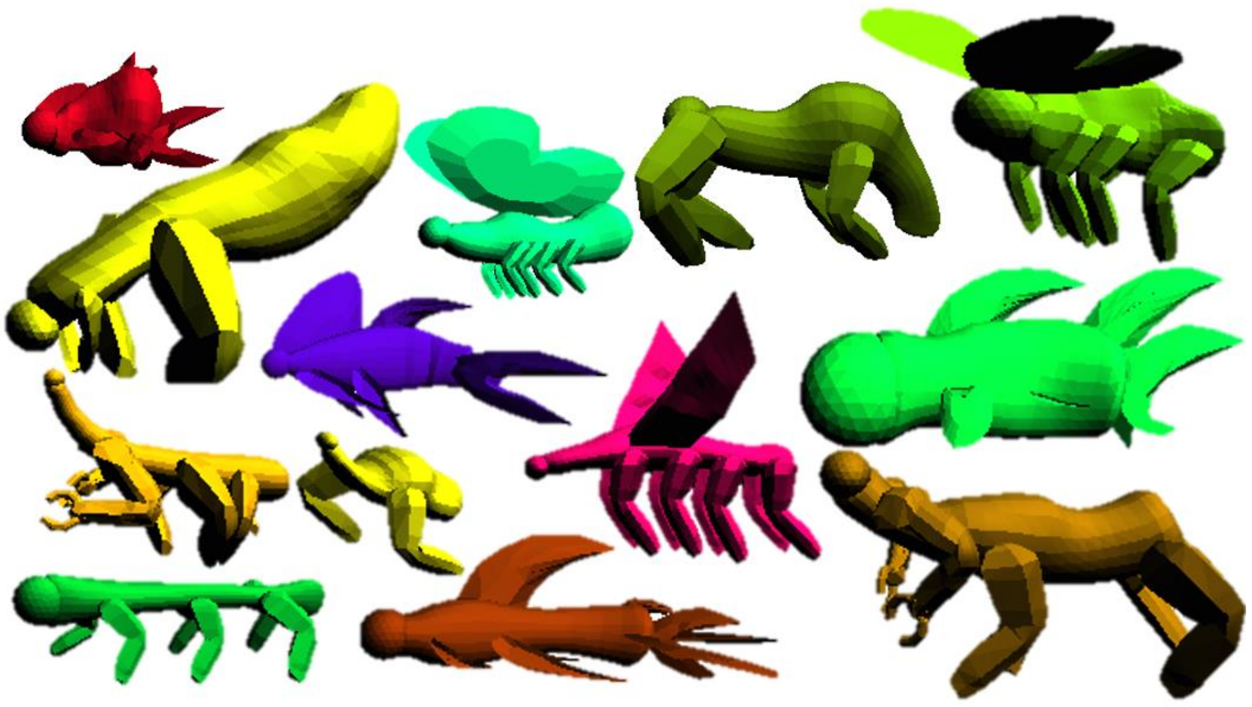
PROCEDURAL CREATURE GENERATION AND ANIMATION FOR GAMES



Alex Christo

Msc Computer Animation and Visual Effects

Bournemouth University

20th August 2022

Abstract

This report documents the design and implementation of a real time procedural generation system for creatures in video games.

The final system, implemented in unity and written in C#, utilizes free form deformation techniques and automatic rigging to create a base body. A novel limb attachment algorithm then attaches, and scales pre-made body parts to create one of three creature types: flying, swimming, or walking. Each creature type has procedural idle and movement animations. Of particular note is the procedural walking system for walking creatures, which uses the size and speed of the creature to create natural looking motion.

The project successfully generates a wide array of random creatures in real time, with reasonably natural looking animations for all. While there are limitations to the visual fidelity and range of animations for each creature, it is hoped that this project offers a more open-ended procedural creature generation system than has come before and can aid in future work in this area.

Acknowledgements

Thank you to Jon Macey, Jian Chang and Jian Jun Zhang who offered useful feedback and suggestions throughout this project’s development.

Contents

1 Introduction

In the context of video games, procedural generation refers to the generation of content in the game algorithmically, rather than manually by an artist or game designer. When implemented well, procedural generation allows for the creation of a vast amount of content that would be completely infeasible to construct by hand. This can greatly extend the lifespan of a game and relieve some of the burden from artists and designers.

Traditionally, procedural generation is used for tasks such as level or map generation. With a correct set of building blocks and rules in place this can be implemented quite effectively (see Minecraft, Spelunky, Binding of Issac and more). While most procedural generation systems essentially combine existing characters, props and environment pieces, some games have also attempted to procedurally generate these individual elements. This is often a much more complex task, that in many cases is best left to artists and designers.

An interesting specific case of this is procedural creature or animal generation. Most notably attempted by the game No Mans Sky (2016), randomly created animals and monsters can lend a game world extra variety and keep the player engaged for longer. Creating varied creatures from scratch, which all look and move convincingly is non-trivial. This project will focus on developing such a system in the game engine Unity, which will eventually be implemented into a game currently in development called “Duel Space”. Given the time constraint, and the researchers initial unfamiliarity with much of the technology, the scope is limited to creating low fidelity creatures, each with a movement, idle and turning animation (all driven procedurally).

2 Research and Design

2.1 Previous Work and Technology

Procedural creature generation is a broad topic, with many component parts. As well as academic research into specific elements such as procedural animation, it is prudent to examine approaches to the problem in current video games. By far the two most notable examples of creature creation in games are Spore (2008) and No Mans Sky (2016). It is worth noting that an obvious limitation is the fact that none of the code or algorithms for either game are available to the public. As such the analysis done here is done primarily based on observation of the finished product, as well as on some sparse developer interviews.

Spore was a steppingstone for both procedural content generation and creature creation in games. However, the creatures that populate Spore are actually not procedurally generated at all, at least not entirely.

Spore offers an in-depth system for users to design creatures. The core stages are as follows: First the player decides on the number of segments the spine will have and how it will bend. Each segment of the spine can be increased or decreased in size which informs how thick the body is around that segment. From this point the player can drag and drop a variety of parts and pieces on to the creature: eyes, mouths, arms, legs and accessories like spikes, feet, hands etc. These pieces can be attached to almost any point on the created base body. Legs will always extend from the attachment point to the ground, where they meet the floor. The player can drag each joint of arms or legs to reposition them, creating wildly disproportioned limbs. Every creature has a wide variety of animations such as walking, attacking, eating and singing which all adapt to the skeleton of the given creature. For example, a creature with very large legs will walk at the same speed as a creature with smaller legs, but the step size will be much greater and the frequency much less.

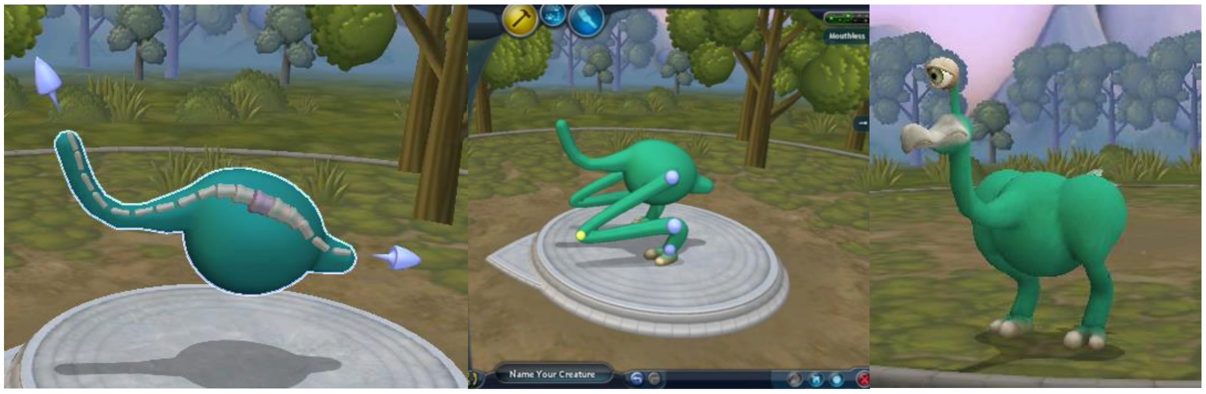


Figure 1: Stages of creature creation

While the worlds which players can inhabit in Spore are procedurally generated, the creatures themselves are not. The reason for this is undocumented. While it could be argued that this was done to allow for more interesting, artist created designs, that seems to go against the core ethos of the game. As such it seems more likely that implementation difficulty was the main issue. The Spore creator is incredibly open ended, allowing for as many arms, legs, mouths eyes as the player desires, and each can be attached to any point on the creature. This allows for some great customization but creating generative methods to constrain this huge array of options would likely be challenging (although again, this is purely speculative).

There were a few notable takeaways from Spore which formed some early ideas for this project:

Starting from the base body and using this to inform the creature, then attaching limbs after this stage; Limiting the scope of what is possible to make generation easier; Adapting animations procedurally as the size and shape of limbs is altered.

Around 8 years later Hello Games promised to deliver on a fully procedural world, including procedural creatures, in their game No Man’s Sky (2016). While many of the touted features didn’t make it into the games original release, it boasted an impressive procedural creature generation system. Specific details are hard to come by, however it is possible to infer some inner workings. In an interview with game informer (A Behind-The-Scenes Tour Of No Man's Sky's Technology, 2014) studio head Sean Murrey goes into some detail on creature generation and gives some brief glimpses of internal developer tools. He states that artists create thousands of component parts which creatures can be made with, this is to be expected of course. More revealing however is when Murrey describes each creature type as a “silhouette” explicitly created by artists, for example lizard, rat, fish, cow etc. There are apparently hundreds of these creature types, each able to create a wide variety of “variants”. Catching a glimpse of the developer tools which generate these variants gives more information.

Looking at three examples (figure 2), it appears that each variant has essentially the exact same rig. For example, each lizard variant has a tail, head at the front, backwards knees at the front, and a double knee at the back. Each part of the body can be scaled, and attachments such as head, shell, wings etc. can be added from a vast pool. Animations modify to fit the rig however it is scaled, similarly to Spore. Even without knowing the exact algorithm used, it is clear that No Man’s Sky’s procedural system benefits from a huge bank of artist created assets as well as some strict predefined animal types. This reinforces the idea that having a limited scope is important for effective procedural generation. As well as this, the idea of having some predetermined animal types was a useful takeaway which was utilized in this project.

