TerraForm

Procedural Generation of Earth-like Ecosystems

Cezar Mocan

Advised by Holly Rushmeier

Computer Science

Yale University

December 2016

1. ABSTRACT

Procedural generation is a growing topic within the gaming community, with recent games such as *Spore* or *No Man's Sky* which allow for almost infinite in-game universe exploration of automatically generated unique planets, flora and fauna. Not only can procedural generation provide a form of entertainment, but it can also have incredible benefits if used in conjunction with the human mind and craft, in order to design digital environments, creatures or objects. There is great potential for this yet young wing of game programming, and our project aims to understand and implement the basics of using procedural techniques in order to explore different levels of digital world creation.

The goal of this project is building a planetary simulator able to produce varied types of terrain, trees and creatures from (almost) scratch—through using the least possible amount of pre-existing assets. We are well aware of the fact that even the most iconic games using procedural generation still use a wide range of 3D designed prefabs, and therefore we do not advocate for using none. As in most situations, a hybridized approach, a productive human-computer interaction in designing and building the simulation world is probably the most fruitful path.

The project has two almost disjoint aspects: the game world and the flora, tackled by Spencer Villars's CPSC 490 project, and the fauna, tackled by Cezar Mocan's project. This report focuses on the latter, investigating the challenges and lessons learned in creating procedural 3D animals from zero and placing them in the simulator provided by the other half of the project.

2. INITIAL EXPERIMENTS

The first steps of the project consisted in analyzing how already existing 3D characters function, as a way of finding a good starting point for building ones from scratch. We imported a series of pre-made creatures into Unity and started deconstructing them, in order to understand their structure. Our choice was the *Allosaurus* asset library[1], a free Unity asset consisting of a dinosaur with a few different skins and sets of animations for different actions—walking, running, jumping, etc.

There were a few key findings at this stage, all related to the way Unity treats character structure. These findings ended up loosely dictating the steps we pursued in the project—with different levels of attention awarded to each one, reflecting the project's timeline.

First of all, we learned that each creature is built on top of a skeleton—a series of objects arranged in a hierarchical (tree) structure (e.g. the subtree encoding for a leg is a child of a node representing the hip.) A good real-life analogy is to imagine the objects composing the skeleton as equivalent to the joins composing a living being's skeleton.

Second, we understood that the shapes are displayed using meshes. Unity uses two types of mesh renderers—the (regular) `Mesh Renderer`, and the `Skinned Mesh Renderer`. The former is used for any kind of mesh which does not have animations attached to it, while the latter requires the mesh to have a bone structure which dictate the mesh appearance, as well as support for attaching animations to it. The process of skinning a mesh refers to performing the necessary mesh deformations, using the underlying skeleton, in order to build up to different add movement to a static mesh. As the *Allosaurus* dinosaur had been designed in a 3D modelling software (e.g. Blender or Maya) and later ported into Unity, it is using the Skinned Mesh Renderer. At this point we realized the fact that even though Unity allows for easy integration of assets built in different software, that is not going to be useful for the project, as our goal was to procedurally generate most of our assets.

[1] https://www.assetstore.unity3d.com/en/#!/content/7477

3. MESH CREATION

Upon understanding Unity's mechanisms of working with creature-like objects, we moved on to implementing the main deliverable of this project, the procedural generation of characters. At the time, the most crucial step of the project appeared to be the generation of a mesh from a pre-existing skeleton, so we chose to tackle that problem first. In the research phase, we stumbled upon two publications related to the issue—*B-Mesh: A Modelling System for Base Meshes of 3D Articulated Shapes*[2] and *Converting Skeletal Structures to Quad Dominant Meshes*[3]. Both present relatively similar methods of creating a mesh on top of a pre-existing tree-structured skeleton, and (somehow different) methods of smoothing the resulting mesh.

Briefly, the algorithm (without the smoothing part) for getting from a pre-existing skeleton represented as a hierarchical tree of spheres to a mesh goes as follows:

1. Build the skeleton and add a tree-like hierarchy between the spheres (*Figure 1*, *Figure 2*)
2. Fill out all of the chains in the tree with spheres, such that all neighboring consecutive spheres intersect each other. (*Figure 3*) A chain is defined as a connected path starting from either the root or the daughter node of a node with multiple children, and ends either in a leaf, or in the parent of a node with multiple children.
3. Cross a square planar surface (quad) through the center of each sphere, tilted at an angle halfway between the sphere's angle and the child's angle. (*Figure 4*)
4. For each chain, create a mesh out of the corners of the resulting quads. (*Figure 5*) This will leave the spheres with multiple children un-meshed.
5. For all the un-meshed spheres, do a convex hull of the corners of the quads crossing each of the children spheres, as well as the parent sphere. This will complete the mesh. (*Figures 8, 9*)

The figures below present steps 1—4 for a single chain (*Figures 1* to *5*), as well as a mesh which includes the convex hull in step 5.

