Evolving Planets

Giovanni Bollati

July 2025

Contents

1 Introduction

The project consists in the procedural generation of arbitrarily shaped, three-dimensional closed surfaces, which thematically represent planets. The generated planets will become assets for a video game where the player has to safely land a spaceship onto the planets; the application game does not belong to the project, but exists as reasoning for the optimization direction.
The generated planets should be varied in shape (the player should perceive them as different planets), and they should offer a certain degree of difficulty. The difficulty is related to how much the gravity force induced by the planet diverges from the player's expectations; for example, the player could expect gravity force to be perpendicular to the surface, like on the Earth, however, the non-spherical shape of the generated planets would induce a non-perpendicular gravitational field. In addition, even if the player anticipates the gravity force direction, the spaceship cannot land if the direction is not perpendicular; therefore, the player must look for a plateau.
The generation consists of two stages: initialization, which preliminarily mutates spherical planets in a completely stochastic manner, and optimization, which is driven by an evolutionary algorithm. While the initialization stage is mandatory, the optimization one is optional, because initialization already produces functional individuals. Evolutionary algorithms implement principles of natural evolution and selection to update a population of individuals while optimizing a fitness value, defined for each individual. This class of algorithms is chosen because they easily provide diversity in addition to optimisation.
The generation process involves the choice of a genotype, which is the underlying representation of the individuals: the chosen representation is the B-Spline formalism, which allows local but meaningful control over the planet shape, while granting surface properties such as smoothness across the random manipulations. Once generated, the planets can be exported as 3D models, ready for becoming assets of the target video game. If the optimization stage is included, then the planets must be generated during the development stage and included in the game as assets before shipment, because of performance concerns, which will be later discussed in the results chapter; if the optimization stage is skipped, initialization by itself could be executed in real-time during the game execution, for example during a loading phase.

2 Method

In the following chapters representation and the generation algorithms will be discussed; the representation consists of a phenotype, which is the object by itself, a genotype, which is the internal formulation, and a mapping function that generates a phenotype from a given genotype.

2.1 Phenotype

The evolving individuals are curved, tridimensional planets; their density is homogeneous; thus, they are entirely represented by their surface. A planet is a solid body; therefore, its surface must be closed and cannot autointersect. The planets should meet certain quality requirements regarding appearance: the surface should be smooth and seamless. In addition, the phenotype should be easily manipulable by the evolutionary algorithm without breaking smoothness or validity.

2.2 Genotype

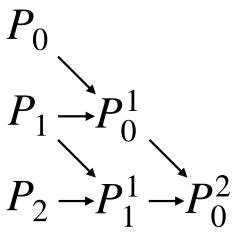
The planets must be represented by a triangle mesh for the fitness computation and for graphic rendering purposes. However, while a triangle mesh can effectively approximate a curved surface by tessellating the latter, directly and arbitrarily manipulating the triangle mesh would easily violate the phenotype's constraints, leading to illegal shapes. Therefore, the chosen genotype is the B-Spline formalism: it can represent curved surfaces by defining a coarse-grained polyhedron; its vertices do not belong to the surface, but they are interpolated through a convex combination, weighted by polynomial functions called basis; the surface can be easily manipulated by moving the vertices of the control polyhedron. To represent a closed surface, the B-Spline is periodic on one axis, while on the other axis it converges to two poles. The B-Spline formalism grants $C2$ continuity whichever manipulation is done, as long as the polyhedron represents a legal surface, satisfying the phenotype's smoothness and ease of manipulation requirements. A specific pole management grants $C2$ continuity even there.In the following chapters the genotype definition will be described in greater detail. For simplicity, the reasoning about the B-Spline formalism will be presented for curves instead of surfaces.

2.2.1 Bèzier Curve

A Bezier curve is the image of a function $P(t) : [0, 1] \to E^k$; its value is derived from the recursive linear interpolation of the vertices of its control polygon as described by the De Casteljau algorithm: given a collection of vertices $P_0 \dots P_n$ where n is the degree of the curve, these are pair-wisely linearly interpolated by the parameter $t \in [0, 1]$, repeating recursively on the obtained values until we have a single value, which is the image of the curve for t. The algorithm is visually described in figure 1