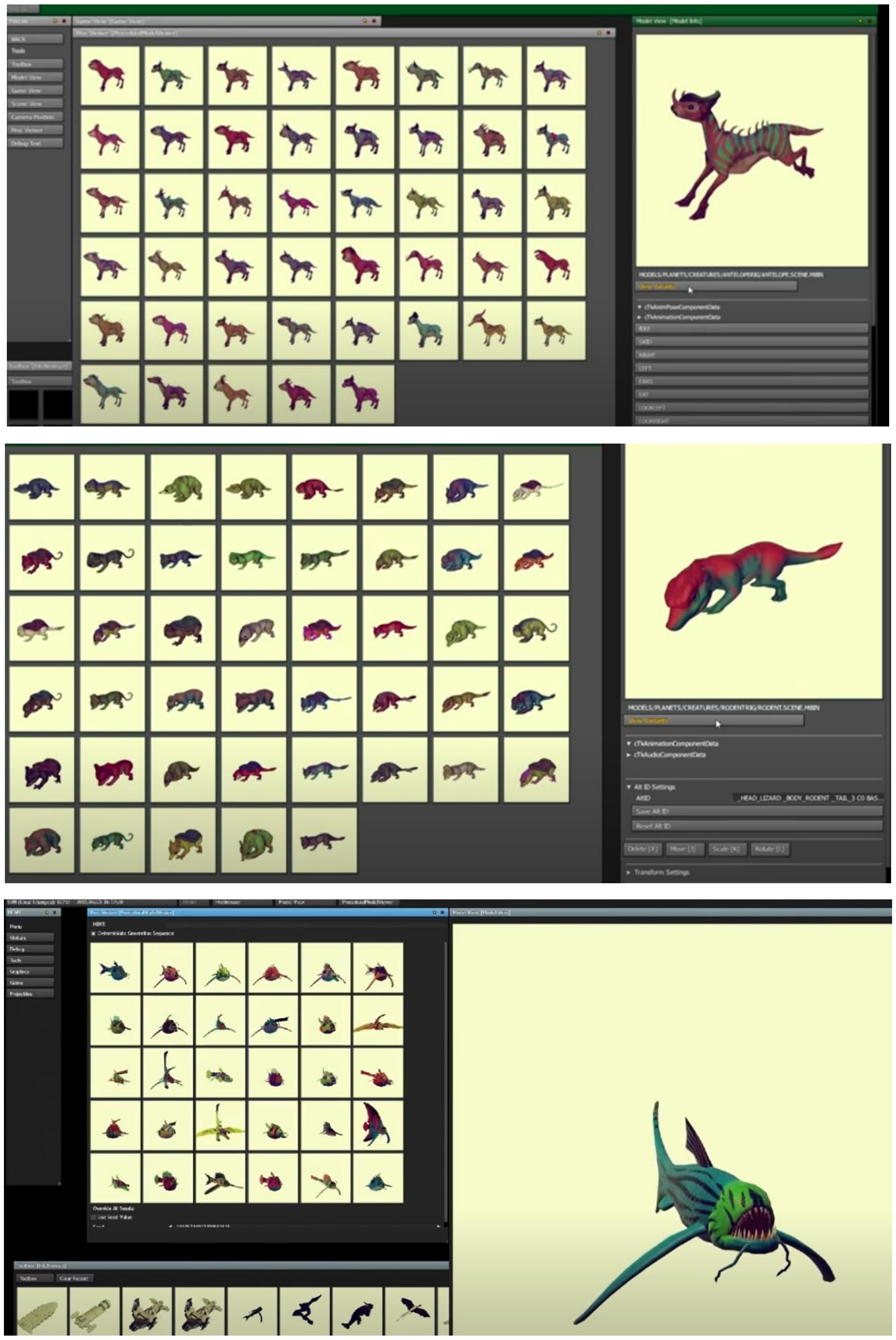

MODELS/PLANETS/CREATURES/ANTELOPERIG/ANTELOPE.SCENE.MBIN
MODELS/PLANETS/CREATURES/RODENTRIG/RODENT.SCENE.MBIN

Figure 2: Three creature types and different variants

After garnering a high-level understanding of how Spore and No Man's Sky approach creature creation, academic works relating to both procedural creature generation and animation were surveyed for techniques which could be useful.

"Creature generation using genetic algorithms and auto rigging" (Hudson, 2013) is a Houdini based system which deals purely with creature generation and rigging, with no considerations made for animation. The main algorithm essentially builds a creature from a series of segments which correspond to bones in the rig. It starts by modifying the size and angle of the base body sections, before choosing a random number of legs, arms and number of sections for the wings, arms and fingers. Arms, legs and heads attach to predetermined segments of the main body. Working in segments like this means that automatic rigging is straightforward. The results are impressive (figure 3) though there are some limitations.

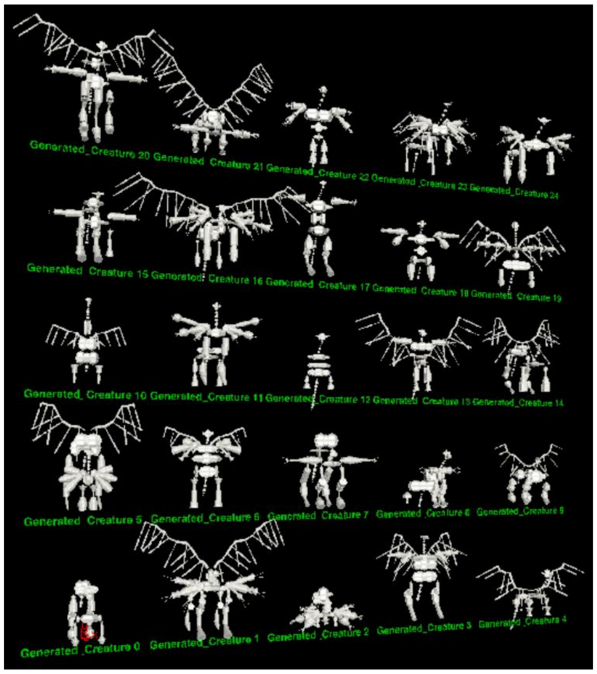


Figure 3: A set of randomly generated creatures (Hudson, 2013)

Legs can only be added in biped and quadruped styles, and arms attach to the same point at the shoulder, no matter how many there are. The generated mesh should be replaced with one of higher fidelity and subsequently reskinned, which is not ideal. Genetic algorithms also play a part here, all features of the creature are stored and can be used to "breed" existing creature designs to create new ones. This feature is useful for the intended use of this project (creating creature designs for artists to work with further). A similar approach could potentially have been used to optimize the creatures produced in this project. By cross breeding creatures until they fit a set of success criteria, it could allow for fewer bizarre unconvincing looking creatures to be generated. However, it is quite possible that converging to a suitable solution would be time consuming, and therefore not suitable for real-time applications. As such this idea was not explored. Instead, the core takeaway from this project was the idea of using the skeleton of the base body to dictate the attachment points for limbs.

Procedural animation is a well-researched topic, and several interesting approaches have been used for characters/creatures. Physically based animations are animations driven by some underlying physics system, which the animated character lives within. Karl Sims' work (1994) is one of the earliest examples of procedurally generated creatures and it relies primarily on physically based methods. Similar to (Hudson, 2013) and (Hornby and Pollack, 2001) creatures are created from a series of segments. Segments are combined, with the points between them acting as points of articulation. Genetic algorithms are used here to combine segments to create creatures capable of moving in specific conditions such as walking, jumping, or swimming, by way of various fitness functions.

Other physically based methods (Geijtenbeek, 2013), (Tan, Gu, Turk and Liu, 2011) vary in implementation approach but share the same general principal of tuning parameters of an animation over a series of generations, such that a character can successfully move themselves through space with physics acting on their body. While these approaches are effective in their domains, they are not well suited to this project for two main reasons. Firstly, as was mentioned previously, optimization approaches such as these can often take many generations to converge on an acceptable solution. This makes these approaches less suitable for real-time domains such as video games. Secondly, the animations generated by these methods often do not look very natural. This is because it is easy to create success criteria for a creature moving from one place to another, but a lot harder to optimize for if it 'looks right'. When dealing with entertainment media, it is usually more important for the end product to look visually appealing than to be physically accurate. As such, other approaches were explored.

As with many problems in animation, neural networks have been proposed as a solution for creating real-time procedural animation (Samyn et al, 2014). Using either motion capture data, or handmade animation as training data, a neural network is capable of created a parameterized model. This model is then able to adapt an existing animation to a new scenario based on it's training data. Using Fuzzy controllers, artist friendly parameters can be tuned to modify the finished animation. This capacity for artist control is the main appeal here, as traditionally procedural animation techniques have existed somewhat separately from traditional animation artists. However, the finished quality of the animations produced is highly dependent on the quality and breadth of training data supplied to the neural net. Given that this project is focused on creating animations for creatures that do not exist (and could have any number of limbs) the reliance on quality training data here is a major pitfall. This project also has no real need for artist control, as such utilizing machine learning in this way was deemed infeasible.

Many procedural animation methods leverage Inverse Kinematics. In contrast to physically based methods, kinematics is purely concerned with the motion of objects, without considering any of the forces causing it. In the context of animation, Inverse Kinematics deals with solving for the positions/rotations of a series of joints, given a desired destination for an end effector and a position for the root of the joint chain. For example, when animating a leg, usually the foot would be the end effector and the hip would be the root. If a new goal position is supplied for the foot, then the hip and knee joint rotations will be solved for, such that the end effector is moved as close towards the goal as possible (while respecting any constraints such as knee direction). If you were to move the hip (root), then the leg joints would bend so that the foot stays in the same position. Various solutions exist (Badler and Zhao 1994) (Aristidou and Lasenby, 2009) each with their own benefits and drawbacks. Most 3D digital content creation (DCC) tools, including game engines, contain some form of inverse kinematics solver. For the purposes of this project the exact IK implementation details are not necessary and can be taken as a black box.

