UNIVERSITY OF TARTU

Institute of Computer Science

Computer Science Curriculum

Madis Janno

Procedural Generation of 2D Creatures

Bachelor's Thesis (9 ECTS)

Supervisor: Raimond-Hendrik Tunnel

Tartu 2018

Protseduuriline 2D olendite genereerimine

Lühikokkuvõte:

Käesoleva bakalaureusetöö raames arendati 2D olendite genereerimise süsteem ning selle süsteemi implementatsioon programmeerimiskeeles JavaScript. Süsteem tekitab mitmekesiseid olendeid ning nendega seotud andmed, sealhulgas skelett, geomeetria ja tekstuur. Bakalaureusetöö sisaldab süsteemi kirjeldust. Süsteemi iga sammu kohta on välja toodud tähtsamad põhimõtted ning seletatud mõned implementatsiooni üksikasjad.

Töös analüüsitakse süsteemi tervikuna ning selle implementatsiooni. Tuuakse välja süsteemi probleemid ning nõrgad kohad ja mõõdetakse implementatsiooni jõudlust. Töö lõpus tuuakse välja süsteemi kasutusvõimalused ja võimalused selle edasi arendamiseks.

Võtmesõnad: Arvutigraafika, protseduuriline genereerimine, JavaScript, metapallid

CERCS:

P150	Geomeetria, algebraline topoloogia
P170	Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)
P175	Informaatika, süsteemiteooria

Procedural Generation of 2D Creatures

Abstract:

The purpose of this thesis is the development of a system capable of generating a large variety of 2D creatures and their associated data, such as skeletons, meshes and textures. A JavaScript implementation of the system was developed for this thesis. This thesis contains a description of the developed system and a description of each step of the generation process and its principles with some additional notes about the specifics of the implementation.

The creature generation system as a whole and its implementation are analysed and their advantages and drawbacks brought out. The performance of the implementation is also tested. Several possible improvements are proposed at the end of the thesis, as well as possible uses.

Keywords: Computer graphics, procedural generation, JavaScipt, metaballs

CERCS:

P150 Geometry, algebraic topology

P170 Computer science, numerical analysis, systems, control

P175 Informatics, systems theory

Table of Contents

1 Introduction

Procedural content has appeared in many popular computer games both recently and in the past. Minecraft[1] (2011), with its endless procedurally generated terrain, reached 144 million sales at the beginning of 2018 [1]. Spore[2] (2008), with procedurally generated creatures, planets, buildings and more, was one of the most awaited games of 2008 [2]. More recently No Man's Sky (2016), with procedurally generated planets flora and fauna, was considered one of the most anticipated games of 2016 largely because of its generation systems [3].

With procedural generation, data is generated by algorithms, rather than by artists, level designers or modellers. This can include terrain [4], cities [5], planets, trees [6], models, music and much more. This means that a team of programmers can create a relatively large amount of content, which would traditionally require large teams of artists. An example of this is SpeedTree[3], which is a piece of software used for generating vegetation in movies and games. SpeedTree allows artists to manually alter the important parts of a generated tree model, while letting procedural generation do a large portion of the work.

The focus of this thesis is the procedural generation of creatures, which has been previously done by games such as Starbound[4] (2016) and the previously mentioned Spore and No Man's Sky. Spore and No Man's Sky are analysed more closely in chapter 2 of this thesis. The reason for this thesis is that not many specifics are known about how these systems work, due to their closed-source nature as commercial products. The system described in this thesis most closely resembles that found in Spore. What sets it apart from Starbound and No Man's Sky is that creatures are not generated from a pre-generated list of parts, but rather are freely alterable.

The purpose of this thesis is the development of a system that can procedurally generate a large variety of 2D creatures, which can also be animated. For this purpose the system generates a skeleton, mesh, texture and all the necessary data to unite them into a single animated and textured 2D object. The necessary data is explained in chapter 3 of this thesis. The system is intended mainly for real-time use, rather than as an external utility, meaning the generation time is kept relatively low. The system is not intended to be used exactly as described in this thesis, but rather will likely be heavily altered depending on the specific

[1] https://minecraft.net/en-us/
[2] http://www.spore.com/
[3] https://store.speedtree.com/
[4] https://playstarbound.com/

use-case. Most notably the first stage of generation (the skeletal model generation) can be entirely omitted and done by hand for the best results.

A proof of concept JavaScript implementation was developed as part of the thesis. The implementation uses Three.js[5] as its graphics library and serves as proof that the system can be integrated with and generates data suitable for a modern graphics library. The implementation is publically available through a GitHub repository under the name ProceduralCreatures[6]. The system and its implementation are described in chapter 4. Chapter 4 also contains an analysis of the system and its performance, and proposes several possible improvements and use cases for the system.

The source code of the implementation can also be found in the files accompanying this thesis. The code written for this thesis is contained in index.html, and it can be run in a browser without any further setup. For information about the rest of the files refer to *README.md* or *readme.txt* among the accompanying files. Note that all included files reflect the state of the system at the time of the writing of this thesis, and some development may have occurred afterwards, meaning use of the repository is advised.

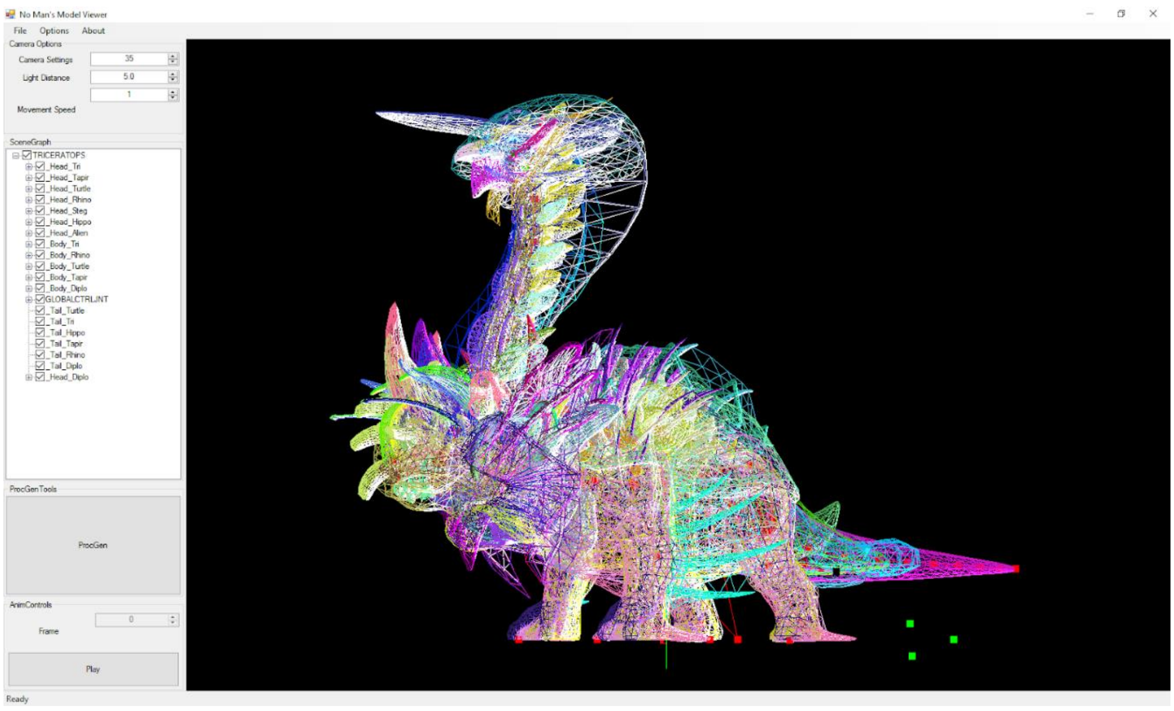
[5] https://threejs.org/
[6] https://github.com/madisjanno/ProceduralCreatures

2 Related Work

Procedural creature generation is not a new concept. It has been done before in computer games such as No Man’s Sky and Spore. There is no standard way to procedurally generate creatures, so each game has a different system. This chapter explores some of the drawbacks and benefits of the systems found in these games.

2.1 No Man’s Sky

No Man’s Sky (2016) has a large amount of procedural generation, from planets to spaceships. For the purposes of this thesis, the most relevant part is creature generation. There is no official information about the specifics of its generation systems, but a blogger going by the name of *gregkwaste* has reverse-engineered some details about the system based on the game files [7].



Camera Settings 35
Light Distance 5.0
Movement Speed 1
Frame 0

Figure 1. Triceratops model [7].

As can be seen in Figure 1, a procedural creature in No Man’s Sky is composed of parts. The procedural generation system decides which parts to use for the actual creature and which to hide, in order to put together a set of parts which will be used for the creature [7]. This means that each part has to be created by an actual artist.

This system has the drawback that players may begin to recognize certain parts of a creature, meaning they will not treat a new creature they find as a new creature, but rather as something they have seen before, but with a few parts swapped out. This problem can be alleviated by having a larger amount of parts, but that requires additional artists to create those parts. An upside of having creatures with a fairly stable shape, is that animations can be created specifically for that kind of shape. Some generated creatures can be seen in Figure 2 and Figure 3.

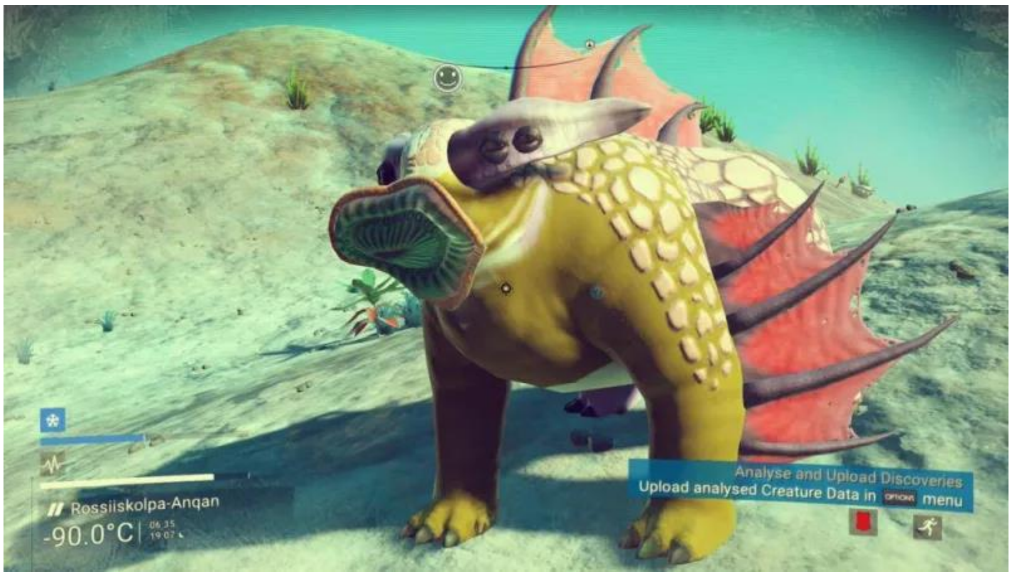

Analyse and Upload Discoveries
Upload analysed Creature Data in OPTIONS menu
Rossiiskolpa-Anqan
-90.0°C

Figure 2. Example of generated creature. [8]

