Volcano: An Interactive Sword Generator

Daniele Loiacono
Dipartimento di Elettronica,
Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
Email: daniele.loiacono@polimi.it

Renato Mainetti
Dipartimento di Informatica
Universitá di Milano
Milan, Italy
Email: renato.mainetti@unimi.it

Michele Pirovano
Dipartimento di Elettronica,
Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
Email: michele.pirovano@polimi.it

***Abstract*—In this work, we introduce *Volcano*, a tool for the procedural generation of 3D models of swords. Unlike common procedural content generation tools, it exploits interactive evolution to reduce as much as possible the effort of the users during the generation process. Indeed, *Volcano* allows to *forge* the desired type of swords through a rather simple visual exploration of the design space. The 3D models generated with the tool can be directly used as game assets or further developed with a standard modeling software. A prototype of *Volcano* was tested by 30 users, including both students and game developers. The feedbacks received are very positive: tools like *Volcano* might be useful both for players, to create user contents, and for developers, to speed-up the design of game contents.**

I. Introduction

Creating game content is currently among the most expensive and time consuming activity in the game development process. To deal with this issue, game developers typically follow two approaches: (i) they exploit procedural content generation to reduce as much as possible the costs and (ii) they heavily rely on user generated content. However, procedural content generation tools might take a lot of trial and error to actually generate high-quality content; moreover, these tools are often rather limited to just some specific type of contents, e.g., terrain, trees, buildings, etc. Finally, user generated content requires a very dedicated community and the development of additional software, such as editors or SDKs, that often are too complex for the average user. A promising technique to deal with these challenges is the Search-Based Procedural Content Generation (SBPCG) [1]. In fact, SB-PCG aims at easing the generation of high-quality game content by means of a stochastic search algorithm, such as a genetic algorithm.

In this paper, we introduce *Volcano*, a tool to generate 3D models of swords. Our tool offers a simple user interface and allows to *forge* the desired swords with a very simple and limited user interaction. In fact, *Volcano* features a SB-PCG algorithm to capture the user preferences through the selection of previously generated content and to search accordingly the design space. Any model generated with *Volcano* can be exported in a standard format to either use it immediately as a graphical asset or to further develop it in any 3D modeling software.

We implemented a working prototype of *Volcano* and performed a preliminary test with few users. The feedback we received is very encouraging: users reported that *Volcano* was indeed capable of generating new swords based on their preferences and that they were generally happy with the quality of the generated swords. Moreover, users appeared quite interested in using tools like *Volcano* either to create user content for games they play or as a design tool for games they develop.

The paper is organized as follows. First, in Section II we briefly provide an overview of the related work. Then, in Section III we describe how we represents swords to procedurally generate them and in Section IV we illustrate how *Volcano* works. Thus, in Section V we discuss the results of the performed user study. Finally, in Section VI we draw some conclusions.

II. Related Work

Procedural content generation (PCG) was introduced in the early 1980s to overcome the limited memory resources: game content that could not fit the main memory was generated on the fly. Today, memory is not an issue anymore and PCG is mainly used to reduce the time and costs necessary to create a large amount of game content. Notable examples of commercial PCG tools that are widely used in the game industry are *SpeedTree* [1], that is specifically devised for generating trees and plants, and *CityEngine* [2] that allows the generation of buildings and cities.

Recently, PCG has also attracted the interest of several researchers from the field of computational intelligence, due to the introduction of the *search-based procedural content generation* (SB-PCG) [1]. SB-PCG combines search-based methods — typically evolutionary algorithms — with procedural content generation: the trial-and-error process that PCG usually requires (e.g., setting parameters values, implementing heuristics, etc.) is mapped into a search problem and automatically solved. So far, SB-PCG already proved successful for several type of game content [2]–[11]. Nevertheless, SB-PCG involves several challenges, the most important being how to evaluate the generated content to guide the search-based methods. Several approaches have been proposed in the literature to deal with these issues (see [1] for an overview). In this work we follow an approach that is inspired by the field of *interactive evolution* [12], a branch of evolutionary computation that replaced the usual fitness function computation with an interactive evaluation provided from one or more users. Interactive evolution has been successful applied to several domain including fashion design [13], ergonomic design [14], landscapes [15], images [16], music [17], and art in general.

[1] http://www.speedtree.com/
[2] http://www.esri.com/software/cityengine

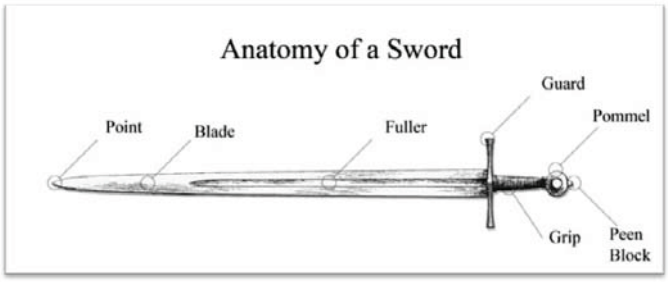


Fig. 1. The different parts of a generic sword.

Recently, interactive evolution has been applied also to generate game content. Hastings et al. [18] developed Galactic Arm Race (GAR) [18], a multiplayer shooter game that features a novel weapon system that evolves during the game based on players' actions. Later, Risi et al. [19] developed a Facebook game, Petalz[3], that applies interactive evolution to breed procedurally generated flowers. Finally, Cardamone et al [20] introduced TrackGen, a web service that exploits interactive evolution for the procedural generation of tracks for a racing game.