[2] Ji, Zhongping, Ligang Liu, and Yigang Wang. "B-Mesh: A Modeling System for Base Meshes of 3D Articulated Shapes." *Computer Graphics Forum*. Vol. 29. No. 7. Blackwell Publishing Ltd, 2010.

[3] Bærentzen, Jakob Andreas, Marek Krzysztof Misztal, and K. Weŧnicka. "Converting skeletal structures to quad dominant meshes." *Computers & Graphics* 36.5 (2012): 555-561.

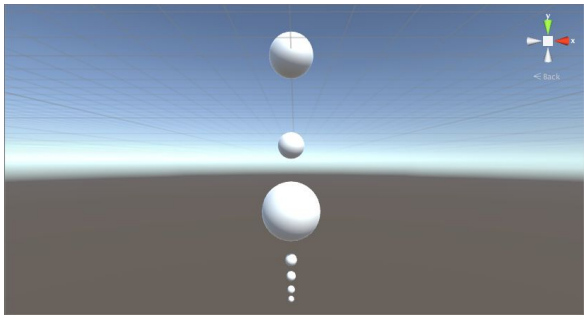

Figure 1—Skeleton.

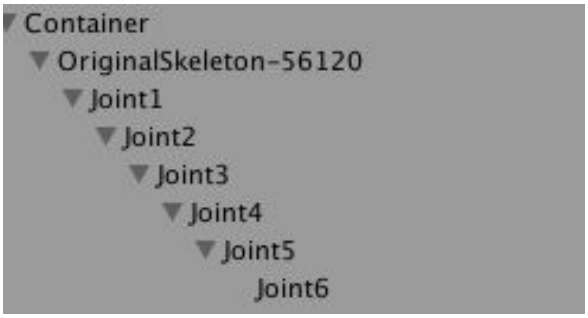

Figure 2—Tree hierarchy of the skeleton.

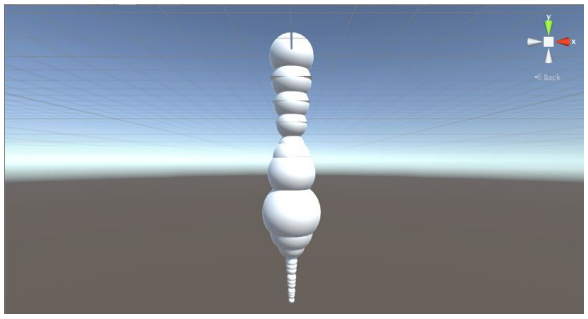

Figure 3—Skeleton with spheres filled out.

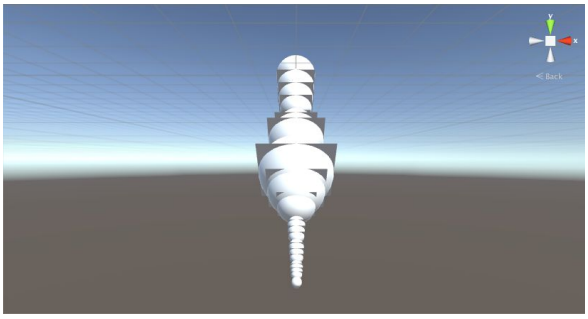

Figure 4—Quads crossing through the centers of the spheres.

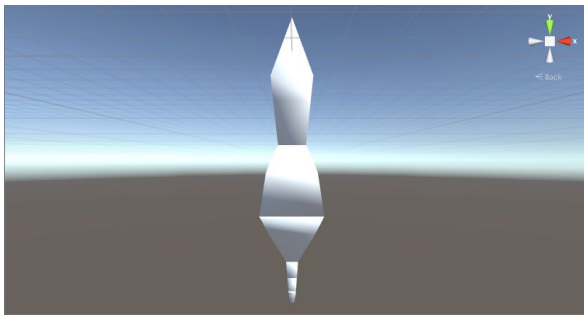

Figure 5—Mesh resulting from connecting all quad corners.

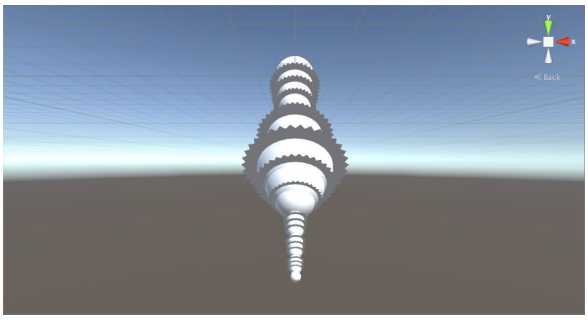

Figure 6—Fig. 4, with 9 quads for each sphere, for a smoother mesh.