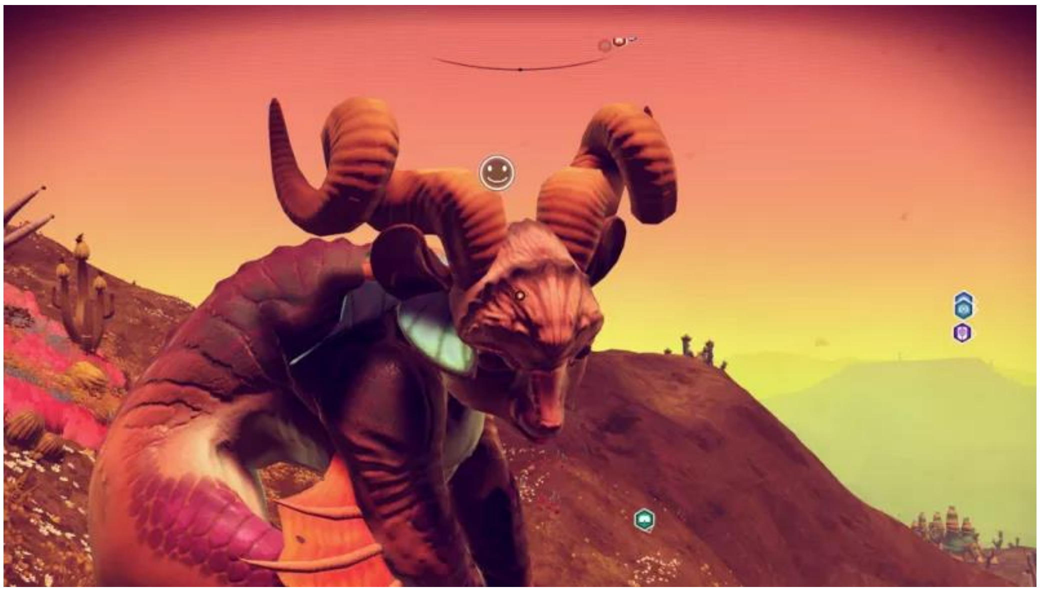


Figure 3. Example of generated creature. [8]

2.2 Spore

Spore (2008) has an extremely freeform system of creature generation and players can create their own creatures. The system allows for a high degree of control over the shape of the creature's body and the placement of limbs onto any location on the body and altering the sizes of those limbs. The system also animates the resulting creature. All creatures are player or developer designed, but the models are procedurally generated within the game. Chris Hecker wrote about his contributions to the game and to the creature generation system on his website [9] and Ocean Quigley (the game's art director) wrote about the texture generation system of the game on his blog [10]. Not all parts of a creature are fully procedural, as things such as eyes and mouths are separate parts. However they can be placed at any location.

The system used by the game has to generate and regenerate the polygons describing the creature each time the body or limbs are changed in any way. Chris Hecker chose to represent the creature through blobby implicit surfaces (often called metaballs), which are then converted into polygons. A method for calculating the placement of metaballs is mentioned, but not elaborated upon by Hecker [9].

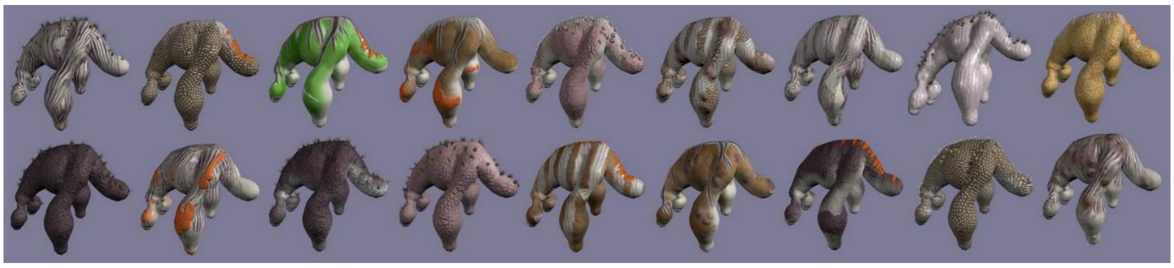


Figure 4. The first attempt to generate lots of different types of skins [10].

The system also has to generate a texture for the creature. A system was made, which can apply brushes onto the creature, convert the 3D locations to the 2D textures, and draw onto the location on the texture. Textures were then generated by a particle system, where the particles applied brushes onto the creature, and crawled around on its surface. The system could create a large variety of textures (see Figure 4) [10].

An upside of the system is that it allowed for a large amount of variety in its creatures. What could be considered a downside is that the variable shapes of the creatures required a custom animation solution which would be capable of retargeting animations onto any creature, no matter how ridiculous [9]. Figure 5, Figure 6 and Figure 7 demonstrate some player created creatures.

Figure 5. Example of a complex player created creature [11].

Figure 6. Example of a simpler player created creature. [11]

Figure 7. Example of a player created creature with many limbs. [11]

3 Theoretical Base

Due to the author of this thesis not being an artist, the system developed for this thesis was designed to make maximum use of procedural generation, rather than using any predefined parts. As such the main inspiration for the system is Spore and its method of generating creatures using blobby implicit surfaces. The system generates skeletons, meshes, textures and bone weights for a 2D model, using metaballs and a variant of the marching squares algorithm. The following subchapters describe these concepts and algorithms and the reasons for choosing those algorithms. Their uses are further elaborated upon in chapter 4, which deals with the actual details of the system.

3.1 2D and 3D Models and Skeletal Animations

The shape of models in computer games and other computer graphics applications is generally defined by a mesh of 3D triangles. This includes 2D models, which can be considered to be on a single plane in 3D. The mesh is composed of a set of points, called vertices, which define the coordinates of the corners of the triangles. A model often also contains data about how these triangles are textured or coloured [12].

Textures are essentially images which are placed onto the mesh. The vertices in a mesh are given data about what parts of the texture to use for a triangle. These are called UV coordinates, and generally range from (0, 0) to (1, 1).

One method of animating meshes is skeletal animation. Under that system a virtual skeleton composed of virtual bones is attached to the mesh, and the mesh is deformed according to the transformations (position, rotation, scale) of the bones. The bones form a hierarchy where each bone has a single parent bone, for example an arm bone would be attached to a shoulder bone. Attaching the skeleton to the mesh is called skinning. Each vertex is given values describing which bones it is attached to, and a weight describing how much those bones affect it. The sum of weights should add up to 1. The end result is that animations can be applied to the relatively simple skeleton to animate the mesh itself [13].

3.2 Metaballs

The planned system for creature generation requires a way to define the actual shape of the creature, beyond simple lines. The chosen method in this thesis is through the use of mathematical objects called metaballs. This chapter explains how metaballs work and why they were chosen.

Metaballs are organic-looking objects used in computer graphics, developed by Jim Blinn and published in 1982 [14]. The original paper did not refer to them as metaballs at the time. The initial use case was for visualizing molecular models in a more accurate way. It defined shapes as sums of density functions, with a threshold determining which parts were inside or outside the surface. The result is that the shape is defined by a smooth curve instead of sharp edges or polygons. This presents the problem of displaying them, as modern graphics technologies are built around rendering polygons instead of raytracing as in Blinn's original rendering algorithm. This is dealt with in chapter 3.3 and in more practical terms in chapter 4.4 of this thesis.

In mathematical terms metaballs can be defined as

$$D(x, y, z) = \sum_i f(r_i),$$

where x, y and z are coordinates of a given point, f is a falloff function describing the density field, and

$$\mathrm{r}_i = \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2},$$

where $x_{i,}$ y_i and z_i describe the location of the metaball. The result is that metaballs describe circular shapes, but also create connections between each other, when the sum exceeds the threshold outside of the defined radius of the metaball, generating more complex shapes. The way metaballs connect is demonstrated in Figure 8.

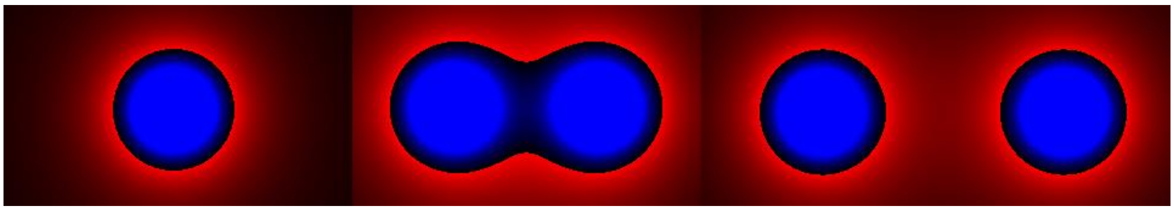


Figure 8. A single metaball (left), two metaballs close to each other (middle), two metaballs distant from each other (right). Red indicates values below threshold, blue indicates values above threshold

The behaviour of creating connections and adding together is what makes metaballs useful for this thesis, as they can for example create a natural looking joint where two parts of a creature meet. The generated shapes are also round and lack sharp edges, which should make the model look more natural and organic.

Another useful property is that it is simple to assign attributes to metaballs, and to calculate an interpolated value for that attribute at any given coordinate. The weight of a particular metaball in interpolation can be found through the formula

$$w_i = \frac{f(r)}{D(x, y, z)},$$

where w_i is the proportion of the metaball with the index i. This property is used in both skinning (chapter 4.5) and texture generation (chapter 4.6).

3.3 Marching Cubes and Marching Squares

The use of metaballs requires a method of extracting either the surface of the generated shape or an outline of it, as the alternative is calculating values for every displayed position every frame. The marching squares algorithm was chosen for outline generation. This chapter explains the basics of the algorithm and why it was chosen and its 3D version called marching cubes.

Marching cubes was developed by William E. Lorensen and Harvey E. Cline and published in 1987. The purpose of the algorithm is to process 3D data, and to generate a constant density surface (isosurface) based on that data. Its original use case was in medical visualizations, as it could be used to generate 3D models based on computed tomograph (CT) and magnetic resonance imaging (MRI) data [15].

Marching cubes generates a surface by sampling 8 neighbouring data points in the shape of a cube and then moving onto another cube until all the data is processed. The user or the application specifies a threshold value which defines the surface. The value located at each vertex of the cube is compared against the user specified threshold and given a value of 0 if the value at that point is lower than the user specified threshold (and the vertex therefore outside the surface) and 1 if the value is higher (and the vertex therefore inside the surface). The result is 8 vertices each with 2 possible states, resulting in 256 ways the surface can intersect the cube. The original paper lowers this to 14 cases through symmetries and complementary cases (see Figure 9). Triangles are then generated based on the particular case and later the positions of the triangle vertices are adjusted through linear interpolation.

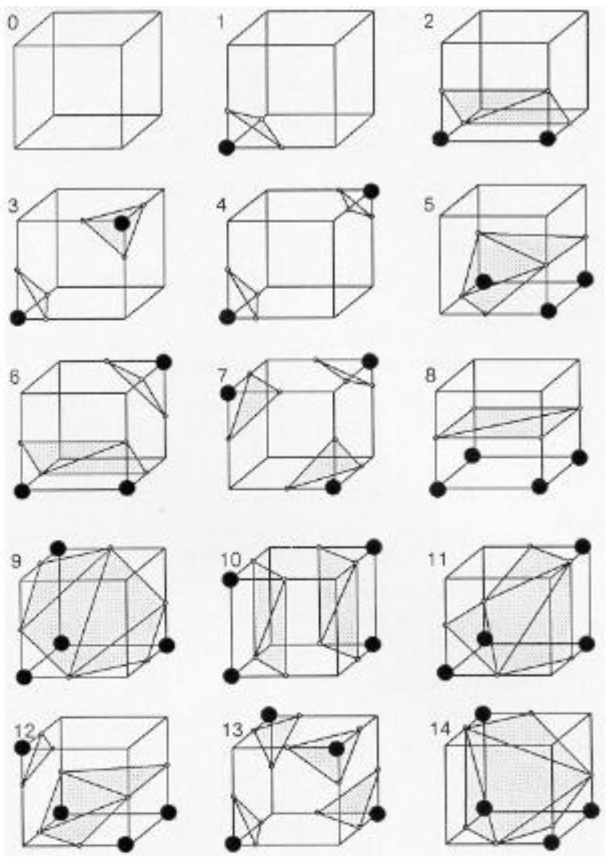


Figure 9. Triangulated Cubes. [15]

Marching squares, as the 2D equivalent, generates isolines instead of isosurfaces. As the name implies, it samples 4 neighbouring data points in the shape of a square. The result is a set of 16 cases. The algorithm was chosen for this thesis as it and its cases are very simple and easy to implement. The algorithm however does not generate triangles, but rather the outline of a shape in lines. As a triangular mesh is required, the outline must be processed further and converted into triangles. This is dealt with in chapter 4.4. The algorithm used in this thesis is an altered variant of marching squares. The details and cases of the variant are described in chapter 4.3.