III. Procedural Swords

In this section, we describe the parametric model we use in *Volcano* to represent and to procedurally generate sword-like [4] shapes. We used a sword as a basis since it is a common, simple, and varied weapon, which gives us plenty of different shapes to work with a few simple constraints. We designed a simplified parametric model through the observation of known swords, determining what set of parameters would better describe the shape of the whole weapon in the simplest terms. We leveraged information on historical sword manufacturing techniques from online resources [21] to determine these parameters.

For simplicity, we define a sword as a composition of its four main different parts: the blade, the guard, the grip, and the pommel (see Figure 1 for an example). These parts are connected to each other in a sequential fashion to form a sword. We model the different parts as tapered curves, thus identifying for each part (i) an open curve that defines the overall part shape and direction (spine), (ii) an open curve that defines the tapering alongside the spine (taper curve), and (iii) a closed curve that defines the cross-section (section curve).

In order to support many different sword shapes, we parameterize the curves of each weapon part. For each part, we define the spine, taper, and section curves by selecting appropriate parameters and plausible value ranges that define the chosen shapes. We restrict parameters to values that produce plausible results, for example by limiting the grip's width to a fixed value and fixing its spine to a straight segment so that all swords can be wielded. We also provide templates of section curves for simplicity. We subdivide the blade's taper into two parts, the point and the edge, so to define the two independently (see

[3]http://petalzgame.com/

[4]Despite our model being mainly focused on classic swords, it also allows the generation of different kinds of hand-to-hand weapons, such as axes, knives, spears, etc. However, for the sake of simplicity, in the whole paper we refer to the contents generated by *Volcano* simply as swords.

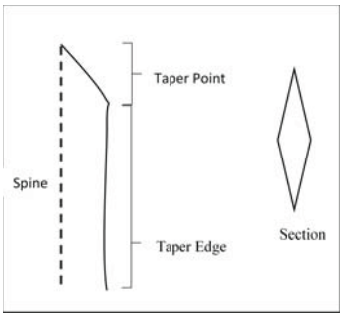


Fig. 2. The spine, taper (point and edge), and section curves for a simple gladius-like blade. Image taken from Wikipedia.

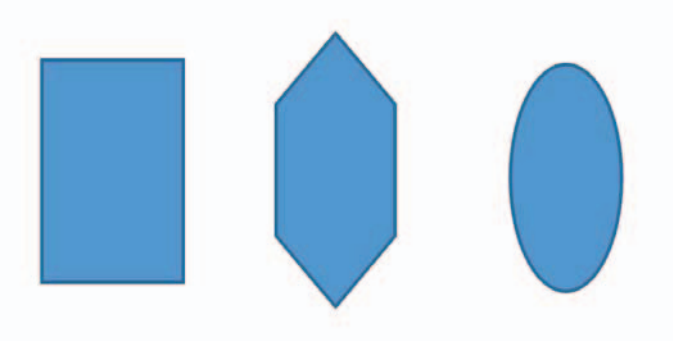


Fig. 3. The templates used for section parts. From left to right: square, diamond, and round.

Figure 2). Table I lists all the parameters of the sword model described above. Note that the parameter groups are for the most part independent from each other, so that we can switch a grip with another without affecting the blade, and vice-versa. We however enforce some slight dependency between a few parameters: for example, the grip length is constrained to be larger than the blade's width to avoid unrealistic weapons. By default, we fix the composition of the different parts to achieve a sword-like appearance. However, to increase variation in the final result, we add a few parameters that allow us to achieve diverse structures and thus build different weapons. Table II describes all these additional parameters that control the number and relative placement of the weapon's features.

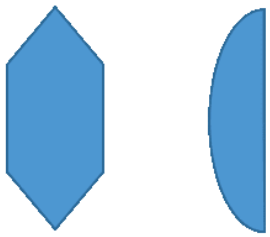


Fig. 4. Two examples of diamond sections. The left section has a curvature of 0, a symmetry ratio of 1 and a diamond ratio of 0.5. The right section has a curvature of 1, a symmetry ratio of 0 and a diamond ratio of 0.

Blade Parameters

Name	Type	Description
Blade Length	float	Length of the blade
Blade Width	float	Width of the blade
Blade Spine Taper	float	The distance between the swords center point at the hilt and at the point. Determines the angle alongside the swords spine.
Blade Spine Curvature	float	Determines the curvature of the sword, creating scimitar-like shapes.
Blade Spine Wave Amplitude	float	Determines the amplitude of waves in the blade alongside the swords length, producing kris-like shapes.
Blade Spine Wave Frequency	float	Determines the frequency of waves in the blade alongside the swords length, producing kris-like shapes.
Blade Spine End Curvature	float	An additional parameter that curves the point of the blade to produce.
Blade Point–Edge Ratio	float	The ratio between the points length and the total blades length. This parameter determines how much of the blade is considered part of the point, and how much part of the edge.
Blade Edge Taper	float	Tapering angle of the blades edge.
Blade Edge Curvature	float	Curvature of the blades edge.
Blade Edge Wave Amplitude	float	The amplitude of waves in the blades taper.
Blade Edge Wave Frequency	float	The frequency of waves in the blades taper.
Blade Point Curvature	float	Curvature of the blades point (notice that the taper angle is automatically computed in order to achieve a pointed tip).
Blade Section Type	integer	Defines the general shape of the blades section (see Figure 3).
Blade Section Width Ratio	float	Normalized thickness of the grips section in respect to the Blade Width.
Blade Section Diamond Ratio	float	Normalized ratio of diamond shape (square) versus simple triangle shape (see Figure 3).
Blade Section Curvature Ratio	float	Curvature of the section segments (see Figure 4).
Blade Section Symmetry Ratio	float	Normalized ratio of symmetry of the blades section. A value of 0 defines a single-edged blade, while a value of 1 defines a perfectly symmetrical dual-edged blade (see Figure 4).