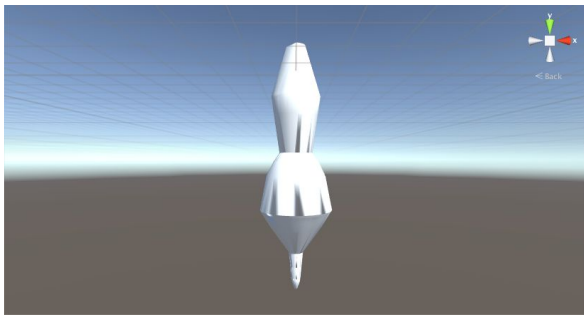

Figure 7—Mesh resulting from connecting all quad corners in Fig. 6

A basic optimization for increasing the fidelity of the mesh is presented in *Figures 6* and *7*—the addition of multiple quads for each sphere, 9 in this case, instead of just one. *Figures 8* and *9* below show the skeleton and the resulting mesh for a more complex tree structure (which requires the convex hull to run as well.)

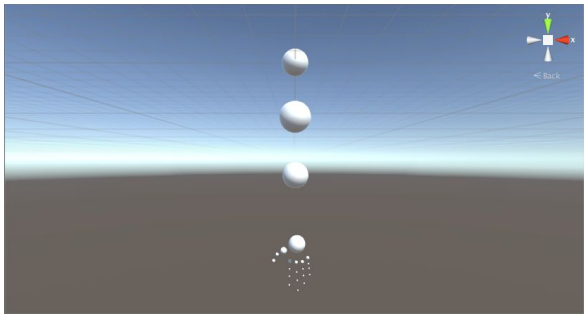

Figure 8—Skeleton of an arm.

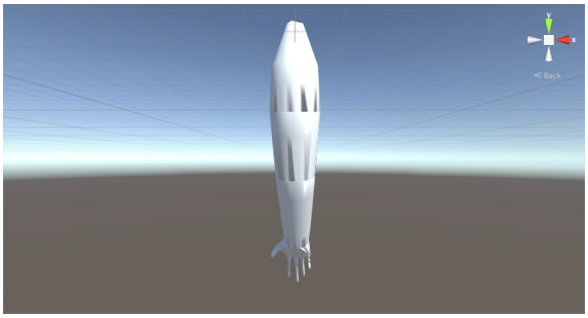

Figure 9—Generated mesh for the arm skeleton.

Aside from the conceptual difficulty of grasping the two papers and implementing some of the solutions they are proposing, this part of the project brought the first larger programming environment challenge—understanding the way meshes are structured (in general, and in particular in Unity), and using the structure for semi-regular shapes such as these. We are using the term semi-regular because the meshes may look like arbitrary 3D shapes, but they are built additively, using layers of coplanar points (the quad corners.) The `Mesh` data structure in Unity is composed out of the following components:

1. `vertices`—a list of 3-dimensional point coordinates, representing where the vertices of the mesh are located.
2. `triangles`—a list of triplets, representing the indices of the points which make up each triangle present on the mesh. They need to be specified clockwise. Vertices and triangles are enough in order to have a mesh shape, but in order to be able to apply textures, and have shading work correctly, the following attributes need to be set as well.
3. `uv`—the UV mapping coordinates of each vertex. Used in texture mapping.
4. `normals`—the normals at each mesh vertex.

In the initial version of the algorithm—the one with a single quad per sphere—the vertices and triangles were assigned manually, by dividing each quad into two triangles, and moving down each chain (well, assigning vertices is easy. The order of the triangles is the bigger challenge.) Once we introduced the multi-quad optimization, we let a 3D Convex Hull library[4] take care of that.

[4] https://designengrlab.github.io/MIConvexHull/

4. SKELETON AND MESH MUTATION

Another step in laying out the groundwork for procedural generation consisted in being able to either generate skeletons from scratch, or mutate existing skeletons and shapes, in order to be able to obtain diverse set of possible body parts, and eventually build a library of them. In a publication from 1991, Sims[5] describes the use of genetic algorithms in procedural gaming at a high level, proposing the following four mutation strategies for tree-like structures:

1. Changing internal parameters of a node (in our case: scale, rotation of a sphere / joint)
2. Adding random nodes to the tree
3. Changing parameters of an edge (in our case: the distance between two nodes)
4. Addition of new random connections, removal of existing connections.
    a. Garbage collection for the nodes that become disconnected after step 4.

He also proposes two strategies for cross-breeding two similar tree-like structures:

1. Crossover—replacing a sequence [X...Y] in the topological sort order of parent 1 with parent 2.
2. Grafting—moving a subtree from parent 1 to parent 2.