4 The Developed System

The intended use case for the system is the real-time generation of visually interesting and varied 2D creatures with a minimal amount of limitations on the shape and nature of the creature. A proof of concept implementation was written in JavaScript for this thesis with the Three.js library. The language and graphics library were chosen due to familiarity by the author, and in the case of the language, to make it possible to develop and run the application in any system with a web browser.

Due to having to work with a modern graphics library, the system has several outputs which should be combined by the graphics library: a skeleton for animations, a mesh for rendering, and a texture to be placed onto the mesh. These are the usual components of an animated 3D object. The developed system does not create animations, but only the capacity to be animated. The implementation does have some extremely rudimentary procedural animations for testing purposes.

The developed system is composed of 6 steps (see Figure 10). The specific data sent from one step to the next is described in each individual subchapter of this chapter. The steps mostly correspond to the actual structure of the code, where each step constitutes a single function, or a part of the generation that requires a different set of inputs. A notable exception is that the mesh generation step has greater importance in the implementation, as it collates the generated data into a model and also contains the code for skinning.

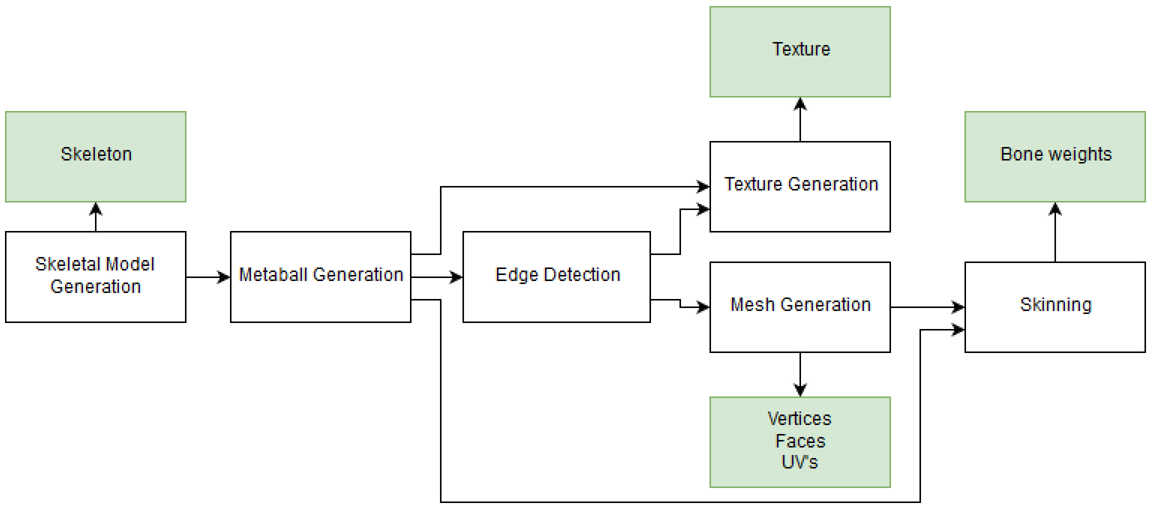


Figure 10. Steps of the developed generation process, green signifies final outputs, arrows indicate data flow.

The following subchapters each analyse a step in the creature generation process, with the last chapter being an analysis of the complete system as a whole.

4.1 Skeletal Model Generation

The first step of creature generation is the generation of bones and the skeleton of the creature. The skeletal model serves as the initial input for the rest of the steps, and determines how the creature will look and how it will act when animated. Both of these are very use case specific. The generation system outlined here is therefore meant for demonstrating and testing the capabilities of the system, but will most likely be heavily altered in actual use. This chapter outlines what data is generated during this step, describes the requirements for a quality system of skeleton generation, and explains the skeletal generation system created for this thesis. Alternative systems are proposed in the final analysis of the system in chapter 4.7.4.

Only two systems interact directly with the skeletal model. It serves as the input for metaball generation and serves as the basis for animating the created model. Therefore, the system must generate data for both purposes. The implementation generates a number of bones as the skeleton for the creature. Each bone is then given variables outlined in Table 1, which will serve as instructions for the metaball generation step (chapter 4.7.4). The number and nature of the generated variables will naturally change if either system is altered, but the same principles will likely still apply regarding how these variables propagate through the system.

The requirements of a quality skeleton generator are hard to define, as they will vary wildly depending on the actual use case for the system. For example, a horror game and a cute game will require differing systems of generation, as some strange shapes and behaviour could even be desirable in the former case. However, some generalizations can be made. Due to its nature as the input for all further systems, the skeletal model has to account for flaws or limitations of the other systems. The next following paragraphs describe some principles that should be followed.

The skeleton should avoid self-collisions and limit proximity between limbs. If two parts of the model are too close to each other, they will be connected due to the nature of metaballs. This causes areas that will be highly stretched (see Figure 11) during animations and therefore should be avoided.

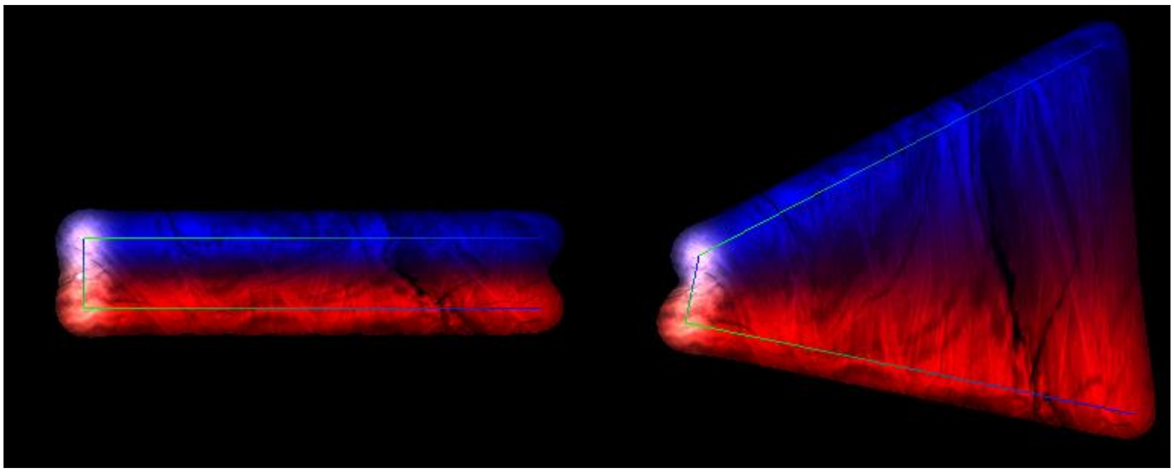


Figure 11. Example of connected limbs and stretched regions.

The skeleton generator should avoid placing smaller bones deep inside the internal surface of larger bones. For example, a thin limb inside the surface of a thick spinal bone. While not an overwhelming source of further problems, this will cause the limb to rotate around a point inside the surface of the creature, meaning it will very easily begin colliding with the rest of the creature or have a strange look when animated (see Figure 12). This is mostly an aesthetic choice, but should likely be avoided due to creating visible seams in the creature.

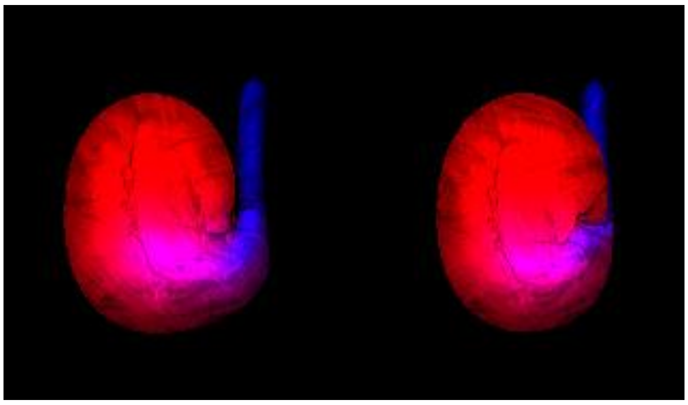


Figure 12. Example of a limb outside the body (left) and a limb inside the body (right) rotated 90 degrees.

Adjacent bones should have similar thicknesses. Due to the skinning data being affected by metaball size, adjacent bones of unequal thickness will make the larger bone cause movement in areas which would logically belong to the smaller bone. This is especially apparent for example when two limbs are attached to a thin spine, as the limbs can cause the middle area to stretch in odd ways (see Figure 13).

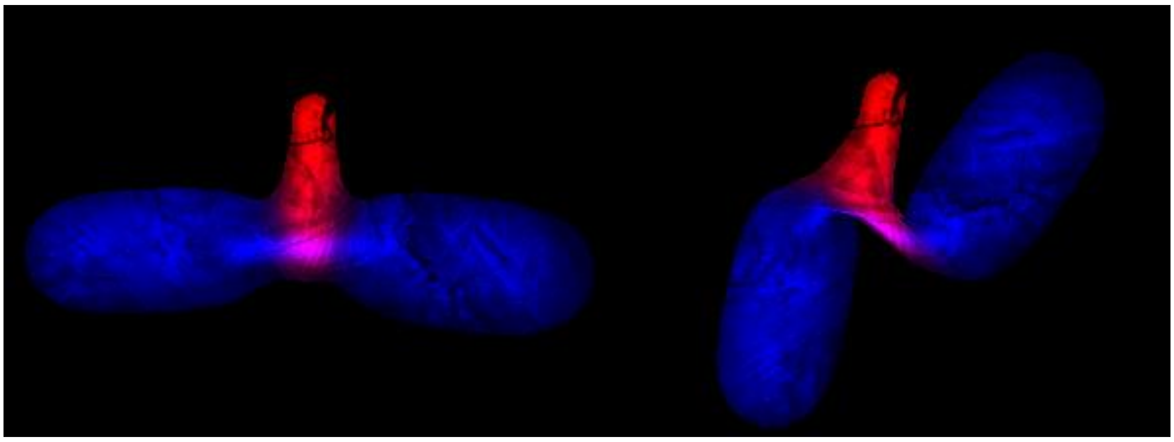


Figure 13 Example of unbalanced limbs. In the default pose (left) and with rotated limbs (right). The spine is not rotated.

The actual system of generation in this thesis is relatively rudimentary and makes heavy use of randomized parameters. While this is most likely far too chaotic to be used in an actual application, it serves to illustrate the capabilities of the system.

First it randomly chooses a number of spine bones to generate. These spine bones are given randomized values (see Table 1) which describe the bone. The last bone is considered the head and is given a randomized amount of eyes, placed around the tip of the bone. The eyes are for aesthetic purposes only and the next steps of generation are unaware of them.

After this, limbs are generated. Each spine bone is given a 50% chance to spawn two limbs. If a limb is spawned, first a limb socket bone is attached to each side of the spine. The limb socket bones placed at a distance equal to the thickness of the spine bone from the spine. This is to keep the point of rotation of the limbs mostly outside of the internal surface of the body. After that, limb generation proceeds similarly to the spine generation without branching further. There is also the limitation that the thickness of limbs must be smaller than double the minimum length of spine segments, to avoid limbs from coming into contact with each other.

Table 1. Variables generated for bones.

