Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Procedural Animations with SoftBody Physics in Godot

Master Thesis

Name of the Study Programme

International Media and Computing

Faculty 4: Informatics, Communication and Economy

from

Elisabeth Kintzel

Date:

Berlin, 23.03.2026

1. Supervisor: Prof. Dr. Tobias Lenz

2. Supervisor: Prof. Dr.-Ing. David Strippgen

Affidavit

Academic integrity declaration

I hereby declare that I have prepared all parts of this thesis independently and have not used any sources or aids other than those specified in the thesis, and that the thesis has not been submitted in the same or a similar form in any other examination. All verbatim or in-sentence copies and quotations have been identified and verified.

No AI-based tools have been used in the process of this thesis.

Berlin, 23.03.2026 

(Place, Date, Signature)

Abstract

This paper explores the possibilities of procedurally generated animations that are adaptive to user input and environments in combination with deformable meshes, simulated through SoftBody physics. Due to a focus on applications in the indie game scene, the goals of this paper are not to find the most realistically accurate solutions, but rather to achieve a blend between performance lightening optimizations and visually pleasing stylizations, all while relying exclusively on freely available, open-source tools, like the Godot game engine and the 3D modeling and animating software Blender.

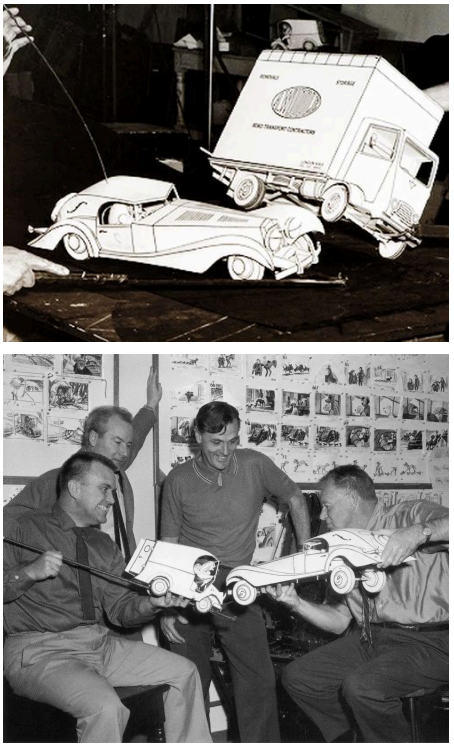
The results are a character mesh, which has been separated into distinct main body sections, to allow for faster and more stable physics simulation created by a custom written SoftBody physics class, based on the Mass-Spring-Model. Additionally, the mesh has been internally subdivided to allow for secure attachments to a procedurally animated character rig, without interfering with the soft physics of the outer mesh surface.

Overview

1 Introduction

Back in the times of traditional, hand-drawn animations, such rigid, hard-surface objects were one of the hardest parts of animation. While animators had plenty of practice of how to stretch, squash, twist, or otherwise deform an object with the motion, for solid props, all dimensions and corners had to be kept perfectly accurate, or they would look deformed and out of place. To make the animation process as quick and easy as possible, digital technologies got incorporated to facilitate the traditional process early on. So did Disney Studios, for example, start using the first few digital shortcuts as early as the 1980s for a few scenes in The Little Mermaid movie. [1]

More interesting is, however, what tricks animators came up with before the rise of computer graphics, to manage the dreaded task of drawing rigid objects. In the 60s, Disney worked on their movie 101 Dalmations, which had a car chase scene as its grand climax, with various cars crashing into each, falling apart, or getting stuck in snow banks. Instead of resigning to drawing every single frame per hand, hoping to get the dimensions of each vehicle consistent across hundreds and hundreds of frames, the animators had a different idea. They build miniature scale models of the cars, painted completely white, with black lines drawn onto the surface, following the contours of the object. (Img.1) The complex scenes were then acted out as stop-motion clips with the help of these models, from which the cars' outlines could be extracted and used as a base for the rest of the scene. (Img.2)



Img.1: car models created by Disney animators for *101 Dalmations* [2]

Img.2: stop-motion car crash scene (left) vs. final animation (right) [3]

With the rise of digital 3D modeling, animating became increasingly easy. All you need is the spatial information of the object, and moving it around through uniform operations like rotation, translation or scaling becomes as trivial as a simple matrix operation; something that any standard computer is capable of calculating. So creating an animation of a static object like a car poses only a minor challenge, no matter how complex the movement or perspective is. Instead, digital animation has the opposite problem: simulating an object that is soft and flexible and deforms on impact or through movement requires advanced calculations or smart work-arounds to achieve what any decent animator used to be able to do with just a pencil.

Digital animations for films have the computing power and time to afford to fine-tune each individual movement or pose, to achieve the best looking result for this exact situation in that exact environment seen only from the exact point of view of the camera. So not only do they include more predictable parameters, in which any minor detail can be manually tweaked to absolute perfection, it also offers the space to allow for much longer render times.

Animations made for games, in comparison, need to be far more flexible. They need to be done on the spot, while having less than a second available to calculate each new frame. Furthermore, does not only the mesh deformation itself have to be reactive to the specific circumstances of the situation, but the movement of the animation also needs to be adjustable. The same walking animation, for example, needs to procedurally adapt to the environment to still be able to work no matter if the character is currently walking along an evenly paved road or a rough, debris-littered path along the woods.

Therefore, this paper explores potential uses of both procedural animation techniques, as well as deformable SoftBody meshes, to find a visually pleasing but yet performance optimated compromise to combine these two aspects and to create more advanced, reactive video game animations.

For the scope of this thesis, the decision has been made to exclusively work with the game engine Godot (GD), as well as additional support in creating meshes from the 3D modeling and animation software Blender, since both are freely available, open-source programs, made by and for the community. Unlike with engines controlled by large scale companies, where users are susceptible to any predatory changes in pricing models or terms of services, community driven projects offer less reliability and vulnerability to the engine’s providers. Thus, it is

important to support their usages and growth by basing experimental research on these softwares.

Godot is relatively new in the game engine market, only having been released in 2014, in comparison to Unreal in 1998 or Unity in 2005. Due to that, Godot is generally seen as not quite as refined and polished yet compared to these big, established engines. However, in the last few years its community keeps growing rapidly, leading to a steadily increasing amount of volunteering contributors, which keep improving the engine with every update.

Furthermore, it has been shown that technical limitations lead to more innovative solutions, something fundamentally required in a creative medium like video games. While making photorealistic simulations in Unreal, for example, poses no big challenge, many people are severely dissatisfied with gaming scene’s ongoing trend of hyper realistic artstyles, looking boringly alike, and yet requiring increasingly excessive amounts of storage space and expensive hardware performance. Therefore, this thesis has been done in support of smaller indie studios, to foster approaches that, while being less realistic, instead display a creative, individual and artistically expressive artstyle.

2 Procedural Animation

Animation Basics

Modern computer animations typically depend on a rig, which is a specific setup of controller objects that allows for the direct manipulation of a mesh's deformation. For humanoid characters, these typically look and function similarly to how a skeleton would function in a real living body, providing structure, and defining the possible range of motion, hence why the rig is also routinely referred to as a *skeleton*, and the controllers as *bones*.

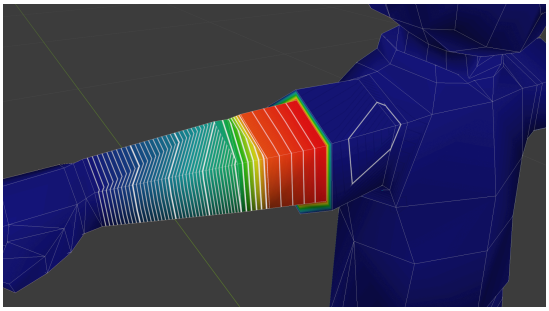
However, a rig is not strictly necessary to be able to animate a mesh. For example, creating the movement of a door opening and closing, can also be done by manually rotating the rigid mesh around a pivot point, while leaving the frame part of the door in place. Similarly, it would be possible to manually select and rotate specific sections of a more complex mesh to achieve deformation directly. However, manually selecting all the points, or vertices, of a character mesh is far more complex, which is why a rig provides an interface for the animator to facilitate this process.

A skeleton consists of multiple bones that are connected to one another in a hierarchical structure. This means, using the example of a human body: all the fingers are parented to the hand, the hand to the lower arm, the lower arm to the upper arm, and so on. Due to this hierarchy, moving the upper arm bone will automatically move everything down to the fingers along with it. Moving just a finger instead, will leave the rest unaffected.

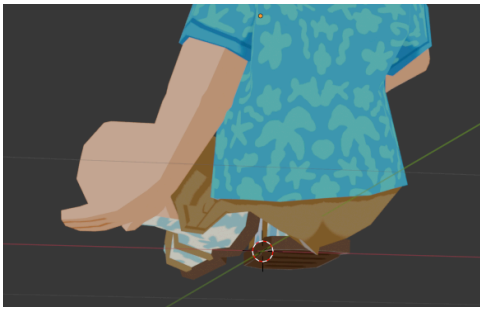
Simpler rigs can include just the core deforming armature, which means only bones that have a direct influence on the mesh. Normally though, rigs include a lot more bones that are used as controllers or reference points for different things, for example, to constrain joints to specified areas of freedom, limiting the maximal allowed rotation for certain axes, to emulate the functionality of regular human joints.

Once a skeleton has been set up, a mesh needs to be connected to it to follow its movement. When this idea was first developed, during the early beginnings of 3D computer animation in the 1980s, this was referred to as *'Joint-dependent Local Deformation'*. [4] Nowadays, this process is mostly known under the term 'skinning', referring to how the mesh is attached to the rig, just like the skin (and meat) of a human is to their bones.

The concept behind it is that each vertex of the mesh can be attached to one or more bones, which prevents breaking the object up into multiple static blocks of skin that are completely fixed in relation to their bone, as the individual components of a mechanical creature might be. Instead, a vertex can be proportionally affected by multiple bones, achieving smooth gradients between joints. To configure how much a vertex is affected by which bone, “each vertex has a per-bone weight giving a measure, in the range [0, 1], of the influence of the particular bone on that vertex.” [5, p.63] These weights can be automatically determined based on the vertex’s proximity to any given bone and afterwards manually adjusted through *weight painting*, an example of which can be seen in Img.3. This can help fix any deformation issues that can occur with the automatic weight assignments, especially in areas like shoulders and hips, where a lot of bones branch out in many different directions and where incorrectly balanced weights can cause the mesh to collapse inwards unnaturally when bending joints too extremely. (Img.4)



Img.3: weighpainting example, red areas signal higher weight values, while blue areas show a weight of 0



Img.4: volume loss example

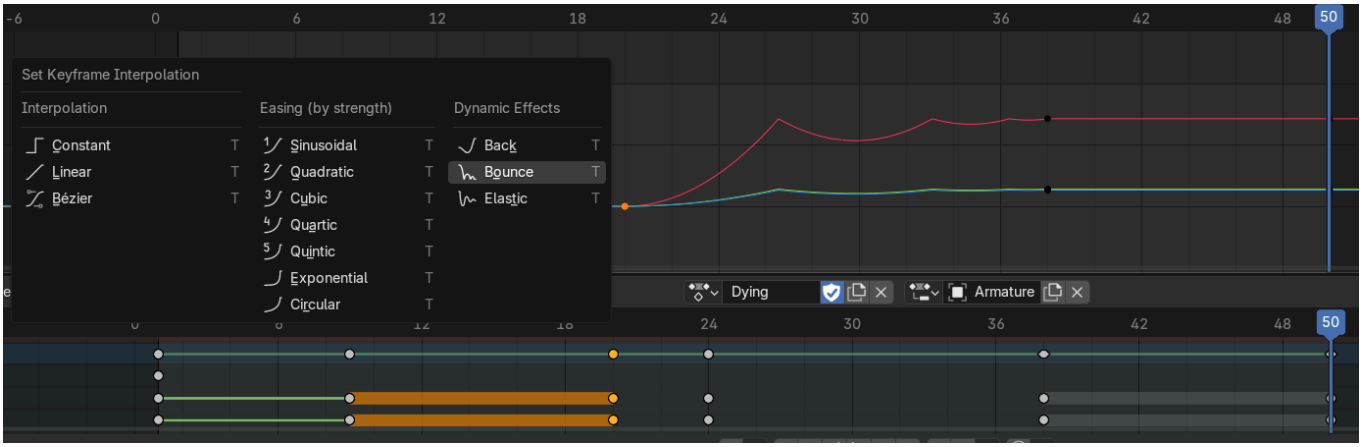
Determining the final position of each individual vertex is done by combining three inputs: first, the modified angles of each bone within the animations is determined. Then, for each vertex, the angles of all connected bones are combined according to their assigned weights. And finally, these modifications get applied to the vertex’s initial rest state that has been set for the mesh. [4]

To animate a finished rigged model, typically a keyframe base with added interpolation curves is used. The keyframes are a set of still poses in which an animation can be broken down into, that show the defining positions that will later on make up the movement. Adding no interpolation whatsoever in between these keyframes, results in harsh jumps from pose to pose without any actual

Vid.1: Test Animatic
https://www.youtube.com/watch?v=3O3XQtklK38

movement connecting them, as can be seen in this progress video[1] (Vid.1) from an animation that has been made as part of a previous project. There, these still poses have been used to roughly map out the overall movements and cuts, similar to how a storyboard would act in traditional animation. For the final piece (Vid.2), a few additional keyframes have been added for secondary movements like head or hips swings, as well as the harsh jumps having been smoothed out by interpolation curves, that describe how to move from one pose to the next.

Vid.2: Dance Animation
https://www.youtube.com/watch?v=DzTh7wAFF60



Img.5: example of *Bounce* Interpolation curve applied to keyframes

The type of curve used can have a huge impact on the final look and feel of the animation. While using linear interpolation results in a very accurate, yet stiff looking movement, which might be suitable for robots or other mechanical animations, a more complex curve can add, e.g., inertia, with the movements briefly overshooting their target position, anticipation, where a movement gets preluded with a pull in opposite direction, or elasticity, with joints quickly snapping towards their destination and then bounce around shortly, before finally settling. So without having to add any additional keyframes that clutter up the animation timeline, the resulting movement can be adjusted individually.

An often used alternative to animating keyframes manually, is through motion capture technologies, that record all movements an actor does while performing a certain action, maps them onto a target rig, and then convert it into keyframes. Depending on the capturing method

[1] Note: Due to inconsistent video support in text-based documents, all referenced videos are represented within this thesis as images, but can be watched by following the provided links underneath each image description.

used, these can be used to create extremely detailed and life-like animations, especially for highly complex subsystems, like facial expressions or hand movements.

One of the most common types for recording these motions is optical tracking, which uses a complex setup of multiple cameras to track the position of strategically placed markers on an actor's body. This produces some of the most accurate and detailed results, making it a standard for motion capturing in high-end productions like cinema films or AAA games. The downside of this approach, however, is the very high cost required for this setup. To accurately track all markers in a 3-dimensional space it requires a dedicated space fitted with at least eight to sixteen cameras distributed within. [6, p.81] This also means this method is confined to work exclusively within such a specialized studio and cannot flexibly be moved to a different location.