We looked for inspiration at a few other publications. First, an M.S. thesis from Edinburgh University[6] proved incredibly useful. It is a project which tackles procedural generation of fish, and needs to perform similar tasks to ours, especially in terms of evolving a mesh using genetic algorithms. Clune & Lipson (2011)[7], and their website Endless Forms[8], which again performs mesh evolution using genetic algorithms. Last, but not least, Min & Cho (2004)[9] apply their particular flavor of evolutionary algorithms to 3D models of flowers, and obtain interesting crossbreeding results.

[5] Sims, Karl. *Artificial evolution for computer graphics*. Vol. 25. No. 4. ACM, 1991.
[6] http://project-archive.inf.ed.ac.uk/msc/20141637/msc_proj.pdf
[7] Clune, Jeff, and Hod Lipson. "Evolving three-dimensional objects with a generative encoding inspired by developmental biology." *Proceedings of the European Conference on Artificial Life*. 2011.
[8] http://endlessforms.com/
[9] Min, Hyeun-Jeong, and Sung-Bae Cho. "Creative 3D designs using interactive genetic algorithm with structured directed graph." *Pacific Rim International Conference on Artificial Intelligence*. Springer Berlin Heidelberg, 2004.

We implemented steps 1, 2, 3 and a modified version of 4 from Sims's paper, and Grafting applied to evolution as opposed to crossbreeding—through duplicating a subtree and finding a random, but reasonable, position for it. (you can see this in *Figure 13*, one of the hands has 6 fingers.)

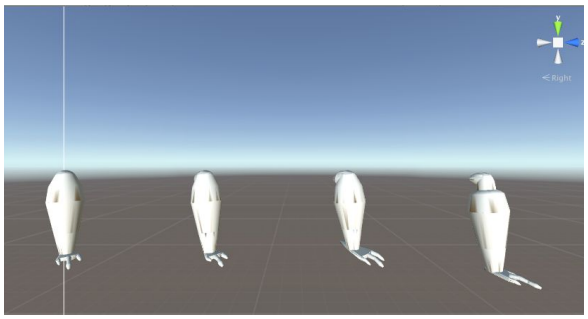

Figure 10—Original leg (left) and mutations.

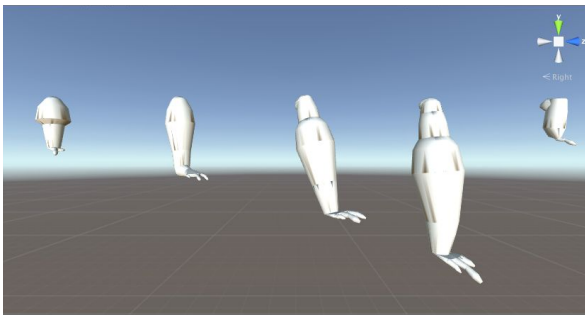

Figure 11—A few other mutations starting from the same original leg.

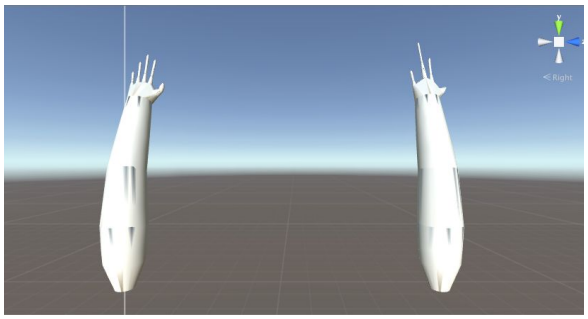

Figure 12.a—Original arm (left) and mutation.

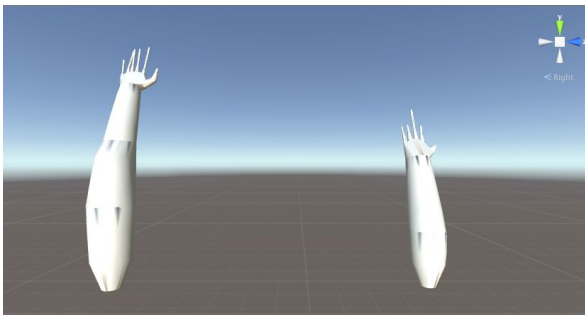

Figure 12.b—More mutations of the same original arm.

The main difficulty in this part of the project consisted in fine-tuning the parameters in order to obtain different-enough, while still realistic results. In the figures below, you can observe some of the mutation strategies in action. *Figure 13* shows an early stage result with high probability of mutation, and wide ranges for the parameters.