Variable	Range	Explanation
distance to parent (length)	$[0, \infty)$	Bones are defined by their location compared to their parent, this means that the length of a bone is also its distance to its parent
thickness	$[0, \infty)$	Thickness serves as a guide for generating metaballs, as they will be generated to match the specified thickness. 0 indicates no metaballs should be created.
colour	$(0, 0, 0)$ to $(1, 1, 1)$	Colour is used for texture generation.
texture weights	$(x, y, \dots)$ where $x + y + \dots = 1$ $x, y, \dots$ are positive	Texture generation uses several base textures for adding detail to textures, this specifies which ratio of textures should be used for the bone. Implementation supports up to 4 textures, but uses 2.
metaball multiplier	$[1, 1.6]$	Value describing how bumpy the generated shape should be. See chapter 4.2 for more detailed explanation

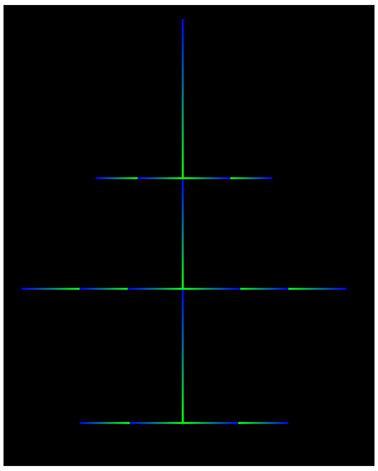


Figure 14. Example of generated skeleton

After completion, the skeleton (see Figure 14) and its associated data is sent to the metaball generation step, which is outlined in the next chapter. The skeleton is an output of the system, and is a part of the finalized creature. All additional data beyond the transforms of the bones are not important after metaballs are generated and then can be discarded.

4.2 Metaball Generation

After the skeleton is generated, the next step is to convert the generated bones and their associated thicknesses into actual shapes. The chosen method for doing this is through metaballs (see chapter 3.2). Simpler methods could obviously be used, such as extruding the bones into rectangles and combining them, or using oval, but metaballs can be used further in skinning and texture generation. They also generate more natural looking round shapes as they do not create sharp edges.

The input for the metaball generation step is the skeleton. During generation, each bone is considered separately. Metaballs are placed along the bone so that it creates a shape as close as possible to the desired thickness of the bone. This placement needs to be calculated, both in terms of position and metaball size.

The chosen falloff function for the metaballs is the one used by Ken Perlin in his original paper. That being

$$D(x,y,z) = \exp\left(\frac{B_i r_i^2}{R_i^2} - B_i\right),$$

where r_i is the distance to the center of the metaball, R_i is the radius of the metaball, and B_i is a "blobbiness" attribute, which determines how much metaballs connect to eachother. The chosen blobbiness for this thesis is -0.5. A different blobbiness value or falloff function can be used, but will change the math neccessary to ensure acceptable results. The threshold for being inside the surface is a value higher than or equal to 1.0.

During placement it must be ensured that metaballs connect to each other, creating a limb or body, rather than stay as individual balls. During generation this creates the requirement of a minimum distance between metaballs. In order to minimally connect to each other, the midpoint between two metaballs has to have a value of 1.0. If each metaball generated for a bone is of equal size, and the midpoint is equally distant from both metaballs, this creates the equation

$$\exp\left(0.5 - \frac{-0.5 \cdot r_i^2}{R_i^2}\right) \cdot 2 = 1,$$

which when solved for distance to the midpoint gives a value of approximately $R_i \cdot 1.566$, and the total distance between two metaballs therefore being twice that at $R_i \cdot 3.13$. If placed at this distance, the generated connection will be infinitesimally small, so a slightly smaller distance between metaballs is used in the implementation.

With the minimal distance found, a bone can be covered with metaballs by placing them at thickness times 2.97 intervals, with the metaball radius being equal to the thickness. This creates a bubbled surface, where the maximum thickness is the desired thickness. In the actual implementation, the minimum number of needed metaballs is calculated by dividing the length of the bone by that amount and rounding up. Metaballs are then placed evenly along the bone, including both tips. The rounding creates more metaballs and slightly larger connections than strictly neccessary.

This creates an interesting look (see 1.0 on Figure 15), but more calculations are necessary to make a smoother limb or body. This can be accomplished by creating a larger number of smaller metaballs. The precise sizes and locations of these metaballs can be calculated, but a precise calculation is beyond the abilities of the writer of this thesis, so a simplified calculation found through experimentation is used instead.

A multiplier value between 1 and 1.6 is used for determining the amount of Metaballs, with 1 creating a maximally bubbled outline, and 1.6 a smooth outline. This value is multiplied with the minimum number of needed metaballs. A thickness multiplier is calculated from this, equal to

$$thickness = \frac{multiplier}{\sqrt{2 \cdot \log multiplier^2 + 1}},$$

which is then multiplied with the base bone thickness to get the size of each metaball. While not precise, the thickness of a limb does appear to be fairly stable (see Figure 15).

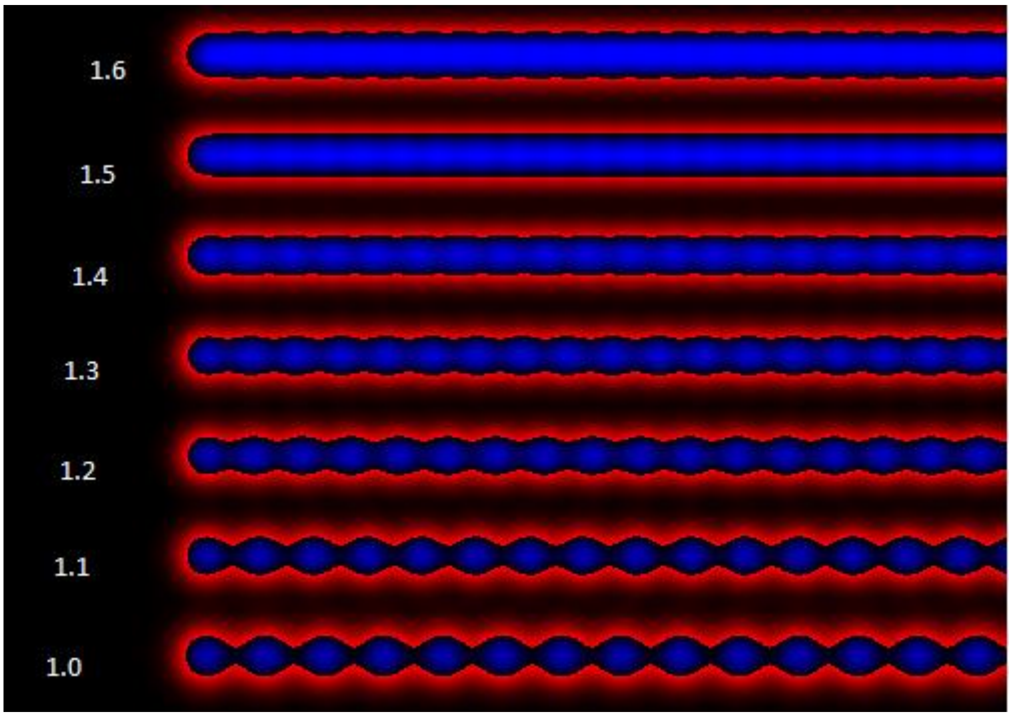


Figure 15. Example of different metaball configurations depending on multiplier value. Blue indicates a metaball density value higher than 1, red indicates value below 1.

Each metaball generated is given additional data: the colour, texture weights, and an id of the bone it was generated from. This data is used for skinning and texture generation (see Table 1 for variables taken directly from the bone). The bone id is used to determine what bone each part of the creature should be attached to.

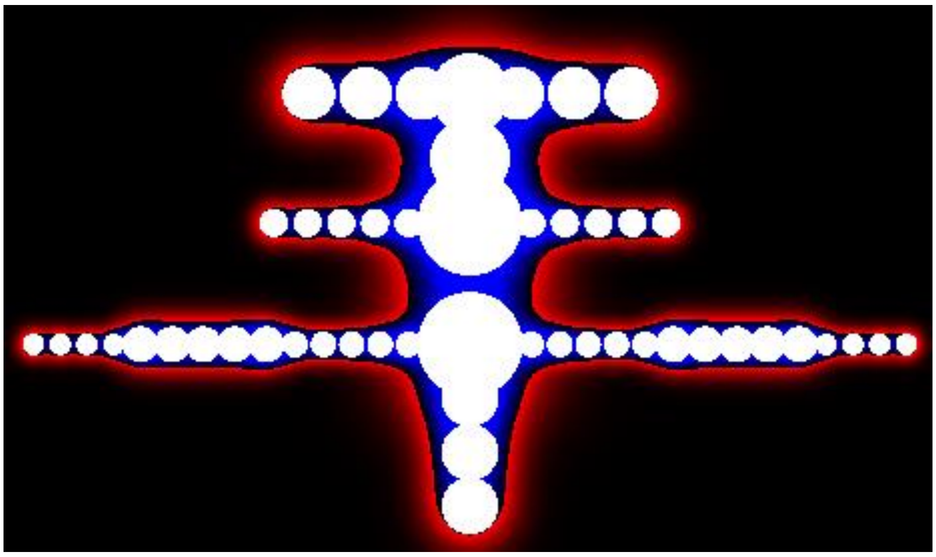


Figure 16. Example of generated metaballs. White indicates metaballs, blue indicates areas inside the surface due to sums of metaballs. Red to black indicates areas outside the surface.

The generated metaballs (see Figure 16) are then used in the edge detection step in order to create an outline of the creature (see chapter 4.3), in the skinning step to assign bone weights to all vertices (see chapter 4.5), and in the texture generation step to generate the texture (see chapter 4.6). The metaballs are not an output of the system and can be disposed of after the final model is generated.

4.3 Edge Detection

After the generation of metaballs, the next step is to generate the actual 2D mesh based on those metaballs. This involves finding the edge of the metaball, which describes the outline of the body, and generating the internal topology of the the model based on that outline. The internal topology is generated in the next step. In this step, the density field described by the input metaballs is sampled in order to generate the outline.

A variant of the marching squares algorithm is used for generating the outline. The algorithm is altered for better performance. Traditional marching squares iterates through a whole rectangular area, which will be mostly unoccupied by the creature, thus most of the sampling will the done in areas where there is no outline. Because of this, a large amount of relatively useless samples would have to be calculated. This puts limits on outline quality, as sampling at a higher rate would generate higher quality outlines, but also require a lot more calculations and processing time. The variant was made to lower the amount of needed samples.

The variant does not iterate through the whole area, but rather finds an initial square that contains the outline, and then iterates clockwise along the outline, until it reaches the starting square, ending iteration. Because of this, sampling is only done where necessary. Unlike the original, this variant cannot handle multiple disconnected metaballs. However, because metaballs are ensured to be connected in the metaball generation step, this becomes a viable method for lowering the amount of calculated samples.

The initial square is found by sampling values starting from the location of the highest metaball and iteratively sampling while moving upward. When the sample at some location is lower than 1, then the upper-left corner of the initial square is at the location of that sample. Iteration then proceeds as if the square had been entered from the left, as this would create an equivalent left edge (see Figure 17)

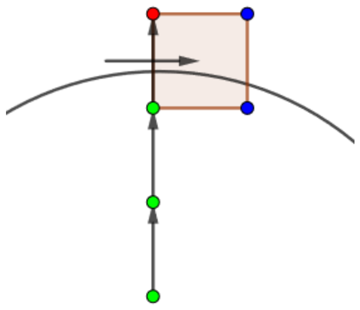


Figure 17. Example of finding initial square and assumed direction of entry. Green signifies values higher than 1, red signifies values lower than 1, blue signifies unsampled values.

The variant is composed of a set of 16 cases, including 4 ambiguous cases (see Figure 18). The cases are divided according to the direction of the previous square and each case generates a single point along a side of the square and describes the direction of the next square. In sequence, the generated points describe the outline. The ambiguous cases are resolved by sampling the midpoint of the square. If the value at the midpoint is lower than 1, then the next square is to the right of the direction of entry, otherwise it is to the left. The positions of the generated points are linearly interpolated along the edges of the squares, based on the sampled values at the ends of the edge.