The algorithm can be made explicit by the Bernstein polynomials, so that the curve is defined as the Bernstein-Bezier aproximation of the control polygon:

$$P(t) = \sum_{i=0}^{n} P_i B_{i,n}(t), \text{ where } B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$



$$P_i^r = tP_i^{r-1} + (1-t)P_{i+1}^{r-1}$$

Figure 1: De Casteljau alogirithm

2.2.2 Piecewise Curve

To represent a complex curve, a high degree Bezier curve could be used; however, a Bezier curve has total support, which means that each vertex influences the whole curve, thus modifying the position of a vertex of the control polygon can lead to undesired changes to other portions of the curve. In order to have local control over the shape, we can subdivide the domain by defining a vector of knots $u_0, \ldots, u_k$ and the curves $P_0, \ldots P_{k-1}$, where P_i is defined over $[u_i, u_{i+1}]$. This formulation defines a piecewise curve, also said spline.
Different classes of continuity can be enforced: C^0 requires $P_i(u_i{+}1) = P_{i+1}(u_i{+}1),\ \forall\ P_i \mid\ i < k-1$; this constraint makes each piece attached at its end point to the next piece. C^1 and C^2 require that the curve is differentiable once and twice, respectively, in the glueing points. Figure 2 shows a piecewise curve with C^0 continuity. However, freely manipulating the control polygons can break continuity where the curves are glued, while a single Bezier curve is infinitely differentiable no matter how the control polygon is mutated. In the next paragraph the blossom notation will be discussed in order to better present the B-Spline formalism.

2.2.3 Blossoms

A blossom $b[t_1, \ldots, t_n]$ is a function that satisfies the following properties:

- Symmetry:
$$b[t_1, \ldots, t_n] = b[\pi(t_1, \ldots, t_n)]$$

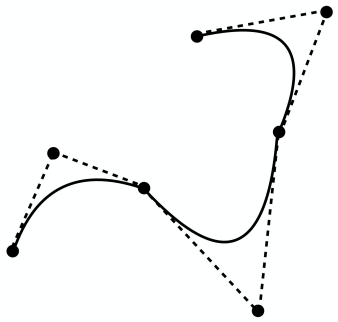


Figure 2: Piecewise curve with C^0 continuity.

- Multiaffinity:

$$b[(\alpha r + \beta s), *] = \alpha b[r, *] + \beta b[s, *];\ \alpha + \beta = 1$$

- Diagonality: if $t = t_1 = t_2 = \cdots = t_n$, then $b[t, \ldots, t] = b[t^{<n>}]$ is a polynomial curve.

We can easily describe the De Casteljau algorithm by using this notation, as shown in figure 3.

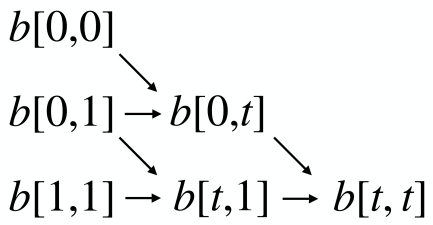


Figure 3: De Casteljau algorithm with blossom notation.

2.2.4 B-Spline

B-Spline is a formalism that allows defining and evaluating a spline while implicitly enforcing a certain class of continuity over the whole domain.
A B-Spline of degree n requires defining a knots vector $u_0, \ldots, u_K$, where $K \geq 2n - 1$; the knots vector identifies a control polygon $d_0, \ldots, d_L$, where $L = K - n + 1$; following the blossoms notation, $d_i = b[u_i, \ldots, u_{i+n-1}]$, i.e. each vertex of the control polygon is identified by a n-tuple of knots. The curve is defined over the interval $[u_n, \ldots, u_{K-n+1}]$, and inside this inverval each non-zero sub-interval identifies a single curve segment of degree n.
The multiplicity of the knot values inside the $[u_n, \ldots, u_{K-n+1}]$ intverval determines the continuity class of the curve: at the knot u_k, the curve is C^{n-r}, where

r is the multiplicity of the knot u_k; the multiplicity of the knots before and after the $[u_n, \ldots, u_{K-n+1}]$ interval determines if the curve passes through the vertices d_0 and d_L: $P[u_n] = d_0$ if $u_0 = \cdots = u_{n-1}$.
A B-Spline can be evaluated by the De Boor algorithm; by using the blossom notation, the De Boor algorithm can be easily interpreted as a generalization of the De Casteljau algorithm. Each $(n+1)$-tuple of vertices from the control polygon determines the curve value for a single segment; given u, the subinterval it belongs to identifies the active vertices of the control polygon; then the De Boor algorithm follows the same procedure of the De Casteljau when using blossom notation: for example, $b[u_0, u_1, u_2]$ and $b[u_1, u_2, u_3]$ are interpolated into $b[u_1, u_2, u]$, where u is expressed as the convex combination of u_0 and u_3 in the same way t was expressed as convex combination of 0 and 1; values are recursively interpolated until $b[u^{<n>}]$ is reached. Like the De Casteljau algorithm, the De Boor algorithm can be made explicit by using the Basis functions, so that

$$P(t) = \sum_{i=0}^{L} d_i N_{i,n}(u)$$

. The basis function is recursively defined:

$$N_l^n(u) = \frac{u - u_{l-1}}{u_{l+n-1} - u_{l-1}} N_l^{n-1}(u) \; + \; \frac{u_{l+n} - u}{u_{l+n} - u_l} N_{l+1}^{n-1}(u),$$

$$\text{with } N_i^0(u) = \begin{cases} 1 & \text{if} \quad u_{i-l} \leq u < u_i \\ 0 & \text{else} \end{cases}$$