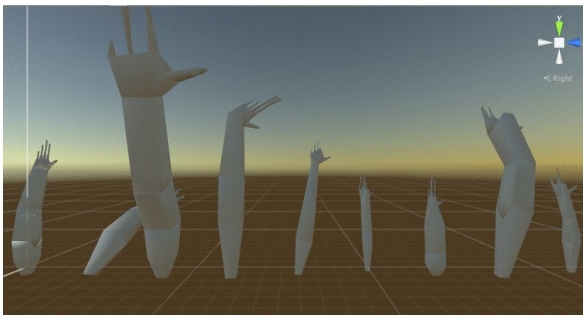

Figure 13—Original arm (left) and mutations.

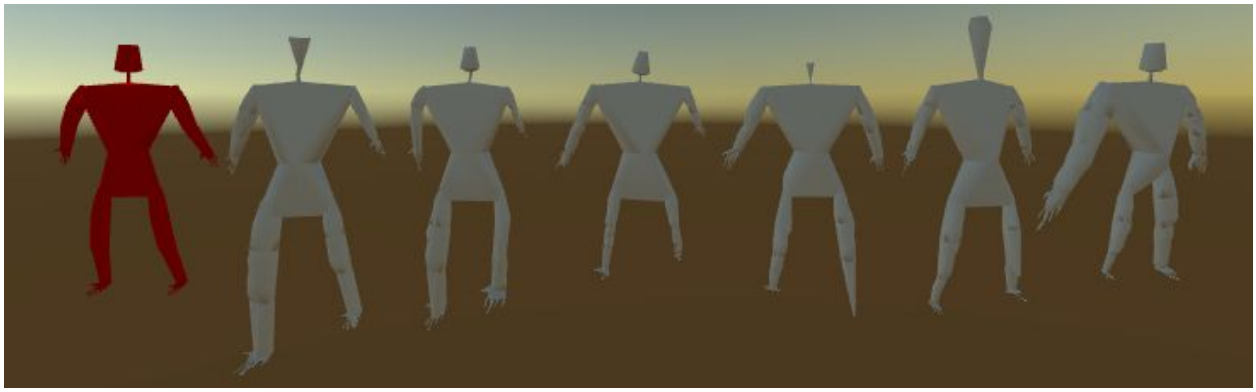

Figure 14—Mutations applied to a humanoid character.

We also implemented a few of the mesh deformation techniques presented in the Edinburgh University M.S. thesis, presented in *Figures 15* and *16*, but ended up not including them in the final version, as the results were relatively unreliable, and caused performance issues.

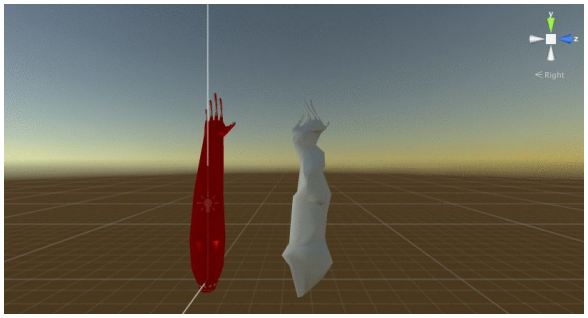

Figure 15—Original arm (left) and mutated mesh (right).

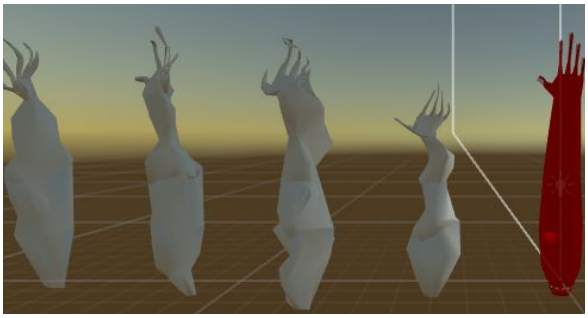

Figure 16—Original arm (right) and mutated meshes (left).

5. FROM LIMBS TO BODIES

Having the ability to generate meshes out of arbitrary tree-like skeletons, and to evolve skeletons to different shapes—in both breadth, and depth (different generations of evolution)—we moved on to creating a library of different body parts, keeping in mind the goal of being able to produce automatically-generated creatures. While, unfortunately, the timeline of the project did not allow for creating a fully automated creature generator, we were able to create a workflow which requires some

human input (especially around putting together different body parts,) but manages to produce reasonably diverse types of outcomes.

By using a simple UI (*Figure 17)* we created the ability to save any generated mutation of a body part, in order to later retrieve it and use it for a new creature. While generating mutations, they are presented to the user with the generated mesh placed on top of them. Saving only writes the skeleton tree structure to the hard drive, and another mesh is generated from scratch when the asset is used in building a creature. You can see a few samples from the arms library in *Figure 18*.