Instead of visual markers, inertia based tracking fits the actor with a set of sensors, which detect rotation and acceleration, from which the body movements can then be inferred. Since these sensors do not measure any specific location within a 3-dimensional space, but instead just record the movement relative to the previous pose, they first need to be finely calibrated to the initial rest position. Additionally, they tend to accumulate smaller errors over time, leading to a noticeable drift when used for longer sessions. [6, p.84] With proper and frequent re-calibration, however, this method produces fairly accurate results without being restrained to special studio environments, making it a flexible method that often gets favored for applications in live performance, e.g., field-based motion studies or sports analytics. [7]

A similar approach of distributing sensors across the actor's body is used by magnetic tracking, which measures the position and rotation of individual body sections within an electromagnetic field. Unfortunately though, these sensors are far more sensitive, especially to interferences from other magnetical influences. Additionally, "magnetic fields decrease in power rapidly as the distance from the generating source increases," [6, p.85] which limits their effectiveness to a much smaller space than other motion capturing types. Although, due to their high accuracy, they are often used to track smaller scale motions, like hand and finger gestures.

A mechanical approach to motion capturing is possible by providing the actor with an elaborate exoskeleton suit, "equipped with sensors, joints, and linkages that directly record body movements." [7, para.3.5] While showing a great robustness against various external influences, like lighting, magnetical interferences or outdoor spaces, the bulky suit impacts the

natural behavior of the actor and restricts their range of motion, limiting the realistic and natural appearance of the motions. [6, p.83]

Finally, the markerless method works without placing any equipment on the person directly, and instead relies entirely on computer vision technologies to analyze the recorded movements. [7] Due to requiring the least amount of specialized hardware and leaving the most physical freeform for the actor, this type gets often used for more casual applications. For example, the Xbox 360 included a separate motion tracking camera, which could be mounted on the TV, and allowed the user to play physical games which focus on, e.g., sport or dancing, while only using their own body as an input method. However, as anyone who owned such a system can likely confirm, this technique of motion tracking is not the most accurate one, and frequently gets affected by low lighting conditions or to uneven backgrounds.

Procedural Animation

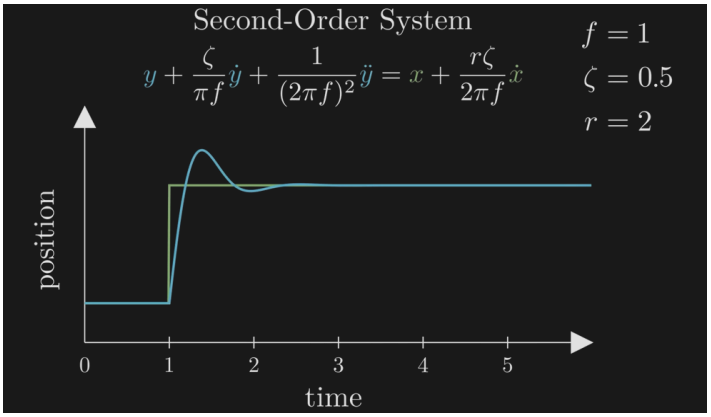
Animations that do not statically rely on premade keyframes to drive the movement, but instead are adaptive, driven by direct impacts of the environment or the character, are referred to as Procedural Animations. This umbrella term can include many different types of programmatical adaptations. While it can refer to an animation that is completely procedurally generated, it can also describe a mix between manual keyframe animations with variable parameters.

In Bereznyak's talk [8], for example, they showcase a way to reuse pre-made animations built for one specific rig, and apply them to another skeleton through *retargeting*. They show a variety of different body types and limb ratios, which are all animated based on the same rig and the same motion captured movements. By defining different structures in the rig, e.g., defining the left leg as starting at bone X and stretching towards bone Y, the animation for said leg can easily be adapted to another different skeleton, as long as it has the same defined chains. This can be used to achieve a greater variety of characters and NPCs, while only needing one singular rig and set of animations, reducing both working time and file sizes.

In the case of scaled skeletons, Bereznyak mentioned how they can result in altered walking speeds, due to the change of leg length and thus stride width, and why simply slowing down the animation to achieve the same distance per time is not a feasible solution. They state that there is only a 10% to 15% range in which the animation can be sped up or down before the manipulation becomes very apparent as it conveys completely different information about the

velocity and mass of the object. Instead, their proposed solution to this issue is making the step size adjustable, and counter-scaling it by the inverse of the rig scale. This way a taller character will still move at the same speed, but simply make smaller steps in relation to their leg length, to mimic the same steps as the unscaled skeleton.

Since fully procedural animations do not rely on keyframes to drive the motion, it is also not possible to apply interpolation curves to modify their transitions. Instead, a video by the user t3ssel8r [9] explains how they used mathematical functions to recreate the behavior of interpolation curves but with continuous values. By using a time derivative they determine the rate of change of the incoming values and transform it to modify the slope of a curve. The green line in Img.6 represents the values of the determined position for a given bone, which gets set to a new discrete value at timestep 1. Instead of directly passing these values onto the bone unchanged, they get converted into the blue line instead, using the formula above the diagram, the outcome of which can be modified via the three variables on the side, with f adjusting the vibration frequency, ζ the damping strength and r the response time. In the case shown in Img.6 it creates a smooth transition with a small target overshoot and elastic bounce back, to achieve a more natural, expressive movement.

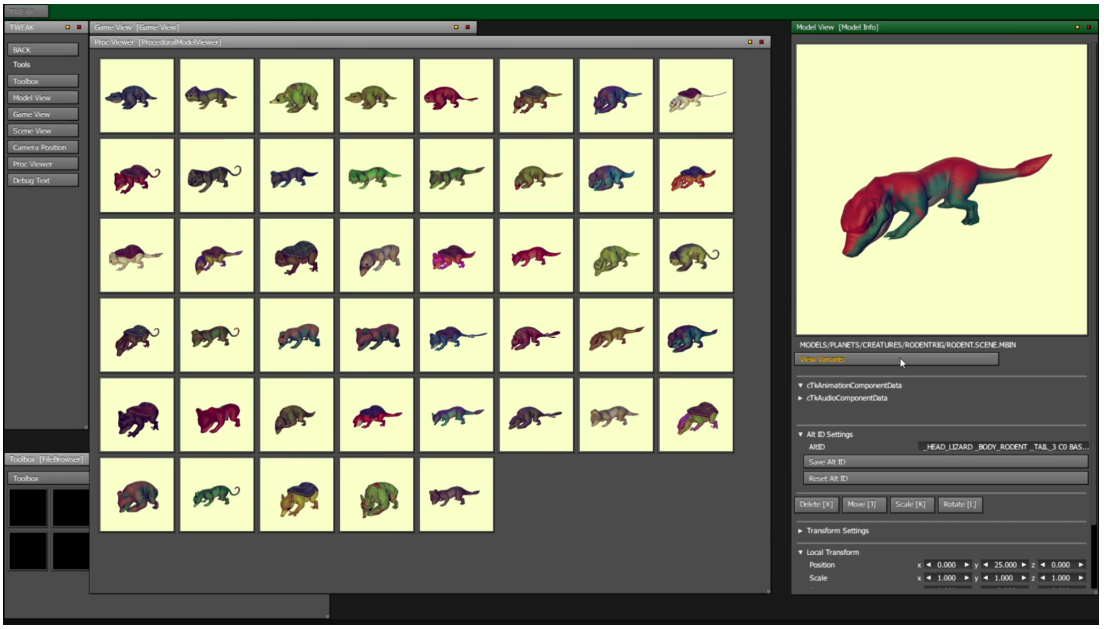


Img.6: discreet input values (green line) getting converted into continues output values (blue line) [9]

One issue of this method which t3ssel8r addresses is that due to the use of a time derivative, the entire system turns into a feedback loop, in which small errors can quickly pile up, leading to catastrophic failure. This can occur when the timesteps in between updates become too large in comparison to the parameters, e.g., due to lags. The proposed solution to prevent this behavior, is to determine a timestep for each update that, if it gets exceeded, would lead to the entire system losing its stability, and then constrain the actual time that passed since the last update to be below that critical value.

Meanwhile the game ‘No Man’s Sky’, does not just procedurally create animations, as developer Murray explains. [10] Instead, they are procedurally generating entire planets along with various types of animals that inhabit them. To achieve this, they defined some basic

silhouette types, like cow, rat, lizard or fish, each of which has a base rig that can be retargeted to the final procedurally generated animal, as well as a huge array of component parts with their individual attachment rules. For example, a ‘lizard’ type creature is always defined as some variation of a small scaly animal, with a forward facing head, two pairs of legs, with the front set having backwards facing knees, and the hind legs having forwards facing ones. Any variation within these parameters will always share the same base rig. However the final outcome within these parameters can be quite varied, as can be seen in Img.7.



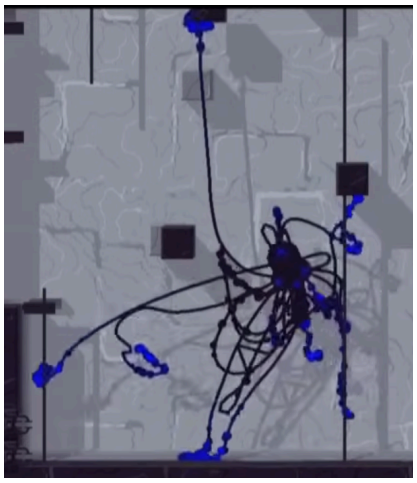
Img.7: No Man’s Sky procedural lizard animal type variation [10]

One game which is known for implementing very elaborate procedural animation systems is ‘Rain World’. In a talk by the creators Jakobsson and Therrien, [11] they explain how they created a unique ecosystem that, even when full of creatures that are fantastical creations or creative mash-ups of multiple animals, like its ‘slugcat’ main character, still appears believable and ‘realistic’ in itself, by using procedural animations to give their creatures not only individual movements that are reactive to their immediate environment, but also clearly convey their emotions of the NPCs. An example of this the developers show is of an enemy, trying to get to the main character, presumably to eat it, but getting visually more agitated and frustrated by its path being blocked. The developers stress how this very clearly visible cause and effect for behavior increases emotional attachment in the player to not only the game’s main character, but the entire world it inhabits.

To make the game feel like a complete ecosystem, that stays alive independently of the player, the developers decided to continuously simulate the entire environment, no matter if it is

currently visible on screen or not. Since this approach would quickly lead to lags or other performance issues, Rain World implements various tricks to separate the visuals from the physics/behavior simulation.

To achieve the game's heavy visual stylization, the developers explained how they prioritized an art focused approach that relies on optical illusions and 'faking' certain physicalities. As an example for this they showed the step-by-step creation of one of their enemies, a huge tentacle monster, which climbs through the level by using its various appendages to grab onto surrounding surfaces. (Img.8) However, instead of simulating actual rope physics on each of these tentacles, the body has a defined goal position which it 'floats' towards. If enough valid support points in the vicinity can be found for the tentacles to grab onto, the body moves towards its goal position faster and more directly. Should less arms be able to grab on, however, the body instead gets more strongly influenced by gravity instead, pulling it down and away from its target.



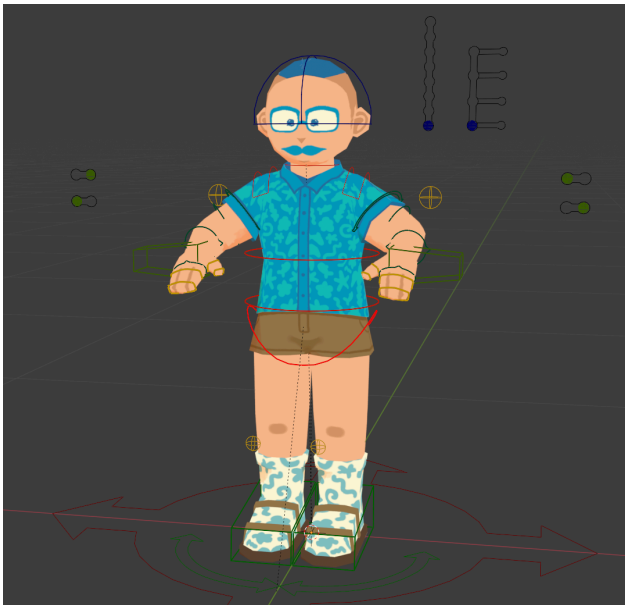
Img.8 climbing tentacle monster from the game *Rain World* [11]

The underlying layer of this behavior calculation therefore only simulates a floating central point, and a number of valid support points in reach of it. Only when the creature actually appears on screen does the additional visual layer get calculated as well, that gives these points the shape of a moving body, and figures out a way to connect the tentacles to the many grabbed points.

One aspect that makes this technique work so well is the fact that all creatures in Rain World are some other-worldly fantasy creatures, which often appear even further removed from real-world animals than the colorful reimaginations that inhabit the 'No Man's Sky' planets. Simulating real-world-adjacent animals adds the challenge of matching their movements and behavior close enough to their real respective counterparts. With fantasy creatures, however, the behavior can be much more creative since it is unrestrained by such expectations, and instead only relies on looking plausible by itself.

2.1 Inverse Kinematic

As previously explained, (p.4) a rig is constructed through a hierarchical set of bones, in which a bone will also be affected by all transformations of the parent bone it is attached to. This behavior of parents affecting children is referred to as *Forward Kinematics* (FK).
On the other hand *Inverse Kinematics* (IK) is, as the name suggests, the inverse of that: a way for the child bones to control their parents. With this method it is only necessary to specify the location of the root bone and tip of a chain (also referred to as the *end effector*), and the sections in between get automatically rotated to make the positions possible. This can be used, for example, to place a hand at a designated point in space to grab or otherwise interact with an object, or to firmly place the feet on the ground without having them slide around, no matter the movement the rest of the body performs, an example of which can be seen in Vid.3.

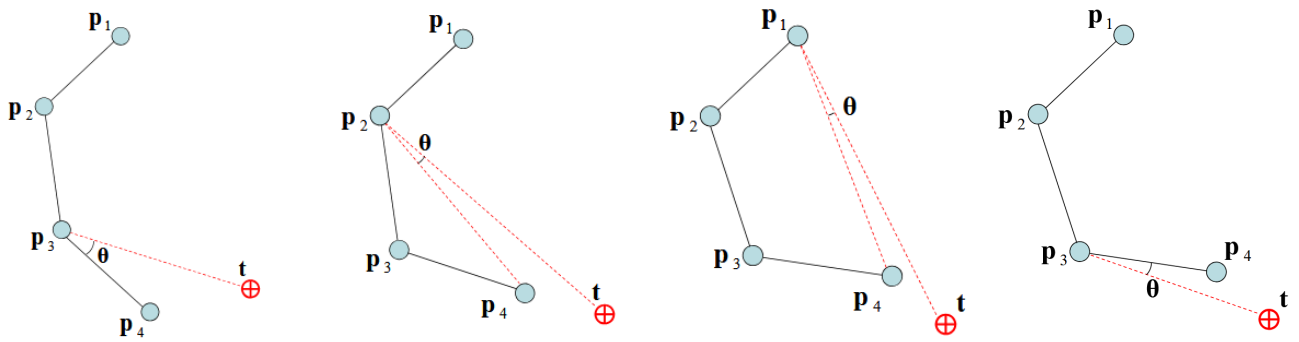


Vid.3: IK example
https://www.youtube.com/watch?v=JujNuvd81D8

There are different methods to find a solution to these IK chains, some of which are iterative approaches, in which the solution gets closer and closer approximated, instead of definitively calculated. The following three iterative solutions are commonly found in game engines and are also the three IK versions that are implemented in the Godot engine, since the GD4.6 update which released in January 2026.

