum of roughly 20 seconds to determine how many different bugs each grid contained. We also asked them to choose the bug they liked the appearance of the most to gather information on what features are the best candidates to continue work on.

**Culling Evaluation**

We wanted to ensure that the bugs used in the final evaluation would be the best possible fit for the game according to the stakeholders and the best liked by children (the games main target audience). As we have no automated way of evaluating the bugs and only keeping those of acceptable quality, we decided to manually cull a generated set of 273 bugs with the help of the stakeholders and 11 children. The first culling with the stakeholders reduced the number of bugs to 104.

These 104 bugs were then presented to 11 children aged 7-11. They were asked to score all the bugs between 1 and 5 points based on how
much they liked them. The bugs were sorted after the mean values of their score. This ordered list of bugs was then used to select what bugs to use in the final evaluation.

**Final Evaluation**

The final study (mix of 23 students and teachers) evaluated how well our final results reached our goals and was divided into two parts.

**Defining Features**

We consider features to be any visual aspect of a bug that stands out and can potentially be used to identify it, regardless of how it was generated. Our goal was to identify which of these features are most important to users when differentiating between bugs. Examples of features included base colour, type and colour of markings.

Similarly to the midway study we placed 5-7 unique bugs for each grid from the 29 previously selected (score of 3.5 or higher) in 5x5 grids using duplicates to fill the remainder of the grid. The unique bugs were selected so that four grids contained bugs of similar colour or shape. The fifth grid contained an even mix of bugs with various colours and shapes. Using these properties, we hoped to identify the most defining features by using these grids for the distinguishability test described below and analysing the results.

**Distinguishability**

We repeated the same procedure as in the midway study and asked our test subjects to determine the number of unique bug-types present in the grid, given only 15 seconds per grid. The given time is strictly limited like this because it is the ability to quickly differentiate between bugs that is of interest as the bugs will be visible on the screen in the game for only a short time. The lower time limit compared to the midway evaluation is because we assume that our bugs are easier to distinguish, as more time has been spent refining their creation. We then repeated this process with each of the five grids we had prepared as above. We intentionally tried to make this test more difficult than it would be in-game, using 25 bugs visible at once and a strict time-limit. The goal was to make sure that the bugs are very easily distinguishable and if users did well in the test, they should have no problem with the game.

**Relative Quality**

Naturally, being able to distinguish between our various bugs is not the only requirement of acceptable bug graphics, it is also important to find out what features users like the most, so that we can generate more bugs that have those qualities. Some balance is still required though, as reusing the same features on every bug just because they are well liked could ruin the distinguishability.

To evaluate our bugs relative quality, amongst each other we had our users sort the images by using pairwise forced choice as described by Mantiuk et. al. [8]. We chose to implement the quick-sort algorithm suggested by Mantiuk as it requires less comparisons than a more naive approach and using the test subjects' input as the comparison operator in the algorithm. The best 12 (score of 4 and higher) images selected in the culling evaluation as well as the 4 (score of less than 2) worst were selected and displayed in a minimal web application.

Like during the midway evaluation, we asked the users which bug they liked best to check for recurring choices.

**IMPLEMENTATION RESULTS**

**Base Bug**

As a starting point for automatic bug generation, we executed a function that repeats actions performed manually to create a simple bug-shaped set of objects (see Image 1). The base bug consists of a head, a body, two mandibles and three legs. The leg-textures are then mirrored in game to conserve texture size, this is more important for the legs and mandibles than for other body parts since they are animated in fifteen frames.

At this point, the bug is coloured in the default grey colour that all objects are given initially in Blender. It also lacks defining features such as stripes or shape modifications.

![Image 1: Base bug as generated by the program before getting modified.](image-url)
Modification

A modifier is represented as a function pointer combined with the arguments it has to be called with in order to produce a specific result. A bug consists of a list of body parts as generated above and a list of modifiers that will be applied to it to generate the specific visuals for that instance.

The first modifier that is always applied to every body part is `create_base_material`. This modifier applies a simple gradient with discrete steps as defined by its parameters: a start colour and an end colour along with a list that determines how light intensity affects what colour is displayed.