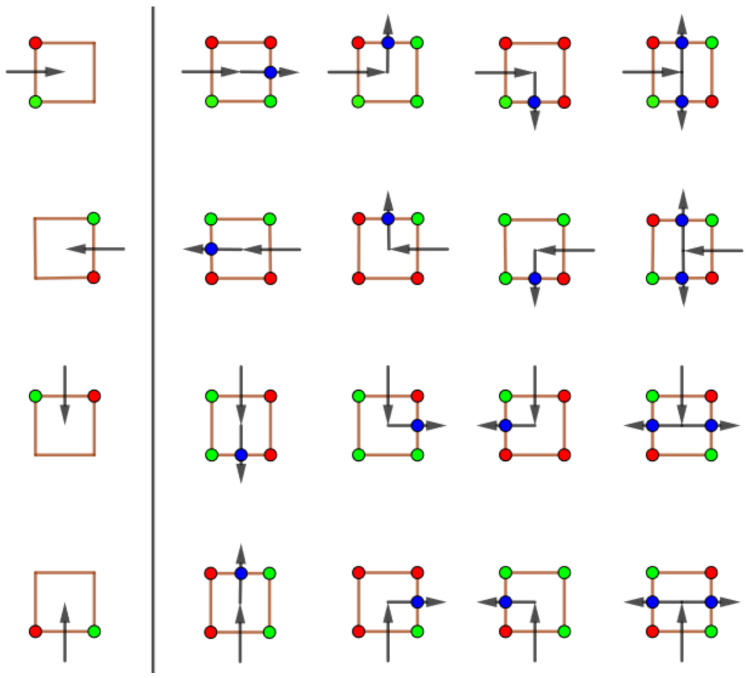


Figure 18. Cases with outline points and direction of the next square. Green signifies values higher than 1, red signifies values lower than 1. Blue indicates outline points. First column shows entry directions and known samples. Last column is ambiguous cases.

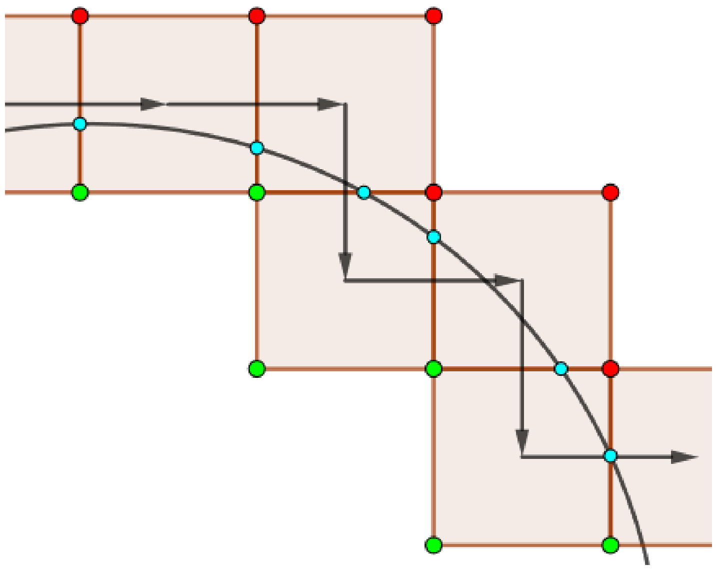


Figure 19. Example of iteration. Green signifies values higher than 1, red signifies values lower than 1. Cyan indicates linearly interpolated locations of outline points.

The result of this step is a sequence of points (see Figure 19), which describe the outline as a closed sequence of lines. This is used further in the mesh generation step (chapter 4.4) to generate an actual 2D mesh. The outline is also used for calculating a bounding box of the whole model, which is passed into the texture generation step (chapter 4.6). The outline itself is only useful for the generation system and can be discarded after the actual mesh is generated.

4.4 Mesh Generation

After the outline is generated, it needs to be converted into a 2D mesh so it can be rendered via a standard graphics pipeline. Meaning in this case a set of lines or a polygon is converted into triangles. This process is triangulation. Since the outline has already been found, this step mainly alters the internal surface of the creature and how it will change depending on the animations. This step also generates texture UV coordinates for each vertex and alters the z value of the vertices to combat z-fighting.

The algorithm used for triangulation generates a constrained Delaunay triangulation, based on a set of points and edges. This is done by an external JavaScript library called cdt2d[7]. As the library handles the specifics, this thesis is interested only in its properties. Delaunay triangulation generally avoids sliver triangles, which is good for animation purposes. Constrained Delaunay triangulation has the addition of constraints, which force the

[7] https://www.npmjs.com/package/cdt2d

triangulation into a particular shape. Simpler methods of triangulation can be used, but ear clipping for example generates a lot of sliver triangles which look strange when animated.

Acceptable results can be gotten simply via triangulation that uses the outline as edge constraints and creates no new vertices (see Figure 20). This has the problem of being mostly composed of sliver triangles and looking unnatural during animations. When animated, the slivers connect distant edges of the creature, making it look as if it is composed of segments like an accordion. Therefore some amount of internal vertices needs to be created.

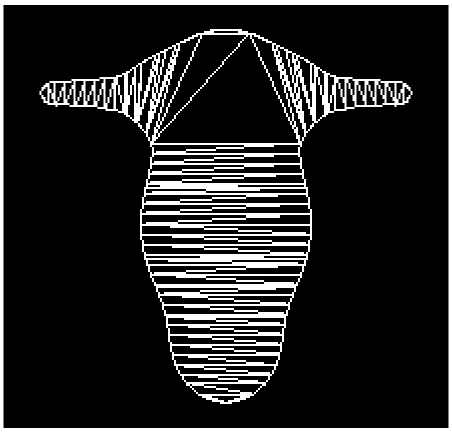


Figure 20. Example of triangulation using only the outline.

Another tested alternative was placing vertices at the locations of the metaballs. This creates an internal surface of a visibly higher quality, but still generates a lot of slivers and an additional amount of larger triangles (see Figure 21). It also looks visually off in some way, which the author of this thesis speculates is based on the difference between the sizes of the triangles.

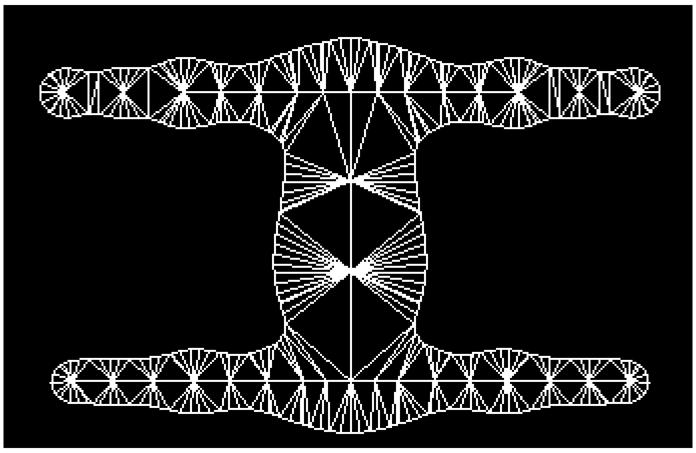


Figure 21. Example of triangulation using outline and metaball locations.

The last possibility tested and used was placing points uniformly across the surface of the creature (see Figure 22). Visually this had the most natural looking animations, and also generates a far smaller amount of slivers. It also generates triangles of the most consistent size. This method does have the downside of generating more triangles than necessary, as it generates complex topology even in places which will not change meaningfully during animations. It was chosen as the method used in the actual implementation.

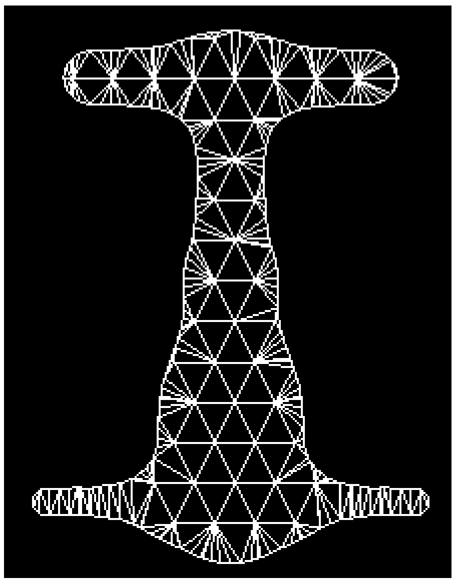


Figure 22. Example of triangulation using outline and uniformly placed points.

Additional possibilities include a smarter placement of points, and Delaunay refinement algorithms, which would improve the quality of the triangulation to eliminate slivers. These were not tested during the course of this thesis, but remain promising possibilities.

After the vertices are placed and triangles are generated, the next step is the calculation of UV coordinates for each vertex. Since the generated mesh is flat and does not overlap with itself, the simplest way to map UV coordinates is by treating it as a cut out piece of a rectangle. The rectangle used is the bounding box around the mesh, which is also sent to the texture generation step for that reason. A vertex at the bottom-left corner of the bounding box gets UV coordinates of (0,0), a vertex at the top-right corner of the bounding box gets UV coordinates of (1,1) and all values are interpolated according to their locations in the box (see Figure 23).

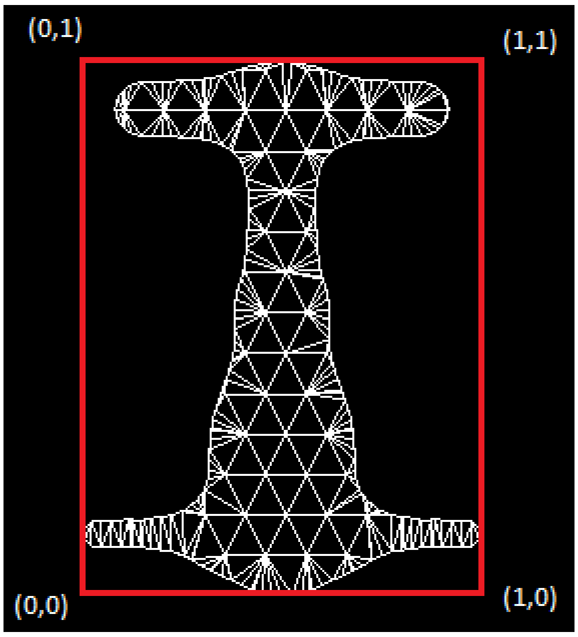


Figure 23. Example of UV coordinates. Red signifies bounding box.

Another problem dealt with at this step is the question of what happens in case of self-collisions. When two triangles at equal depths meet, it is fairly random which will lie on top and which will lie on the bottom. This is called z-fighting. Since the 2D mesh is actually just a 3D mesh, where every vertex has a *z* coordinate of 0, we can add a slight variance to the *z* coordinates of vertices to deterministically set which parts should stay on top and which on the bottom. This is accomplished similarly to the UV calculations, except only the vertical component is used. The *z* value at the top being 1, and at the bottom being 0. This effectively means that a higher limb will consistently be on top of lower limbs, solving the problem of *z*-fighting in this instance and allowing overlapping limbs (see Figure 24).

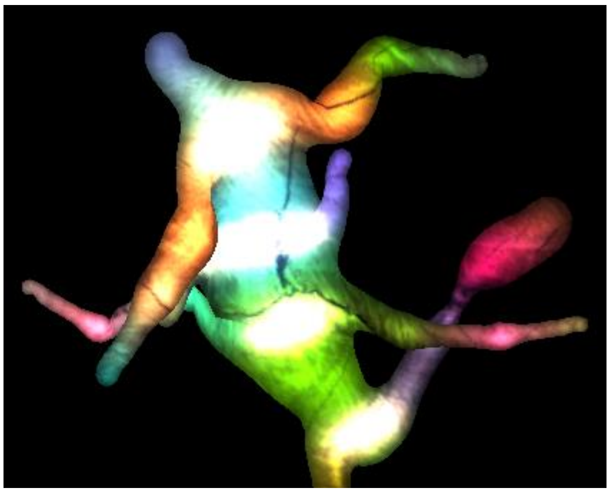


Figure 24. Example of overlapping limbs.

After this the mesh is effectively complete, only lacking skinning data to tie it to the skeleton. The mesh is sent to the skinning step, and the vertices, faces and texture UV coordinates are final outputs of the system.

4.5 Skinning

After the mesh is generated, it needs to be tied to the skeleton for animation purposes. This is done through the assignment of bone weights and indices to each vertex of the mesh. Within the proposed system, these weights and indices are calculated based on the metaballs and the bones associated with them. The metaball derived values are calculated similarly to the texture generation step (chapter 4.6). In the implementation itself, skinning is a part of the mesh generation function due to convenience. In this thesis they are treated as separate steps, because overall mesh generation does not require any use of metaballs.