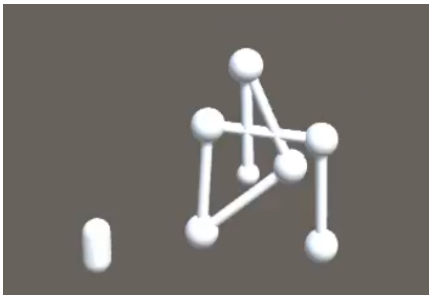
CCD IK

The ***C**yclic **C**oordinate **D**escent* works by continuously rotating each bone segment to point the end effector towards the target position, by starting from the end of the chain, and moving backwards towards the root, as is illustrated in Img.9. Once the root is reached, the whole process starts again from the tip (Img.9.d) and gets repeated as often as needed, until the end effector is at an acceptably small distance from its target position. [12, p.19]



Img.9: CCD IK. Adapted from [12, p.20]

The advantages of this method is the fairly fast performance and the simplicity with which it can be implemented. However, once the chain starts having multiple end effectors, for example, like a Y shape, where two ends share the same root, the implementation quickly becomes more complex. [12, p.20] While generally producing very stable results, in some specific cases CCD can lead to unrealistic or erratic motions, especially when very large distances need to be crossed, or when the target is close towards the root position. [13] This is due to the latter bones not considering what exactly is happening in front of them, instead, they only consider the vector between themselves and the end effector, causing them to overcorrect, as can be seen in Vid.4.

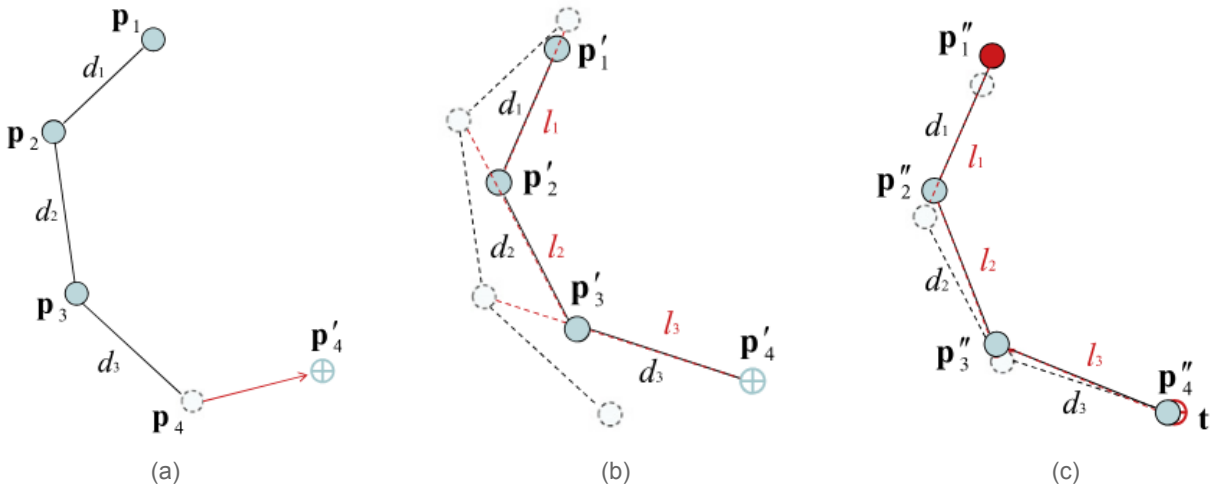


Vid.4: CCD example [13]
https://drive.google.com/file/d/1MWI4nTCuRzfCYVktQqbeh9cJGDa7BQGM/preview

Additionally, since the joint adjustments are always made from the end effector backwards, the method tends to overemphasise the movements of latter joints, which might not be the desired behavior. [12, p.20] However, most of these issues are far less pronounced for a more limited amount of bones, [13] making it a great approach for the more commonly occurring two bone IK-chains, like legs or arms.

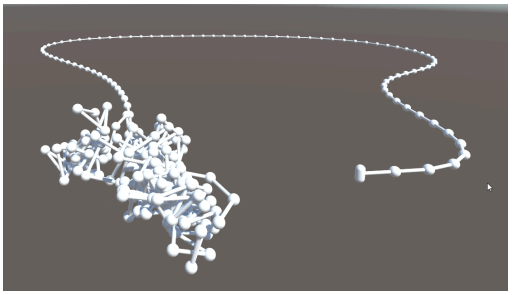
FABRIK

The ***F**orward **A**nd **B**ackward **R**eaching **I**nverse **K**inematics* (FABRIK) approach essentially works by connecting the individual joints with distance constraints, mimicking the way actual physical links of a chain work. That means, if one end is being moved, the connected bones are being dragged behind it. After the end of the chain is being moved to the target position (Img.10.a) each consecutive joint is being moved towards it in a straight line, until all original distances are restored. (Img.10.b) Afterwards, the same procedure is repeated, however, unlike with the CCD algorithm which always starts from the tip again, FABRIK continues on from the root bone and moves back towards the tip again. (Img.10.c) This continues until both the root and the tip of the chain rest sufficiently close to their respective target positions. [12, p.22]



Img.10: FABRIK. Adapted from [12, p.22]

“This method, instead of using angle rotations, treats finding the joint locations as a problem of finding a point on a line” [12, p.22] which makes the implementation very easy and keeps the overall performance cost low, making it the fastest performing IK solution out of all previously addressed. It is also easily expandable to work with more complex chains, which, for example, include angle constraints between individual joints or have multiple chains branching off of it. Additionally, by changing the iteration direction routinely, FABRIK returns a very smooth, even result, without erratic motions.



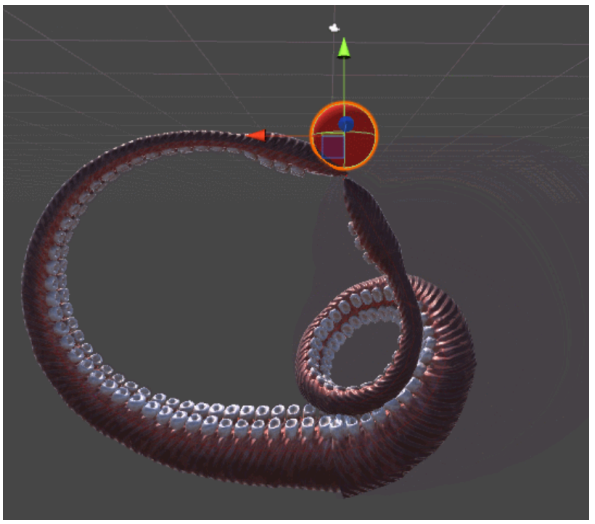
Vid.5: FABRIK extra long chain example [13]
https://drive.google.com/file/d/1fuTiJo2v7gd1ATYmU8BouA_ZQn-FwHCn/preview

Finally, due to being modelled after a physical chain, it recreated the behavior very accurately and realistically in that regard, as is demonstrated clearly in Vid.5. However, this might not always match the desired outcome, e.g., in cases where one would expect the motion to be primarily driven from the root bone on.

Jacobian IK

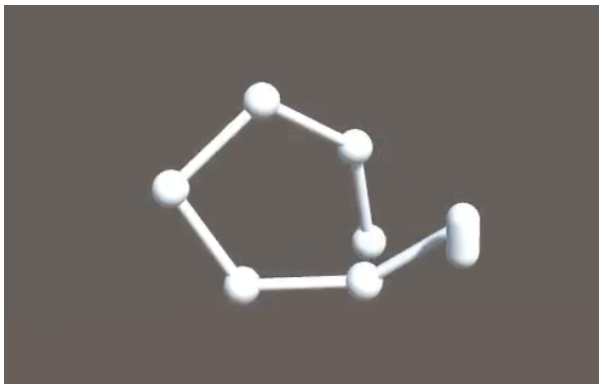
Instead of the previous geometrical approaches for posing IK chains correctly, the Jacobian Method is a numerical approach, which creates a linear approximation of the end effector's movements. [12, p.12] In a process called *'Gradient Descent'*, the system attempts to iteratively minimize a specified error function, which eventually moves the end effector closer and closer to the target. [13]

In Zucconi's IK tutorial, [14] they are additionally introducing a comfort function, which describes a natural rest pose for the IK chain. Instead of evaluating the error function solely on the distance between end effector and target, it additionally evaluates the overall distance from said rest pose, resulting in more natural poses. An example, in which they show off this approach, is a pair of octopus tentacles, reaching for a ball. As can be seen in Vid.6, the left tentacle reaches for the target in a big arch, making it look robotic, stiff, and overall unnatural compared to the behavior we would expect from an actual octopus. However, the tentacle on the right, which evaluates multiple criterias in its error function, seems far more convincing in the way it is curling around itself. Besides the distance towards the target ball, this arm also attempts to position its end effector with the same rotation as the goal object. Additionally, it tries to minimize the average angle between all of its joints, resulting in this organic looking spiral formation.

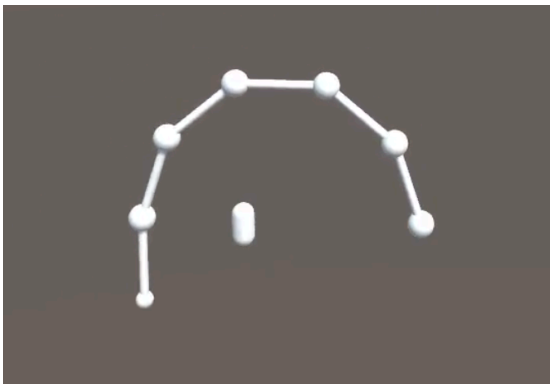


Vid.6: Tentacle with simple error function (left) vs. Tentacle with additional comfort function (right) [14]
https://www.alanzucconi.com/wp-content/uploads/2017/04/tentacle3.gif

The advantages of Jacobian solutions are the very smooth and even results it produces, as well as its customizability in terms of behavior, as is demonstrated by the tentacle example. Additionally, Vid.7 shows how this method remains stable, even in cases where it needs to cross large distances, or where the end effector comes very close to its root.



Vid.7: Jacobian IK example 1
https://drive.google.com/file/d/1zrUAUwQyYXh3PczscdTu4tn8ZxatKBZF/preview



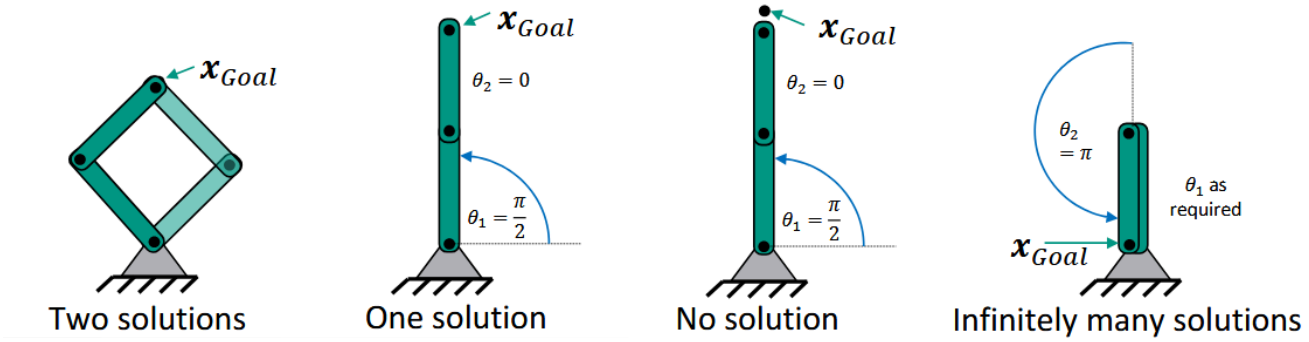
Vid.8: Jacobina IK example 2
https://drive.google.com/file/d/1SNNxW59F2s481fuJXEDcIKVpsULysObq/preview

However, due to the more complex matrix operations required, Jacobian IKs are more difficult to implement, as well as perform far slower than the previous two solutions. While this might be less of an issue for pre-rendered cinematic animations, it makes it unsuited for many applications in realtime game animations. Besides the high calculation time, it additionally takes quite long to converge to the solution, a behavior that is especially strongly pronounced in Vid.8. While it is possible to increase the steps taken towards the solution to speed this process up, this needs to be done very carefully since it can quickly lead to the chain overshooting the target position, which will result in vibrating around the correct solution. Determining the appropriate scaling factor for the steps is possible, but requires even more resource intensive calculations, making it less suitable for video game applications.

Pole Target

Besides their performance, iterative solutions are often favored since one definitive mathematical solution does not always exist. Img.11 shows the different cases that exist for a simple two bone chain that exists in merely two dimensions. Most scenarios fall into the case of Img.11.a, which include a target position that is somewhere *inside* the chain's maximal range of motion. Already, this 2D, two bone case offers two equally possible solutions to choose from, the number of which rapidly increases with additional dimensions or chain segments.

To decide which one out of the array of possible positions should be chosen, different strategies can be utilized. The advantage of an iterative solution is that it does not need to *decide* since it will simply choose whichever pose it finds first, and thus also guarantees that the 'chosen' solution is a natural continuation of the previous pose.



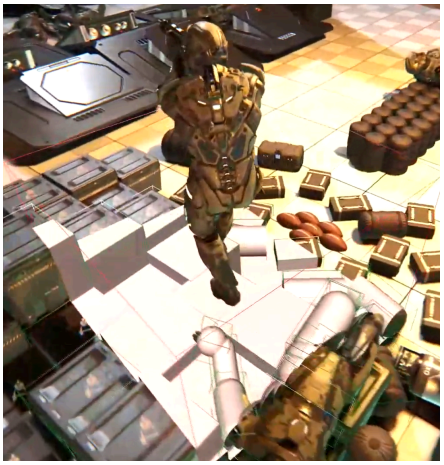
Img.11 a) goal is within the range b) goal is at exactly the maximum range c) goal is outside of range d) goal is at root position [15, p.17]

Otherwise, the selection process can be configured to favor certain options, by imposing stricter angle constraints on the individual joints. While this in most cases still leaves a number of possible options, it can help eliminating many undesirable cases early on.

The solution that gets implemented most often, however, is introducing an additional pole object that 'pulls' the curvature of the chain towards it. This is an especially helpful technique for human or animal rigs, since it allows you to set knees, e.g., as being forward or backwards facing, while also allowing for the most freedom in positioning the knee as needed for specific movements, independently of the hips and feet.

Gait System

In a talk by Herzog, a developer for the upcoming game Star Citizen, [16] they explain how they used IK chains to develop a procedural gait system. The first step of this process is to adjust the height of the feet's IK controllers to the surface of the terrain, as well as tilting the hips accordingly to match the legs' pose comfortably. However, this alone can result in frantic up and down movements in certain cases, e.g., when running over small debris. To combat this, they needed their characters to not only react to environments, but instead to make strategic steps in advance to prevent things like stumbling off an edge or sliding down slopes. This challenge is also briefly touched upon by Bereznyak, [8] who describes it as a '*Nostradamus Mode*' due to the needed prediction of the upcoming terrain.