The `line` modifier (`material_line`) creates a new material using the same arguments as `create_base_material` and replaces the material of faces in a line shape on the bug. The arguments specify where to start a line, how long and wide it should be and its angle. This modifier is applied, twice, symmetrically along the bug.

The `ring` modifier (`material_ring`) works like the `line` modifier, except that it changes the material in a circular pattern instead of a line. It is also symmetrical.

The second set of modifiers alter the shape of the bug, rather than its material properties. The first of them are the `front_curve` and `back_curve` modifiers. They alter the curvature of the ends of the bug, making it rounder or sharper in a defined portion of the length of the bug.

The `dent` modifier (`dent`) increases or decreases the width of the bug in a region by given amount. This modifier often gives a sectioned look to the body.

The `ridge` modifier (`ridges`) does not actually alter the shape of the generated texture as its modification is only visible as a line and by its effect on how lighting affects the bug. This modifier is called similarly to the `line` modifier, but without arguments for material. Instead it creates new vertices on the surface and pulls them towards the camera, deforming the mesh into a ridge. This modifier is also symmetrical and creates two matching ridges.

Procedural Generation

As mentioned the `create_base_material` is always applied first, but after that point the amount, type and arguments for modifiers and the order in which they are applied is determined individually for each bug, which is how they end up with their respective visuals. Certain modifiers are limited in which body parts they can be applied to, for example dents could deform the head or legs in a bad way so they are forbidden from appearing there.

Colours are also slightly dependent on body parts, as legs and mandibles are forcibly made darker than the rest of the bug. Details such as rings and lines are given a second user assigned colour.

Front and back curves are limited to a maximum of one each per body part that is allowed to have them, as the combined effect of multiple curve modifiers in the same place counteract or amplify each other greatly. Dents are similarly restricted but multiple are allowed as long as they do not overlap for the same reason as with curves.

User Intervention

Using the process above, the program can generate a large number of bugs without any user input being required. However the end-results are often not what is being sought, so the user has to sift through the resulting images and determine which bugs they want to keep and which to discard. Each generated bug-texture is accompanied by a serialised representation of the Python object required to recreate it. So if a bug that is almost right is found, it can easily be recreated and modified by hand until it matches the sought visuals, operations such as adding/removing or modifying arguments for any of the modifiers on the bug is possible, as they are all stored in the serialised object.

Since the bugs are generated for a video game, they also require death animations to complete the game-play experience. However generating these for each new bug would be excessively time-consuming since many bugs end up not being used. Instead the user is required to load the serialised bug and call a function to generate the death animation of their choice (a selection of 6 were developed). To maintain a
thematic style all death animations are created as variations on the same idea, another consideration was that they do not look too violent as the game is at least partially aimed to children. All deaths are created by generating a fragmented version of the body, by using a plugin called cell fracture and using Blender’s built-in physics simulator for collision interaction between the fragments and various non-rendered objects.

**EVALUATION RESULTS**

**Midway Evaluation Results**

Table 1 illustrates the number of bugs in the five grids as estimated by the users. Grid 2 contained bugs of very similar cyan colour, all bugs in grid 4 shared roughly the same shape.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid 1</td>
<td>-1</td>
<td>+1</td>
<td>5.4</td>
<td>6</td>
</tr>
<tr>
<td>Grid 2</td>
<td>-1</td>
<td>+1</td>
<td>5.8</td>
<td>6</td>
</tr>
<tr>
<td>Grid 3</td>
<td>-1</td>
<td>+2</td>
<td>7.1</td>
<td>7</td>
</tr>
<tr>
<td>Grid 4</td>
<td>0</td>
<td>+3</td>
<td>9.4</td>
<td>8</td>
</tr>
<tr>
<td>Grid 5</td>
<td>0</td>
<td>+2</td>
<td>5.6</td>
<td>5</td>
</tr>
</tbody>
</table>

*Table 1: Results of midway distinguishability tests*

We were surprised by how few times the users seemed to have mixed up bugs. We expected it to be harder to differentiate between them, especially with grid 2. The large number of overestimates in grid 4 might be a result of there being too many bug types to keep track of. We adjusted the final evaluation because of this so that no grid contains more than seven bugs.

The selection of favourite bugs was mostly focused on a single or a pair of bugs per grid receiving most votes. Many bugs received no votes at all and each grid except the last had a bug that received four of the ten participants’ votes.