For each vertex the value of the metaballs is calculated at the location of the vertex, and an array of bone weights is generated. Each metaball contributes its value to its bone in the array. The highest 4 bone weights (maximum in Three.js, may vary in other graphics libraries) are then chosen as the bones for the vertex. Due to the relative simplicity of the generated skeleton, having more than 4 would likely have little effect. The 4 weights are then divided by their total sum in order to make their sum equal to 1. The bone weights and their indices are then added to the vertices of the mesh.

The quality of the skinning depends largely on the placement of metaballs and the shape of the original skeleton. Some problems that should be accounted for were described in chapter 4.1.

No further processing of the data generated during this step is necessary, and the generated bone weights are one of the final outputs of the system.

4.6 Texture Generation

The texture generation step gives the 2D mesh colours and texture. It is a very important component of how the creature will look. The texture is generated based on the values assigned to metaballs during metaball generation (see chapter 4.2) and made to match the UV coordinates assigned to the vertices of the mesh in the mesh generation step (see chapter 4.4). In order to match the UV coordinates this step requires a bounding box calculated after the edge detection step. Much like skeleton generation, texture generation will likely be heavily altered in actual use. Therefore the system described in this chapter is meant to illustrate capabilities and possibilities rather than to create textures of particularly high aesthetic worth.

This system of texture generation attempts to display some ways in which textures could be altered and changed by parameters. For this purpose, the generated textures have the illusion of depth through the use of calculated metaball values, which could also be used for other purposes. The textures can also vary in terms of colour, determined by metaballs. A method for adding details to the texture is also shown through the use of more detailed premade base textures.

The implementation generates 256 by 256 textures, but differently sized textures can be generated without problems. The texture is generated on the graphics card via a custom shader, but the same results can be accomplished without using the graphics card. Due to being computationally heavy, not using a graphics card may require a different method of texture generation.

The generated texture is not fully procedural, but rather uses base textures for adding more detail. However simple colours and the brightness of any given area of the texture is fully procedural. The values used in generation are the colours and texture weights of the metaballs, and a bounding box of the mesh.

The first step of the texture generation is transforming the coordinates of each point in the texture to make the texture correspond to the bounding box of the outline. That makes it possible to calculate the value of the metaball density field for any point on the texture. When the outline is not a square, then the texture will appear stretched when seen directly, but correct when applied onto the mesh. This is calculated by multiplying the UV coordinates of a pixel with the width and height of the bounding box, and adding the coordinates of the lower-left corner of the bounding box.

Since base textures are used, they also need to be scaled to counteract the stretching that occurs when the bounding box is not square. This is done by calculating the ratio between the width and height of the bounding box. If the ratio is bigger than 1, then the u coordinate of the pixels UV coordinates is multplied by the ratio, and if the ratio is smaller than 1, then the v coordinate of the pixels UV coordinates is divided by the ratio. The result is UV coordinates that can be used for sampling the base textures.

After the coordinates of a pixel are transformed into the coordinate space used by the metaballs, it is possible to begin calculating the colours and base textures of that pixel. This is done similarly to the skinning step (see chapter 4.5). The value of each metaball at the location of the pixel is calculated and summed, with separate sums made for colours and texture weights. These sums are later divided by the overall sum to get a value that is smoothly interpolated between the effects of different metaballs.

After these calculations, a pixel has an overall sum of metaball values, which describes if and how deep inside the creature this pixel is and creating the illusion of depth. It also has a colour and a set of texture weights, which describes how the base textures should be sampled. After the textures are sampled, the final result is the multiplication of these values.

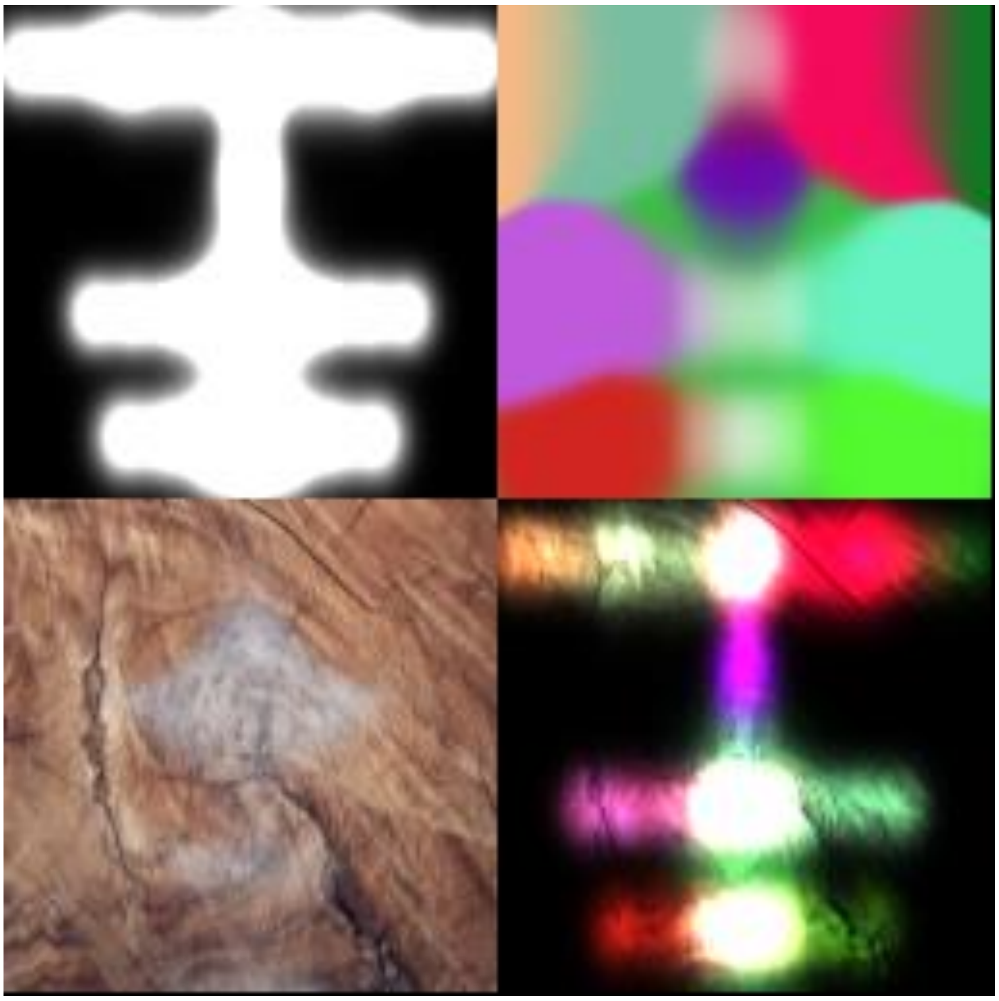


Figure 25. Example of the parts of a generated texture. Top-left is the sum. Top-right is the colour. Bottom-left is the combination of base textures. Bottom-right is the final combination of these values multiplied together.

As can be seen in Figure 25, the sum is not directly usable for texture generation due to being much larger than usual colour values. It describes the actual shape of the creature, and is known to have a value of 1 at the outline and has a much larger value directly above the metaballs. Therefore it needs to scaled. A generally colourful, but not too bright value seems to be 75% of the sum (see Figure 26).

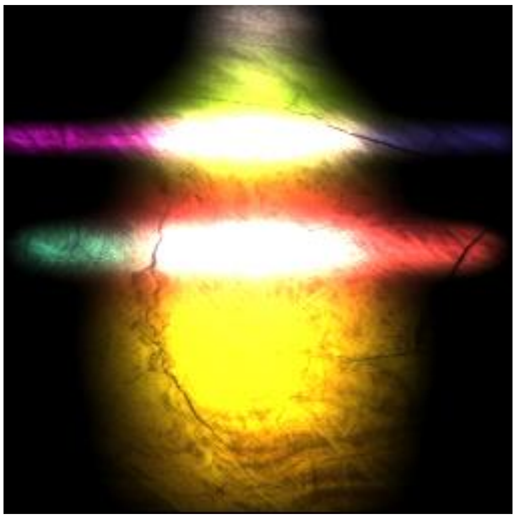


Figure 26. Example of texture, with 75% metaball value contribution.

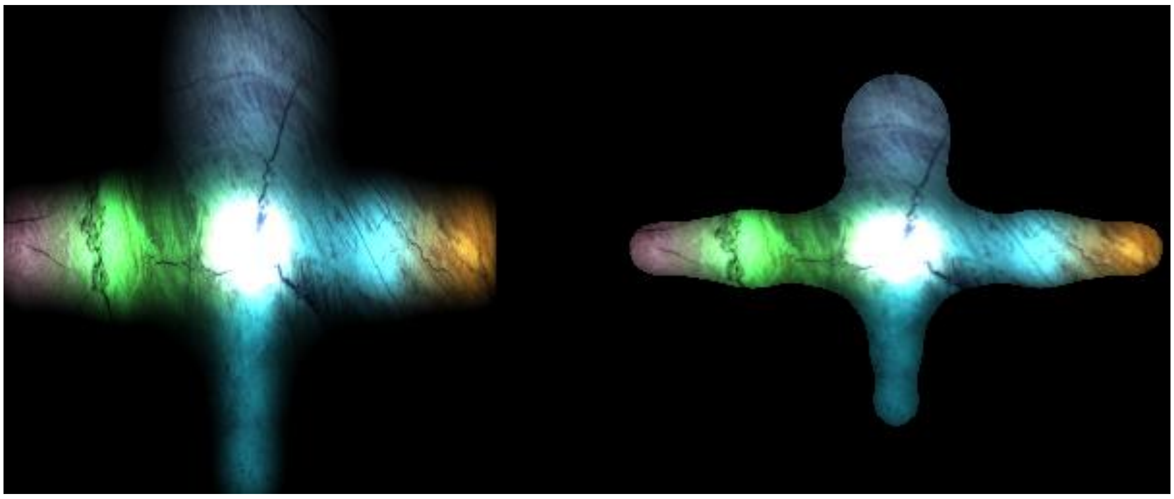


Figure 27. Example of texture applied to generated mesh.

The generated texture is a final output of the system, and requires no further processing. The generated texture can be applied directly onto a generated mesh (see Figure 27). If both the mesh and texture are fully generated, the creature is complete.

4.7 Analysis of Complete System

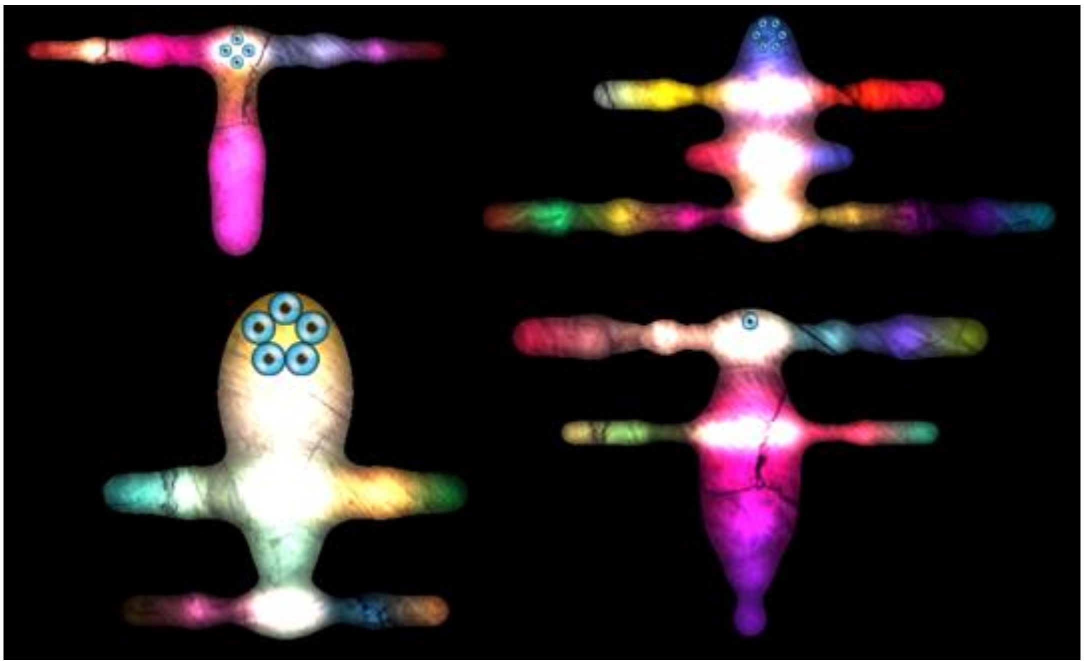


Figure 28. Some generated creatures. Creatures are in default pose.