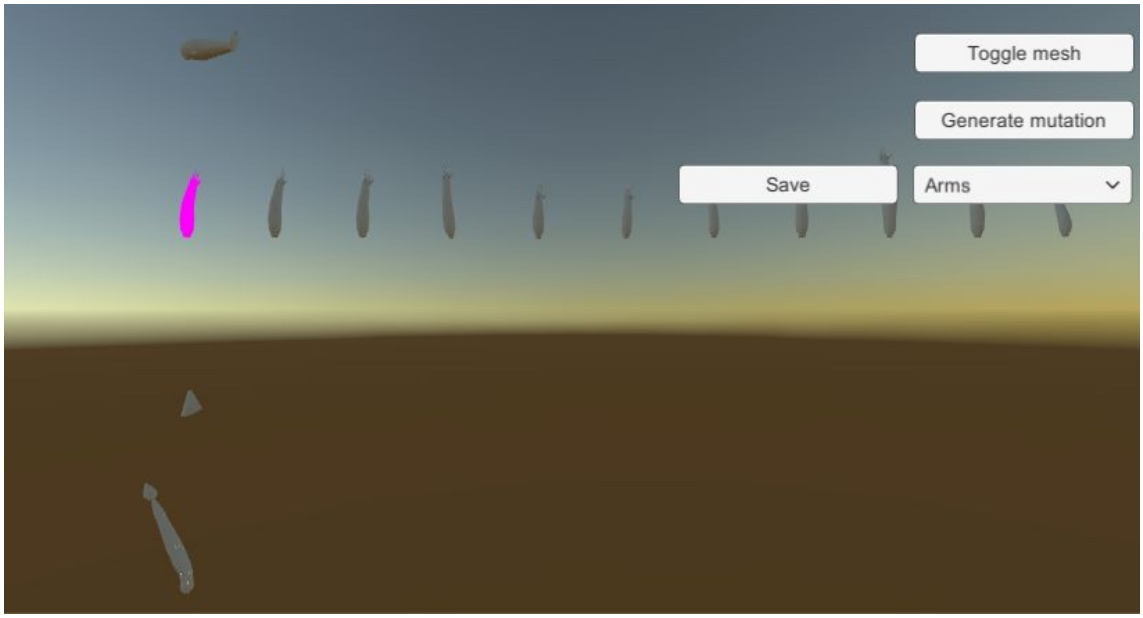

Figure 17—Body part evolution UI. One starts with a leg, an arm, a head, and a spine/body, and can generate mutations for each.

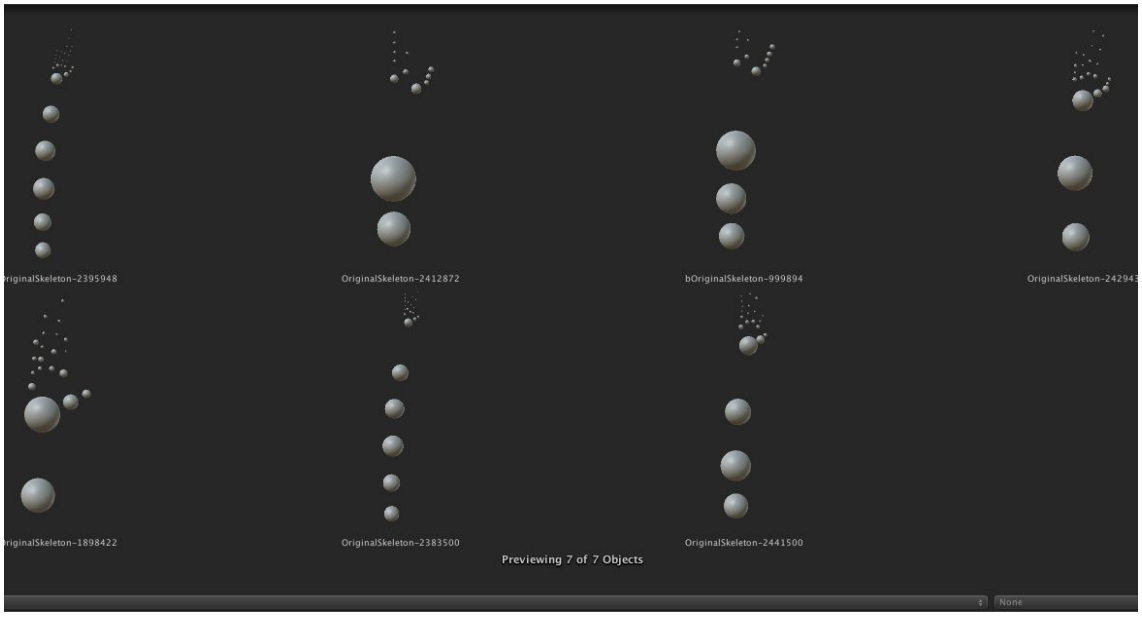

Figure 18—Snippet from arms library.