Guard Parameters

Name	Type	Description
Guard Section Type	integer	Defines the general shape of the guards section (see Figure 3).
Guard Length	float	Length of the guard (to the sides).
Guard Height	float	Height of the guard (the thickness alongside the swords length).
Guard Width	float	Width of the guard.
Guard Taper Tapering	float	Tapering of the guards taper.
Guard Spine Tapering	float	Tapering of the guards spine.
Guard Spine Curvature	float	Curvature of the guards spine.
Guard Spine Left Curvature	float	Curvature of the left point of the spine.
Guard Spine Right Curvature	float	Curvature of the right point of the spine.

Grip Parameters

Name	Type	Description
Grip Section Type	integer	Defines the general shape of the grips section (see Figure 3).
Grip Length	float	Length of the grip.
Grip Width	float	Width of the grip.
Grip Taper Curvature	float	Curvature of the grips taper.
Grip Taper Tapering	float	Tapering of the grips taper.
Grip Section Width Ratio	float	Normalized thickness of the grips section in respect to the Grip Width.

Pommel Parameters

Name	Type	Description
Pommel Section Type	integer	Defines the general shape of the pommels section (see Figure 3).
Pommel Length	float	Length of the pommel.
Pommel Width	float	Width of the pommel.
Grip Taper Curvature	float	Curvature of the grips taper.
Pommel Section Width Ratio	float	Normalized thickness of the pommels section in respect to the Pommel Width.

TABLE I. ALL THE PARAMETERS THAT DETERMINE THE SHAPE OF THE DIFFERENT SWORD PARTS.

Name	Type	Description
Guard Enabled	boolean	Is a guard added to this weapon?
Pommel Enabled	boolean	Is a pommel added to this weapon?
Axe-like	boolean	The blade is on the side of the grip, instead of on the top of the grip, similarly to an axe.
Scythe-like	boolean	If Axe-like is enabled, the blade is also rotated to protrude from the sides instead of pointing upwards, like a scythe.
Double Headed	boolean	If Axe-like is enabled, the blade is duplicated on the other side of the grip to form a double headed weapon.
Swallow-like	boolean	The blade is duplicated on the bottom part of the grip. Note that this can be combined with the previous parameters to form a weapon with 4 blades.
Blade Position Ratio	float	If Axe-like is enabled, this parameter determines the distance alongside the grip, from its tip, where the blade is placed at. This is normalized along the grips length.

TABLE II. ADDITIONAL PARAMETERS THAT DEFINE THE OVERALL STRUCTURE OF THE WEAPON.

IV. INTERACTIVE SWORD GENERATOR

Volcano consists of three major components: (i) the *user interface* that allows the interaction with the user and triggers the generation of the swords; (ii) the *evolutionary algorithm*, that performs the search in the parameters space of the sword; (iii) the *procedural backend*, that generates the 3D models of the swords from the parameters identified by the *evolutionary algorithm*. In this section we briefly describe how these three components work.

A. User Interface

Volcano was intended to be used by a single user at once and it is based on the idea that the content is generated within a *user session*. Whenever a user wants to generate one or more swords, he has to start a new *user session* through the user interface (UI) and a set of N swords is immediately generated. The swords in this set, called *current set*, are rendered and displayed by the UI along with an additional set of swords, called *starred set* (see a screenshot of the UI in Figure 5). The *starred set* is empty when a new *user session* starts and cannot contain more than S swords. During a *user session* it is possible to perform the following actions:

Fig. 5. A screenshot of the user interface of *Volcano*.

- **star** any of the sword in the *current set*, adding it to the *starred set* (the action is possible only when the *stared set* does not contain already S swords);
- **unstar** any of the sword in the *starred set*, removing it from the *starred set*;
- **select** one or more swords either in the *current set* or in the *starred set*;
- **export** any of the sword in the current set as an *fbx* model;
- **adjust** the *randomness level*, a parameter that controls the stochasticity of the generation process, choosing among 7 levels ({`extremely low`, `very low`, `quite low`, `moderate`, `quite high`, `very high`, `extremely high`}) encoded as integer values from -3 to +3;
- **generate** a new set of swords based on the currently selected swords;
- **stop** the current *user session* and start from scratch a new one, with the desired parameters setting;
- **quit** the interactive sword generator.

Before starting a new *user session*, it is also possible to set a few parameters that affect either the UI or the evolutionary algorithm used to generate the swords. In particular, the user can set the size of the *current set* (N) and the size of the *starred set* (S). The user can also set the parameters μ and σ of the `mutation` operator of the evolutionary algorithm (see Section IV-B). Finally, he can choose a $template$ from a list (e.g., basic swords, rapier, knife, axe, katana, etc.) to add some constraints on the type of swords that can be generated by the system.