**Final Evaluation Results**

Our hypotheses for the final study were that the users should be able to distinguish the bugs with less effort compared with the midway evaluation and, based on that study, that no particular feature should be easier or harder to tell apart.

For the relative quality we expected the best 12 according to our culling study, to be ranked better than the 4 selected from the very worst.

**Defining Features**

Looking at the mean correctness deviation being consistent across all grids in the final study (see table 2), we believe our hypothesis that no particular features are better for distinguishing bugs from one another to be correct.

**Distinguishability**

In table 2 we present the results of the final distinguishability test. A total of 23 people took part in the test, two of which were removed as their results were too irrational. Another three were ignored when compiling the data as they had previously participated in our midway study and we did not want this to affect the results. We still tested these people to see if their results were different but did not get hold of enough repeat subjects to make any clear conclusions. Overall there was no significant difference, but that might be unrepresentative.

There was a noticeable improvement in correctness of the answers between the midway study and the final study (compare table 1 and table 2). Considering the final study only allowed for 15 seconds instead of the original 20 per grid we consider our hypothesis to be validated. Notable is that no grid contained 8 bugs this time as we noticed this was too many for people to keep track of in the first study. No significant difference can be observed between the grid with 7 bugs and those with fewer. The lower accuracy for the first grid can likely be blamed on unfamiliarity with the task.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid 1</td>
<td>-1</td>
<td>+3</td>
<td>5.5</td>
<td>5</td>
</tr>
<tr>
<td>Grid 2</td>
<td>-2</td>
<td>+1</td>
<td>6.78</td>
<td>7</td>
</tr>
<tr>
<td>Grid 3</td>
<td>-2</td>
<td>+1</td>
<td>5.83</td>
<td>6</td>
</tr>
<tr>
<td>Grid 4</td>
<td>-2</td>
<td>+1</td>
<td>5.67</td>
<td>6</td>
</tr>
<tr>
<td>Grid 5</td>
<td>-1</td>
<td>+1</td>
<td>5.17</td>
<td>5</td>
</tr>
</tbody>
</table>

*Table 2: Results of final distinguishability tests*

**Relative Quality**

We compiled the results from the individual users bug sort and analysed the data. The results indicated that our hypothesis was mostly correct, three of the four bugs that were disliked in the culling study ended up at the very bottom. The fourth one also got a low rank, but unexpectedly ended up above the two top
There was no distinct pattern to how our users sorted the bugs, it seemed very individual which bugs they liked and disliked, e.g. six of the bugs even ended up on both of the extreme ends for different people.

Compared to the midway study, the favourites selected from the grids showed a more even spread of votes. The maximum number of votes for any bug was 8 out of 23 and very few bugs received a single or no votes at all.

DISCUSSION

Method

Using an efficient sorting algorithm such as quicksort for evaluating relative image quality may result in reduced accuracy as demonstrated by Silverstein et al. [10]. However they also stated that comparing images of very different quality is of lesser relevance and that the performance gains thus outweigh the loss of accuracy. This allowed us to compare a larger amount of textures without inconveniencing the users.

The way quicksort is implemented, all elements have to be compared with the first pivot element chosen. We believe that this might affect the users choices as it forces them to repeatedly choose between the pivot element and another bug.

Our assumption that roughly 20 seconds per grid is reasonable might not be optimal, we chose this number since it is close to the maximum amount of time the bugs are present on screen in game. While actually playing the game, the users would have many opportunities to learn how the bugs look, making it much easier to differentiate between them.

We do not know whether doing the project in a 3D-environment was the best decision to make or not. We think that it was easier to, for example, change the colour of the bugs in a natural way by changing lighting conditions and projecting them from 3D to 2D. It is however difficult to determine if the development strategy was superior to the alternatives considered. We experienced less difficulty than expected creating models that looked good in 2D. Most of the problems encountered were Blender specific and caused by known issues, although other 3D-tools likely would have had their own share of similar issues.

When gathering opinions from children in the second part of the preparation study, the bugs were printed on paper. This resulted in colours not quite matching their look on the computer screen used during development and gameplay. It may have had an effect on the scores given, but we assume this effect is negligible.