The basis function $N_i^n(u)$ is non-zero only for the active vertices of the control polygon: for each sub-interval of the domain, only $n+1$ basis functions are non-zero, thus $n+1$ vertices of the control polygon determine the value of the curve in that domain interval. It is important to note that the number of vertices of the B-Spline control polygon is smaller than the number of vertices of an equal piecewise curve: for example, for three cubic curve segments, we would have 10 vertices for the piecewise, but only 6 for a uniform B-Spline: this apparent reduced degree of freedom is the reason behind the implicit enforcement of continuity.This specific B-Spline formulation is the one presented by *Gerald Farin* in his book *Curves and Surfaces for CAGD* [4].

2.2.5 Surfaces

The reasoning made for curves is valid for surfaces as well: the domain becomes bi-dimensional, and the control polygon becomes a polyhedron. If P is a Beziér surface of degree r for u and s for v, then

$$P(u, v) = \sum_{i=0}^{r} \sum_{j=0}^{s} P[i, j] B_{i,r}(u) B_{j,s}(v)$$

. A Planet is represented by a bicubic uniform B-Spline surface; by default the control polyhedron is 14x14, which results in 11 cubic segments.

2.2.6 Closeness

A B-Spline surface is open by default, as is its domain, like a piece of paper; therefore, in order to have a closed surface, we must make it periodic on the horizontal axis, and make it degenerate vertically, defining two poles. Periodicity is imposed on the u-axis by making the last 3 vertices of each parallel (for the cubic case) equal to the first 3 ones. The poles are implemented by making the first and the last parallels degenerate in one point and imposing that the meridians lie on the control polyhedron for $v = 0$ and $v = 1$ through the multiplicity of the knots vector for the v-axis.

2.2.7 Poles Continuity

The poles consists in degenerate parallels; however, they raise a concern about continuity. C^0 is guaranteed by the multiplicity of the knots vector: every meridian ends in the last vertices, which is the pole. C^1 and C^2 are guaranteed by designing a plateau, which consists of two additional fixed parallels adjacent to each pole; in fact, the first and second derivatives on the vertical axis depend on the last 2 and 3 vertices, respectively; therefore, the last two parallels are circumferences concentric around the pole, with fixed height along the y axis of the 3D image space; this guarantees that each meridian ends on the pole with the same first and second derivatives, making the pole fit the C^2 requirement.

2.2.8 Validation

In order to be valid, a surface must not intersect itself; the test is performed by converting the B-Spline to a collection of triangles by tessellation, and then checking for intersection for each triangle against every other triangle. Every surface manipulation can lead to an illegal shape; therefore, the auto-intersection check must be performed after each modification to the control polyhedron.

2.2.9 Genotype Summary

The genotype defines a surface that is smooth (C^2 continuity granted by the B-Spline formalism), closed (the surface is periodic and degenerate on the poles without breaking continuity) and easily manipulated (the surface can be manipulated by moving the vertices of the control polyhedron without breaking the other requirements, thanks to the B-Spline formalism). Figure 4 shows the complete genotype representation.

2.3 Optimization

An evolutionary algorithm is a stochastic optimization algorithm based on the principles of biological evolution and natural selection. A population of individuals is subject to multiple evolution cycles; each cycle consists of mutation, crossover, and selection. The mutation and crossover phases are responsible for the stochastic evolution of the population, while selection determines the

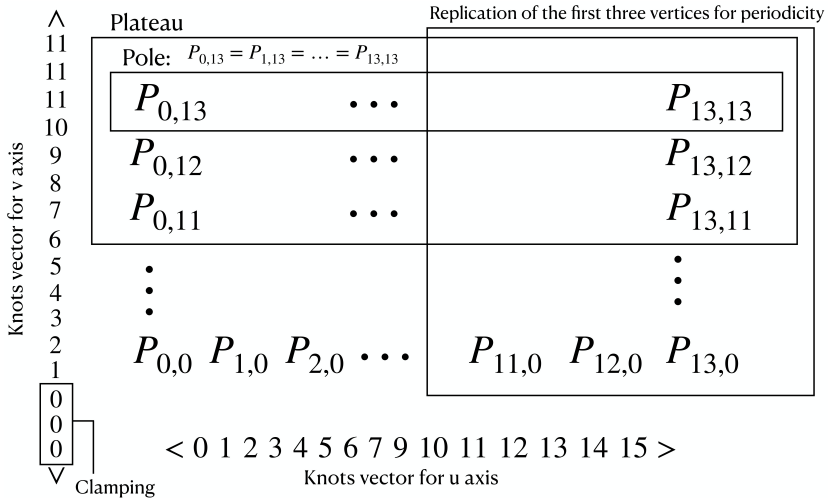


Figure 4: Genotype Final Representation.