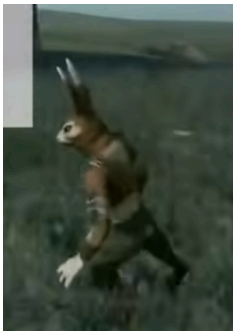


Img.12: simplified colliders in ray intersection area around character [16]

However, Herzog explains how relying on Ray-World-Intersection to determine the complex surrounding terrain is too resource intensive, making it too slow of an approach to apply to the required fast character movement. Instead, they limit the ray scanning to only a small bounding area around the character. This area scales dynamically along with the character's move direction and speed, to incorporate the least amount of scanning area necessary for predictions and reactions. Additionally, inside this bounding area, the obstacles are approximated with simplified primitive shapes, making the ray intersection calculation far faster, while still being accurate enough to allow for both vertical as well as horizontal foot placement adjustments at runtime, even while the character is sprinting.

A different approach for a gait system that partially relies on keyframes is used by Rosen, the developer of the game Overgrowth. In their talk [17] they explain how they developed the main character controller, starting with focusing exclusively on the movement and the player input before adding any kind of animation. For that they are treating their character as a central point of mass with extended colliders (Img.13.a), making sure it can move around the environment, without getting stuck on any obstacles, and always keeping the same distance from the ground, thanks to the lower collider detecting the floor height. Only once that basic interaction works, do all of the visuals get implemented.

(a)

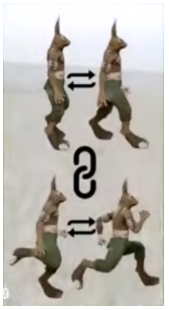


(b)

The first step is integrating the base mesh and rotating and tilting it along with the acceleration generated from the player input. (Img.13.b) Afterwards, a few selective keyframes are being introduced to be procedurally applied to create the walking animation. Those keys are limited to just two different poses, the pass pose and the reach pose, which simply get mirrored for the step of the other leg, plus two additional poses for a separate running animation. (Img.13.d) Instead of relying on the placement of the keyframes on a timeline, as is done with regular keyframe animations, Rosen determines when each pose needs to be

applied during the walk cycle, by measuring the amount of distance covered during the walk (Img.13.c), which additionally ensures that the footsteps are always in sync with the ground and the feet do not slide across the floor unnaturally.

(c)



(d)

Img.13: D.Rosen's animation process. Adapted from [17]

Since this approach does possess a keyframe base, it is possible to finely blend between different keys as needed. For example, to achieve an intermediate jogging speed between walking and running, the animation does not just interpolate between the keyframes of either the walk or the run; instead both options are blended together to procedurally generate two 'new' jogging keyframes, adjusted perfectly for the current movement speed.

This technique can be further expanded by separating animations into different layers, e.g., the movement of the legs, the movement of the arms, the curvature of the spine and the rotation of the head. This layered system makes it possible to easily combine different animations, without them impacting one another in unwanted ways. Bereznyak uses the same technique in their adaptive base skeleton work, [8] where they use animation layers to create features like arms and general upper body bones, that are being used for animations that carry or otherwise interact with props, without the base walking animation of the legs being affected.

2.2 Physics Based Ragdolling

Ragdolls are a type of procedural animation where the entire body of a character gets moved through physics simulations. Instead of relying on the posing of bones, each body section is modeled as a physics object that approximated the real shape and mass of the limb, and is connected to the other sections through constraints that replicate the range of motion that their real counterparts exhibit. Each of the body sections then gets physically simulated, causing the overlaying body mesh to follow their movements, similarly to how it would when skinned to a regular bone rig.

Ragdolls are commonly used to create more interesting death animations in games, since these movements need to be more reactive to the exact situation and environment the character is in at the moment of death. After all, once someone dies, they stop being an articulated character, affected by the actions of their muscles, and instead act more like an object which gets subjected purely to external influences. So these reactive, physics-based motions often create more convincing deaths than a static pre-made death animation would be able to.

During a previous project, focused on learning the different physics simulations Blender offers, the creation of such a ragdoll has been explored. As a base, a character model has been used, which had been provided by Laura Unverzagt[2], as a test object to practice rigging on.

The objects had been specifically modelled to approximate the shape of each major body section, as can be seen in Img.14, which were then simulated with Blender's RigidBody Physics. To transform this collection of loose body parts into a completed figure, different types of constraints connect the individual body parts, the behavior of which depended on the type of constraint chosen.

Img.14: (L) full model with rig overlay (R) proxy model

The Point Constraint, for example, allows both connected sections to fully rotate around their shared center, while keeping the distance the objects had from each other initially. If one of the objects starts moving away, like falling due to gravity, it will drag the other along with it. The

[2] https://lilotm.github.io/

allowed rotation of two connected objects can also be restricted, however, e.g., with the use of a Hinge Constraint instead, which allows rotation exclusively along one axis. With this, it is possible to regulate the body parts' movements to a more humanly natural range.

After constraining all neighboring sections, the result is a figure that, although visually disconnected, acts as one interlinked object, and behaves physically plausible. Afterwards, the rig's bones can be set to follow their corresponding proxy object's transformations, and the original character mesh, which is skinned to the rig, would follow.

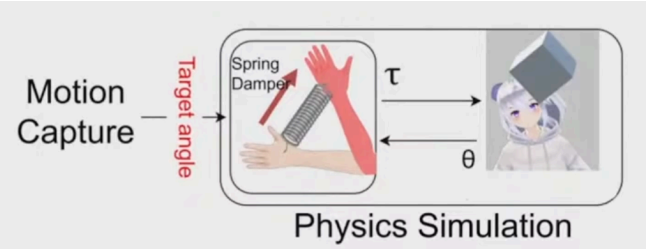
Finally, there was one more challenge this rig provided: the character had a tail that needed to be simulated as well. Instead of adding a long chain of several small bones to enable the flexible tail movement, a special bone type Blender offers has been used, called Bendy Bone (BBone), which, as the name suggests, is flexible and thus perfect to achieve the smooth bend of the long, thin tail. The curvature of each end of a BBone can be copied from another bone, essentially creating handles that allow the shape of the bone to be manipulated similar to how a curve tool would work. That meant, instead of having to create ragdoll proxies for each of the tail's '*vertebrates'*, a cube has been added at the tail tip, and connected to the hip proxy object using a Spring constraint, which added a bouncy, elastic movement between the base and tip of the tail.

Vid.9: Ragdoll on stairs
https://www.youtube.com/watch?v=GlVi1bkIsq0

Besides limply subjecting to gravity, however, it is also possible to blend the effects of ragdolls with pre-made animations. One approach is to only simulate certain sections of the model, to apply some secondary physics to attachments like hair strands, tails, or ears, like Rosen [17] did for their rabbit character.

Another option is fully simulating the rig, but additionally constraining it to an animation. Abreu explains in their tutorial [18] how they achieved this effect by splitting the rig into two separate objects: a static reference that ignores any physicality and only plays the animation, and the ragdoll, that tries to imitate the reference object, by rotating every bone to match the orientation of the corresponding bone, while simultaneously being subjected to the physics simulation.

Sugimori et al. [19] are using the same animation-matching approach, however, they are focusing specifically on virtual avatars that are tracking the motions of a user at realtime. To enable their avatars to react appropriately to collision or physical interactions with other avatars, the physical rig is connected to the tracked input rig with spring connections that elastically pull the avatar towards its intended position. (Img.15)



Img.15: Diagram of the physics-modified avatar tracking process. Adapted from [19]

Vid.10: comparison between regular motion capturing avatar methods, and physics-modified avatar tracking. Adapted from [19]. Visible at time stamp 1:07
https://youtu.be/GpNKBbl6OLk?si=yJp-9r3VTFWCDbq4&t=67

3 SoftBody Physics

To construct a 3-dimensional model, the position of each vertex is saved relative to the object's origin point, along with how these points are connected to form the model's surface. This way, even if the entire object gets moved around, all vertices will stay at a fixed distance from one another, keeping the shape of the object consistent. To physically simulate such a RigidBody (RB), the mesh can generally be reduced to one central point, for which forces like gravity or other accelerations are calculated. Meanwhile, the mesh's surface only needs to be considered if detecting collisions with other objects is relevant.

A SoftBody (SB) on the other hand, is an object that shows some flexibility and deformability, by allowing the individual vertices to move in relation to each other. Yet, the object is expected to keep its general shape or at least attempt to return to it, as opposed to fully deflating or stretching infinitely under pressure, be that through gravity, collisions or other forces.

A subtopic of SoftBodies is Cloth simulation. Oftentimes, Cloths are 2-dimensional objects, like, for example, a flag, which is simply a rectangular plane, as opposed to complete 3D meshes, like, for example, a bouncy ball. Even if the topology of a Cloth is flat though, simulating it is not simply a 2-dimensional problem, since the object still exists and freely moves around in a 3-dimensional space. However, there are also Cloths with a more 3-dimensional form, for example, a shirt, which approximates a general tube shape. The biggest difference to full SBs is that Cloths do not have a defined volume that needs to be preserved, which makes simulating them a little more simple. While a flag has no volume at all, a shirt's volume is variable, depending on the body it is worn on. Meanwhile a bouncy ball has a fixed volume that should not be able to change, even if the ball gets temporarily deformed.

Even still, the topic of digitally simulating any type of SB is naturally far more complex than RigidBody Physics, since instead of just one point needing to be constantly simulated, suddenly each object consists of several, interconnected points that need to be re-calculated for every frame. This also transforms the problem not only into a more advanced physical calculation, but also into a performance optimization issue, especially for use in video games, or any comparable applications that require a framerate high enough to allow for user interactability. The following chapter will introduce multiple approaches to simulate SBs, that each vary in physical accuracy, complexity, and performance, to analyze the advantages and disadvantages of the different ideas and techniques.

3.1 The Mass-Spring Model

The Mass-Spring Model is used to simulate multiple interconnected, yet distinct mass points. Kenwright et al. [20] explore in their work how this model can be used to create flexible, elastic SoftBodies. This approach works by imagining each vertex of the mesh as its own mass point, and all edges as springs, connecting these masses together. The vertices are then simulated as if they were independent objects, reacting to external forces like gravity or collision, with one of these forces being the connecting springs, ensuring that two points do not move too close or far from each other. Hooke's Law (ut tensio, sic vis - "as the extension, so the force") states that the force created by a spring is proportional to its deformation, which means that the more a spring gets compressed or stretched, the stronger will be the force that pushes them back to their original distance.

While the resulting mesh possesses an elasticity that is suitable for cases in which the simulated object is made from very stretchy or bouncy material, it does need a few more modifications to be also applicable to other situations.

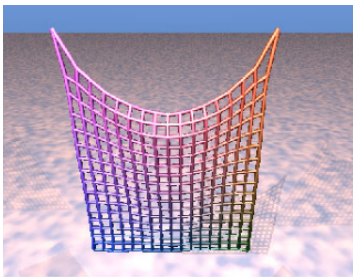
For example, depending on the intended properties of the simulated SB, the model also needs to be able to account for permanent deformations. Kenwright et al. do that by defining an elastic limit for the springs that functions as a point of no return, after which Hooke's Law will be ignored, as it "will no longer accurately approximate its elastic properties". [20, p.4] Instead they recalculate the spring's baseline length and position, setting the current, deformed shape as the new default.

Meanwhile Provot [21] describes another approach to circumvent what they refer to as '*super-elastic*' deformations. Since their work focused mostly on Cloth objects, a permanent deformation is not a feasible solution, since most fabrics would realistically tear in such instances, instead of stretch.

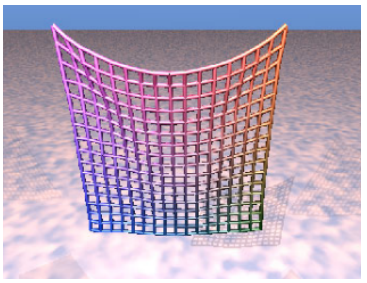
They also note that simply increasing the springs' stiffness does not produce satisfying results either, since it not only fails to provide the needed stability some materials require, but it also negatively impacts the performance. Due to the high force that gets generated by a '*super-elongated*' spring, it is possible for the two connected points to overshoot their desired distance and to oscillate, which causes an erratic and jittery behavior. Meanwhile, with increased spring stiffness, the time these masses need for one full swinging period decreases.

If a critical stiffness value is exceeded, the swinging period becomes shorter than the timesteps in which the simulation is being computed, thus requiring an increase in recalculations per timeunit, causing the algorithm to become more expensive.

Instead of relying on spring stiffness, Provot implemented a maximum deformation rate to limit how much a spring can deviate from its rest length. For that, first the regular, unrestrained deformation of each spring is calculated. Afterwards, each spring that exceeds its individual maximum gets scaled back, by moving both connecting points evenly towards the middle. In cases where one of these masses is fixed in location, e.g., one side of a flag being attached to a flagpole, the free end is moved towards the fixed one, until the spring's maximum length is reached.



(a) Deformation with unrestrained spring



(b) Deformation with applied constraints

Img.16: Deformation of an elastic model of a sheet hanging by two adjacent corners. Adapted from [21]

If this belated shifting of the vertices causes any neighboring springs to exceed their maximum deformation, they then need to be restrained as well. In theory, this could noticeably impact the performance again, should there be many such instances where restraint is necessary. However, in Provot's experiments, that focused on partially pinned Cloth objects, cases of such extreme stretching concentrated primarily in small regions, as can be seen in Img.16.a, so applying the constraints method propagates the deformation more uniformly throughout the mesh.

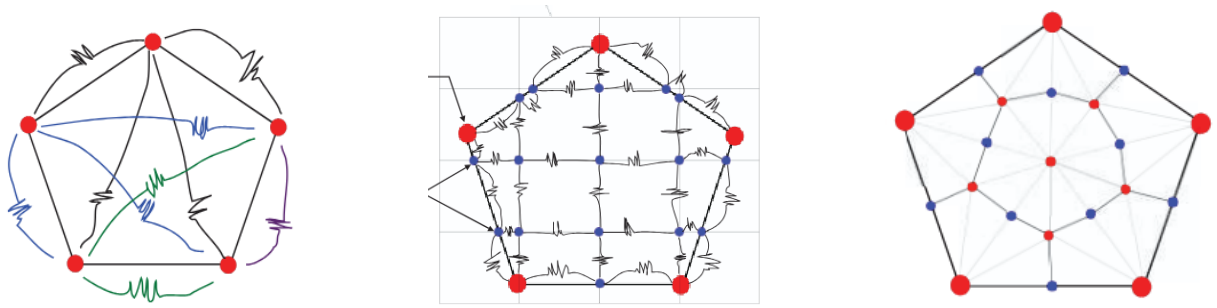
Furthermore, Provot added more springs to the surface to provide additional support and stability. While the regular Spring-Mass Model only simulates structural springs that correspond with the edges of the mesh, connecting each vertex to its direct neighbor, Provot introduced new shear springs and flexion springs.

The shear springs are located inside of faces, connecting the opposite vertices of it that have no edge between them, to make the individual face more resistant to collapsing and deforming. It is important to note that this approach only works with quad-meshes, which are meshes that consist entirely of faces that are made up of exactly four vertices. Tris-faces in comparison, are drawn between only three vertices, so adding shear springs would not be possible, since all possible interconnections between these three points are already covered by the structural springs. Meanwhile, faces that are constructed of more than 4 vertices, referred to as n-gons,

should generally be avoided in non-static meshes, since, due to their irregular shape, they do not deform as neatly.

Secondly, the flexion springs connect each vertex to its second neighbor, thus adding more stiffness to the structure by preventing individual faces from folding towards each other.