The workflow for creating a new character goes as follows: first, the user needs to generate mutations for 4 types of body parts (arms, legs, head, body) until finding ones that they are content with. The second step is saving them as assets, using the Save button in the *Figure 17* UI and selecting the correct folder for the specific type of body part. Next, they need to create a new empty Unity GameObject, a container which will be the root of the entire body skeleton tree structure. Importing the body as a subtree of the container follows. Once the body skeleton is in place, the user needs to place the head and the limbs, by selecting the spheres which are going to be the subtree roots of the respective body parts, one at a time, and importing the assets from the library. The only thing left to do is adjusting the limb positions, such that they are symmetrical and at a certain distance from their root node. Although this process allows for making manual adjustments to the current skeleton (for example, changing the angle of a joint, or scaling the limbs), it is still quite clumsy, and could use more automation, making it a perfect candidate for future work on this project.

Once the user has created their new creature's skeleton, they need to run the mesh generation algorithm on it. This is fortunately easy—it just involves attaching a script to the root object of the skeleton—and the algorithm we have implemented is general enough in order to work on arbitrary

complicated skeleton trees in one run (as opposed to having to generate meshes for each limb and joining them manually.) Although the produced results are far in quality from today's gaming standards, this process can be 1) improved to generate better meshes and more realistic body parts, and 2) leveraged in order to (semi-)automate character design, especially if diversity is a goal. You can see two creatures built using this workflow in *Figures 19* and *20*.

Figure 19—Biped creature with tiger texture applied.

Figure 20—Quadruped with hippopotamus texture.

6. MOTION

We started investigating ways of adding movement to the characters from the very beginning of the project, and, unfortunately, had to take a more low-level path than just using Unity's `Animation`, as the feature is used mostly for imported animations from different pieces of software. Unity's animations usually work in two steps: first of all, an `Animation` component is attached to creatures, which contains static animations of them doing certain gestures—walking, running, jumping. The component is accompanied by scripts which implement the actual movement of the whole creature body inside the world is it placed in, in accordance with its body parts movements rendered by the animation. We took a similar path, without using the `Animation` component.

Adding movement to a skeleton can be as simple as adding a script which produces a pendulum-like motion in the limbs, and that can be entirely manipulated through the rotation of the limb's parent in the tree hierarchy. Although we started out by just using a sine function for these movements, more realistic renderings can be obtained through more complex functions. One challenge we encountered

at this step has been the fact that skeleton limb rotations are easy, but accomplishing the same with the mesh is more dfficult.

The brute-force approach to obtaining movement in our meshes is by just reconstructing the mesh on top of the skeleton at each frame, but, not surprisingly, this led to significant performance issues—even for a single creature. Given our mesh-building algorithm, one way of circumventing this issue is by recomputing the mesh only at the joints where movement is performed (usually one joint per limb)—meaning a 3D convex hull of 20 to 100 points per moving body part. This is a better long term solution than the brute-force algorithm, but we ended up taking a different approach: precomputation using the brute-force algorithm.

We are simulating a very simple type of movement—running, in which all limbs move at the same pace, in the same direction with no offset relative to each other. We generate 15 different limb positions which together create one full cycle of running, and save meshes for all 15 of them. From a user experience point of view, generating this for an existing skeleton is as easy as attaching a script to it and running the code—even though running can take up to one minute, given the intensity of the mesh generation algorithm. *Figure 21* shows 6 of the 15 movement meshes for one of our creatures.

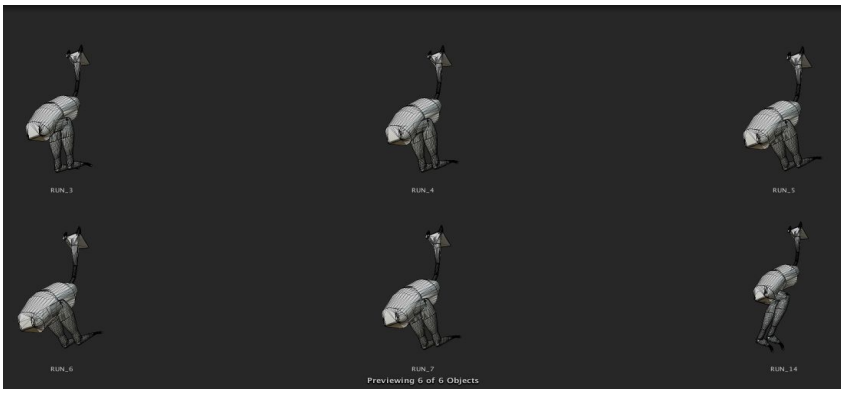

Figure 21—Movement frames as meshes—#3, #4, #5, #6, #7, #14

The script for making the creatures move in the world synchronously with their limbs will be discussed about in the next section.