B. Evolutionary Algorithm

The evolutionary algorithm used in *Volcano* to generate the parameters of the swords is a slightly modified version of a simple genetic algorithm, where the selection process is directly controlled by the user. Accordingly, in *Volcano* the underlying evolutionary algorithm is triggered only by some specific actions performed through the user interface: the request to start a new *user session* and the request to generate a new *current set*. In the following we describe how the evolutionary algorithm works in *Volcano* by illustrating the most important procedures.

Algorithm 1 Start a new user session.

```
1: procedure START(N, template, μ, σ)
2:     for i ← 1, N do
3:         current[i] ← init(template, μ, σ)   ▷ Generate a parameter vector
4:     end for
5:     return current
6: end procedure
```

START: this procedure (see Algorithm 1) is called to generate the swords' parameters of the *current set* when a new *user session* begins. It receives four input parameters: N, the size of the *current set*; μ and σ, that are respectively the probability and the *intensity* of the mutation operator; $template$, a vector that defines the standard values of the sword parameters. Through the $template$ parameter, the user can focus on the generation of a specific *class* of swords, such as *classic swords*, *knifes*, *rapiers*, *katanas*, *axes*, etc; if the user does not want to choose a specific class of swords, the procedure will be called setting the template parameter to **None**. The procedure returns $current$, which contains the parameters vectors of N swords to render and to display as the *current set*.

Algorithm 2 Generate a random parameters vector of a sword.

```
1: procedure INIT(template, μ, σ)
2:     param ← new parameters vector
3:     if template is None then
4:         for i ← 1, length(param) do
5:             param[i] ← U(MIN_i, MAX_i)   ▷ Generate a value in the range of i-th parameter with a uniform distribution
6:         end for
7:     else
8:         for i ← 1, length(param) do
9:             param[i] ← template[i]   ▷ Set i-th parameter to the default value of template
10:        end for
11:        mutate(param, μ, σ)   ▷ Apply mutation operator to parameters vector
12:    end if
13:    return param
14: end procedure
```

INIT: this procedure (see Algorithm 2) generates a random parameters vector of a sword. It receives three parameters: μ and σ, that are respectively the probability and the *intensity* of the mutation operator; $template$, a vector that defines the standard values of the sword parameters. If $template$ is **None**, each parameter of the sword is generated using a uniform distribution (line 5 in Algorithm 2, where MIN_i and MAX_i are respectively the lowest and the highest value of the i-th parameter). Otherwise, each parameter is set to the default value defined by the $template$ provided and the MUTATE

operator is applied to the resulting vector. At the end, the generated vector of parameters, $param$, is returned.

Algorithm 3 Generate a new current set.

1: **procedure** EVOLVE(N, $selection$, $template$, μ, σ, ρ)
2: $\mu \leftarrow \mu \cdot k^{\rho}$ ▷ Adjust μ and σ based on randomness level ρ
3: $\sigma \leftarrow \sigma \cdot k^{\rho}$
4: **for** $i \leftarrow 1$, N **do** ▷ Create a new $current$ set of swords based on $selection$
5: **if** $selection$ is **empty then** ▷ If user did not select any sword, a new random set of swords is generated
6: $current[i] \leftarrow init(template, \mu, \sigma)$
7: **else** ▷ Else the new set is based on selections
8: $parent_1 \leftarrow$ random element from $selection$
9: $parent_2 \leftarrow$ random element from $selection$
10: $current[i] \leftarrow crossover(parent_1, parent_2)$
11: $current[i] \leftarrow mutate(current[i], \mu, \sigma)$
12: **end if**
13: **end for**
14: **return** $current$
15: **end procedure**

EVOLVE: this procedure (see Algorithm 3) is called as soon as a request to generate a new *current set* of swords is received from the user interface. The procedure receives as input the parameters N, μ, σ, and $template$, described before. In addition, it receives as input $selection$, a set that contains the parameters vectors of all the swords selected by the user in the UI before requesting the generation of a new *current set*. Finally, the procedure receives as input also the parameter ρ, that encodes the desired *randomness level*: the seven possible values introduced in Section IV-A are mapped to an integer value in the range $[-3, +3]$ (e.g., `extremely low` is mapped to $\rho = -3$, `moderate` is mapped to $\rho = 0$, and `extremely high` is mapped to $\rho = +3$).

The EVOLVE procedure works as follows. At the beginning (see line 2– 3 of Algorithm 3) the mutation parameters μ and σ are adjusted by multiplying them to K^{ρ}, where K is a constant[5]; accordingly a *randomness level* below `moderate` ($\rho < 0$) would result in decreasing the values of μ and σ, while a level above `moderate` would result in increasing them [6]. Then, a new set of N parameters vectors is generated. If the user did not select any swords in the *current set*, i.e., $selection$ is an empty set, the new parameters vectors are generated from scratch by using the INIT procedure (line 6 in Algorithm 3). Otherwise, the new parameters vectors are generated as follows (line 8–11 in Algorithm 3): at first, each one of the new parameters vector is generated as the result of the CROSSOVER operator applied to two *parent* vectors, randomly chosen from $selection$; finally, MUTATE operator is applied to this parameters vector just generated. All the parameters vectors so generated are included in *current* that is returned by the procedure.