In [21] the deformation constraint is only applied to the structural and shear springs, leaving the flexion springs unconstrained. The reason stated for this decision is that fabric sheets, the likes of which Provot was focusing their work on, are generally able to be folded without problem. And while these methods result in great results for non-closed, plane-like surfaces, simulating a closed object in this fashion still results in a rather unstable model that easily collapses into itself, due to missing internal supports.



(a) Brute Force Interconnection (b) Uniform Grid (c) Recursive Subdivision

Img.17: Different approaches to place internal springs. Adapted from [20]

The simplest solution to this issue, referred to as the *Brute Force Method* in [20], is to connect every mass not only to its neighboring vertices, but to all available points of the mesh (see Img.17.a. While this method does provide adequate internal stability, it also is unreasonably costly due to the huge amount of additional springs that need to be calculated. Some other alternatives they explore rely on different versions of voxelation or tessellation instead, to place the internal springs more strategically.

Matyka and Ollila [22] describe a different approach of providing structural stability, which relies on internal pressure. Their approach is based on how Cloth objects are being manipulated by an external wind force, with the difference that the pressure/wind force is being placed inside of a closed model. The linear wind pressure is then replaced by a more advanced pressure calculation, called Ideal Gas Approximation. Furthermore, unlike a wind force being orientated in one static angle, the pressure is being calculated for each face of the model separately, to be scaled proportionally with the face's surface area and directed to be parallel with its normal

vector. Finally, each vertex gets readjusted by averaging the determined pressure based displacement of all its adjacent faces.

While this method produces decently fast and accurate results, it also is considerably complex in its implementations, since it relies on thermodynamic approximation to calculate the pressure of an internal gas on the outer surface. A slightly simplified version of this approach is presented in the video form Argonaut, [23] where they use volume preservation to provide structural stability. If the calculated volume becomes too small compared to its default state, the object gets inflated, proportionally to its volume deviation. Should it become too large instead, it gets shrunken down.

Both [22] and [23] methods depend on continuously determining the object's volume, a task that can be its own rather complex topic, when working with highly irregular 3-dimensional objects.

Since only 2-dimensional objects are used in [23], they utilize a method called Shoelace Formula, which splits any potentially irregular polygon into trapezoids. Iterating through the edges, each one is used to form a trapezoid together with the x-axis, the area of which can then be easily calculated. If the edges flow in a clockwise direction, which means that vertex v_0 has a x coordinate that is less than v_1, the calculated trapezoid is added to the area of the polygon. If the edge is orientated in the opposite direction ($v_{0x} > v_{1x}$), the trapezoid is instead subtracted from the polygon's total area. While this technique is very effective and easy, it is not applicable for 3D meshes.

Matyka and Ollila [22] use a bounding object approach instead, to approximate the meshes volumes. Bounding objects are bigger, simpler objects, usually in the shape of primitives like boxes, spheres or ellipsoids, that completely enclose the actual object. Since these primitives are much faster to calculate, they are often used as a pre-check for tasks like collision detection or ray intersection. If nothing intersects with the bounding object, the actual mesh itself cannot possibly intersect with it either, so it is unnecessary to perform any more expensive test for it. Naturally, a bounding object is not going to reflect the mesh's volume precisely, but, as Matyka and Ollila demonstrate, it is possible to approximate the general change of the volume, which is sufficient to determine the intensity for the internal pressure.

A more accurate alternative they mention is the Monte Carlo Method, which relies on randomized sampling to approach a solution. In Czapla and Pleszczyński work, [24] they show how this technique can be used to determine a volume. First, the object, whose volume is wanted, gets a bounding box, similar to the last method. However, instead of simply relying on the bounding volume, the box gets covered with several, evenly spaced sample points, each of which then gets evaluated for intersection with the object. The mesh's volume can then be determined, since the ratio of samples outside vs. inside of the object will be the same as the ratio of the bounding volume vs. the object volume. Despite the fact that this method also approximates the volume, it still produces more precise solutions than the bounding objects. Additionally, the accuracy is easily adaptable, by in- or decreasing the amount of samples, depending on how accurate a volume is required, or how strongly optimized the performance needs to be.

In conclusion, the Mass-Spring Model is a popular approach that can be individually adjusted to best approximate the behavior of different types of elastic or deformable materials and objects. One of its biggest advantages is its fairly easy and customizable implementation, as well as its inexpensive calculation cost, which makes it such a favored method especially for use in digital animation or video games, since such cases primarily require results at frame rates that allow for interactive responses.

However, the Mass-Spring Model's biggest downside is that its fast performance comes at the expense of physical accuracy. The resulting simulations are merely visually plausible, which makes it sufficient for computer graphics, but lack the precision and correctness that would be required for analytical simulations.

3.2 The Finite Element Model

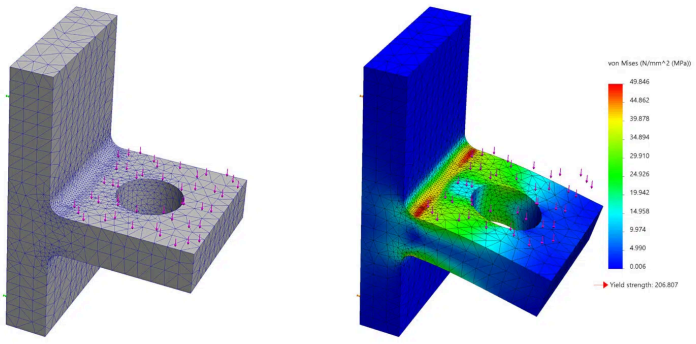
The Finite Element Model, also referred to as Finite Element *Method* (FEM) or *Analysis* (FEA), is a mathematical approach to approximate the solution of differential equations, which can be applied for a variety of problems and situations, which are closely detailed in the textbook *The Finite Element Method: Theory, Implementation, and Applications* by Larson and Bengzon. [25]

One use case of this technique is the physically accurate computation of the behavior of non-rigid objects. This is achieved by dividing a model into several separate pieces, each defined with its own physical properties. Based on physical laws and relation to surrounding elements, it is possible to determine a set of mathematical equations, which can be solved with the FEM technique to predict the parts' behavior.

Shumak [26] describes why FEM is such a popular method for engineering work, like structural analysis. Engineers can specify tests with various internal, as well as external influences, to predict how the entire structure would behave under different conditions or loads, simulating real-world operating settings. Though, instead of having to analyze the entire physical structure as one, FEM turns the problem into a more manageable set of smaller, interconnected, mathematical evaluations. By iterating through the individual elements of the model, each section is solved for its minimal potential energy, revealing possible structural weak points.

The advantage of this method is that it creates very accurate simulations that give detailed insight into how structures respond to different scenarios and exactly which sections need redesigning to reduce strain. It also allows for testing of multiple alterations and optimizations, without having to produce expensive physical prototypes for each tested variation.

However, the high computing power required for this procedure, as well as its overly complex setup, does make it unsuited for applications like video games, that require increased performance optimization.

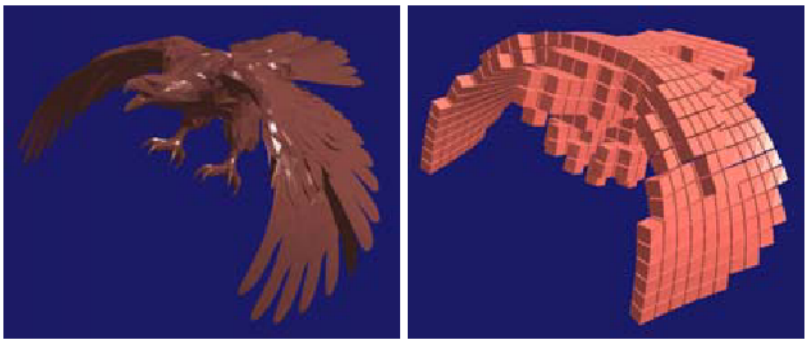


Img.18: Example of a FEM analysing stress distribution in a bracket [26]

3.3 RigidBody Based Deformations

As previously touched upon, the simulation of RigidBodies is less resource intensive than calculating SoftBodies, (p.23) so problems should be transformed into rigid physics whenever possible, to improve the performance. Exactly that is the idea behind this technique: using multiple RBs to create one SB.

Müller et al. [27] used this approach in their work with the goal to turn surface meshes into a volumetric mesh, for which physical behavior can then be calculated. To achieve this, they voxelized the original object, breaking it up into multiple small RB cubes. This collection of cubes then got constrained to one another, allowing each individual cube to move moderately in regards to its neighbors, while overall still causing them to move together like one large structure. The underlying RB structure then affects the original high-resolution mesh similarly to how a rig influences the mesh skinned to it: the mesh's individual faces conform to the transformation of their nearest underlying rigid cubes, thus following any movement or deformation of it. (see Img.19)

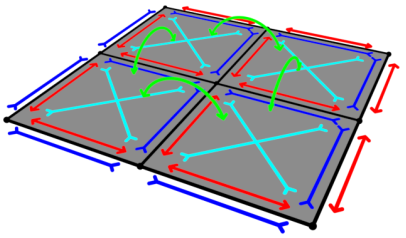


Img.19: Eagle surface mesh (right) for which a cube mesh was generated (left) to simulate elastic deformation. [27]

While the voxelized structure can be constructed exclusively with RB physics, it can also be expanded on by the implementation of SB based voxels, as done by Müller et al, [27] who utilized FEM algorithms to simulate the cubes, or James et al, [28] who use, what they refer to as '*Squashing Cubes'*, as a base for their voxels. Since the form of a regular cube provides more 3-dimensional stability with fewer needed steps for volume preservation, this technique can still improve performance even with SB cubes. However, the final results as well as calculation speed is highly dependent on the resolution of the voxelization, with an increase in cubes naturally increasing the calculation steps required. James et al. also note that "Limitations of coarse voxel resolutions are (a) close geometric components may be undesirably joined, and (b) the bending of thin surfaces can reflect voxel 'stair-casing.'" [28, p.1]

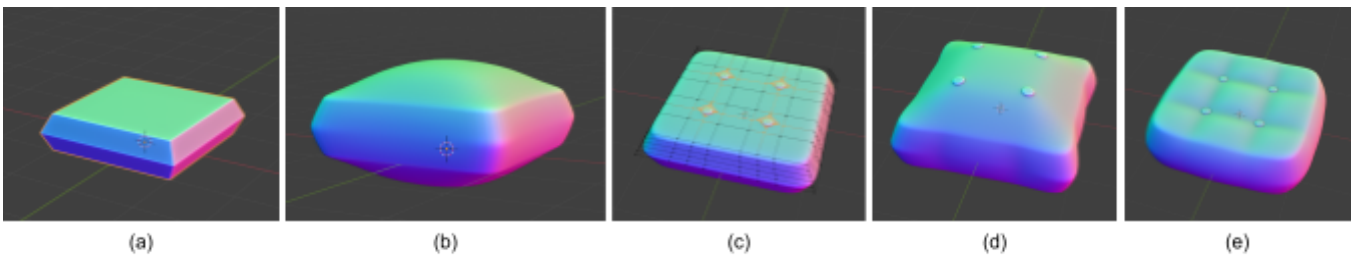
3.4 Blender Physics

While SoftBody and Cloth physics share a lot of similarities in functionality and utilization, Blender separates both into their own distinct simulation types.
Still, both of them are based on the Spring-Mass Model, but differ slightly in what sort of springs they simulate. The SoftBody type by default only enables the structural springs (Img.20, blue and red arrows) along the mesh's edges. This results in the objects immediately collapsing in on itself, due to features like shear springs (Img.20, cyan), self-collision or volume preservation, having to be separately enabled in the settings first.



Img.20: different spring types simulated on a Cloth object [29]

Blender's Cloth physics work similarly, with the biggest difference being the added '*angular bending springs'* (Img.20, green), which are a derivation of Provot's flexion springs [21]. However, instead of keeping the distance between two 2nd neighbor vertices, they are drawn between two adjacent faces. Functionally they serve the same purpose though, to prevent the mesh from folding in on itself. While Cloth Simulation is optimized for 2-dimensional objects, this approach still handles fully closed 3D objects remarkably well. They can be additionally improved by adding internal springs, making it function very similarly to how Blender's SoftBodies work. Alternatively, the Cloth Simulation has a pressure option to achieve the 3-dimensional volume, similarly to [22]'s approach. However, this setting seemed less intended for continuously simulating the behavior of a flexible mesh, but instead as a quick and simple way to model a soft object, as seen in the example in Img.21. The final Cloth shape can then be applied, to permanently save the mesh in the current shape, and potentially add a new layer of Cloth or SoftBody simulation to it, to make it reactive to the environment.



Img.21: creation process of a Cloth simulated pillow (a) simplified cushion shape model (b) base cushion with Cloth simulation (c) added geometry for buttons pin groups (d) button cushion simulated without pinning enable (e) button cushion simulated with pinned groups

Since Blender's physics systems are specifically set up to allow the user as much freedom as possible, each specific parameter of each simulation step can be fully customized as needed. While this includes technical aspects, like render time steps or accuracy, and circumstantial aspects, like gravity influence or air resistance, it also includes very accurate settings for fine-tuning the stiffness of each type of spring.

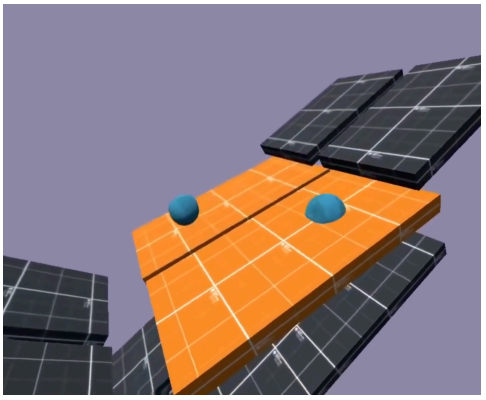
Additionally, the Cloth Physics section has a list of available presets that instantly set the various Cloth settings to imitate common fabric types such as Cotton, Denim, Leather, Rubber, or Silk. These settings are a good starting point to achieve exactly the kind of simulation required, but it is important to remember that the preset is still heavily dependent on the geometry of the object itself. The Mass-Spring Model works by interconnecting the mesh's vertices, so the amount of vertices per area is an important factor in how the end result will turn out. There are countless Blender compilation videos[3] online, which repeatedly show certain types of physics simulations with continuously increasing face count, clearly showcasing the effect a mesh's resolution can have on the final outcome, even if all simulation settings stay the same. Hence, e.g., a limb, which has a very different ratio of vertices, surface area and volume, will naturally behave differently than the main-body section of the same mesh.

To accurately simulate complex, non-convex meshes, like humanoid or animal bodies, Blender enables most simulation settings to be combined with the *Vertex Group* features.
These are specialized data structures that save a unique value for each individual vertex. While most commonly being used for skinning a mesh to a rig, they can be used for a variety of additional purposes. For example, in the pillow creation process (Img.21), the buttons can be added to a Vertex Group that gets pinned in place, leaving only the rest of the surface to be affected by pressure. Another use of this feature is to adjust the physics parameters of different mesh sections, without having to separate it into individual pieces. In the example of Vid.11, the big section of the body has a more bouncy, flexible appearance, while the attachment areas of any limbs or other appandages, in this case the antenna-like ears on top, are stiffened to prevent them from becoming too weak to be held upright.