7. INTEGRATION INTO THE TERRAFORM WORLD

As this project was created jointly with Spencer Villars's final project, one of our goals was to integrate the creatures built through this project with the terrain and game-like world generated by Spencer's. The main challenge of the integration consisted in having the creatures move on non-flat terrain, and act under the forces of gravity.

Unity's physics engine made it relatively easy to start out by testing movement on a flat surface. The script handling the motion of the animated mesh is simplistic, but can be built upon in order to create more realistic movement. As of now, it uses the collision detector in order to find the points and moments of contact between the creature's limbs and the surface it is moving on, and accelerates the body's movement the moment right after contact with the surface, after which it quickly starts decreasing the velocity. Movement works with a simple linear function right now—this is another important feature for future improvement.

Transitioning from flat surfaces to terrain with a heightmap required rotations of the entire creature object on the X and Z axis, such that all four (or two for bipeds) limbs stand on the ground, as opposed to just one limb touching the underneath surface. This was done using ray casting in order to determine the normals of the surface, and computing two perpendicular lines on it—one of them being the creature's forward direction, and the other one being perpendicular to the creature's forward direction. *(Figures 22* and *23.)*

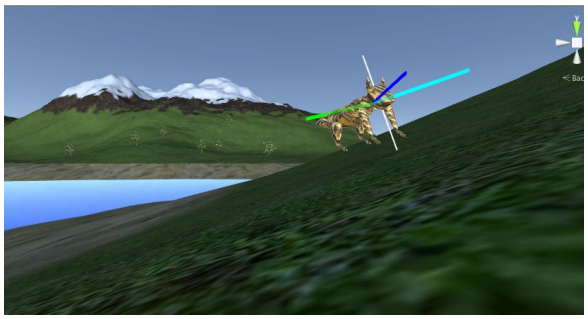

Figure 22—Creature running on a hill. Lines encoded as follows:
White—surface normal;
Dark blue—forward direction, perpendicular to normal (for front-back body angle adjustment);
Light blue—perpendicular to normal and forward (for lateral body angle adjustment)
Green—back direction;

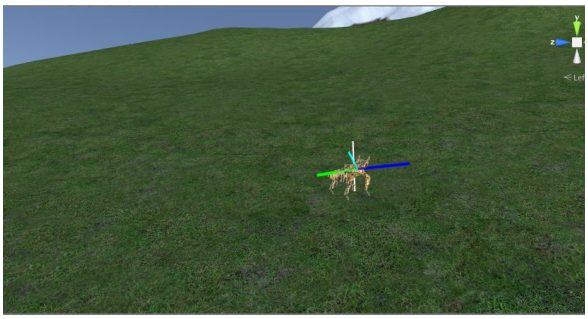

Figure 23—Same creature on the same hill, different camera angle.

The last addition consisted in creating an automated script which populates the world with the existing animated creatures in the library, and giving them slightly different characteristics, such as scale, texture, direction of movement (which changes every now and then.) A snapshot of the final result of the integration is presented in *Figure 24*.

Figure 24—TerraForm world populated by creatures

8. CONCLUSION AND FUTURE WORK

This study provided a great opportunity to learn about multiple aspects of 3D (game) programming, as well as systems of procedural generation, 3D mesh geometry, mesh generation and software development in Unity. Having started without a background in the field, I believe one of the mistakes on the fauna generation side of the project was the fact that its focus was too broad for the timeline and skills involved in its creation. The wide scope definitely contributed to the learning process—as it gave a comprehensive high level overview, but this led to a lack of accuracy or complexity in several aspects, some of them detailed below:

1. **Mesh smoothing**—future work needs to be done in order to achieve smoother meshes, an essential aspect of realistic creatures.

2. **Textures and materials** for the creatures—this side of things was almost entirely left out, due to time constraints. Although we have a few textures we are mapping on creatures, they could use much better normal and height maps.
3. **Improved workflow for creature building** by eliminating some of the actions users need to do manually (select assets, import them, place limbs, etc.) Ideally, the user would just need to select the desired body parts, and make small position and angle adjustments.
4. **Realistic movement**—movement still needs plenty of tuning in order to seem realistic, in the current state it works mostly as a proof of concept. (e.g. moving more than one joint of the limbs, head movements, better mathematical functions for moving the limbs, etc.)
5. **Faster animation generation**—either by using the proposed algorithm, or using an entirely different method.
6. **Macro-level interaction** in the simulation world—by having multiple types of behaviors for the creatures (looking around, jumping, walking, etc.)

9. ACKNOWLEDGEMENTS

I would like to thank Professor Holly Rushmeier for her continued feedback and support in the development of this project, as well as Spencer Villars, without whom this work would not have existed.