survival of the offspring by evaluating each individual through a fitness function. Evolutionary algorithms are said to be metaheuristics, since they produce sub-optimal solutions. In our context, there is no optimal planet, but we can identify planets that are better than others, that is why we start from a random population and progressively improve it by subjecting it to evolution.

2.3.1 Differential Evolution

Differential evolution is a particular type of evolutionary algorithm. The name comes from the implementation of the mutation step: an individual $\overline{x}$ is mutated into $\overline{x}'$ by adding a perturbation vector defined as the difference between two other random individuals, scaled by a scaling factor F: $\overline{x}' = \overline{x} + F\ (\overline{y} - \overline{z})$. The individuals are represented by lists of real values; in our case, the real values are the 3D coordinates of the vertices of the control polyhedrons. Crossover is uniform, which means that each gene of the child is randomly chosen between the first and the second parent based on a probability $Cr \in [0,\ 1]$. Selection is deterministic and elitist, i.e. an individual is never replaced by one with lower fitness. Regarding the variants of differential evolution algorithms, we use DE/rand/1/bin, which is the standard version described above.

2.3.2 Initialisation: Gaussian Terraformation

The algorithm is initialised with a population of completely random planets: each planet is a sphere that is deformed by repeatedly applying a procedure that performs a single random mutation. The procedure chooses a random vertex P from the control polyhedron, a random direction d and a random magnitude value M within a specified range; then, each vertex Q of the control

polyhedron is moved along the chosen direction by an amount proportional to a gaussian distribution, centered in p:

$$Q = Q + e^{\frac{|Q-P|^2}{2\sigma^2}} Md$$

After each single mutation, the auto-intersection check must be performed to prevent illegal shapes. In order to preserve continuity on the poles, the plateau is moved as a single entity.

2.3.3 Mutation

Having a population $P = \langle \overline{x}_1, \ldots, \overline{x}_\mu \rangle$, a mutant vector $M = \langle \overline{v}_1, \ldots, \overline{v}_\mu \rangle$ is created; for each $\overline{v}_i$, three individuals among the population are randomly chosen: $\overline{x}_r$ as the base individual to be mutated, $\overline{x}_s$ and $\overline{x}_t$ to define the perturbation vector.

$$\overline{v}_i = \overline{x}_r + F(\overline{x}_s - \overline{x}_t)$$

Each mutation should be repeated until the auto-intersection test succeeds, within a limit of attemps; if the limit is exceeded, $\overline{v}_i$ is set to $\overline{x}_i$ mutated by multiple gaussian terraformations.

2.3.4 Recombination

Having the population P and the mutant vector M, a trial vector $T = \langle \overline{u}_1, \ldots, \overline{u}_\mu \rangle$ is constructed, where u_i is the child of $\overline{x}_i$ and $\overline{v}_i$; for each gene, Cr, also said mutation rate, is the probability that the gene is taken from $\overline{v}_i$ instead of $\overline{x}_i$. It must be guaranteed that at least one gene is inherited from the mutant parent, so that the child does not replicate the parent from the previous generation. Crossover should be repeated until the auto-intersection test succeeds, within a limit of attempts; if the limit is exceeded, the crossover operation falls back to the continuous crossover: each gene of the child is the average of the parents' genes. Continuous crossover has better chances to succeed; however, if it does not within a certain number of attempts, $\overline{u}_i$ is set to $\overline{v}_i$.