The intended use case of real-time generation of interesting and varied 2D creatures creates several avenues for analysing the system. Due to the subjective nature of the quality of the overall generation, this analysis will concentrate mostly on the quality of skinning and performance, both of which can be better quantified in terms of objective problems. In terms of creature variety, the system is of sufficient quality to satisfy the author of this thesis (see Figure 28), although improvements could be made, which are described in subchapter 4.7.3.

4.7.1 Skinning

There are some problems with animations and skinning which become apparent during closer inspection, although they are relatively hard to spot during constant movement. These could and should be improved upon to improve the quality of the system. These problems are not apparent in all creature, nor even most of them.

The first is a problem explained in chapter 4.1. Due to the nature of metaballs, skinning can cause bones to affect vertices that they logically should not affect very strongly and the mostly randomized nature of the skeletal model can connect bones of wildly different thicknesses. Figure 29 illustrates this problem, as strange behaviour can be seen in the curvature of the “neck” and where the limbs connect to the body. The most likely fix for this is either changes to the skeleton generation and metaball placement to generate better joint regions or a change in the metaball equation to lower the effects of metaballs outside their actual radius.

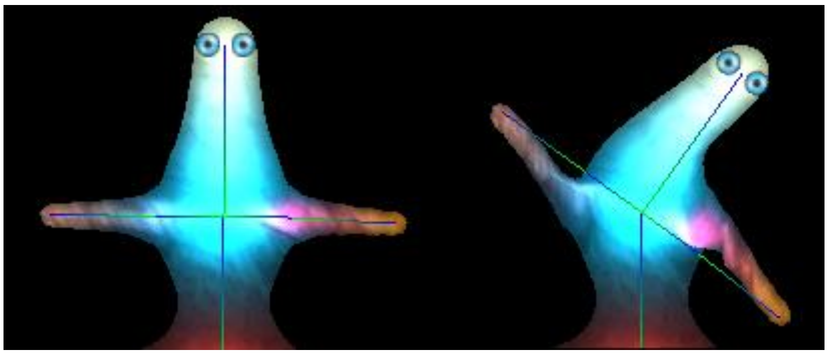


Figure 29. Creature in the default pose (left) and with a rotated upper-body (right).

A second problem illustrated by Figure 30 is the effect of the solution to z-fighting described in chapter 4.4, where higher parts of the mesh have a higher z-value. Because of that, lower limbs stay behind upper parts of the body. This is of relatively little importance, but causes limbs to look better when rotated down rather than up. An ideal solution would allow limbs to stay in front of the body in all cases, but also needs to make sure overlapping limbs do not begin to move through each other, rather than being above or below each other.

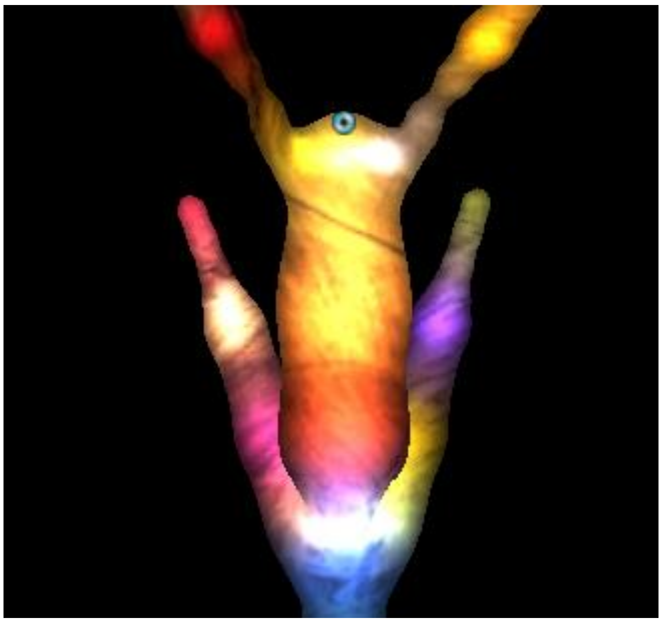


Figure 30. Example of limbs rotated upward.

4.7.2 Performance

The intended use case was that the system would be capable of working inside a game without requiring pre-generation or a lengthy generation period. A ballpark figure for acceptable performance was that it should not take over a second. The system in general has not been heavily optimized.

Performance was tested on two pieces of hardware. A desktop computer with Intel i7 6700K @ 4.5 GHz, Nvidia GTX 1060 6 GB VRAM was used for primary testing with a weaker laptop with a Intel i5-5200U CPU and Intel HD Graphics 5500 serving as an example of the system running on weaker hardware.

On the weaker hardware, skeleton generation on average took 3 ms, metaball generation 5 ms, outline generation 15 ms, texture generation 46 ms and mesh generation 416 ms, for a total generation time of 485 ms. On the stronger hardware skeleton generation took 0.3 ms, metaball generation took 0.4 ms, outline generation took 1.4 ms, texture generation took 12 ms and mesh generation took 70 ms for a total of 84 ms. These numbers are also expressed in Figure 31. From these numbers it becomes apparent that the vast majority of generation time is spent generating the mesh. Of that time, most is spent generating the faces with the cdt2d library, meaning that for improved performance either an alternative library must be used, or some way to use it more effectively has to be found.

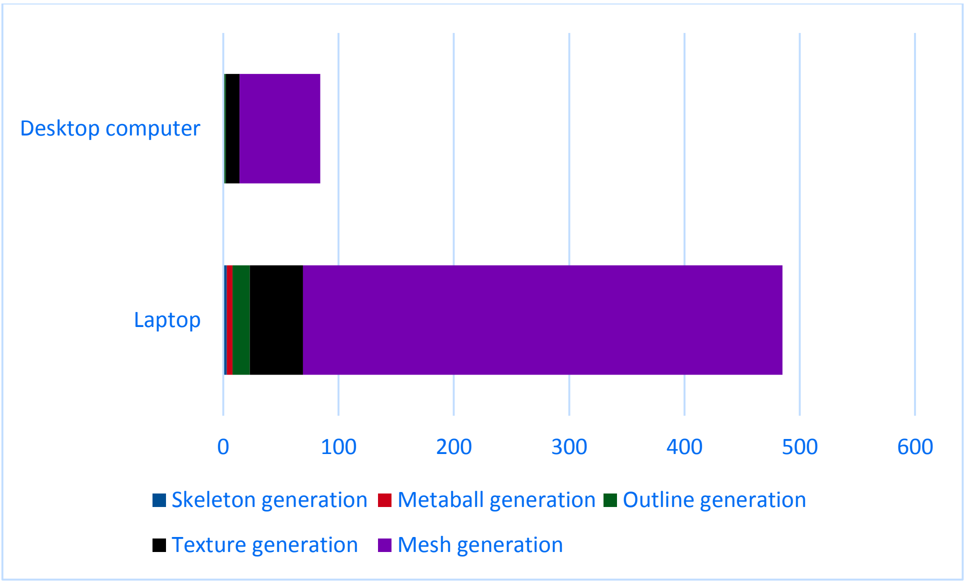


Figure 31. Time taken by individual steps (ms).

The system was also tested for how variations in certain parameters would affect performance. For this testing, the full generation system was disabled, and a simple skeleton was used to replace the skeleton generation system to make generation more consistent. Consequently the generation time is much lower, but it should work for the purposes of testing how the system scales.

The first variable tested was the effect of altering the distance between samples during outline generation (see Figure 32), which affects the quality and precision of the generated outline. The data shows that there is a correlation between a higher distance between samples and higher performance, due to lowering the amount of data mesh generation needs to generate. However at higher values the quality of the outline of the mesh visibly degrades, causing sharp edges (see Figure 33). With the values displayed in Figure 33, the right mesh has poorer quality, but takes nearly half the time for generation. Therefore this is a variable that can be tweaked according to aesthetic needs and the performance of the target hardware.

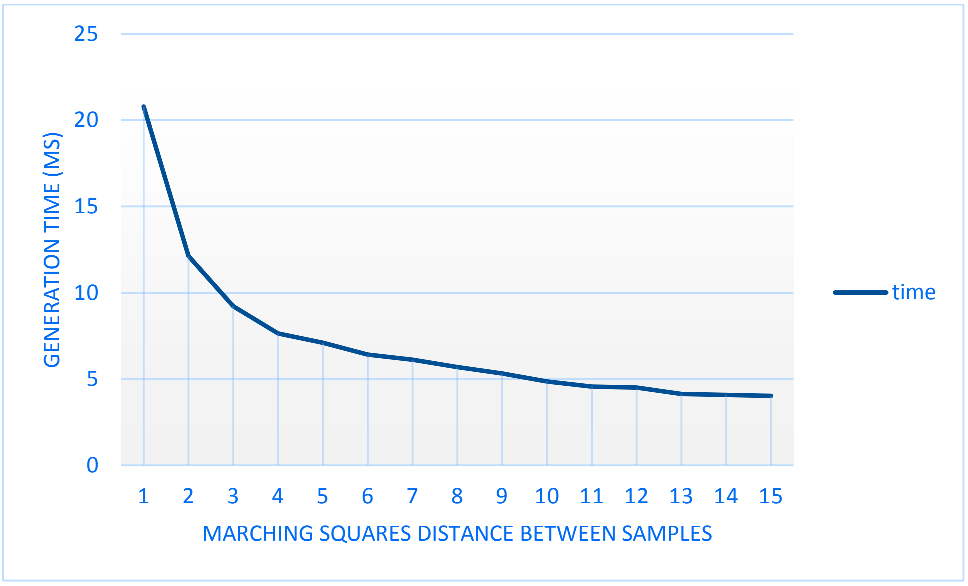


Figure 32. Performance effect of altering the distance between samples for matching squares.

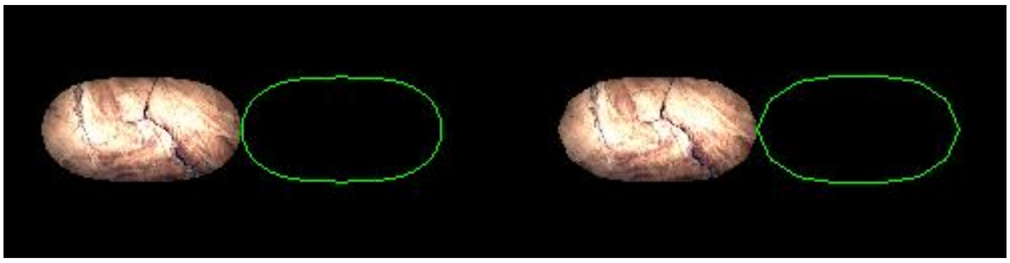


Figure 33. Mesh generated with distance between samples of 5 (left) and 15 (right)

The next variable tested was the effect of altering the distance between the internal vertices of the mesh (see Figure 34). Altering this variable alters the quality of animations, by changing its effect on the texture. The graph shows that its effect on performance becomes minor quite quickly. The effect on animation quality is very difficult to show in images, and the effect is relatively subtle. The data seems to suggest a value of around 30, although it could be lowered if animations seem to be without issues.