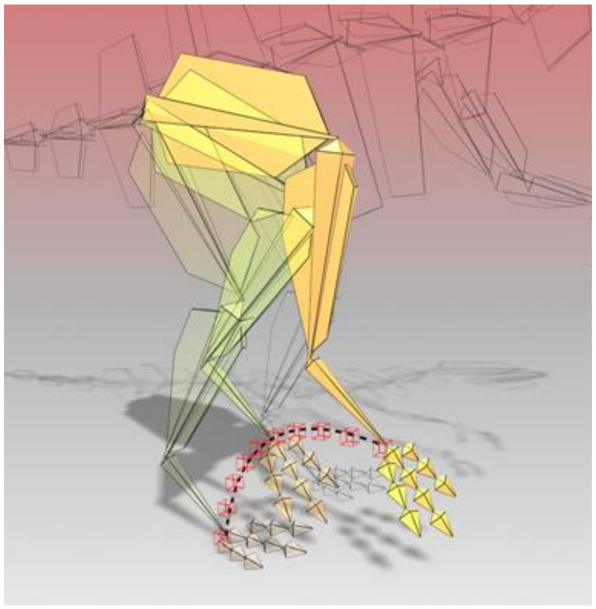


Figure 4: An IK leg setup (Autodesk, 2016)

Inverse kinematics has successfully been leveraged for the goal of creating convincing, procedural animation), particularly for walking/running. Since access to motion capture is not feasible, only purely procedural methods such as Chung and Hahn (1999) were considered. Implementations vary but generally a series of parameters such as step length, step frequency and walk speed are used to predict where a footstep will be made, then drive a foot position based on how close the character is to this predicted step position. This foot position acts as the new goal position for the end effector. Modern methods (Skovbo Johansen, 2009) usually involve a ray being cast onto the environment to place each step. This method is highly suited to real-time due to low computation and is highly adaptable on the fly due to its parameter driven nature. This makes it well suited to procedural creatures as things like leg size, speed and gate will vary from creature to creature and being able to tune a simple set of parameters to reflect this in animation is ideal. As such this approach was the foundation of the procedural walking system created here.

2.2 Initial Idea

Post examination of both creature generation and procedural animation techniques, a high-level series of steps for the creature generator was constructed which outlined the core problems that would need solving. Taking inspiration from Spore it was decided that generating a base body would act as the starting point for the creature. From this point the base body would need to be rigged so that it can move, with the bones acting as reference points for attaching limbs (as in (Hudson, 2013)). These limbs would then be driven by an IK based procedural animation system, which would adapt the animation as needed depending on the size and speed of the creature. The goal was to get as close to the level of freedom offered by Spore as possible, but procedurally generated.

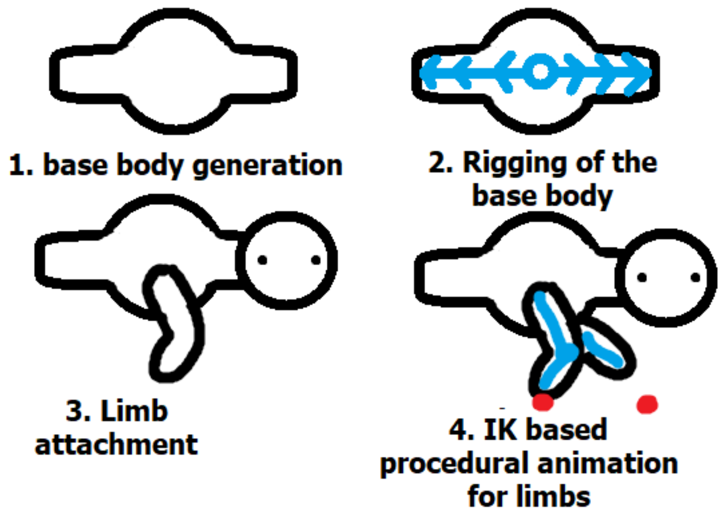

1. base body generation
2. Rigging of the base body
3. Limb attachment
4. IK based procedural animation for limbs

Figure 5: An early illustration of the stages of creature creation

2.3 Requirements

A small set of requirements was designed. These requirements helped both guide the direction of implementation, and act as success criteria for evaluation upon completion of the project.

Number	Requirement	Priority	Dependencies	Justification
1.	The system must be built using the Unity game engine.	H	/	The creature generator is intended to be incorporated into a Unity based game.
2.	Each creature should have an idle animation.	M	/	An idle animation gives life to the creatures but is less important than a procedural walk, as it is less specialized to the creature than a walking animation.
3.	Each creature must have animations to move forward and turn left / right.	H	/	This is a good example animation for the creature as the movement animation will need to vary procedurally based on the limbs, size and speed.
4.	The generator must be able to produce creatures at a variety of sizes.	H	/	Important to add variety.
5.	The generator must be able to produce creatures at a variety of speeds.	H	/	Important to add variety.

6.	Walking animations must scale convincingly with the speed and scale of the creature.	H	3, 4, 5	Important for visual fidelity.
7.	The generator should be able to produce creatures with a variety of movement styles eg. Swim, Fly, Walk	M	3	Important to add variety. No Man's Sky is an example of this approach being necessary.
8.	The generator must be able to create a variety of different looking base bodies.	H	/	Important to add variety.
9.	The generator must be able to create creatures with a variety of different limb styles (eg. Bipedal, quadrupedal).	H	/	Important to add variety.
10.	The creatures created must look generally feasible + convincing.	H	/	Creatures need to be as convincing as those which were designed by hand, e.g. No creatures with tiny legs at the front which look like they would topple over.
11.	Creature generation should take no more than half a second.	M	/	Creature generation needs to effectively be real time to be useful in the context of video games.

3 Technical Background

As well as inverse kinematics, which was touched on earlier, there are two concepts which play an important role in the creature generator implementation: Free Form Deformation (FFD) and the rigging/skinning of 3d models. These concepts will be explained here so as not to bog down the implementation section, which is mostly concerned with the novel aspects of this project.

3.1 Free Form Deformation

FFD is an older technique (Sederberg and Parry, 1986), used to smoothly deform objects by manipulating a three-dimensional grid of points surrounding the mesh.

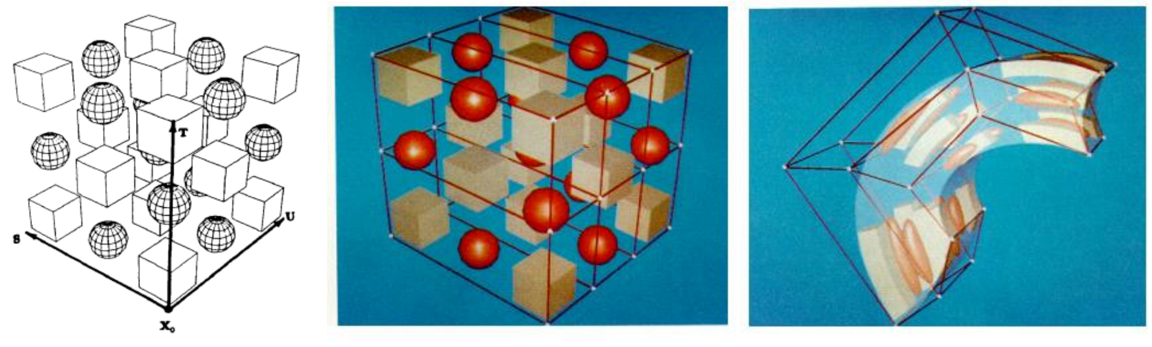


Figure 6: FFD in action (Sederberg and Parry, 1986)

Vertices of the mesh to be deformed are placed in a local coordinate space defined by three vectors S,T and U. A point P using the system is given by $P = P_0 + Ss + Tt + Uu$. The s,t,u components of a point P can be found using simple linear algebra:

$$s = (T \times U) \cdot (P - P_0) / ((T \times U) \cdot S)$$

$$t = (U \times S) \cdot (P - P_0) / ((U \times S) \cdot T)$$

$$u = (S \times T) \cdot (P - P_0) / ((S \times T) \cdot U)$$

In order to modify the points, the local coordinate system must be transformed. To do this the region encompassed by S, T and U is split into a grid of points such that there are l points in the S direction, m points in the T direction and n points in the U direction.

A single control point P (i,j,k) is given by $P = P_0 + i/l\, S + j/m\, T + k/n\, U$. Deformations are achieved by manipulating the control points, either algorithmically or by hand. The deformed position of a point in global space can then be found by computing the points s,t,u coordinates and then evaluating its trivariate Bernstien Polynomial:

$$P(s,t,u) = \sum_{i=0}^{l} \binom{l}{i} (1-s)^{l-i} s^{i} \sum_{j=0}^{m} \binom{m}{j} (1-t)^{m-j} t^{j} \sum_{k=0}^{n} \binom{n}{k} (1-u)^{n-k} u^{k}$$

Although an older method, it allows for very smooth, natural deformation since multiple control points contribute to the final vertex position.

3.2 Rigging and Skinning

Rigging and skinning are fundamental parts of the animation pipeline, and are crucial to understand when developing an animated character from scratch. After creating a mesh, a rig is created for it. A rig acts as a skeleton for the mesh. It is comprised of “bones” which can have other bones connected at either end. Bones themselves can be stretched or deformed, but generally when dealing with characters such as humans or animals with ridged skeletons bones have a fixed length and size. Each connection between bones acts as a point of articulation for the rig. A bone can be rotated via its connection point to a parent and any child bones will follow its transformation. For example, if using traditional forward kinematics (FK), a bicep bone can be rotated about the shoulder connection, any forearm and hand bones will follow the same motion as they are connected. The rig will have a “root” which acts as the source of all other bones, transforming a root bone will also transform all other bones in the rig. In human characters the hips usually act as the root of the character. Constraints can be added to joints to give them realistic limitations, for example restricting an elbow to not rotate past 180 degrees. As well as FK motion, where each bone follows its parent, a rig can use inverse kinematics. As mentioned previously, IK takes a root node and a position for the end effector and calculates the rotations of the bones in between so that these positions can be honored.

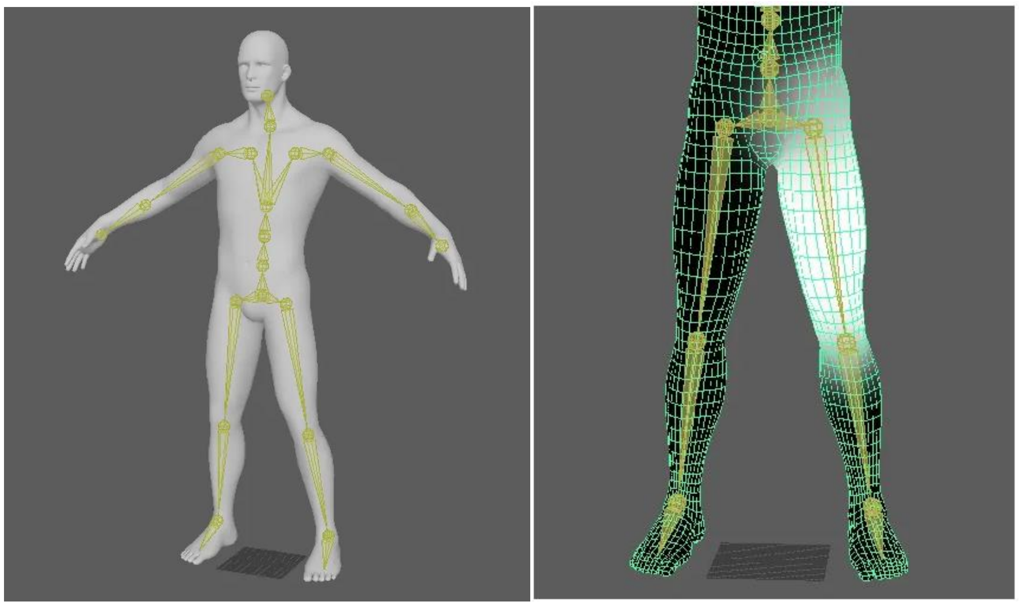


Figure 7: A standard human rig and skinning of a leg bone (Game Dev Insider, n.d.)