2.3.5 Fitness Function

In the next chapters, we will consider fitness as a value that decreases if optimized: the lower, the better, even if the semantics of the name suggests the opposite. Fitness consists in how much gravity induced by non-spherical planets differs from the player expectations; the lower the fitness, the higher the difference. In particular, we presume that the player expects gravity force to be perpendicular to the surface; therefore, fitness becomes how much gravity force diverges from the surface normal direction (divergence increases if the ftness value decreases, being represented as the dot product of directions). In particular, we define the fitness model as

$$f(P) = \frac{n(P)}{d(P)}$$

$$n(P) = \sum_{u=0}^{k}\sum_{v=0}^{k} \frac{\vec{g}_{u,v,k}}{\|\vec{g}_{u,v,k}\|} \cdot \frac{\vec{n}_{u,v,k}}{\|\vec{n}_{u,v,k}\|} \cdot \frac{\|\vec{g}_{u,v,k}\|}{\max_{u,v}\|\vec{g}_{u,v,k}\|}$$

$$d(P) = \sum_{u=0}^{k}\sum_{v=0}^{k} \frac{\|\vec{g}_{u,v,k}\|}{\max_{u,v}\|\vec{g}_{u,v,k}\|}$$

$$\text{with } \vec{g}_{u,v,k} = \vec{g}(P(\frac{u}{k}, \frac{v}{k})) \text{ and } \vec{n}_{u,v,k} = \vec{n}(P(\frac{u}{k}, \frac{v}{k}))$$

where $P(\frac{u}{k}, \frac{v}{k})$ is one of the k^2 positions sampled on the surface, $\vec{g}$ is the gravity function and $\vec{n}$ is the direction perpendicular to the surface.

This fitness model requires the ability to compute the gravity force induced by the planet at a given location. The computational method is taken from another project described at `github.com/bolla99/gravity_field_sampling`. The surface is tessellated and transformed into a triangle mesh; the triangle mesh is required for the generation of a volumetric representation, which in turn allows the computation of gravity at any point in the space.

2.3.6 Selection

Selection is deterministic and elitist: having P and T, the i-th individual of the next generation is $\overline{x}_i$ is $f(\overline{x}_i) < f(\overline{u}_i)$, $\overline{u}_i$ otherwise. We defined the fitness function as a unary function; however, in order to preserve diversity and prevent premature convergence and similarity between the resulting planets, fitness is corrected by introducing a diversity evaluation:

$$f_c(P, i) = f(P[i]) + F_d \cdot f_d(P)[i]$$

where f_c is the corrected fitness, f the original unary fitness function, $F_d \in [0, 1]$ a scaling factor, and f_d a function that evaluates diversity for a given individual. The diversity function f_d will be discussed in the immigration section.

2.3.7 Computational Concern

The necessity of performing the auto-intersection test after each mutation and crossover raises a significant concern about performance; its execution is a bottleneck, making the other operations negligible. The auto-intersection test has quadratic complexity with respect to the number of triangles; we observed that to make the test reliable, the $(0, 1)\times(0, 1)$ domain should be tessellated by a step of 0.01, leading to two thousand triangles, i.e. four million tests for intersection between two triangles (however, each single triangle-triangle test can be accelerated by heuristics such as a preliminar sphere collision detection). The auto-intersection test must be performed until the operation succeeds; this repetition is limited by a parameter set by the user; if the limit is reached, the operation fails, and the execution goes on. Even if the complexity of the auto-intersection test is linear with respect to the size of the population, the coefficient is very high: $T^2 \cdot limit \cdot 2$, where T is the number of triangles. The fitness function

computation is computationally demanding, too; however, it is linear instead of quadratic with respect to the number of triangles; in addition, the tessellation resolution can be tuned down, since the gravity computation does not require the same precision as the auto-intersection test. The computational difficulties lead to the necessity of keeping the population size at a minimum, which, in turn, causes premature convergence and reduces diversity. These issues are partially solved by designing a particularly aggressive immigration technique, described in the next chapter.

2.3.8 Immigration

Immigration is a technique that consists in replacing some individuals with new ones in order to artificially increase diversity and avoid premature convergence. In our case, a new individual is a completely random planet, such as the ones created during initialization. At the start of each epoch, a certain number of individuals are substituted by immigrants; however, the individuals that are going to be replaced are not randomly chosen. For each pair of planets P and Q, given $P_1, \ldots, P_k$ and $Q_1, \ldots, Q_k$ their control polyhedrons, diversity $d(P,Q) = \frac{1}{k}\sum_{i=1}^{k} distance(P_i, Q_i)$. Given $P = P_1, \ldots, P_n$ a population of planets, D a $n \times n$ matrix where n is the population size and $D[i,j] = d(P_i, P_j)$, we define a vector $v = v_1, \ldots, v_n$ where

$$v_i = (v_i^{(1)}, v_i^{(2)}) = \left(\min_{1 \leq j \leq n} D[i,j], \quad \text{argmin}_{1 \leq j \leq n} D[i,j] \right), \text{ with}$$

$$i \neq v_i^{(2)} \text{ AND } v_{v_i^{(2)}} \neq i$$

The individuals that will be replaced are the ones with the lesser values from the vector v:

$$\text{if } I = argkmin_{1 \leq i \leq n} v_i^{(1)} = I_1, \ldots, I_k, \text{ then } P_{I_1}, \ldots, P_{I_k} \text{ will be replaced}$$

The definition of v grants that each relation of diversity appears only once, i.e. if P_i is the most similar to P_j and vice-versa, v_i will be the diversity value between them, while v_j will be the diversity value of P_j and the most similar one among the others minus P_i. This guarantees that each relation is broken once, i.e. only P_i will be replaced.