Vid.11: SB simulated Bunny Blob
https://youtu.be/WUl4bdsk364

[3] example: https://www.youtube.com/watch?v=E60ZkWbloRY

3.5 Godot Physics



Vid.12: Jolt vs GD physics parkour [30]
https://www.youtube.com/watch?v=2fP9uioaEaA

The initial research for this thesis encountered multiple Godot users heavily suggesting to not rely on the standard GD physics engine for SoftBodies, since it is infamous for not handling SBs particularly well, especially when it comes to collision handling. This is demonstrated well in Vid.12, which features an obstacle course for the SB sphere to traverse, in which the GD physics sphere repeatedly glitches through various colliders. Instead, the Godot community largely recommends using the open-source alternative Jolt physics. [31]
What started in 2014 as a personal learning project by Jorrit Rouwé, one of the Game Tech Programmers at the Guerrilla Games studio, kept being improved until it eventually turned into a very robust alternative for established physics engines. Rouwé [32] started this project to address the issue of threads being locked by one another when multiple of them try to affect the same physics object. Since physics engines tend to employ multithreading to accelerate their calculations, while the interaction inputs from player scripts are coming from a singular thread, it creates a bottleneck that wastes a lot of time every frame with threads waiting for other threads.

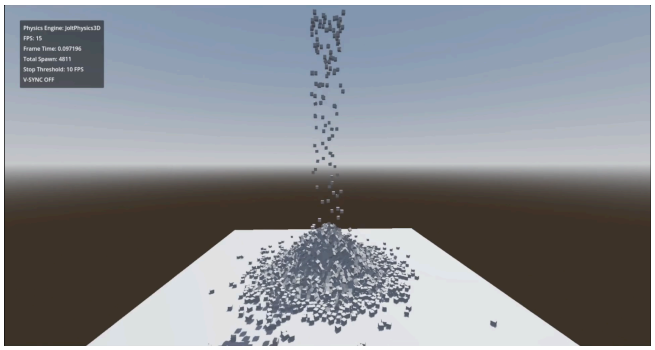
In 2021, Rouwé compared their hobby project to the commercial physics engine Guerilla employed at that time, finding that their engine actually performed better. Since then Jolt has been used as the primary physics engine for two major titles of the Guerilla studio, namely *Horizon Forbidden West* and *Death Stranding 2: On the Beach*, where it, according to official statements from the Guerilla studio, “saved memory, executable size and [...] double [the] simulation frequency while using less CPU time.” [33, para.2]

Jolt’s SoftBodies are based on the Extended Position-Based Dynamics (XPBD) approach. [34] The idea behind PBD [35] is using Verlet integration techniques to determine a point’s new position based solely on its previous position, along with any external accelerations influencing it. Instead of saving the velocity of each point, and modifying repeatedly to include acceleration, the velocity is newly determined at each frame through the difference in current and previous position. This way, all movements caused by spring constraints or collision will be automatically considered, leading to a smoother continuous motion.

However, PBD is known for having some issues with being too dependent on iterations and time steps, causing “constraints [to] become arbitrarily stiff as the iteration count increases, or as the time step decreases,” [36 p.49] which can cause especially huge problems when trying to simulate interaction between multiple materials of different stiffness, like mixing SB and RB objects. XPBD addresses these problems by implementing an improved new constraint formulation, although the underlying logic of the technique stays the same.

Before implementing its own physics engine, Godot used to base its 3D physics system on the Bullet physics engine. Over time, however, some issues with Bullet became apparent, amongst them being Bullet’s intended focus on physics simulations instead of game physics, resulting in different optimizations regarding performance and accuracy.
Therefore, with the release of Godot 4, the developers made the decision to shift to their own physics system, enabling them to build it custom suited exactly to Godot specific needs, while also being less dependent on external contributors for such an integral function of the game engine.
However, the results of GD Physics left many people disappointed, leading to several requests and petitions to either return Bullet or include Jolt. In a discussion on Github regarding the integration of Jolt Physics, Hugo Locurcio, one of Godot’s developers and maintainers, defended their previous decision, stating “we don’t want to rely on third parties to maintain a physics engine. If we do this, we are reliant on any decision they make (including stopping the maintenance of the engine, shifting the focus towards non-game use cases, etc).” [37, Sec.2]

Since then there have been many comparisons from the community, extensively testing the performance and reliability of jolt, providing many examples of its advantages, not only for SB simulation, but for all types of 3D physics. For example, Vid.13 demonstrates Jolt’s increased performance, with it being able to handle far more objects at far better framerates than GD Physics is capable of. Meanwhile, Vid.14 provides a great example of how a construct, built out of multiple RBs constrained together, behaves with far greater accuracy and stability with Jolt.



Vid.13: Jolt vs GD physics fps test [38]
https://www.youtube.com/watch?v=B3fEdhKw8mg

Vid.14: Jolt vs GD physics custom vehicle test [39]
https://www.reddit.com/r/godot/comments/1hkod6c/attempt_of_making_custom_vehicle_jolt_vs/

Due to the overwhelming demand from the users, a community developed extension, aiming to port Jolt to Godot quickly emerged, with Jolt creator Rouwé himself contributing to the project, massively speeding up the entire process, as well as improving the quality and trust in this integration, leading to the GD maintainers reconsidering their stand and officially endorsing Jolt physics since the release of GD4.4.

As of January 2026, with the release of GD4.6, Jolt now officially replaces GD physics as their default 3D physics engine. [40]

4 Implementation

To combine the aspect of SoftBody physics with procedural animations, different approaches were tested, to see what options already existed in the used programs, and what tools had to be custom created to fully suit the requirements.

Since this procedure included a lot of experimenting and switching between different focuses, note that the following chapter has been sorted into thematic sections, and does not necessarily recount the working process in chronological order. Hence there are instances where certain issues are being reiterated, to highlight additional consequences they caused, even though the underlying problem had already been fixed by concepts explained in earlier sections.

4.1 Blender SoftBody Animation

To begin, a deeper look into how Blender archives the combination of physics simulation and animation was taken, to see which concepts could be useful, and what could be recreated in Godot. Additionally, due to pre-existing practice with the associated tools in Blender, it allowed for quicker testing of new ideas and concepts, before committing to them further.

According to the Blender documentations, their SBs were “designed primarily for adding secondary motion to animation, like jiggle for body parts of a moving character.” [41, para.1] To combine the SB simulation with a keyframe animation, the SB includes a *‘Goal’* setting, which defines how much motion from an animation system gets applied, reaching from 1 = no simulation effect to 0 = no animation effect. Through the use of Vertex Groups, it is possible to assign each vertex its individual goal strength, applying the simulated secondary movement only to certain sections of the mesh, while having the main part of the body still controlled solely by animation.

Blender’s Cloth physics, on the other hand, do not have this function. While it is possible to simulate only part of a mesh, by using the pinning function to define a Vertex Group for all the mesh areas which are supposed to be left unaffected by the physics simulation, these pinned sections cannot be skinned to a rig. However, it is possible to parent the entire Cloth object to another object or bone, to move the pinned sections around.

To compare these two approaches, once again the character model from the Ragdoll chapter is being used. (pp.20-22) Since this model has a tail, the goal was to simulate the tail movement, during an otherwise pre-made walking animation. For the SB version, Vertex Groups give most of the mesh a goal setting of 1, while leaving the tail portion to be affected solely by physics. (Vid.15) However, due to the complexity of the mesh, the performance got so severely impacted that it caused Blender to lag intensely until it eventually crashed.

Vid.15: SB simulated Tail
https://www.youtube.com/watch?v=73KiwYkiWII

The same approach has been applied to the Cloth simulation, pinning down most of the body. While the results are far smoother (Vid.16) and rendered faster, they are not a viable solution either, due to the pinned section being unable to follow the rig deformation.

Vid.16: Cloth simulated Tail
https://www.youtube.com/watch?v=hasdvqUq0PM

To work around the Cloth pinning restrictions, once again proxy objects are employed to run the physics simulations, similarly to what has been done before with the Ragdoll. For the tail this required only a simple cylinder, with enough subdivisions to allow for deformation, the top of which got pinned to the tail base bone of the rig, making it follow the general movement of the hips. Applying a Surface Deform modifier allowed the tail section of the original mesh to follow the simulated movement of the proxy object, without having to apply any physics simulation to the character mesh itself.

Vid.17: SB animated Tail
https://www.youtube.com/watch?v=S6diu83oDdw

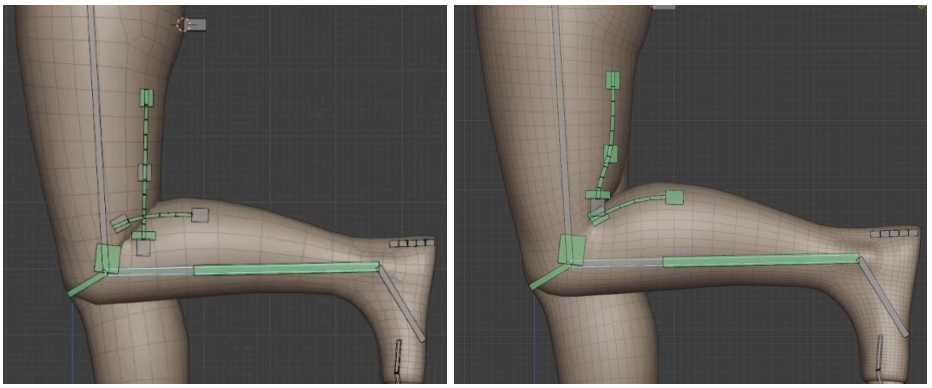
Vid.18: Cloth proxy animated Tail
https://www.youtube.com/watch?v=Yp_ILnIJvx8

Vid.17 is the result of the regular SB simulation, which due to the long baking time and the multiple crashes, barely even sped up the animation process compared to manual tail bone keyframing. Additionally, the results are very unnatural, resembling more the behavior of a rubber rope than a tail. While the movements of the proxy Cloth solution (Vid.18) look better than the SB variant, the results are still quite dissatisfying. Since the proxy cylinder cannot be allowed to collide with the body, without also interfering with the tail portion of it, there are some frames where the tail slightly clips into the legs during the walking animation.

A completely different approach to create soft deformations, which does not rely on physical simulation, can be seen in the work of Rakenval [42], who, besides creating original stopmotion animations, also uploads videos revealing huge parts of their working processes.

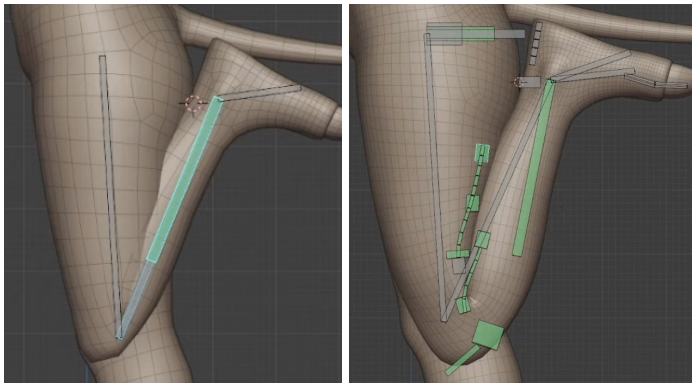
The rigs they create are far more advanced, imitating not only the skeleton of a creature, but also using several additional bones, to recreate the effect of muscles, body fat, etc. This can be seen especially well in one of their timelapse videos showing their rigging process. [42] Using the area of the knee as an example, Rakenval added multiple sets of bendy bones to the backside of the leg, to be able to control the way the fat there is affected by movement and collision during the bending of the knee. The left part of Img.22 shows the unaltered knee deformations, driven just by the regular rotation of the lower leg bone upwards. The two sections behind the knee are clipping into each other, resulting in a very unnatural crease to form. Meanwhile, the right section of Img.22 is the adjusted result, in which the BBones on the back of the leg are re-adjusted to simulate them colliding with one another and getting squished together. This adjusted position is then coupled to the orientation of the lower leg bone. This

way, whenever the leg gets bent backwards, the adjusted pose is automatically triggered, as can be seen in their timelapse video at the 3:08 minute mark.



Img.22: regular knee deformation (left) vs. adjusted deformation (right).
Adapted from [42]. Visible at timestamp 3:08.
https://youtu.be/EPiHGtRFCR0?si=TYLQEs5BJ0oWcJVi&t=188

When bending the knee backwards even further, the unaltered deformations are even worse. (Img.23.A) Half of the lower leg section clips into the upper thigh, resulting in the leg appearing to have far less volume than it is supposed to have. To correct this, first Rakenval alters the way the knee itself bends. Instead of simply rotating the lower leg around the central knee joint, the lower leg needs to move in a much larger arch to lay neatly next to the upper leg. For that, they created a secondary bone, which generally follows the primary lower leg bone, but can be manually moved outwards to accommodate for the extra space the knee requires. Then, the two BBone lines on the backside of the leg get adjusted again to lay neatly next to each other. This adjustment pose is also saved, for the affected bones to automatically move towards this position whenever the main bone exceeds a rotation of over 90° backwards. The final result can be seen in their timelapse video at the 3:52 minute mark, showing the leg being able to bend back and forth by just posing the main bone, with all adjustment bones automatically assuming their set positions, making it appear as if the meat on the bones actually interacts with itself and deforms softly.

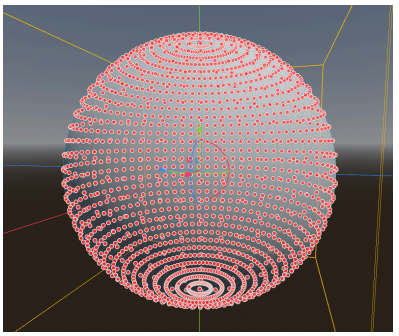


Img.23: regular knee deformation (left) vs. adjusted deformation (right).
Adapted from [42]. Visible at timestamp 3:52.
https://youtu.be/EPiHGtRFCR0?si=aVdLO2igkTS4ONly&t=232

Even though this method requires a lot of extra time to create the rig, it results in the most detailed and customizable results, which are therefore a favored method for creating animation films in Blender. However, for use in external programs, this setup is not suited, since a lot of the necessary tools, like BBones or various bone constraints, do not have exact equivalents in other programs. Thus these complex bone setups cannot be imported into any typical game engine. While it is possible to use such an advanced rig to create game animations in Blender, and only import the finished result into a game engine, these animations would not be combinable with most procedural practices, without manually recreating the functionality of these Blender specific bone constraints in the game engine as well.

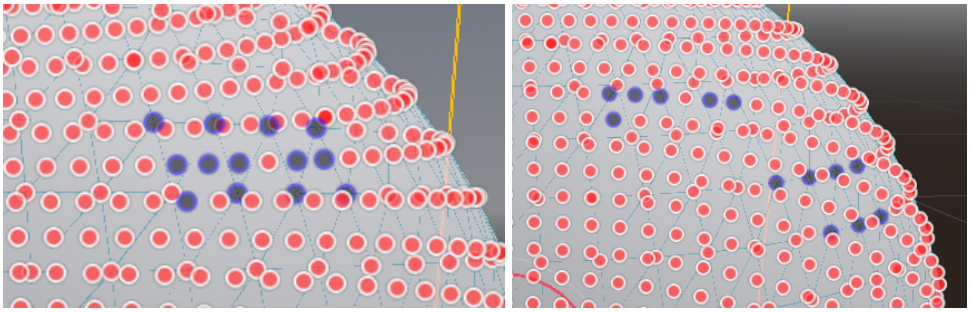
4.2 Godot SoftBody Animation

Just like Blender, Godot also includes a feature which attaches a SB to a rig or another object, to make it follow its movement. Unfortunately though, this tool is so complicated and tedious to use, that it could be described as unusable for any even slightly more complex case.