Skinning is the process by which bones are mapped to vertices on the mesh. To maintain a smooth look for organic characters multiple bones can contribute to the final position of a vertex. At a knee, both the upper and lower leg bones would likely contribute. Weight painting is the process by which artists (or in our case an algorithm) dictate the weight of influence each bone has on the mesh vertices. Any vertices with no associated bone weighting will not be transformed along with the rig. Traditionally, both rigging and skinning are done by hand using DCC tools, however recently several solutions to algorithmic rigging and skinning have become popular to alleviate the artist workload (Mixamo, n.d.).

4 Implementation

The creature generator was written in C# using the Unity game engine. Unity was selected for a few reasons. Unity is an industry standard and has wide adoption, this creature generator is intended to play a role in a Unity based game which is currently in development. It also offers support for compute shaders more readily than its main counterpart, Unreal Engine. While they were not utilized here, they offer a good avenue for future optimization of some parallelizable tasks. Additionally, the researcher of this project had little prior experience with Unity and felt that implementing in an unfamiliar engine would be a valuable learning experience, even if progress was slower.

The final implementation is fairly similar to the initial idea presented section 2.2. Each stage in the process and the reasoning behind it is explained here in step by step manor. Some specific code level details are omitted for brevity but the source code is available at *https://github.com/kingkristo*. Comments should be verbose enough to answer any outstanding questions. An additional element which was decided on during production was having three core creature types (similar to No Man's Sky): walking, flying and swimming. Each would build upon the same base generation system, but with some specific variations for each creature type.

4.1 Body Generation

4.1.1 Linear Interpolation

The first step of generation is creation of the base body, this base body then informs the attachment of limbs. A few methods of generating a base body were considered. Taking a base sphere mesh and modifying each vertex programmatically was deemed too complex, especially to create a body that looks like a creature. Choosing from a set of base bodies with pre-set limb attachment points was too limiting.

In the end a hybrid of these two methods was implemented. Using a single pill shape as the base, a set of nine base bodies were constructed, based loosely on different animals and their body types. It should be noted that while real life animals were used to inspire the different base bodies, accuracy to any specific real-life animals is not an aim of this project. A wide variety of interesting, strange looking fantasy animals is more desirable. If the aim is to create animals more closely tied to existing ones, a more restrictive artist centric approach (such as that of No Man's Sky) would be more suitable.

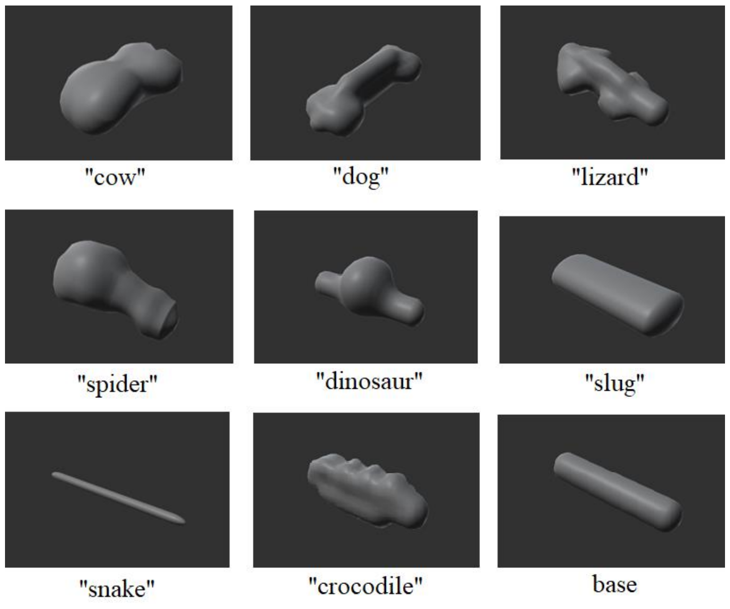


Figure 8: Base bodies

Three of these bodies are selected at random, and linearly interpolated between using a random weight between 0 and 1. For this reason it was important that each base body had the same number of vertices, and they were ordered in the same way.

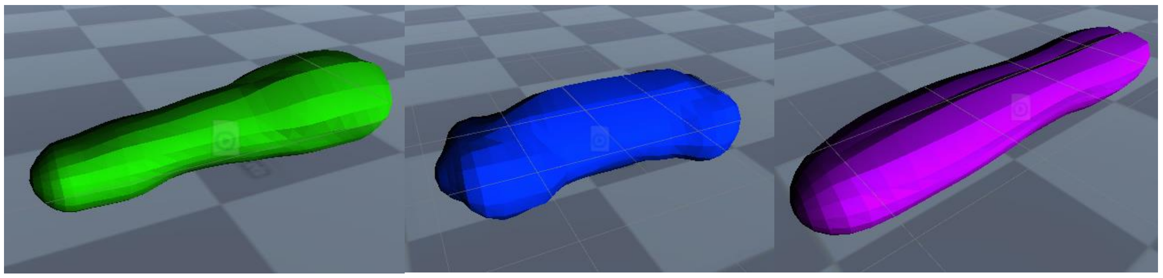


Figure 9: Lerped bodies

It was already decided that the creatures would be created in a low polygon (low poly) style, in order to fit in with the game the creature generator is intended to be a part of. This had the added benefit of greatly increasing the performance of any parts of the algorithm where every vertex on the mesh is inspected (for example lerping base bodies). However, when some premade limbs were added it became clear that unity was doing some smoothing to the body, which made it contrast the low poly limbs.

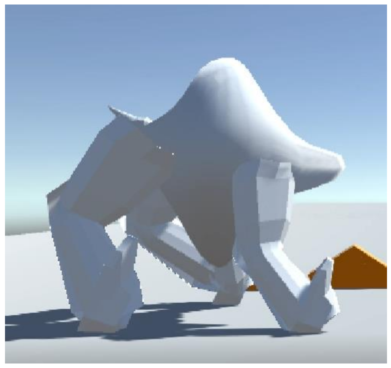


Figure 10: An early creature

To remedy this, a shader was constructed to do per face shading on the creature, each face of the mesh would be assigned a single colour, with no gradients or shading. The shader itself simply splits the material position into DDX and DDY partial derivatives. Final face colour = ((DDX x DDY) · ld) * mc, where ld is the light direction and mc is a material colour.

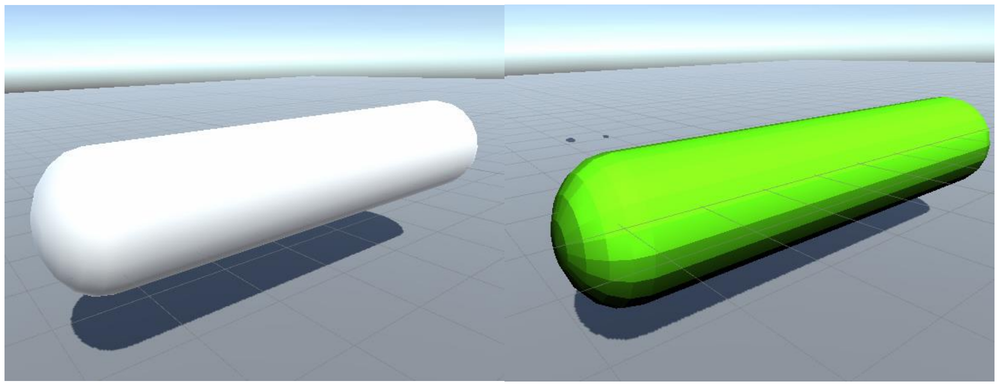


Figure 11: A capsule with and without low poly shader applied

4.1.2 Free Form Deformation

While lerping base bodies was a good starting point, it was decided that it didn't offer enough variation on its own, the bodies were all generally the same size. A few methods, such as using blend shapes, were explored to add more variation. In the end Free Form Deformation was selected due to its ability to create smooth, organic looking deformation easily. The method implemented is the same as that of (Sederberg and Parry, 1986). A 3 x 3 x 11 grid of control points is constructed around the mesh. Each 3 x 3 "slice" of the grid is randomly scaled in x , y and z directions. The swimming creature type has a smaller x and z, and larger y range of random values than walking / flying creatures. This is because fish tend to have flatter bodies than land animals.

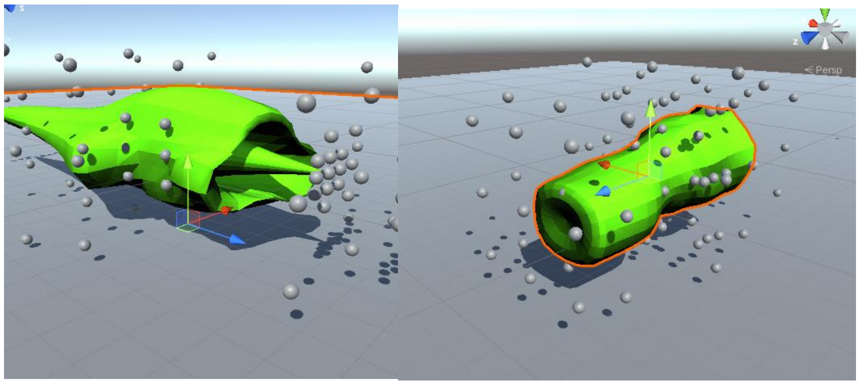


Figure 12: Initial attempts at FFD

Randomly scaling the slices in the z dimension meant that they would frequently get out of order, resulting in the body being concave on either end. To remedy this, instead of just randomly scaling the z dimension of each slice, the z value of the previous slice is stored, and some random amount is added to this to get the z position of the current slice. This method successfully added a lot of variation to the base bodies.