MUTATE: this procedure (see Algorithm 4) implements a standard mutation operator; it has three input arguments: $param$ is the parameters vector to mutate; μ and σ are respectively the probability and the *intensity* of the mutation. Each parameter of the sword is mutated with probability μ (line 3 in Algorithm 4). Depending on the parameter, a different type of mutation is applied: if the parameter is *boolean* a flip mutation [22] is applied (line 5 in Algorithm 4); if it is an *integer*, uniform mutation [22] is applied (line 7 in Algorithm 4) with an interval size computed as $\sigma \cdot (MAX_i - MIN_i)$, where MAX_i and MIN_i are respectively the lowest and the highest values of the i-th parameter; if it is a *float*, gaussian mutation [22] is applied (line 10 in Algorithm 4) with standard deviation computed as $\sigma \cdot (MAX_i - MIN_i)/6$.

Algorithm 4 Mutate operator.

1: **procedure** MUTATE($param$, μ, σ)
2: **for** $i \leftarrow 1$, $length(param)$ **do** ▷ Each parameter is mutated with probability μ
3: **if** $rand() < \mu$ **then**
4: **if** $param[i]$ is boolean **then** ▷ Flip mutation is applied to boolean parameters
5: $param[i] \leftarrow notparam[i]$
6: **else if** $param[i]$ is integer **then** ▷ Uniform mutation is applied to integer parameters
7: $\delta \leftarrow (MAX - MIN) \cdot \sigma/2$
8: $param[i] \leftarrow U_{int}(param[i] - \delta, param[i] + \delta)$
9: **else** ▷ Gaussian mutation is applied to float parameters
10: $param[i] \leftarrow N(param[i], (MAX - MIN) \cdot \sigma/6$
11: **end if**
12: **end if**
13: **end for**
14: **end procedure**

Algorithm 5 Crossover operator.

1: **procedure** CROSSOVER($parent_1$, $parent_2$)
2: $cut \leftarrow U_{int}(2, length(parent_1))$ ▷ Chose the cutting point for crossover.
3: **for** $i \leftarrow 1$, $length(parent_1)$ **do**
4: **if** $i < cut$ **then**
5: $param[i] \leftarrow parent_1[i]$
6: **else**
7: $param[i] \leftarrow parent_2[i]$
8: **end if**
9: **end for**
10: **return** $param$
11: **end procedure**

CROSSOVER: this procedure (see Algorithm 5) implements a simple single point crossover. The operator receives as input two parameters vectors, $parent_1$ and $parent_2$, that will be *mixed* together to generate a new sword. First (see line 2 in Algorithm 5), a *cut point* is chosen by selecting a random position in the vector (except the first one). Then, a new parameters vectors, $param$ is initialized as follows: each element of $param$ in a position before the *cut point* is set to the the corresponding element of $parent_1$; instead, from the *cut point* position to the end of the vector, each element of $param$ is set to the corresponding element of $parent_2$. The

[5]We set K to 1.35 based on an experimental analysis.

[6]We decided to introduce ρ instead of letting the users to adjust directly μ and σ to keep the user interaction as simple as possible.

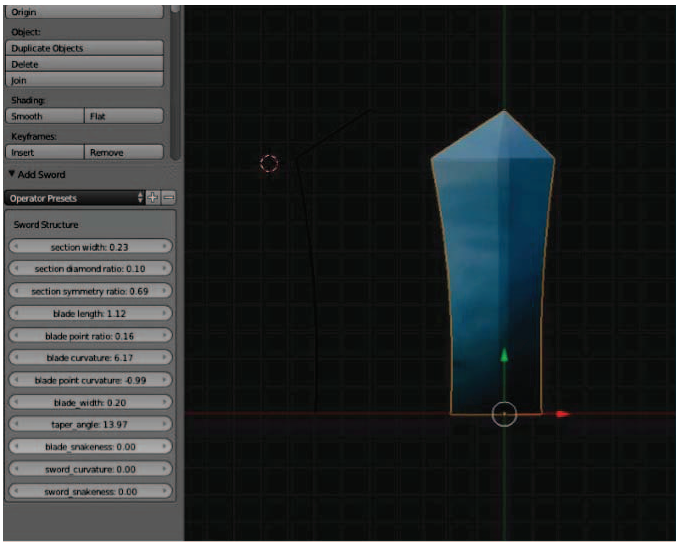


Fig. 6. A blade created with our blender plugin. On the left, the parameters can be altered, and on the right a simple render of the resulting blade is shown.

Fig. 7. A set of our procedurally generated weapons shown in real-time as virtual physical objects inside a Unity3D scene.

resulting parameters vector, $param$, is finally returned by the operator.

C. Procedural Backend

The procedural backend of *Volcano* is the component in charge of generating the actual 3D models of the swords as well as rendering them to images so that they can be displayed in the UI. In particular, we implemented the backend as a plugin for the Blender3D suite [7] that can be called either from the evolutionary algorithm or from the UI to generate and/or to render a 3D model from a given parameters vector. When the plugin is called, the parameters vector (see Section III for a detailed description of the parameters) is used as input to create three 2D polylines for each weapon part (the spine, taper, and section curve). The polylines are then used as input to a tapering algorithm, which uses the taper curve and the section curve to extrude the spine curve and thus achieve a 3D representation. Thus, we convert the curve representation to 3D meshes and generate both normals and UV coordinates, we assign textures and materials to the various weapon parts, we define lights and a centered camera view, and we render the result to an image file. It is also possible to export the resulting mesh as .obj or .fbx files alongside the resulting baked textures to use them as game assets (see an example in Figure 7) or to import them in other modeling tools.