2.3.9 Termination

In general, evolutionary algorithms have the anytime property, which means that they can be terminated anytime while still yielding valid solutions, eventually not enough optimized for the user requirements.
The user can choose to set a diversity limit below which the evolution is terminated; diversity computation is based on the technique described in the immigration chapter: for a population P, a vector $d = d_1, \ldots, d_n$ is generated, where d_i is the minimum diversity of the planet P_i with respect to any other planet;

the diversity limit is compared to the average value of d_i from the current population. In addition, the user cannot set two parameters, a number of epochs N and a fitness threshold T: the evolution will terminate if the fitness improvement from the previous generation is less than T for N consecutive epochs.

3 Implementation Notes

The method implementation is available under the MIT license at `https://github.com/bolla99/evolving_planets`. The gravity computation method is described at the following link: `github.com/bolla99/gravity_field_sampling`. The programming language used is *C++*. The following third-party libraries were used:

- *SDL2* [12] for window and input management
- *Metal* [9] for graphic rendering and GPU computing
- *GLM* [6] for linear algebra operations
- *Dear ImGUI* [3] for the graphical user interface
- *Cereal* [5] for serialization
- *Assimp* [1] for mesh loading and export
- *OpenMP* [11] for CPU parallelization

Blender [2] was used to produce the 3D renders shown in the results section; *Numpy* [7] and *Matplotlib* [8] were used for creating the plots shown in the results chapter. In general, the implementation takes great advantage of GPU parallelization; in particular, the most expensive operations that are performed by the evolutionary algorithm are the fitness computation and the auto-intersection test. The fitness computation is bottlenecked by the gravity computation, which requires the following steps:

- surface tessellation to generate the triangle mesh (performed by CPU)
- generation of a volumetric representation, based on parallel lines, from the triangle mesh (performed on GPU, for each ray intersecting the mesh)
- sum of the gravity induced by each line (performed on GPU, for each sample position for which gravity is required)

The auto-intersection check is the most expensive operation; it requires tessellation for converting the B-Spline to a triangle mesh, and then an intersection test is performed for each triangle against every other triangle. It has $O(n^2)$ complexity, where n is the number of triangles; in addition, a very high resolution is required so that the triangle mesh approximates well the B-Spline curved surface, otherwise the test may give false negatives.Performance of the auto-intersection test is greatly improved by an early exit optimization that consists in performing a spherical collision detection between the bounding spheres of the two triangles; this test is way faster than the *Moller-Trumbore* [10] algorithm, which is executed if the collision is detected, and excludes the majority of couples of triangles. In order to preview the planets' appearance, a basic rendering system has been designed.

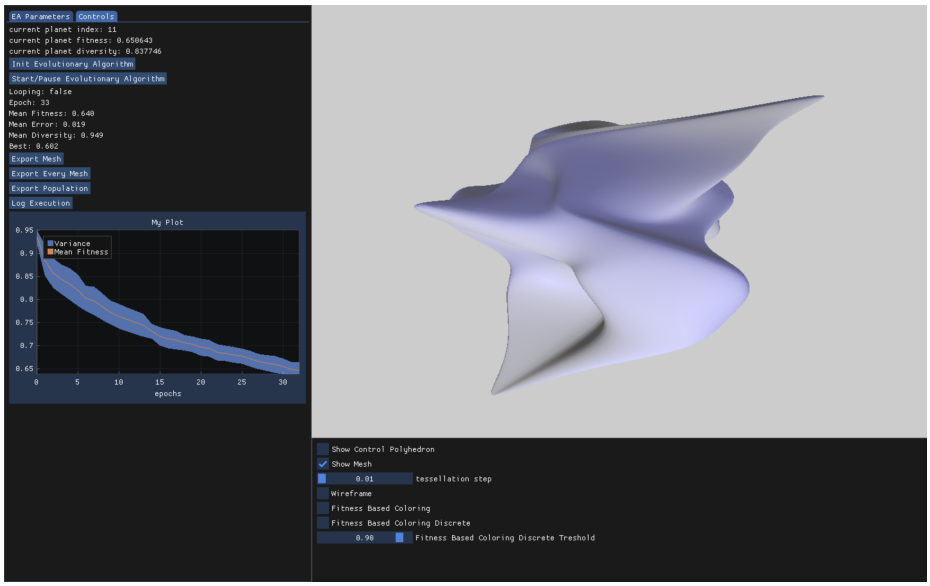


Figure 5: UI of the PCG software.