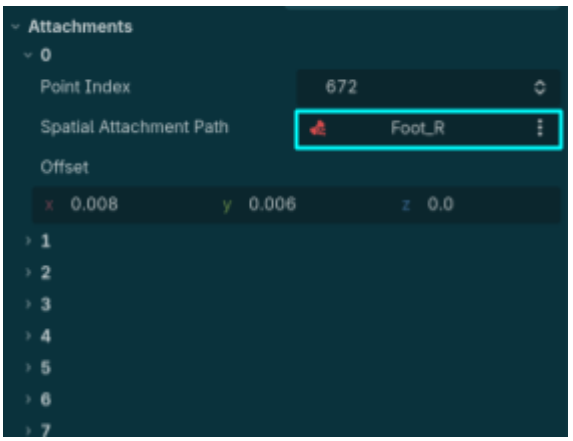


Img.24: Godot SB pinning menu on a sphere mesh

The way the setup works in Godot is by clicking on each individual vertex which needs to be pinned, without any option to select multiple vertices at once, neither by weight painting nor a selection area nor any other means. This approach alone can already result in hundreds of points having to be manually selected, depending on the resolution of the mesh. Additionally though, this procedure is complicated even further by the mesh view constantly showing all existing vertices of both the font and backside (Img.24), with no filtering or culling view available. As Img.25 demonstrates, differentiating between the vertices is very unclear, easily leading to pinning a lot of vertices on the wrong side.



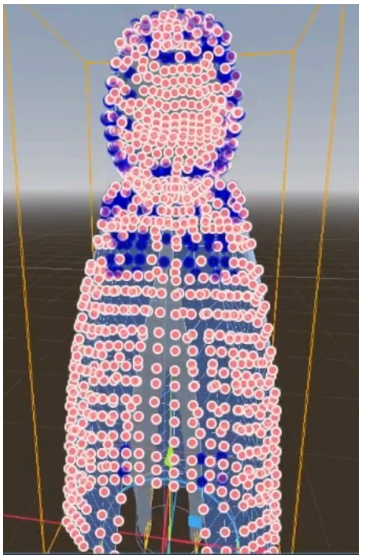
Img.25: pinned vertices on a mesh (left) revealing to be incorrect by changed viewing angle (right)



Img.26: SB object property tab, specifying to which bone (highlighted) the vertex is pinned

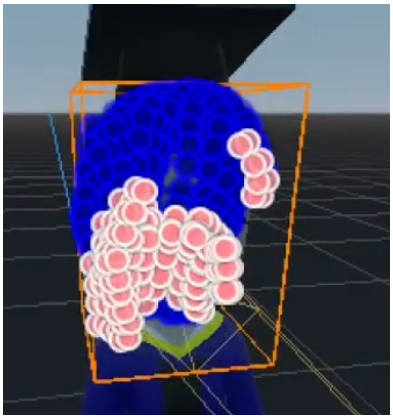
Additionally, clicking on the vertices only pins them to one main bone, which has been defined as the SoftBody's root object. If vertices of separate sections of the mesh need to be pinned to individual bones, as is required for skinning a model to a complete rig, every single one of the pinned vertices needs to have its connected bone specified separately in the object property tab.

While this tool seemed unusable to me, Savallion [43] found a way to make it work for their purposes. They successfully connect a SB cloak to their character rig to have it mimic collision with the main body, but still fall and flow naturally, which they achieved by only pinning as few vertices as possible in strategic places, in which the body otherwise clipped through the cloak. (Img.27) But even with the lower polycount and simplified geometry, this procedure took over 200 vertices which needed manual pinning. While Savallion proved that with enough dedication this feature can be used to combine SoftBodies with animations in Godot, they also explicitly stated how tedious and unintuitive this procedure was, and that they definitely think this system needs a much improved user interface that automates several substeps, before it can actually be used effectively.



Img.27: cloak mesh with scattered pinned vertices [43]

In a video by 최성수 (Choi Seong-su) [44] they show a workaround they discovered, which allows for pinning of several vertices at once. Using a character's hair as an example, they edited the mesh data, to manually reorder the vertex indices, so the vertices are sorted by pinned and unpinned, with all the pinned ones listed first. To add these indices to the SB's object properties, they wrote a temporary script, which simply prints out all consecutive numbers from 0 to the amount of pinned vertices, which can then be copy-pasted into the object data code for the SoftBody. The results are the top half of the hair objects being pinned to the head, with the lower half being simulated. (Img.28)



Img.28: pinned upper half of hair mesh [44]

While this technique is a better alternative for situations that require many vertices to be pinned, it still has several issues. Due to Godot not allowing the pinning influence to be weighted, there is a hard border between the pinned and the simulated vertices, as can be seen in Img.29.a. Additionally, the hair has no collision with the head object, it occasionally clips through the head mesh. (Img.29.b)

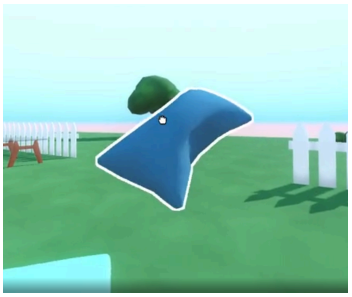
(a)

Furthermore, since this method is only able to automate the vertices getting added to the pinned array, without any further clarification to which bones they are pinned, it is only possible to pin them to one shared object. Should multiple pin targets be required, they would need to be manually added again. Finally, this technique exclusively works with triangulized meshes, since importing a mesh with quad faces causes Godot to convert the mesh in a way that reorders the vertex indices, making this approach impossible.

(b)

Img.29: results of GD SB pinning [44]

In summary, it can be said that the GD SoftBody pinning tool is still too unrefined to be considered a usable option. Even if someone is willing to submit to the tedious process of pinning each vertex by hand, the end result still restricts the outer skin of the object to the pinned bone, which would transform the pinned areas into rigid sections, impacting the softness of the mesh. Instead, the aim is to create a sort of ‘active’ SoftBody, that is still completely soft and deformable on the outside, but can be moved around the scene as a whole, without dragging it by individual areas.



Vid.19: active SB demo [45] https://www.reddit.com/r/godot/comments/14kdqzx/active_soft_bodies_in_my_game_d/

Searching for other approaches to use the provided tools, revealed a demo clip by the user abrasivetroop in a GD forum, that showed exactly the desired behavior. [45] In the discussion of this post, they explained briefly how their SB setup works, stating “I really didnt do much its just a softbody attached to rigidbody and pick up is my own code which I have made a tutorial about and its on my Youtube.” [45] Unfortunately though, trying to find this video remained unsuccessful. Neither searching for the user’s name, nor the title of the game they worked on, yielded any results on Youtube. While a comment on an older post of them included a link directly to their youtube channel, it appears that it has since been irrecoverably deleted.

Trying to recreate the SB behavior, based on the limited description available, unfortunately, also remained unsuccessful. Parenting the entire SB to the RigidBody caused the object to lose their softness, while only pinning some vertices to the RB affected the shape too significantly, as these points would then become rigid.

Eventually, the decision to abandon this lead had to be made, and instead to re-focus on writing a custom SoftBody physics class.

4.3 Custom SoftBody

To recreate the behavior of SoftBodies, first, an effective way to manipulate the positions of individual vertices had to be found. The first idea was to use a vertex shader, which seemed like an easy and performance optimized way to displace individual points of a mesh. After following the tutorial *'Vertex displacement with shaders*' from the GD documentations, [46] to gain an understanding of how shader displacement works, one major problem with this approach became apparent: Since shaders process multiple vertices in parallel to speed up the process, it is not possible to access the properties of neighboring vertices, something that was definitely required to determine the distance between vertices, making it unfitting for these purposes.

Godot Mesh Types

The next approach was looking into the different mesh types Godot offers, to see which ones can be easily manipulated from a script. GD's *Primitive Mesh* includes presets for simple shapes, e.g., cubes, spheres or cylinders, with a set of customizable parameters to configure aspects, such as height or radius, which makes it possible to transform these base shapes to, for example, turn a cylinder into a cone or a cube into a rectangular cuboid. While this mesh type can be great to quickly create placeholder or debug objects without having to import any additional mesh data, it offers a too limited range of functions for the intended purposes. [47]

The *Array Mesh* type takes an array of vertex information and assembles it into a complete 3D object. This type is suitable for more complex geometry and is Godot's standard to display imported mesh data. It works by simply listing the 3D position of the vertices in sets of three to make up the triangle faces. This also means that whenever a vertex is part of multiple faces, it needs to be listed multiple times in the array. [48] Since Array Meshes automatically take over functions like, e.g., calculating the normal of faces, the order in which the vertices of a face are being listed is important. Otherwise the surface will face towards the inside of the mesh. Additionally, it is possible to combine the use of an Array Mesh with GD's Mesh Data Tool or Surface Tools, both of which provide different information about the mesh and can be used to alter meshes dynamically. While this combination could potentially be used to apply SoftBody deformations to the mesh, it is not optimized for being called on every frame, which is something a continuous physics simulation would need. Instead, it is more intended to create

permanent deformations to a mesh, which can even be saved as a new mesh resource file, to be reused throughout the project.

The *Immediate Mesh* on the other hand, is optimized specifically for simpler geometry which need to be redrawn frequently, by using a technique similar to OpenGL to directly draw the mesh onto the screen. [49] Additionally, it includes an interface similar to the Surface Tool, providing additional access points for custom normals, vertex color, etc. However, since it does not rely on the underlying Array Mesh structure, it is not limited by performance issues that come with frequent redrawing.

This made the Immediate Mesh the perfect choice for this project's intentions. The only potential disadvantage of this mesh type is that the documentation explicitly states that "generating complex geometry (several thousand vertices) with this tool is inefficient." [49, para.2] However, since creating SoftBody physics with such a high polycount would likely cause far worse performance issues than inefficient drawing optimization, such complex meshes are excluded from the scope of this thesis regardless.

Mesh Manipulation

Since Godot lacks 3D modeling functions, a way to read in the mesh data of objects that had previously been created in Blender had to be found to import them into Godot. While it would have been possible to first pass the file into the Immediate Mesh unaltered, as part of the setup of the scene, and then retrieve the information about positions and connections of the individual vertices from there, Godot converts meshes to be more native to the way it handles it, which includes slightly altering the original topology of the mesh to triangulate all faces, which would make intentionally modelling objects to be optimized for soft deformations much harder.

Instead, the decision was made to read in the mesh data directly from the imported files, and afterwards pass the information on to the immediate Mesh. Since multiple common 3D model file types exist, which all save the mesh data in slightly different formats, the scope of this project was limited to only work with .obj files, which are text-based files, that list all base information of the model, making it easy to read them in and parse the information into custom data holding classes.

```
public class Vertex
{
    public Vector3 Position;
    public Vector3 PrevPos;
    public readonly Dictionary<Vertex, float> Neighbors = new ();
    public readonly Dictionary<Vertex, float> Shear = new();
```

Img.30: Vertex class properties

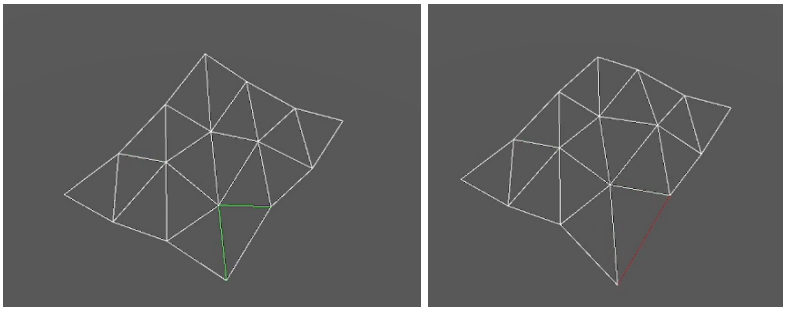
```
public class SoftMesh
{
    public readonly string MeshName;
    public readonly List<Vertex> Verts;
    public readonly List<int[]> Faces;
```

Img.31: SoftMesh class properties

All vertices of the original mesh are converted into instances of the custom '*Vertex'* class, (Img.30) which stores not only the current position of the point, but also where it has been in the frame before; an information that is needed to calculate its continuous movement. Additionally, it includes the different types of connection to other vertices, separated into neighbors, which together form the edges of the model, and shears, needed to simulate the springs inside each face as described by Provot. [21] A dictionary structure stores these connections, since it allows to additionally associate the initial rest length of each of these springs, which is needed to detect any changes in the springs' lengths.

The '*SoftMesh'* data storage class (Img.31) includes a list with all Vertex instances, as well as a list of faces, which will reference the vertices it is consisting of by index. Additionally this class includes a field for a mesh name, which helps differentiating between multiple mesh instances in the scene tree.

The Immediate Mesh type has an additional drawing mode to construct a mesh as just lines, used for wireframe displays, which made it possible to create a debug view, which only shows the connected springs. This became especially helpful later on, after the amount of spring types per model had been increased, and allowed to visually verify their correct placement and behavior inside the mesh. Additionally, by pushing individual vertex colors to the mesh, it was not only possible to differentiate easily between the connection types, but also to color code them dynamically in regards to their length, which made it easy to see excessive squash or stretch of any springs. (see Img.32)



Img.32 two adjacent squashed springs (green) extend in the next frame, causing the spring in between to overstretch (red)

Cloth Simulation

Before tackling the implementation of full SoftBodies, the first test runs were done with a simple falling piece of Cloth. Since this model needed no volume preservation or reinforced surface stability, this allowed a focus on implementing the basics of the spring behavior, as well as identifying some compatibility issues with Godot early on.

One of those challenges was that GD's face drawing order seems to be opposite from Blender's, which caused some confusion and trial and error to find the correct order in which to read in the vertices, push them to the Immediate Mesh, and use them for the calculation of the face's normal vector.