Finally, using Blender's python API, we also developed a GUI-based plugin that integrates with Blender to provide a visual user interface that allows manual modification of all sword parameters through sliders paired with a real-time update of a sword mesh, rendered in the 3D viewport (see Figure 6).

V. User Study

To assess the usability of *Volcano*, we carried out two different user studies. The first study was performed during an Open Day at our university and consisted of 22 user sessions with 22 distinct users (18-22 years old student, mostly male). Before starting each session, we collected few information about the users (i.e., age, gender, gaming habits, etc.) and briefly explained them how the UI of *Volcano* works. Then, we left them free to *play* with the tool for as long as they want. Finally, we asked them three simple questions:

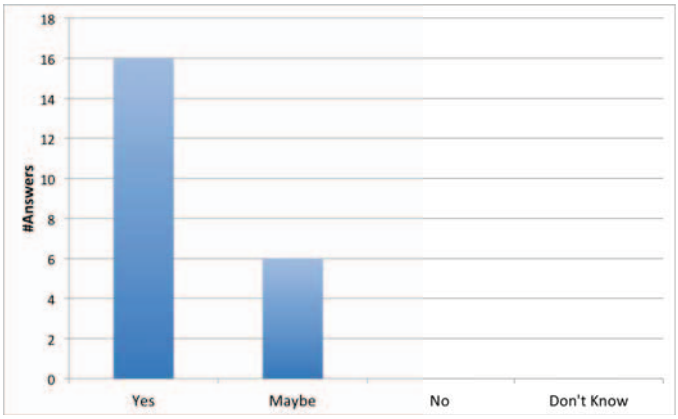
- Q1. Would you use a tool like *Volcano* to create contents for the games you usually play?
- Q2. Are you happy with the swords generated with *Volcano* during your session?
- Q3. Were the swords generated by *Volcano* consistent with your choices (i.e., the selected swords)?

Each user had to answer choosing among a list of options: {Yes, Maybe, No, Don't know} were the possible answers for the first questions, while {Definitely yes, More yes than no, More no than yes, Definitely not} were the possible answers for the second and third questions. Results (see Figure 8) suggest that players actually like the idea of generating their own game content. In fact, Figure 8a shows that 16 out of 22 users would use tools like *Volcano* (i.e., they answered *Yes*) and also the remaining 6 users might give it a try (i.e., they answered *Maybe*). When it comes to evaluate the quality of generated swords, the performance of *Volcano* was pretty solid; in fact, none of the users reported a negative answer, i.e., *more no than yes* or *definitely not*; the most common answer was *more yes than no* and 8 users answered *definitely yes* (see Figure 8b) Similar results were obtained also about the capabilities of *Volcano* to generate swords that are consistent with the users preference (see Figure 8c): 9 users answered *definitely yes*, 12 users answered *more yes than no* and only one user answered *more no than yes*.

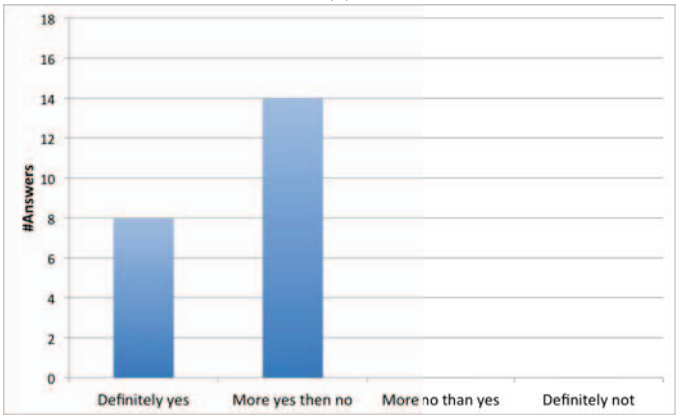
Then, we analyzed how the users interacted with *Volcano* to generate the swords. First of all, the smallest number of swords' sets generated per user session was 5, the largest one was 26, and the median number of generated sets was 9 per session — suggesting a rather good user engagement. Figure 9 shows how the average number of selected swords and of starred ones changes during the user session. Despite, the size of collected dataset is rather small [8], it is still possible

[7] https://www.blender.org/

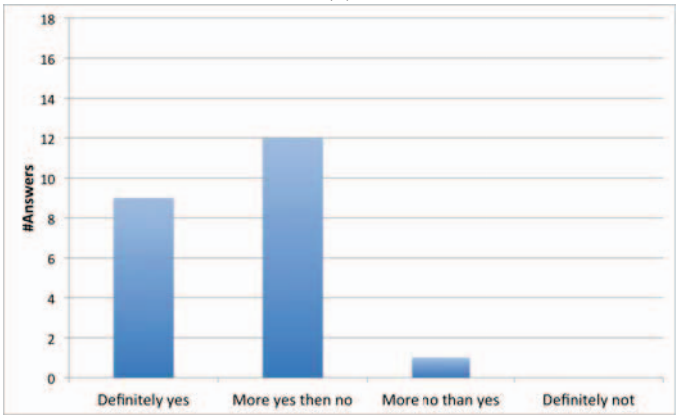
[8] Please notice, that in Figure 9 we show the statistics only for the first 9 generated sets, i.e., the median number of sets generated by the users.