Results

When implementing the bug generation code certain parts were written in as general form as possible so that they can be reused, e.g. the modifiers can act on any body part and utility functions take into account if a body part is animated or not. For some parts of the code generalisation was deemed unnecessary. The creation of a base-bug is one such example, with the reasoning that if another variation is wanted, it would not be a lot of work to create one from scratch, or simply altering the existing function.

The decision to base all bugs on the same, simple objects, instead of creating them from scratch made it easy for us to guarantee that the result would be of roughly the correct shape every time. It also made it trivial to apply the animations for the legs despite their relative positions varying depending on the body’s shape. This creation process, as described in the method section, also makes it easy to group body parts according to type, and since the modifiers are written to take advantage of this, they are easily applied to every body-part of the same type.

There are two main reasons we chose to implement modifiers by storing them in lists on the bug along with their arguments rather than being applied directly. Simplified serialisation and allowing for genetic combination and mutation of bugs.

Serialising the bug before modifiers are applied is easier as Blender’s 3D-objects would require both more complicated methods and storage space due to intentional limitations in what is possible to do with them. The fact that the modifiers are stored along with their arguments also mean that they can be tweaked either externally or between deserialisation and
generating the 3D-model. Currently Pickle\(^7\) is used for serialisation as it required minimal effort to get working, another alternative would have been to use JSON\(^8\), XML\(^9\) or similar formats to improve portability and allow for easy modification of the data externally e.g. in another language or by hand.

If modifiers are seen as genetic traits, then genetic mutations and evolution by combining traits from different bugs should be fairly easy to do with the current implementation of modifiers. We considered implementing code for combining bugs at an early stage but later abandoned the idea as we realised that it did not match the goals of the project. It would also have taken too much time, better spent on other parts of the development process.

The procedural generation currently consists of very specific intervals for random numbers that are allowed in each modifier's argument list. These intervals were chosen based on our opinion of what makes a bug look reasonable and lacks objectivity. It is also hard coded which body parts are allowed to receive certain modifiers and the amount of modifiers that might be created for each body part. Although outside the scope of this project, there is room for improvement or a complete rewrite of the procedural generation code.

Procedural generation creates large volumes of models and materials compared to manual content creation. Blender saves previously used resources, which is normally not a problem, but with the large volumes we generate, a noticeable slowdown can be detected after some time. This problem can easily be solved by reloading the project file or restarting Blender so it is only a minor issue.

We considered using Blender's vertex-paint mode to colour the bugs, but ultimately decided that we could obtain the sought results quicker by changing material properties.

Blender offers substantial support to replicate actions done in the GUI through Python-code. During the project we encountered more difficulty than expected caused by bugs in Blender or functionality designed specifically for use through the GUI.

For example, while editing a mesh through the 3D-view it is intuitive to swap between modes to perform actions, but being forced to change between these modes results in unnecessarily complicated code. That problem is then compounded by a known issue in Blender where mesh data is not updated if changes are applied while in edit-mode. The most severe consequence is that selecting sections of the mesh can not be done in edit-mode, but to perform operations on the same or to de-select, edit-mode has to be active.

One GUI-specific tool that we would have used if possible was the knife tool, to create a new line of edges across the surface of an object which would have been useful to create ridges. Instead we ended up using a Boolean intersect operator to create the new edges required.

Boolean operators came with their own set of issues, while easy to use through code, some information about the objects original mesh is not correctly transferred in the version of Blender we used. Specifically what vertices belong to certain vertex groups. This caused issues as we positioned the body parts relative to each other using constraints depending on certain vertex groups. A number of extra issues also showed up when the ridge modifier was made to generate ridges symmetrically and a lot of time was spent solving those problems.

While the Cell Fracture-plugin we used to create death animations for the bugs was a powerful tool, it has some issues with the models our code generates and others have reported similar problems\(^10\) on multiple\(^11\) occasions. We partially solved these problems by not fracturing the whole bug, limiting ourselves to a single simpler shape by using only the body with its ridges removed. This solves the issue more often than not, but it still fails randomly. We assume that the impact on gameplay experience is minimal since most of the bug's area consists of the body.