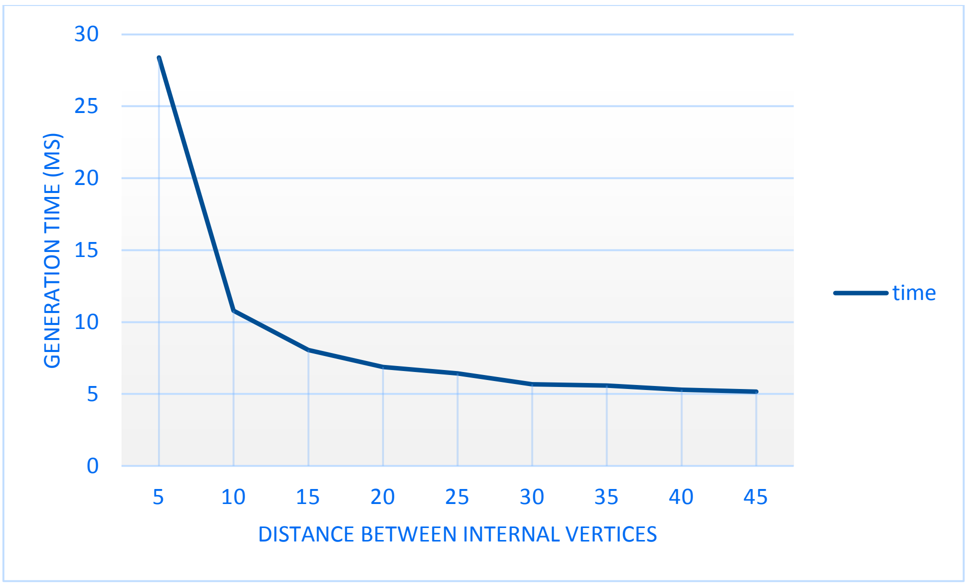


Figure 34. Performance effect of altering the distance between internal vertices

The last testing criteria was how the generation system reacts to adding more segments onto the testing skeleton, essentially creating a long worm, and testing the generation time (see Figure 35 and Figure 36). Note that this was done with powers of 2 (2, 4, 8, 16, ...), unlike previous tests. 128 could not be tested, as that amount of metaballs reached the limits of texture generation. The data suggests that there is a flat performance cost in the system, which slowly becomes more irrelevant as more segments are added. It also suggests that there is some component in the system which scales badly, as large amounts of segments causes the system to become slower again.

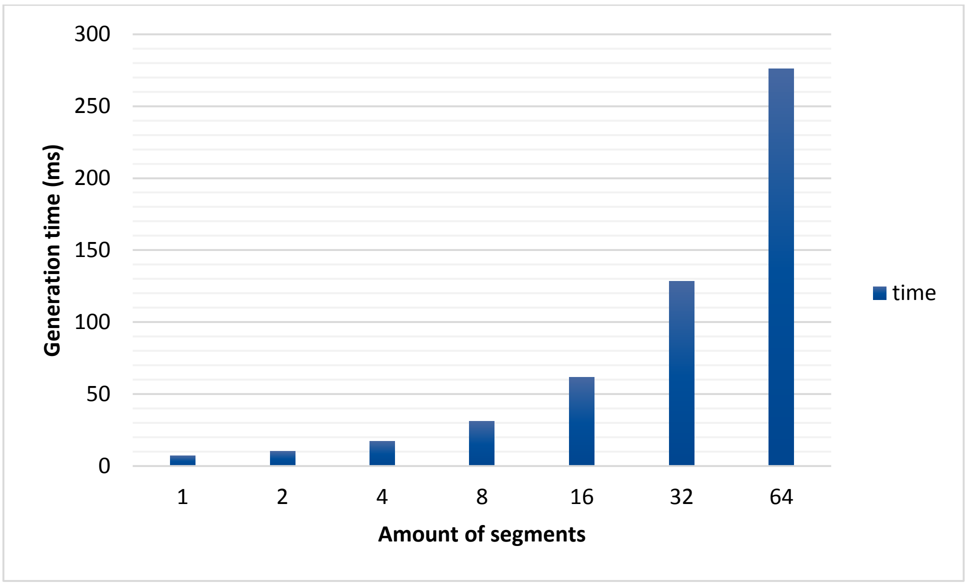


Figure 35. Performance effect of creature with more segments.

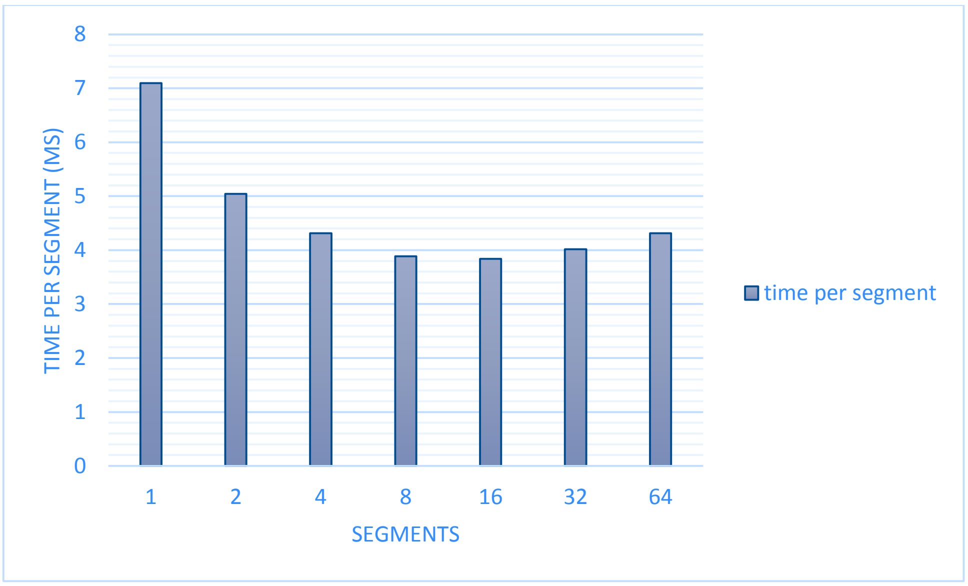


Figure 36. Performance effect of creature with more segments. Displays time divided by amount of segments.

4.7.3 Possible Improvements

There are several avenues for improving the system in its current form to create more variety and better creatures. Nearly every step of the process could be improved in some way to improve the overall quality of the generated creatures.

The first and foremost being changes to the skeleton generation to create body parts such as fingers, and to allow for more complicated shapes than just a spine with limbs. L-systems present an obvious possibility. Although more complicated generation systems also create more possibilities for self-collisions, which need to be mitigated. As the input for the rest of the systems, improvements to skeletal model generation would likely radically alter the quality of the entire system.

Another area of improvement is metaball generation. The system described in this thesis is quite rudimentary, as it only places metaballs onto a single line. More complicated placement algorithms could be used instead to create interesting and novel shapes. Another area of improvement for metaballs would be a more nuanced system for bone weight calculations. In the present system, each metaball is assigned a single bone, but a more nuanced system might show better results in generating something like tentacles.

A third area of possible improvement is texture generation. A problem with the texture generation system is that it does not generate a model that reacts to lighting, as it lacks proper surface normal vectors. Therefore a way to improve the system would be the generation of a bump or normal map. The system already generates the data necessary for a bump map in the form of the “sum” which should make it relatively simple to make a rudimentary system, although it would most likely require some tuning to give good results. In general texture generation is the safest step to alter heavily, as it does not have a direct effect on any other step.

4.7.4 Uses

The system is not intended to be used in its current form, but rather the assumption is that it will be heavily altered to match the actual use case. This chapter outlines some possible ways to use the system.

The first possibility is the removal of the skeletal model generation step, and to let users create the skeleton manually. This would be useful for an application similar to the Creature Creator in the computer game Spore. However this would require improvements to the performance of the system, as constant regeneration with the current generation time would create a bad user experience.

A second possibility would similarly remove the skeletal model generation step, but replace it with developer created skeletons that include random variations in thicknesses of limbs, colours, and other parameters. This creates the possibility of having a creature with a preset skeleton so animations do not need to be procedurally generated, but the creature could still look sufficiently different to be interesting.

A third possibility is simply the further development of the current system, for fully procedurally generating enemies, aliens or animals in some game or application. In this case the skeletal model will most likely be generated in some other way and the texture generation step will be similarly altered, but the general structure of the system will remain the same.

5 Conclusion

This thesis was about a system which can generate a large variety of 2D creatures and their 2D models and skeletons. The data generated for the creatures makes it possible to animate and pose them although this thesis did not deal with animations themselves. A JavaScript implementation of the system was created for this thesis and analysed.

The thesis started with a short theoretical background about creature generation and what data needs to be generated. It summarized the systems used by two computer games and their benefits and drawbacks. One of them was used as inspiration for the initial direction of development. It then gave a short overview about 2D models and the algorithms that were used in the system.

This thesis contained an overview of the developed system and its goals. It also explained some of the features of the implementation and the reasoning behind them. Then it gave a breakdown of every step in the generation process, containing what the system should do, and why the system was designed the way that it is. Afterwards it analysed the complete system as whole and its problems and the performance of the implementation. The system included: skeleton generation as the basis for all further processing, metaball generation which defined the shape of the creature, outline generation using a variant of marching squares, mesh generation based on that outline via constrained Delaunay triangulation, skinning data calculation and texture generation based on metaballs.

The system was consistent with the initial plans in terms of variety of generated creatures, performance and the possibility of further development. Several avenues of improving the system were proposed.

The author of this thesis hopes that this system will serve as inspiration for the creation of games, applications and systems which include creature generation. The author will likely continue development and there are plans for developing an application which uses this system.

6 References

[1] Sutton, J. 2018. Minecraft Sales Hit 144 Million, Mojang Boasts of 74 Million Active Monthly Players. http://www.game-debate.com/news/24411/minecraft-sales-hit-144-million-mojang-boasts-of-74-million-active-monthly-players (06.05.2018)

[2] Onyett, C., Butts, S. IGN PC's Most Anticipated Games for 2008. http://www.ign.com/articles/2008/01/25/ign-pcs-most-anticipated-games-for-2008 (06.05.2018)

[3] Gaudiosi, J. 10 Most Anticipated Games of 2016. http://fortune.com/2015/12/29/10-anticipated-games-2016/ (06.05.2018)

[4] Sepp, A. Protseduuriline lõpmatu maastiku genereerimine. University of Tartu, Institute of Computer Sciences, Bachelor’s Thesis. 2016. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=53657&year=2016 (06.05.2018)

[5] Perli, K. Protseduduriline linnade genereerimine. University of Tartu, Institute of Computer Sciences, Bachelor’s Thesis. 2016. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=53791&year=2016 (06.05.2018)

[6] Tunnel, R.H. Protseduuriline puude genereerimine. University of Tartu, Institute of Computer Sciences, Bachelor’s Thesis. 2012. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=25141&year=2012 (06.05.2018)

[7] Alexandra, H. A Look At How No Man's Sky's Procedural Generation Works. https://kotaku.com/a-look-at-how-no-mans-skys-procedural-generation-works-1787928446 (12.05.2018)

[8] Hernandez, P. The Strange and Disturbing Creatures That People Are Finding In No Man's Sky. https://kotaku.com/the-strange-and-disturbing-creatures-that-people-are-fi-1785021127 (14.05.2018)

[9] Hecker, C. My Liner Notes for Spore. http://chrishecker.com/My_Liner_Notes_for_Spore (12.05.2018)

[10] Quigley, O. Spore's creature skin painting. http://oceanquigley.blogspot.com.ee/2009/04/spores-creature-skin-painting.html (12.05. 2018)

[11] Sporepedia. http://www.spore.com/sporepedia (14.05.2018)

[12] Explaining basic 3D theory. https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_on_the_web/Basic_theory (12.05.2018)

[13] Magnusson, L. Skeletal Animation (Skinning). http://www.it.hiof.no/~borres/gb/exp-skeletal/p-skeletal.html (12.05.2018)

[14] Blinn J. F. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics* Volume 1 Issue 3, July 1982, 235-256. https://dl.acm.org/citation.cfm?doid=357306.357310 (12.05.2018)

[15] Lorensen, W. E. Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics* Volume 21 Issue 4, July 1987, 163-169. https://dl.acm.org/citation.cfm?id=37422 (12.05.2018)

Licence