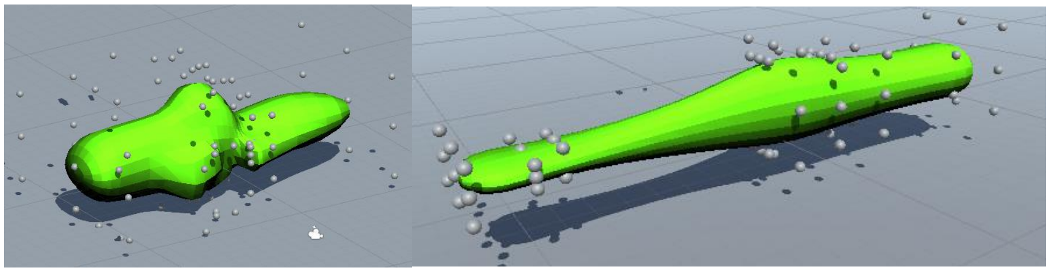


Figure 13: Bodies modified by FFD

4.1.3 Auto Rigging and Skinning

After the body is created, it is then rigged and skinned. The number of bones is driven by the size of the body. Then the “neck” and “tail” vertices of the mesh are found. Originally these were just the two points on the mesh with the greatest distance between them, the idea being that this would allow different angles of spine (for example upright bipedal style). However once FFD scaling was implemented this method no longer worked in all cases.

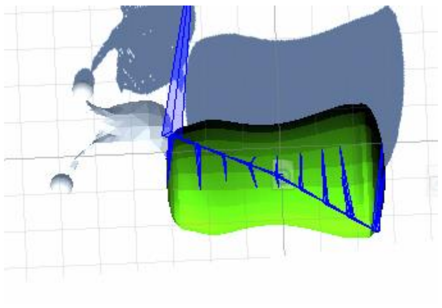


Figure 14: Failed spine placement

Drawing a curve through the body to form a spine was trailed at one point. However, after consulting with riggers and animators it was decided that having the neutral pose be anything but a straight spine could cause issues with achieving natural looking deformation while animating, this is why character models are usually rigged in a T or A pose, with straight limbs.

The initial method was revisited with some additional constraints. In the final algorithm the head and tail vertices are the two furthest away vertices that lie within a small threshold of the y and x coordinate of the mesh center. The head vertex is whichever of the two is furthest in front in the z direction. This successfully places the head and tail at logical positions.

Rigging of the spine itself is handled differently for swimming creatures and flying/walking ones. For the purposes of animating the swimming creatures, the head is the root bone and bones are added up to the tail. For walking and flying creatures, moving the neck and tail is simpler if the center is the root, as such bones project in opposite directions from the center towards the head and tail to form the spine.

The final stage of rigging the body is adding attachment points where body parts can be attached. To do this, at each bone (other than the first and last) the closest vertices on the right side, left side and top of the mesh are found. New game objects are created at each of these points, these act as anchors for body parts to be attached. Each attachment point is parented to its corresponding bone, this ensures that the limbs move with the body. These new game objects are stored in an array for easy access, as well as the distance between the left and right points (used to attach limbs later).

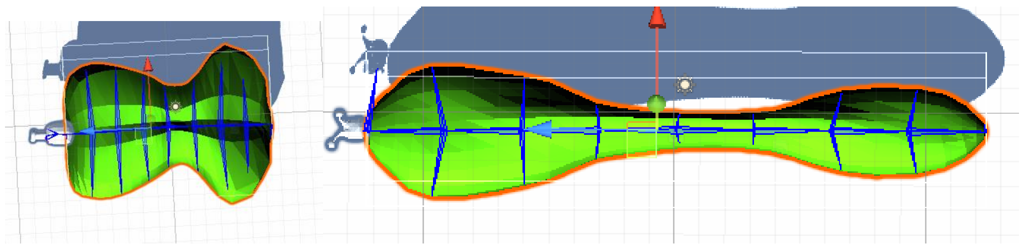


Figure 15: Successful spine placement

A capsule collider is added to the center of the body with a random height, this capsule height dictates how high from the ground the creature will be.

The body is then skinned so that it moves with the bones. The method to do this is simple, each vertex is influenced by its nearest 4 bones. The wight assigned to each bone is dictated by the distance of that bone to the vertex. Having 4 bones influence each vertex ensures that deformations are smooth and there are no harsh bends.

4.1.4 Body Adjustments

After revisiting Spore, it was noted that a crucial stage of creature creation was bending the spine into an interesting shape, this added a lot of variety and personality (figure 1).

This was only implemented for flying and walking creatures as fish generally don't have bent spines. Due to the fact the rig root for walking and flying creatures is the center, rather than the head, bending the tail and neck was simple. First the maximum angle that could be applied to each bone while preventing the neck or tail from rotating past 100 degrees was found (200 / bone number). It was important the spine didn't bend past this point to prevent self-intersection and unnatural looking creatures. Two random angles are found between 0 and the maximum, for both the tail and neck. Each rotation is then applied to every bone in the tail and neck to randomly bend them.

A simple spherical head is then added to the tip of the neck, this is scaled based on the distance between attachment points at the second neck bone. A more involved random head generator was started, but ultimately scrapped due to time constraints.

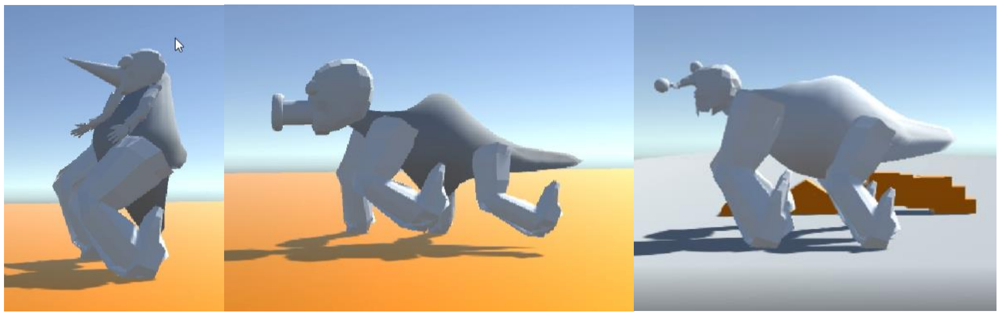


Figure 16: Early creatures utilizing a head generator

If after adding the head or bending the spine any parts of the creature intersect the floor, the capsule height is adjusted to accommodate.

To both add visual variety and test the capabilities of the procedural animation, it was important that both the speed and size of creatures would vary. To this end the body is randomly scaled between 0.5 and 2. This scale is then used to set the initial movement speed of the controller script, which is further modified by some random amount. This resulted in a range of creatures both small and large, as well as fast and slow.

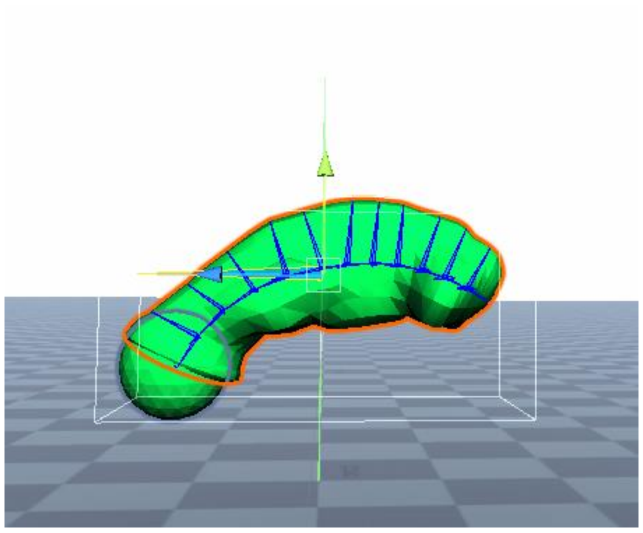


Figure 17: A creature body after adding the head and adjusting spine with a bend

4.2 Limb Attachment

Before animation, limb attachment is where the main differences between flying, swimming, and walking creatures exist. Each has a unique method of limb placement suited to the creature type.

4.2.1 Flying Creatures

Flying creatures have 4 different body parts that can be attached, two styles of "buzzing" wings, one style of "flapping" wings and legs. "Buzzing" wings are modelled after insect style wings, two styles were created: one inspired by butterfly wings and one inspired by fly wings. These wings have a basic script attached to them which rotate the wings at frequent intervals, giving the impression of fast insectoid wings.

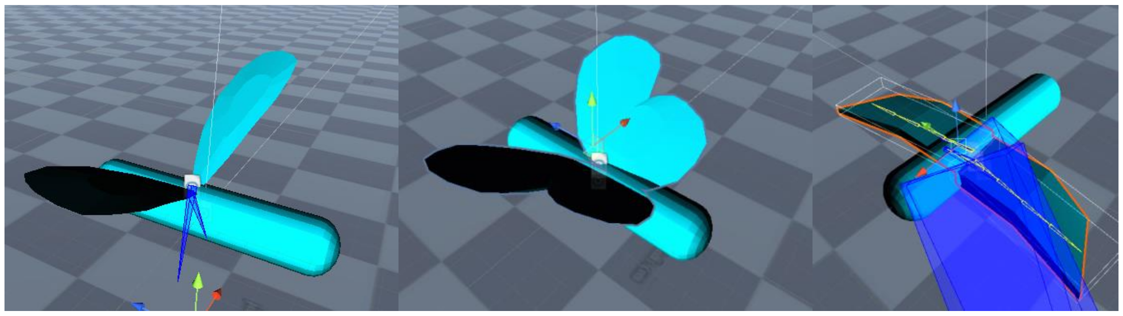


Figure 18: Wing types

"Flapping" wings are modelled after bird wings. Bird wings flap slower than insect wings and can be seen bending with movement. To reflect this, these wings are rigged with 4 bones. Each bone uses a damped transform (Damped Transform, n.d.). A damped transform allows damping of a game object's position and rotation to that of a constraint game object. The constraint game object will essentially follow the transform changes of the source object while aiming at it, with some damping. It's useful for creating flowing objects like tails and tentacles, where each bone follows its parent. Here it creates a convincing flapping effect. It was important to only have a small number of bones to prevent the wings from flopping in the air too much and looking less solid.

Since the flying creatures are always in the air, the legs need to hang in the air convincingly. A few approaches were trailed for this. Ragdoll legs flopped around too much and looked silly. The same damped transform approach used for the wings was attempted, and while the motion itself was good, the knee direction was not respected, and the length of the bones did not stay consistent (legs would appear to stretch at high speeds due to the delay in position change bone to bone). To fix this, the damped transform method was combined with a simple two bone IK for the leg (Two Bone IK, n.d.). A two bone IK rig takes a root bone, middle bone, and tip bone. The root will stay stationary, the tip will follow a target and the middle bone will be rotated to accommodate as needed. This is a typical IK setup as previously discussed. The two bone variant is especially made for things such as arms and legs, and provides a hint object which dictates the direction of bend in the "knee". A simple two bone rig was created, with damped transforms applied. At the tip of this is the target used to direct the leg tip in the two bone IK. This method was perfect for hanging legs, as the foot target moves with the body convincingly due to the damped transform, but the two bone IK ensures that the leg bends in the correct direction bones stay at a constant length.