4 Results

A valid approach for generating a population of planets is the totally random one, which involves the Gaussian terraformation technique described in the previous chapters. The number of mutations can be indefinitely increased till substituting the whole evolution and making initialization the only generation process.

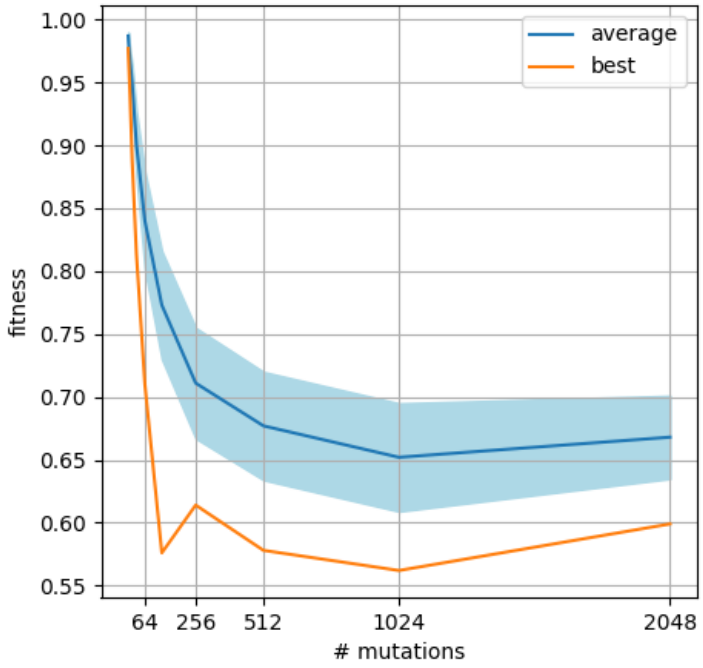


Figure 6: Performance of random approach

The plot at figure 6 shows how fitness varies with the number of mutations. We can see that after a certain number of mutations fitness stops decreasing, with aproximately 0.57 as best reached value. However, this approach has its best advantage in the population diversity: the performed manipulations are completely random, and thus the generated planets result in being substantially different. Time performance is neglectable: even with thousands of mutations, it takes very few seconds (less than ten during our tests) to generate a planet, which is an acceptable duration for a loading phase inside a video game; that is why this approach could be employed in real-time for creating brand new planets as the game is running.
On the other hand, the plot in figure 7 shows a particular execution of the

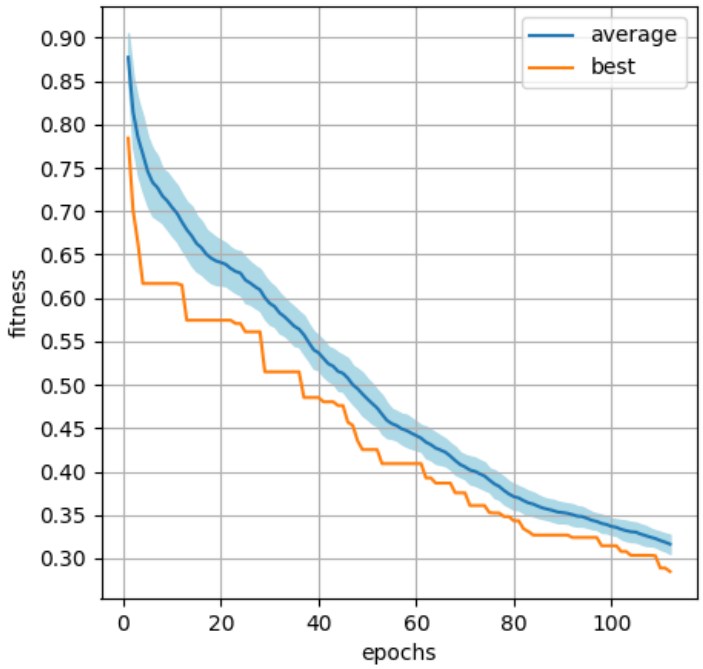


Figure 7: Differential Evolution Approach

evolutionary algorithm; we can observe a great improvement over the random approach, with a reached average fitness of 0.25 and best at 0.194. Analysing the phenotype of the generated planets, we can observe that the resulting population has a very low diversity: as you can see in figure 8, its individuals resemble slightly different versions of the same planet.