(a)



(b)

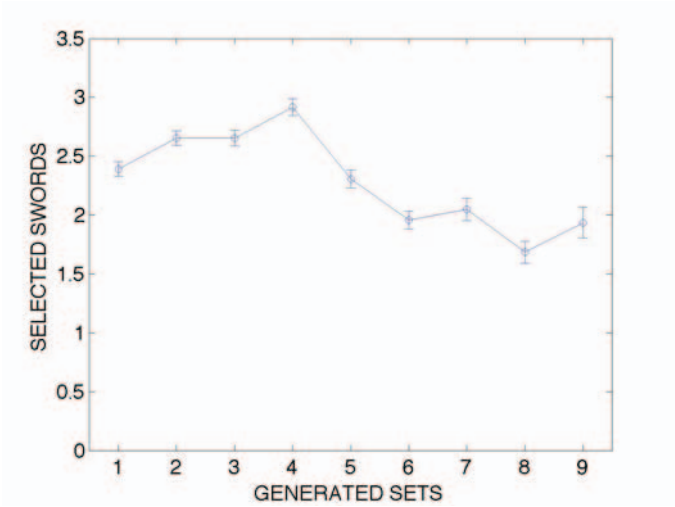


(c)

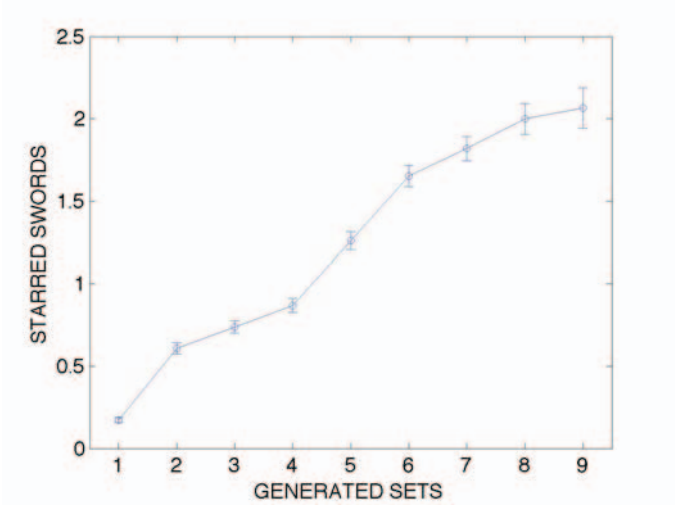
Fig. 8. Answers provided by the users to Q1 (a), Q2 (b), and Q3 (c).

to see some trends. Figure 9a shows that the average number of selected swords slightly increases at the beginning and then decreases as more sets of swords are generated. Perhaps, the users starts by exploring more possibilities but soon focus on just one or two swords they like more. More interestingly, the number of starred swords (Figure 9b) keeps increasing with the generation of more sets of swords. This might suggest that the larger the number of swords generated by the tool, the better is the quality of the swords generated.

The users provided also useful suggestions that allowed us to make some improvements to the UI. In particular, a very common suggestion was to add something that could allow the



(a)



(b)

Fig. 9. User interactions with *Volcano*: (a) number of selected swords per generation and (b) number of starred swords per generation. The plot reports the average values along with the standard error.

user to control the degree of variety in the generated swords. Based on this suggestion, we introduced the *randomness level* control described in Section IV-A, that was not included in the tool at the time of this first test.

Few weeks later, we performed a second user study at an independent game developers festival. The test involved 8 users among developers and game designers. The feedback was similar to the one received from the previous test: all the users evaluated positively the generated swords (we collected 5 *definitely yes* and 3 *more yes than no* answers to Q2); similarly, the evaluation of the consistency with respect to the user actions was very good (we collected 6 *definitely yes* and 2 *more yes than no* answers to Q3). Moreover, when asked whether they would use a tool like *Volcano* for creating the content for their own games [9], the answers were promising (5 answered *yes* and 3 answered *maybe*). In particular, designers and developers pointed out that a tool like *Volcano* might be

[9]In this second user study question Q1 was modified as: "Would you use a tool like *Volcano* to create contents for the games you develop?"

valuable even if not capable to generate assets that can be directly used in games. This kind of tools might be useful to perform a quick exploration of the design space and to generate a set of preliminary assets that a 3D artist can further develop with a modeling software.

Finally, we collected a lot of interesting suggestions to improve *Volcano*, such as supporting modular content generation (e.g., working only on the blade while keeping fixed the grip and the guard), clustering the generated set based on visual similarity, and allowing to browse back the *history* of generated swords.

VI. Conclusion

In this work we introduced *Volcano*, a tool for procedural generation of swords that learns the user's preferences: instead of requiring a long session of trial-and-error parameters setting, *Volcano* exploits interactive evolution [12] to generate new swords based only on a selection of the ones previously generated. Therefore, it allows to perform a *visual exploration* of the design space and to generate novel game content with a very limited effort.