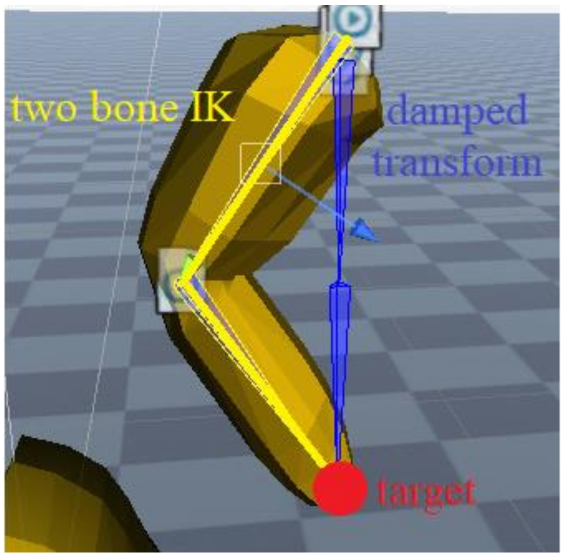


Figure 19: Flying leg setup

The algorithm for attaching these body parts is relatively simple and is presented here in pseudocode:

```
Decide on a wing style.
If "buzzing" style
    Randomly scale wings in x,y and z directions.
    Attach wings to the top attachment point of a random bone in the front half of the spine. //in nature insect wings are always attached on the front half
If "flapping" style
    Uniformly scale wings by a random amount.
    Attach wings to the left and right attachment points of the middle spine bone.
Decide on a number of leg pairs between 1 and 8.
Decide if legs will be closely packed.
If bones are still available to attach limbs
    Decide a random uniform leg scale driven by the number of legs //legs are more likely to be small if there are more of them and vice versa
    If "closely packed"
        Attach the first leg pair to the left and right attachment points of a random bone in the front half of the spine.
        For each subsequent pair of legs
            Attach to the first available bone which will not cause intersection with another pair of legs.
    Else
        Attach legs at evenly spaced intervals so that they take up the length of the body.
```

Since flying creatures don't rely on legs the same way as walking creatures, the algorithm to attach legs is much more basic. The aim was to allow leg placements similar to that of insects such as the dragonfly where legs are clumped at the front, as well as allow for bird like legs and more fantastical arrangements.

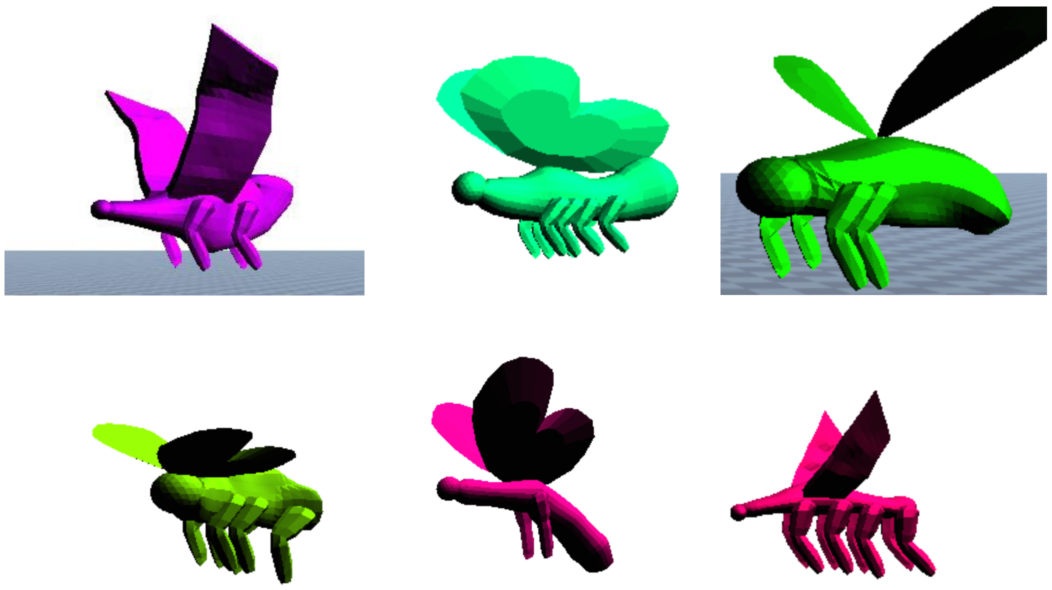


Figure 20: Example flying creatures

4.2.2 Swimming Creatures

Swimming creatures have three types of body part that can be attached: tails, pectoral fins and dorsal fins. Each has two alternate styles. All of them are set up in the same way, with bones starting at the point where the fin attaches to the body, and ending at the point furthest from the fish. Fins have damped transforms applied in the same way as wings, but with a greater number of smaller bones. Increasing the number of bones gives more of a flowing motion than what was desirable for wings, emulating fins underwater quite effectively.

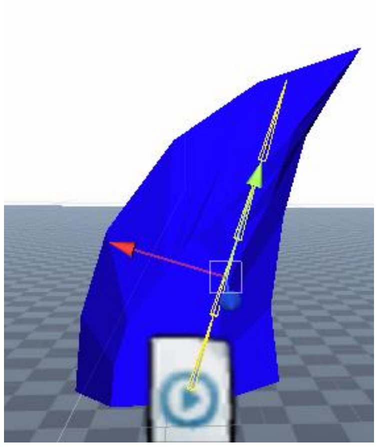


Figure 21: A fin bone setup

Fin number and placement varies from fish to fish, and creating something that reads as a fish/swimming creature to a player is relatively easy. As such the fin attachment algorithm is largely random, with few constraints. It is presented here in pseudocode:

```
-Decide on random number of dorsal fins
For each dorsal fin
    If bones are available that don't intersect with existing dorsal fins
        -Decide on dorsal fin style.
        -Randomly choose an available bone and attach to its top attachment point.
        -Randomly scale fin in x,y and z.
-Decide on random number of pectoral fins.
For each pectoral fin pair
    If bones are available that don't intersect with existing pectoral fins
        -Decide on pectoral fin style.
        -Randomly choose an available bone and attach to its left and right attachment points.
        -Randomly scale fins in x,y and z.
-Decide on tail style.
-Attach tail to the tail bone tip.
-Randomly scale tail in x,y and z.
```

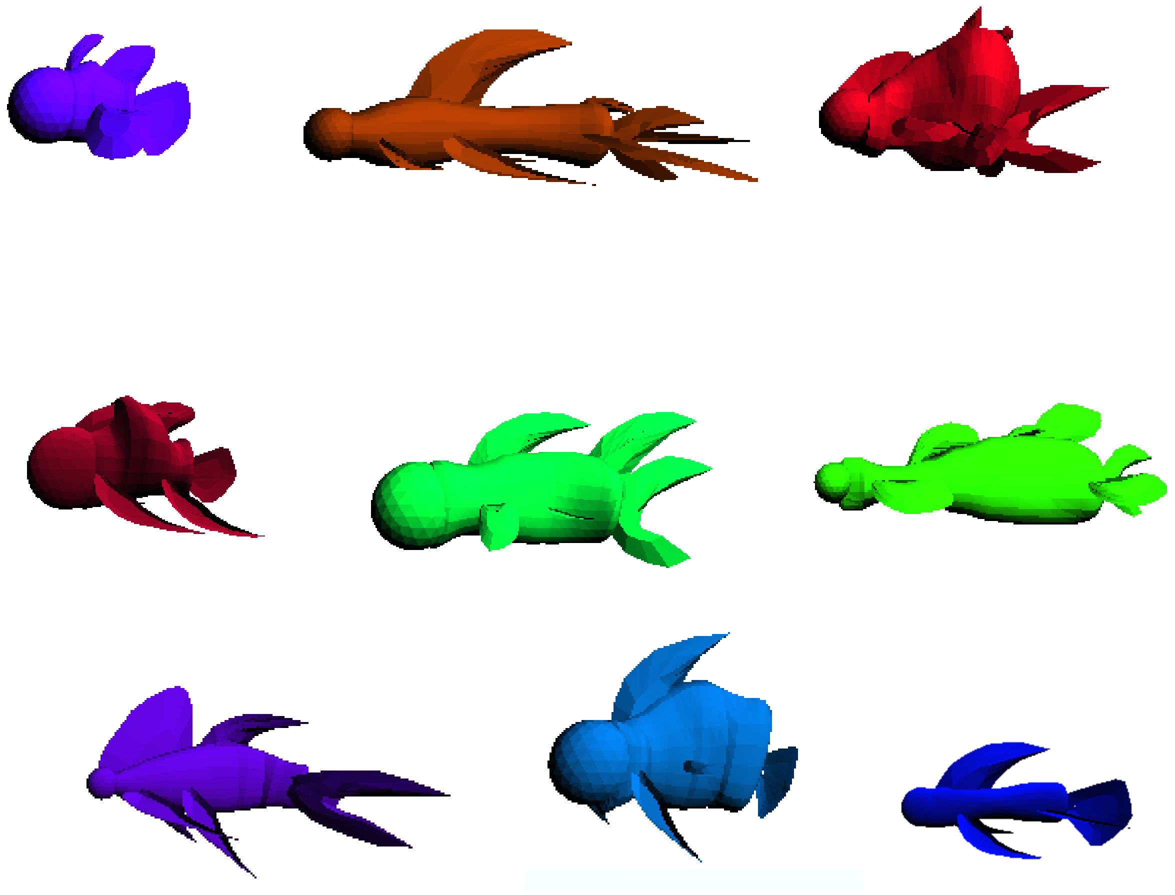


Figure 22: Example swimming creatures

4.2.3 Walking Creatures

Walking creatures have 2 types of body part that can be attached: legs and arms. Each one uses a simple two bone IK rig, similar to the flying legs. The target positions are driven by a procedural animation algorithm which will be explained in a later section. The knee orientation of the legs is chosen at the start of the generation algorithm, it can be either forward, backward or sideways to represent different types of legs that exist in nature.