\(^7\) [https://docs.python.org/3.4/library/pickle.html](https://docs.python.org/3.4/library/pickle.html)
\(^8\) [http://www.json.org/](http://www.json.org/)
\(^9\) [https://www.w3.org/TR/REC-xml/](https://www.w3.org/TR/REC-xml/)
**Evaluation**

**Defining Features**

If we interpreted the results of the grid-test correctly then there is no significant difference between features in regards to distinguishability. We find this to be a positive thing as it allows for changes to any of the features without reducing recognisability.

**Distinguishability**

While the results of the grid test in the final study were better overall it was also more common for users to make guesses at too low amounts. We believe this might be caused by the reduced time limit not giving the users enough time to identify all of the bugs in each grid.

We realised towards the end of the study that it would have been beneficial to have a 6th grid that would not be part of the actual study, but rather serve to demonstrate how to perform the test. Since we did not have this, some users had problems following exactly what we wanted them to do in the first grid and explaining it was a bit difficult with no visual aid.

**Relative Quality**

Comparing the users favourites from the grid study we found it interesting to see that the results were much more even than in our first study. This could be because of the very limited amount of subjects in both studies but we think it could be an indication of a more even level of quality of the displayed bugs. Partially this could be caused by the culling done before the final study, removing bugs of lower quality and leaving only the best, but we also believe it indicates an overall improvement of the generation of bugs itself.

When compiling the data from the relative quality sort we realised that we had misunderstood exactly how Mantliuk et al. [8] had analysed their data. We believed the matrix in their article (Figure 3.) was a sum of the relative order in all users sort orders. When trying to do this for our data, we found that it was no longer a strict totally ordered set, over the whole group of users, some bugs were considered equivalent. Because of this we could not follow the rest of the method to determine the difference in quality between bugs. However we were still able to distinguish how many bugs were considered better or worse than each one and could determine their relative ranks, just not how significant the difference between them was.

We were somewhat surprised by how big variation there was between how different users sorted the bugs. We did not expect for six bugs to show up in both extreme ends of the sort. Also surprising was to find that the two most favoured bugs from the culling study ended up getting ranked so low in the final study, even managing to fall below one of the four bugs chosen from the very bottom.

**Consequences**

We hope to have reduced the emotional impact on the younger audience of the game that the sprites are used for. Some of the previously tested death-animations were considered to be too violent for the game, but our new ones are comparable in style to the LEGO-game franchise\(^{12}\) with fragments falling apart and eventually fading away which we think is less violent and negatively impacting.

The project might also increase the number of people playing the game. Since the game is motion-based it could positively affect healthiness of the players.

**CONCLUSIONS**

In the end we feel that we have answered all of our intended research questions:

- How do we make sure that all textures are varied enough to be distinguishable?

We concluded that users had little difficulty distinguishing between bugs. Blender's ability to easily create large variation in both shape and colour made this task much easier than expected. Simply generating bugs with default arguments directly gave bugs that were different enough to be used in the game. There is the possibility that very similar bugs could be generated as the generation is based on random values, but we believe this to be very unlikely to happen and have not observed any cases of too similar bugs.

\(^{12}\) [http://videogames.lego.com/](http://videogames.lego.com/)
What features do users perceive as the most defining for a texture?

We were unable to find any patterns to which features made it easier or harder to distinguish between the bugs. We do not have enough data to say so for certain but we believe this is because no features are more easily used to distinguish between them.

• What features do users like best?

It seems users do not particularly favour any specific features, rather which features they prefer varies from person to person and appears to be based only on opinion. However most users seem to dislike bugs that have very few features and that look too plain as they are considered to be boring. The most well liked bugs tended to have intricate and varied patterns (see Image 2). Bugs that were too wide were commented to look “fat” and were also discriminated towards, but users seemed to not notice this if the bug had interesting enough patterns.

Image 2: Worst (left) and best (right) according to sorting done by users in final study.

FUTURE WORK

It would be possible to expand the set of modifiers to increase the diversity of the generated bugs.

The restrictions on arguments for the modifiers could be improved as well, preventing combinations that result in inadequate visuals. Particularly the angles and placement of ridges and lines sometimes makes them not visible at all on the resulting bug.

It should be possible to extend the use of modifiers to objects other than bugs.

REFERENCES