We implemented a prototype of *Volcano*, which includes a plugin for *Blender* to actually generate usable 3D models of the generated swords. We also tested our prototype with 30 users, among students and game developers. The result of these tests was very encouraging: users reported to be rather happy concerning both their interaction with the tool and the generated swords; moreover most of the users expressed high interest for the possibility of using a tool like *Volcano* to create game content either as players (i.e., to create user-generated content) or as designers. (i.e., to simplify the development process).

Future works include improving the user interface of *Volcano* following some suggestions received from the user tests (e.g., adding a browsable history, clustering generated swords based on visual similarity, etc.), support the generation of modular game content (i.e., allows the generation of only some parts of the swords at once), and implementing an on-line system to collect more user experiences.

References

[1] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation," in *EvoApplications (1)*, ser. Lecture Notes in Computer Science, C. D. Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcázar, C. K. Goh, J. J. M. Guervós, F. Neri, M. Preuss, J. Togelius, and G. N. Yannakakis, Eds., vol. 6024. Springer, 2010, pp. 141–150.

[2] C. Pedersen, J. Togelius, and G. N. Yannakakis, "Optimization of platform game levels for player experience," in *AIIDE*, C. Darken and G. M. Youngblood, Eds. The AAAI Press, 2009.

[3] K. Compton and M. Mateas, "Procedural level design for platform games," in *AIIDE*, J. E. Laird and J. Schaeffer, Eds. The AAAI Press, 2006, pp. 109–111.

[4] N. Shaker, G. N. Yannakakis, and J. Togelius, "Towards automatic personalized content generation for platform games," in *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010, October 11-13, 2010, Stanford, California, USA*, G. M. Youngblood and V. Bulitko, Eds. The AAAI Press, 2010. [Online]. Available: http://aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/view/2135

[5] D. Loiacono, L. Cardamone, and P. L. Lanzi, "Automatic track generation for high-end racing games using evolutionary computation," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 245 –259, sept. 2011.

[6] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Interactive evolution for the procedural generation of tracks in a high-end racing game," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 395–402. [Online]. Available: http://doi.acm.org/10.1145/2001576.2001631

[7] J. Dormans and S. Bakkes, "Generating missions and spaces for adaptable play experiences," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 216–228, 2011.

[8] J. Dormans, "Adventures in level design: generating missions and spaces for action adventure games," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 1.

[9] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis, "Multiobjective exploration of the starcraft map space," in *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2010, pp. 265–272.

[10] L. Cardamone, G. N. Yannakakis, J. Togelius, and P. L. Lanzi, "Evolving interesting maps for a first person shooter," in *Proceedings of the 2011 international conference on Applications of evolutionary computation - Volume Part I*, ser. EvoApplications'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 63–72. [Online]. Available: http://dl.acm.org/citation.cfm?id=2008402.2008411

[11] P. L. Lanzi, D. Loiacono, and R. Stucchi, "Evolving maps for match balancing in first person shooters," in *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, August 26-29, 2014*. IEEE, 2014, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/CIG.2014.6932901

[12] K. Sims, "Interactive evolution of dynamical systems," in *Toward a practice of autonomous systems: Proceedings of the first European conference on artificial life*, 1992, pp. 171–178.

[13] H.-S. Kim and S.-B. Cho, "Application of interactive genetic algorithm to fashion design," *Engineering Applications of Artificial Intelligence*, vol. 13, no. 6, pp. 635 – 644, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/B6V2M-41TN609-3/2/f36f199cd3de20d072879d123e3a04ff

[14] A. Brintrup, J. Ramsden, H. Takagi, and A. Tiwari, "Ergonomic chair design by fusing qualitative and quantitative criteria using interactive genetic algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 12, no. 3, pp. 343 –354, Jun. 2008.

[15] P. Walsh and P. Gade, "Terrain generation using an interactive genetic algorithm," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, Jul. 2010, pp. 1 –7.

[16] J. Secretan, N. Beato, D. B. D'Ambrosio, A. Rodriguez, A. Campbell, J. T. Folsom-Kovarik, and K. O. Stanley, "Picbreeder: A case study in collaborative evolutionary exploration of design space," *Evolutionary Computation*, vol. 19, no. 3, pp. 373–403, 2011. [Online]. Available: http://dx.doi.org/10.1162/EVCO_a_00030

[17] B. Xu, S. Wang, and X. Li, "An emotional harmony generation system," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, Jul. 2010, pp. 1 –7.

[18] E. J. Hastings, R. K. Guha, , and K. O. Stanley, "Automatic content generation in the galactic arms race video game," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 245–263, 2009.

[19] S. Risi, J. Lehman, D. B. D'Ambrosio, R. Hall, and K. O. Stanley, "Combining search-based procedural content generation and social gaming in the petalz video game," in *AIIDE*, M. Riedl and G. Sukthankar, Eds. The AAAI Press, 2012.

[20] L. Cardamone, P. L. Lanzi, and D. Loiacono, "Trackgen: An interactive track generator for TORCS and speed-dreams," *Appl. Soft Comput.*, vol. 28, pp. 550–558, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.asoc.2014.11.010

[21] [Online]. Available: http://www.myarmoury.com/feature_properties.html

[22] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Lulu, 2013, available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.