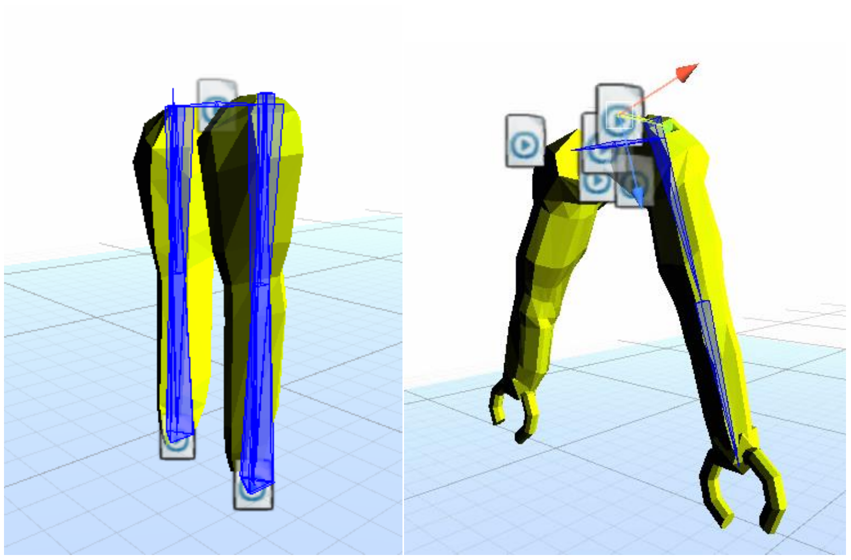


Figure 23: Walking legs and arms

Limb placement for walking creatures was the most complicated to achieve. Because legs are directly tied to the movement animation of land base creatures, having them attach in a convincing way was vital. There were 4 main criteria for attaching legs:

- Do not have legs overlap.
- Make sure legs appear balanced (eg. The creature doesn’t look like it would fall over).
- Have legs attach to the parts of the body that would support legs if it was real (eg. the widest points like hips).
- Having a variety of different style creatures with fun designs.

Balancing these requirements in all cases is challenging given the variety of possible base bodies. Initially, using mesh volume calculation to find the parts of the mesh which would support legs was considered, however a much simpler method was found to be effective. Simply using the distance between left and right “hip” connection points gave a thickness for that part of the mesh very easily and with little computation. The algorithm presented in pseudocode here uses these distances to attach limbs and gives good results in most cases:

```
-Find bones with largest and smallest distance between left and right attachment points //"hip width"
If ( body length is > 3 * distance from the ground OR the smallest hip width is >= 0.75 * largest)
    - Add as many legs as possible without causing intersection/overlap.
    //if the body is close to a tube then add as many limbs as possible (an excuse to make creatures with large legs have more than two pairs of legs)
    //if the body is long but close to the ground then legs will be small, need to add lots to support the creature like a centipede.
Else
    - Add legs to the widest point
    - Find the next widest bone which does not overlap with existing legs
    If (next widest point is >= 0.75* original widest point)
        Add legs to second widest point
        //if the second largest non-overlapping point is a similar size to the first, then we know this is a second wide point of the mesh and not just a small blob, as such it's probably a good place to add legs. The fact that its non-overlapping with previous legs should mean it's far enough away from other legs to be balanced.
    Else
        If the large point where legs have been already attached is in the center (give or take 2 bones)
            - Stick with one pair of legs //t-rex style
        Else
```

```
            -Find the closest available bone to the opposite of the current leg and put
            a leg there // balance
-Randomly decide whether or not to add arms based on how close first pair of legs is to the head
// make it more likely to add arms the more space is between the head and first unusable bone
If add arms
    -add arms halfway between the head and first unavailable socket
```

This method gives good results in most cases however there are some edge cases where the algorithm fails to achieve convincing results. If a very large pair of legs is added to the front or back of the creature to start with, it may overlap all available slots. This can leave the creature unbalanced with no free spaces to add more legs to correct. Secondly, if spine bending results in the thickest part of the creature being very close to the ground, then very tiny legs may be attached (leg size is driven by the distance between the attachment point and the ground). Finally, when legs are added the algorithm finds any bones which the leg will overlap with during its walk cycle and marks them as unusable. However, it does not account for the size of legs when adding them, it simply cares about which bones are available and not. This means that if a very small pair of legs is added to start with, it will only mark a small number of bones on either side as unusable. A large pair could conceivably be added to the first available bone next to the small legs and overlap them. These edge cases could be accounted for given more time, but unfortunately were left unsolved at the time of this project’s completion.

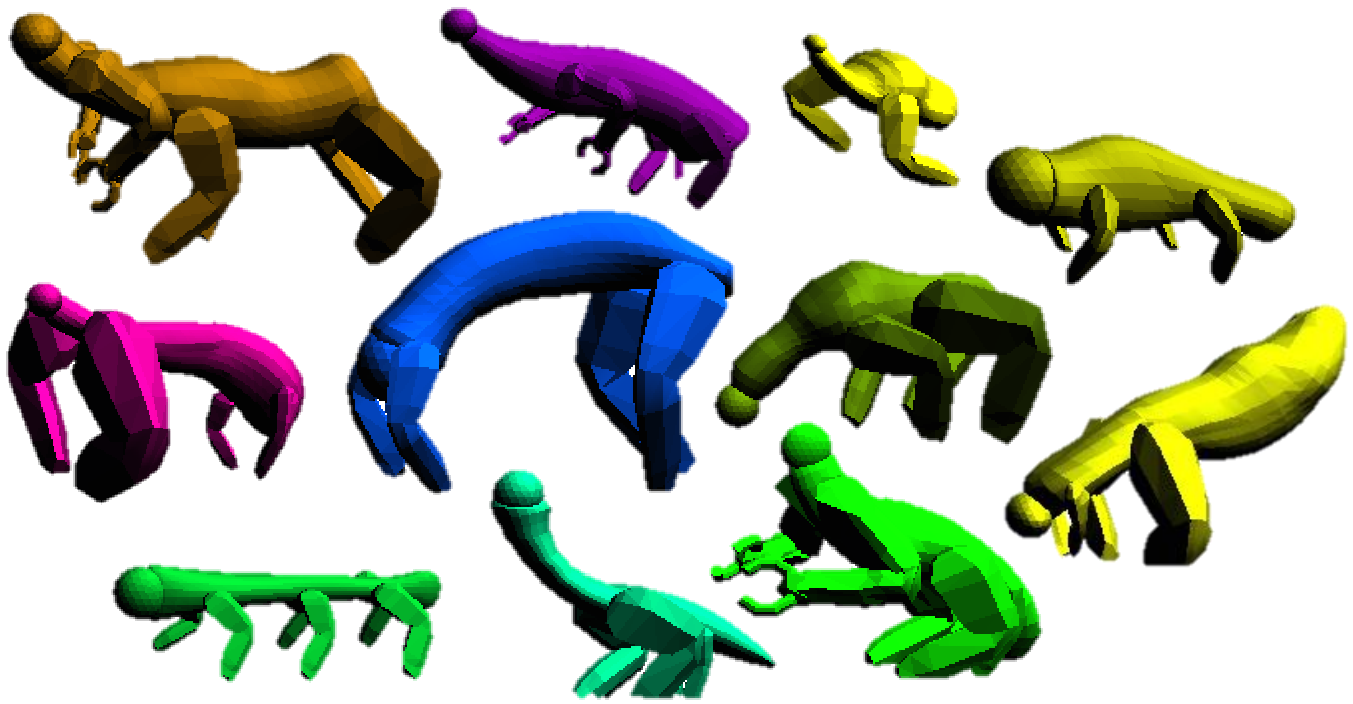


Figure 24: Example walking creatures

4.3 Animation

Each creature type has both movement and idle animations fully driven programmatically. Each creature type has a unique animation solution, with walking creatures being the most complex of the three.

4.3.1 Flying Creatures

Animation for flying creatures is relatively simple. Wings of both types rotate up and down, with insect style “buzzing” wings flapping at a much higher frequency as in nature. Flying creatures move up and down in the air when idle, for “flapping” wings the movement is faster and is in time with wing flaps. Creatures with buzzing wings stop moving up and down when moving forward, flapping creatures do not (to simulate bird style flying). From the middle of the body to the tail tip damped transforms are applied to every few bones, this makes this half of the body move with the movement of the creature. Damped transforms are used sparsely, as with flapping wings, to maintain a solid look to the body. The other half of the body bends in the direction of rotation.

4.3.2 Swimming Creatures

Animation for swimming creatures is even simpler. Damped transforms are applied to every bone from head to tail. By simply moving the head back and forth when moving, the motion propagates through the body in a rippling motion, as if swimming in water. When idle the fish sways slightly as if in water. Any movement in the body also propagates to the fins which sway convincingly with the rest of the body.

Before damped transforms were discovered, a swimming motion was attempted via a shader by using a sin function to drive displacement along the body. While this method seemed to work well on the body, it didn't work once limbs were attached as it would cause them to become disconnected from one another.

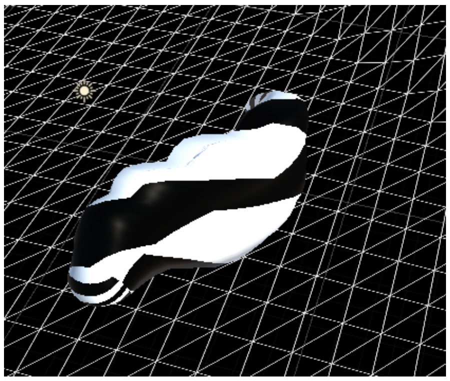


Figure 25: An early swimming creature with swim shader applied

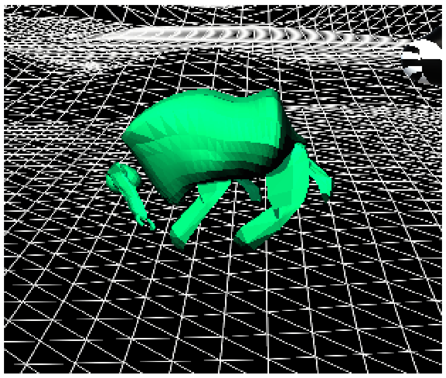


Figure 26: A creature with attached limbs with swim shader applied

4.3.3 Walking Creatures

The idle animation of walking creatures is simply the root bone being moved up and down slowly frame by frame to evoke the creature breathing, the amount of movement is modified based on the creatures' size. When turning the creature bends its neck and tail towards the direction of rotation, with the rotation amount being driven by the number of bones.

Initially these animations were handled within an update method, however once the project was built and packaged the idle animation did not look the same as it had in the inspector. This is because the number of times the update method is called varies depending on the environment. To alleviate this all update functionality was switched to a fixed update method, which is guaranteed to execute once per frame.

The procedural walking algorithm used here is similar to many modern methods used for games (Skovbo Johansen, 2009), with modifications made to ensure movement is smooth for any creature size, speed and number of legs.

Each leg has a target which dictates foot position. The position of this target is driven by a "targetMove" script. The bone which the leg is attached to is used as the "body" of the character as far as the animation script is concerned. On each frame a ray is fired from this bone towards the floor. If the distance between the foot position and the ray hit is large enough, then this means the body has moved far forward enough to warrant that a new step is taken. When this happens a new step position is calculated in front of the body, the distance of this new step from the body is driven by a step size parameter.