The techniques for preserving diversity described in the previous chapters proved to be inadequate: in order to really preserve diversity and generate substantially different planets, the related parameters have to be set at a very aggressive level, so much that they hinder optimization and neutralize the advantage of this approach over the random one; instead, if set at a moderate level, the techniques can help avoiding premature convergence and in fact improving optimization, but they are not able to preserve the level of diversity required to consider the planets different to each other. For this reason, the algorithmic approach proves to be favorable for generating a single planet with very low fitness; however, given the stochastic nature of the evolutionary alogirithm, different executions can generate substantially different planets, as you can see from figure 9, thus it is still possible to produce a very optimized population by executing the algorithm multiple times.

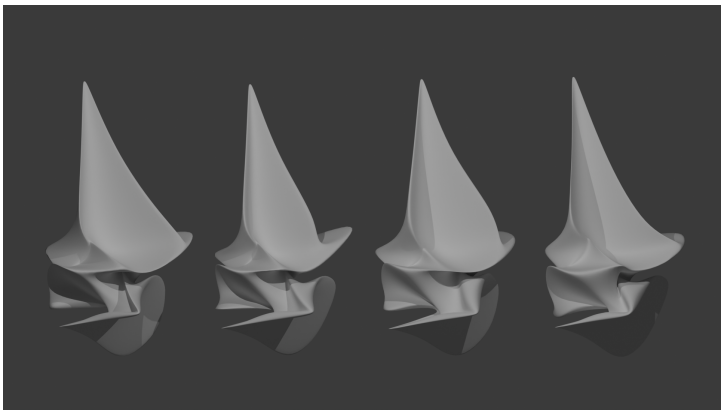


Figure 8: Converged individuals from evolutionary algorithm

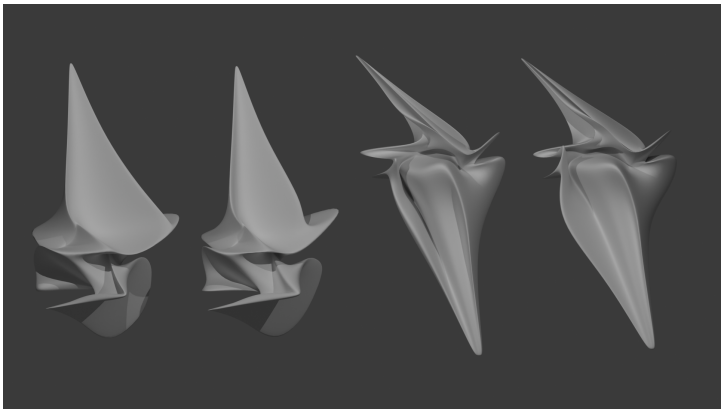


Figure 9: Four individuals from two different algorithm executions

5 Conclusion

The algorithmic approach has proved to be effective in creating planets with very low fitness, even if with reservations: the algorithm is not able to produce a diverse population of planets, especially if the fitness target is low, because of convergence; however, the algorithm can be executed multiple times to produce a population instead of using the one resulting from a single execution. For producing a very diverse population without a strict fitness requirement, the random approach proved to be better. In addition, the random approach could be employed in real time during a loading phase of the video game.
The genotype satisfies the phenotype requirements; however, it does not manage to hide the existence of the poles, which is not desirable.
As for additions to the project, the method should eventually be employed for a real application game; more fitness models should be designed: for example, a planet could be evaluated by the number of safe landing zones (dot product between gravity and normal direction close to zero); even models not related to gravity could be used, and a system that allows the user to specify a custom fitness function could be implemented, so that it becomes a generic generator of closed, curved and smooth surfaces powered by an evolutionary algorithm.

References

[1] *Assimp*. URL: https://assimp.org/.

[2] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Blender Institute, Amsterdam. URL: http://www.blender.org.

[3] *Dear ImGUI*. URL: github.com/ocornut/imgui.

[4] Gerald Farin. *Curves and Surfaces for CAGD*. 2002. ISBN: 978-1-55860-737-8. DOI: https://doi.org/10.1016/B978-1-55860-737-8.X5000-5.

[5] W. Shane Grant and Randolph Voorhies. *cereal - A C++11 library for serialization*. 2017. URL: http://uscilab.github.io/cereal/.

[6] Christophe Groovounet. *OpenGL Mathematics*. URL: github.com/g-truc/glm.

[7] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[8] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[9] *Metal*. URL: developer.apple.com/metal.

[10] Tomas Möller and Ben Trumbore. *Fast, Minimum Storage Ray-Triangle Intersection*. Vol. 2. 1. 1997, pp. 21–28. DOI: 10.1080/10867651.1997.10487468.

[11] *OpenMP*. URL: openmp.org.

[12] *SDL - Simple DirectMedia Layer*. URL: libsdl.org.