Different creatures walk with different gates, but often legs opposite from one another will move at opposing times. For example, humans take a step with one foot and then another. To maintain balance creatures with more legs do the same, creatures with 4 legs will usually take a step at the same time with legs diagonal from one another. This is not always the case, cats take steps with opposite legs at the same time when running for instance. However, when dealing with an arbitrary number of legs it was necessary to place some limitations on the styles of walking and these cases were not accounted for. This is a limitation of the generalized approach to animation taken here, as some specialist types of run/walk had to be abandoned in favor of a more generalizable style.

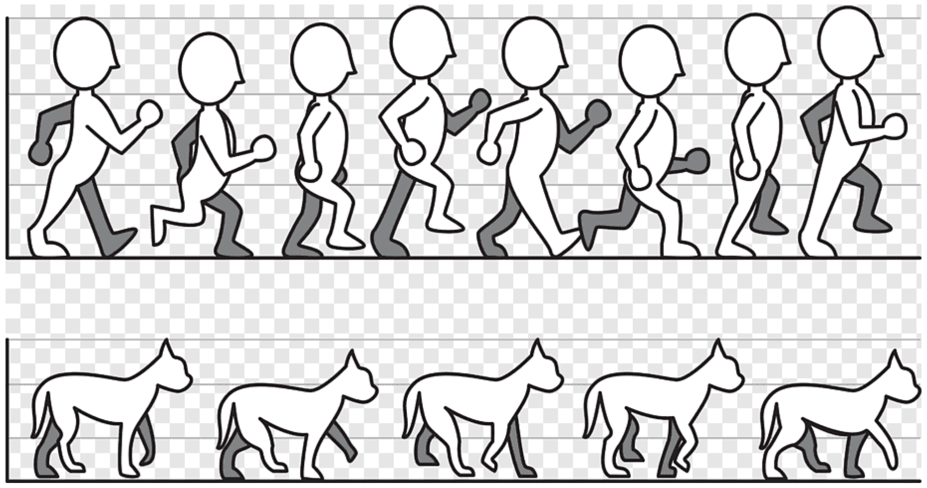


Figure 27: A typical human and animal walk cycle

As a rule, legs opposite from one another take a step at opposing times. This is implemented by ensuring that a foot will only take a step when its opposite is stationary. This caused problems when adding multiple legs, as all legs on one side of the body would take a step at the same time. To remedy this, if more than one pair of legs has been added, the opposite foot reference of the left foot is overwritten with the foot target of the previous left leg which was added. This ensures that not only do opposing feet move at different times, but diagonal feet move at the same time (as the left foot will move only when the left foot behind it is stationary, and every right foot will only move when its opposing left foot is stationary). This only happens if the current and previous pair of legs are of a similar scale (within 80%), as otherwise the scale discrepancy will cause the step frequency to be very different and moving opposite one another will not be feasible.

A step completion variable is used to lerp the foot position in between previous and new step positions. A sin function is used to raise the foot in the middle of the step. At first this step completion was simply incremented by some amount each frame based on a speed parameter. Tuning this speed parameter correctly so that walking looked natural was challenging, so another approach was used. Step completion is instead driven by how close the creature body (the bone legs are attached to) is to the new foot position, this means that no matter the size and speed of the creature the legs will always be constantly in motion when walking, with no awkward pauses because the feet made a step to fast or the feet dragging behind due to being too slow. The problem with this method was that if the creature stopped moving halfway through a step, its feet would stay hanging in the air, which looked strange. To remedy this the initial method using a per frame step increment takes over if the creature is stationary, to complete any unfinished steps and bring the creature back to a neutral pose with all feet on the floor.

The last operation per frame is adjusting hip rotation. At first the movement of legs felt disconnected from the body which stayed stationary. In nature when observing animals their body shifts depending on which feet are carrying their weight. To simulate this, the difference in height between a foot and its opposite is used to rotate bones in the spine. When a creature has its foot in the air, its “hip” will rise on that side, and fall on the other. This was a computationally cheap method of adding what looks like physically based motion to the creature.

Each time a leg is created, its size and speed are sent to the procedural animation script. The size is used to multiply the step distance, and step height. These are further multiplied by the speed if the creature has a speed multiplier above 1, this is to give the creature more of a running gate if it is fast moving.

This process is the same for arms, except they ignore the target position in the y direction, to stay in the air. Arms also have an idle animation, with hands moving up and down slightly when stationary.

The final procedural walking system adapts very well to any creature type and walking looks natural for any size, speed and number of legs. Due to the ray casting method, creatures are also able to accurately traverse uneven terrain. The main limitation is that essentially only one style of movement is represented, future work to expand this method with different styles of running which are hardcoded depending on the number of legs could add even greater variation. This is another example of generalization resulting in a poorer outcome.

5 Conclusion

Overall, this project has successfully delivered on its intended goal of creating an open-ended creature generator. It succeeds in creating a variety of creatures that for the most part look and move convincingly. There are a few limitations of the system which have been mentioned previously, most notably some edge cases in which the leg placement algorithm fails to create convincing walking creatures and the fact that only one style of walking animation is catered for.

With that being said, there is a case to be made for the usefulness of such an approach. Generalizing procedural generation to this degree has notable drawbacks. Chiefly, ensuring the quality of the generated end result is challenging and generalizing prevents variation in some cases. For example, adding an arbitrary number of legs anywhere on the creature body leads to a wide variety of creatures, but prevented them from having bespoke movement animations suited to their leg layout. These could be hardcoded in in future, but at that point you are getting close to simply having a set of pre-determined creature types (as in No Mans Sky). If the access to artists is available, creating a vast pool of premade content is likely important to creating a polished generator of this kind.

However, if access to artists is limited this approach successfully creates a wide variety of creatures from a very limited selection of component parts.

5.1 Evaluation Against Requirements

Revisiting the initial requirements, it can be said that they have all been fulfilled. That is not to say that the end product is perfect, but it does successfully meet the scope of requirements set before its completion. Requirements 1,2,3,4,5,7,8,9 and 11 are objective and were all achieved.

Requirements 6 and 10 (quality of creatures generated and their movement animation) are somewhat subjective so were evaluated via user testing. A series of users familiar with games was given free reign of the generator and asked if each creature looked and moved convincingly. On average out of 100 different creatures evaluated by 4 different testers (25 creatures each). 95% were said to move convincingly and 83% were said to look convincing. The failures in movement were primarily due to the arm animation looking strange for small arms (which move back and forth quickly despite not making actual steps). Failures in the look of creatures was mostly down to problems with the leg attachment algorithm discussed previously. While not perfect acceptance rates, and using a limited pool of testers, 95% and 83% are good enough to say requirements 6 and 10 are satisfied.

5.2 Future Work

There are several key areas of future work which offer avenues to improve the creature generator.

As highlighted in section 4.2.3 there are some edge cases which should be fixed to make the leg attachment algorithm of walking animals more successful.

Although speed was never an issue throughout development, base body generation (specifically FFD, rigging and skinning) could be translated to compute shaders to improve performance further.

A greater focus on visual additions to the creatures generated would add more variation. For example a head generation system, pre made accessories like shells and horns to attach to the body or procedural texturing beyond colour changes.

Different procedural walking styles specifically tailored to the number of legs a creature has would add more variation. For example, kangaroo style hopping for two legs, cat like running for four legs, or snake like slithering for no legs.

Finally, optimization techniques could help ensure the quality of generated creatures, so long as those methods operate in real time.

6 References

2016. No Man's Sky. Hello Games.

2008. Spore. Maxis.

Youtube.com. 2014. A Behind-The-Scenes Tour Of No Man's Sky's Technology. [online] Available at: <https://www.youtube.com/watch?v=h-kifCYToAU&list=UUK-65DO2oOxxMwphl2tYtcw&ab_channel=GameInformer> [Accessed 20 August 2022].

Hudson, J., 2013. Creature Generation using Genetic Algorithms and Auto-Rigging. Msc. Bournemouth University.

Sims, K. 1994. Evolving virtual creatures. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (SIGGRAPH '94). Association for Computing Machinery, New York, NY, USA, 15–22.

Hornby, G. and Pollack, J., 2001. Evolving L-systems to generate virtual creatures. Computers &amp; Graphics, 25(6), pp.1041-1048.

Geijtenbeek, T., 2013. Animating Virtual Characters using Physics-Based Simulation. PhD. University of Utrecht.

Tan, J., Gu, Y., Turk, G. and Liu, C., 2011. Articulated swimming creatures. ACM Transactions on Graphics, 30(4), pp.1-12.

Samyn, K., Hoecke, S., Pieters, B., Hollemeersch, C., Demeulemeester, A and Van de Walle, R. 2014. Real-time animation of human characters with fuzzy controllers.

Zhao, J and Badler, N. 1994. Inverse kinematics positioning using nonlinear programming for highly articulated figures. ACM Trans. Graph. 13, 4 (Oct. 1994), 313–336.

Aristidou, A. and Lasenby, J., 2009. Inverse Kinematics: a review of existing techniques and introduction of a new fast iterative solver. University of Cambridge.

Knowledge.autodesk.com. 2016. Inverse Kinematics (IK) | 3ds Max | Autodesk Knowledge Network. [online] Available at: <https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/3DSMax/files/GUID-516E301F-E911-429F-9337-9FA7FAD49BB6-htm.html> [Accessed 20 August 2022].

Chung, S and Hahn, J. (1999). Animation of human walking in virtual environments. 4 - 15. 10.1109/CA.1999.781194.

Skovbo Johansen, R., 2009. Dynamic Walking with Semi-Procedural Animation. In: GDC. [online] Available at: <https://www.gdcvault.com/play/966/Dynamic-Walking-with-Semi-Procedural> [Accessed 21 August 2022].

Sederberg, T. and Parry, S., 1986. Free-form deformation of solid geometric models. ACM SIGGRAPH Computer Graphics, 20(4), pp.151-160.

Game Dev Insider. n.d. Rigging and Skinning - Game Dev Insider. [online] Available at: <https://gamedevinsider.com/making-games/rigging-and-skinning/#:~:text=Rigging%20is%20all%20about%20creating,move%20when%20the%20rig%20moves.> [Accessed 21 August 2022].

Mixamo.com. n.d. Mixamo. [online] Available at: <https://www.mixamo.com> [Accessed 21 August 2022].

Docs.unity3d.com. n.d. Damped Transform | Animation Rigging | 1.1.1. [online] Available at: <https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.1/manual/constraints/DampedTransform.html> [Accessed 21 August 2022].

Docs.unity3d.com. n.d. Two Bone IK | Animation Rigging | 1.1.1. [online] Available at: <https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.1/manual/constraints/TwoBoneIKConstraint.html> [Accessed 21 August 2022].