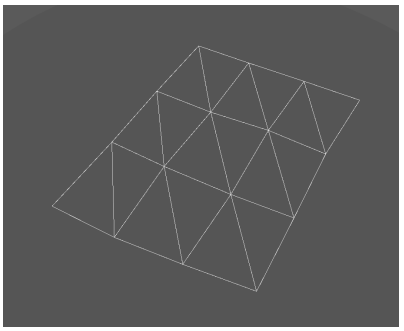
```
var a:Vector3 = ToLocal(softMesh.ExternalVerts[face[0]].position);
var b:Vector3 = ToLocal(softMesh.ExternalVerts[face[1]].position);
var c:Vector3 = ToLocal(softMesh.ExternalVerts[face[2]].position);

var n:Vector3 = (c-b).Cross(a-b);

[...]

mesh.SurfaceAddVertex(c);
mesh.SurfaceAddVertex(b);
mesh.SurfaceAddVertex(a);
```

Img.33: code excerpt of vertices drawing order

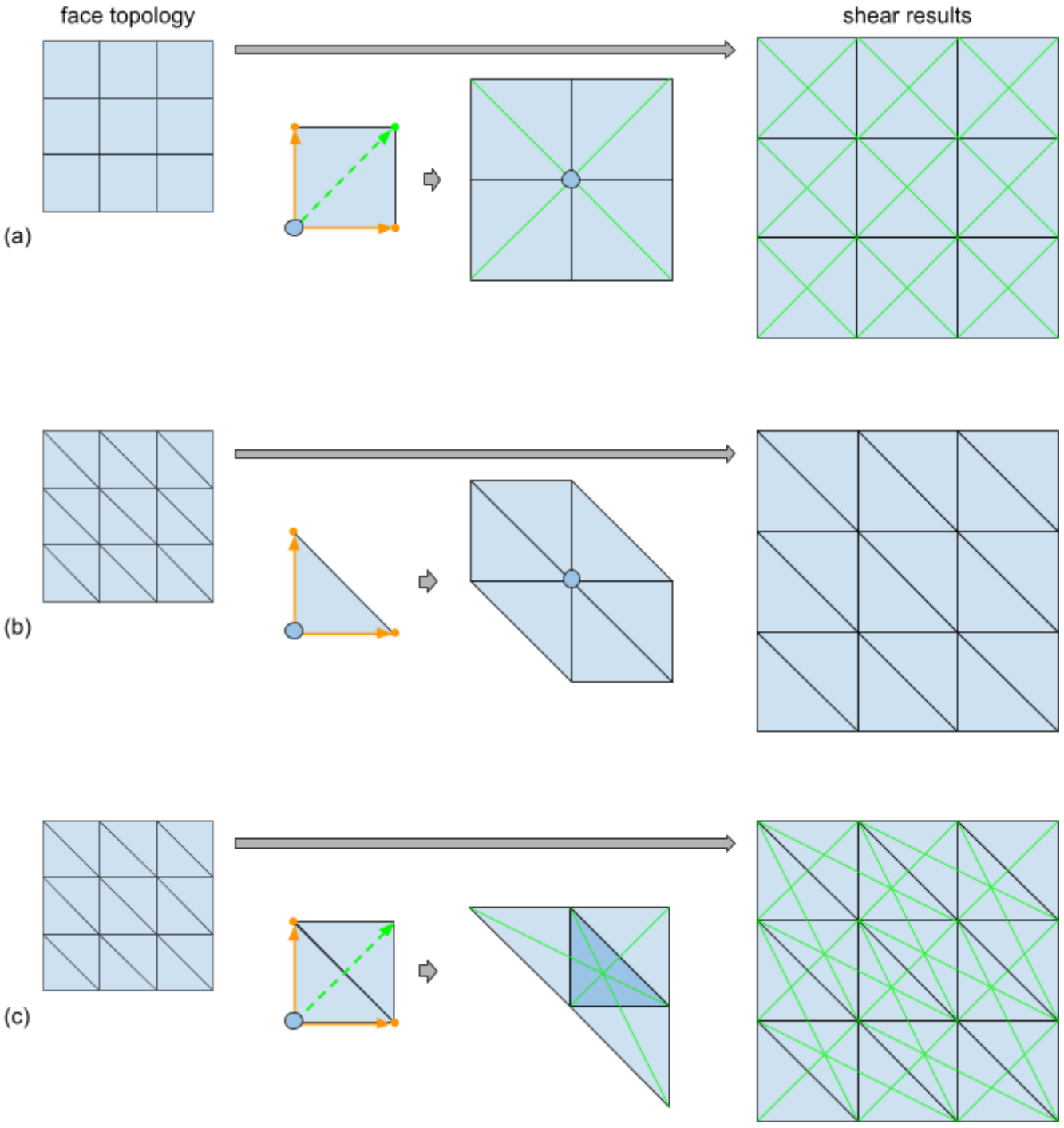
Another issue was that Godot exclusively constructs surfaces from triangles. Therefore, initially a method was implemented to detect all quad faces during the read-in phase of the mesh data, and immediately convert them into two triangles instead.

After the further expansion of the spring setup, however, some issues became apparent, caused by the triangulized mesh. With quads it is possible to simply connect all vertices of a face, which are not connected by edges already, to react with the additional shear springs. However, as mentioned before, (pp.25-26) this is not possible with triangles, since in those, all possible vertex connections are already present. Hence, it is not possible to recreate the same shear spring setup for added surface strength. (Img.35.b)

While tris act like a modified version of shear springs, due to them only being orientated in one direction, it noticeably influences the stretch direction of the final Cloth. (Img.34) While this might be useful for specific cases, in which a realistic portrayal of fabric types that does show a difference in stretch depending on the axis is the goal, it is not applicable for this project.



Img.34: horizontal stretch stronger than vertical stretch due to orientation of triangles



Img.35: different methods for shear spring placement
a) connecting each vertex of a **quad** to all other vertices of the **same** face
b) connecting each vertex of a **tris** to all other vertices of the **same** face
c) connecting each vertex of a **tris** to all other vertices of a **neighboring** face

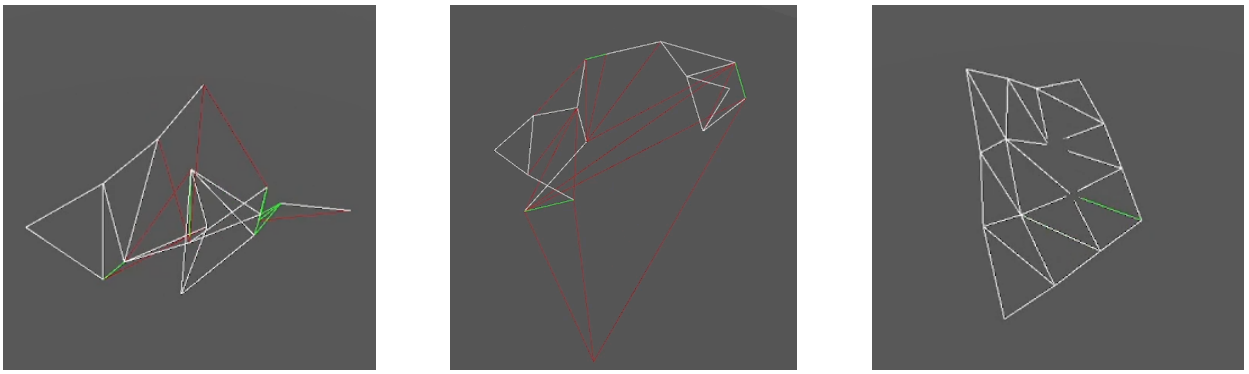
Using a different approach and looking at face-pairs instead to try to place the springs in a way that resembles the quad-based shears, results in an abundant amount of shears. (Img.35.c) Additionally, this triangle setup would cause more problems down the line, once the flexion springs were implemented. Since those connect each vertex to their second neighbor, the increase from four direct neighbors with quads, to six neighbors with tris, would lead to far more flexion springs being generated for the same surface area, which would unnecessarily increase the calculation time due to the surplus of springs.

Therefore, the decision had been made to re-write the code to read in and calculate the meshes with the original quad setup, and only separate them into triangles when pushing them to the Immediate Mesh. Additionally, extra handling for any tris that exist in the original mesh had been added. Since a few limited triangle faces are unlikely to impact the stability of the mesh noticeably, a simple warning is being printed whenever one is encountered, while they are otherwise ignored for the shear spring generating step. However, the handling of n-gons has been completely avoided, since finding a universal approach to stabilizing and drawing them would have required entirely different strategies. Thus, should an n-gon be encountered, an error message is being logged and the construction process of the SoftMesh is terminated.

After the base mesh is successfully constructed, individual vertices need to be simulated. This process has been divided into three phases:

1. External Forces.
   For the scope of this thesis, this is limited to a constant pull downwards, to imitate gravity, but it could be expanded to include additional influences, like wind, fairly easily. In this step, each vertex gets treated as its own, distinct object that gets affected by these external forces, without any regard to the structure it is part of.
2. Spring Constraints.
   In this step, the vertices will be adjusted to keep their initial distance from one another, to preserve the original shape of the object. To allow for slight deformations, springs are not restricted to their exact rest length but instead are gradually moved towards it. This is achieved by moving the vertex not only by the difference to the rest length, but additionally multiplying it by an adjustable spring stiffness value, which during tests achieved the best results with a value between 0.1 and 0.5, depending how stiff the body is supposed to behave. Furthermore, it was attempted to make this spring value dynamic, scaled by how

strongly the spring in question has been removed from its rest length, to allow an even stronger pull back towards its rest state when stretched out excessively, emulating the behavior described by Hooke's Law. (p.24) Unfortunately though, this led to wildly unstable results and thus had to be reverted back to only using a spring stiffness constant.



(a) springs clumping together (b) springs extending infinitely (c) correctly falling sheet

Vid.20: Cloth Simulation, physics update time scale intentionally reduced for test purposes
https://youtu.be/S5WO8fiaD4Q

3. Collision Handling.

   As with the External Forces step, this is a simplified process in the current implementation, which exclusively registers a Y-coordinate of below 0 as a collision and moves the vertices upwards accordingly, to create a ground plane for the meshes to land on. It was contemplated whether this step is best done before or after the spring connections are evaluated, since on one hand, it makes sense to first apply all forces that treat vertices as individuals before applying the springs' connections, otherwise a collision could potentially lead to unnaturally excessive deformations. On the other hand, expanding springs could create new collisions, which in turn could potentially lead to slightly overlapping objects on every frame.

   To prevent both, a recursive system would need to be implemented that checks for both collision avoidance and spring constraints to be satisfied repeatedly, until both are true or below a certain margin of error. For the scope of this thesis, however, this step was simplified by adding fixed collision checks before and after the spring step. With that said, it should generally be preferred to put these as a final step, since overlapping objects, even if it is just marginally, are far easier to see in the final result than slightly over- or understretched SB springs.

To calculate the vertex movement, first an explicit euler method has been implemented, which continuously adds on to the old values of position and velocity (Img.36.a), with gravity as the only source of acceleration, and any additional collision or spring adjustment added separately. This calculation, however, is fairly easy to lose stability, which further contributed to the failing simulations in Vid.20.

Instead, the implementation had been switched to an Verlet Integration approach, which implicitly calculates the velocity solely through a difference in position from the previous frame. (Img.36.c) The advantages of this are that the velocity also includes any movement that the collision or spring constraints add to the vertex, leading to a smoother calculation of their continuous movement.

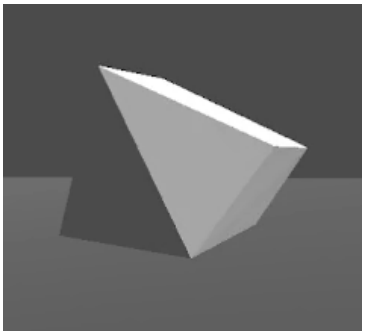
```
    newVelo = velocity + acceleration
a)
    newPos = pos + newVelo

    velocity = pos - prevPos
b)
    newPos = pos + velocity + acceleration

c)  newPos = 2*pos - prevPos + acceleration
```

Img.36 increasingly implicit approach to calculating the new vertex position (pos)

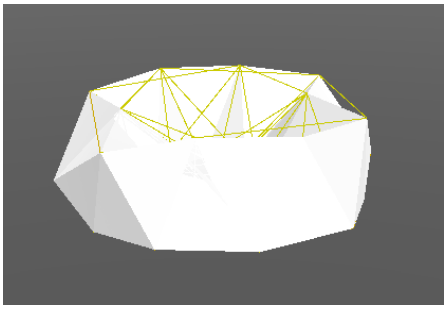
The result is a believable Cloth simulation, which is able to interact with a collision ground plane, and which is individually configurable due to the adjustable spring stiffness.

3-Dimensional Structures

Since the implemented spring setup had been heavily based on the work of Provot, [21] first attempts to achieve 3-dimensional structure in an object were done by heavily constraining the flexion springs, which connect each vertex to its second neighbor. The success of this approach, however, varied greatly with the geometry of the mesh: for cubes, which consist of big faces set at sharp angles to one another, the flexion springs form a rather stable, internal structure that connects corners of opposing sides of the model. While the resulting object still behaves rather strange and unnatural, as can be seen in Vid.21, the box is mostly able to keep its shape, or return to it if deformed.



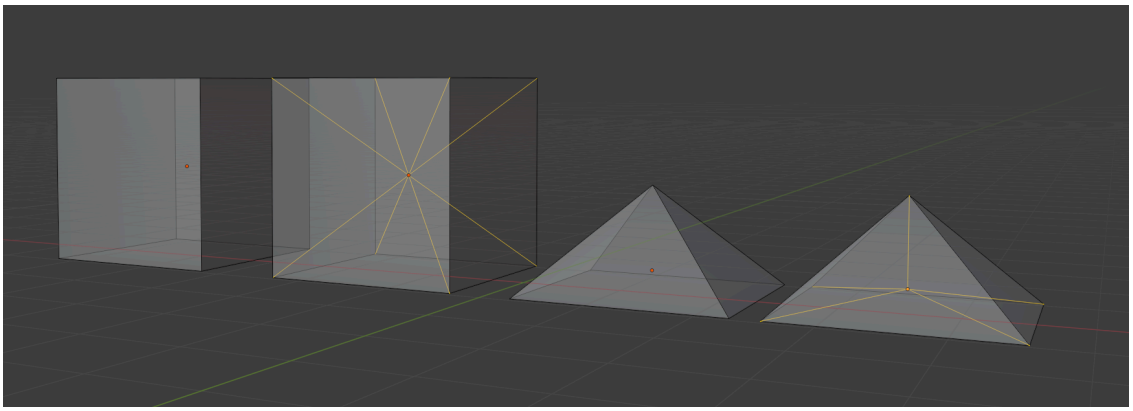
Vid.21: first SoftCube attempt
https://youtu.be/asslWx7tenE



Img.37: collapsed sphere mesh

The same algorithm applied to a lowpoly ball, however, causes the model to deflate and collapse into itself. (Img.37) Due to the smaller faces, the flexion springs act less like internal support beams and more like a reinforced skin, which helps to keep some of its shape, but still allows the mesh to collapse.

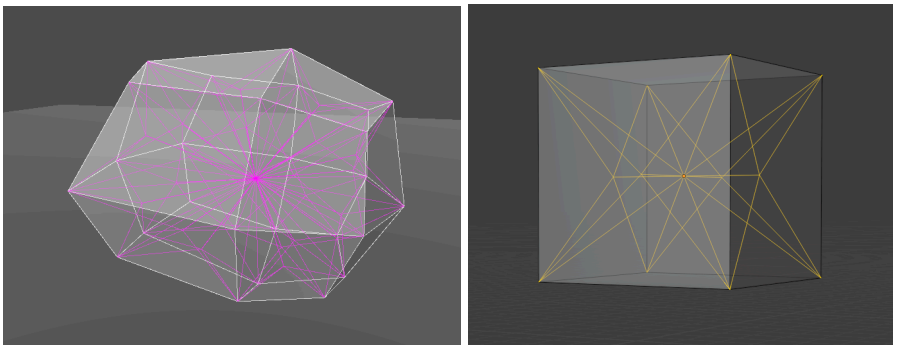
Therefore, instead of relying on flexion springs, a new, more robust way to place internal support springs has been created, that works regardless of the surface geometry. The developed method repeatedly subdivides the volume into smaller sections, each of which can then get interconnected. This process has been visualized below in Img.38 on the example of a simple cube model.



Img.38: (a) Base Mesh; (b) Subdivided Mesh; (c) Fragment Section; (d) Subdivided Fragment

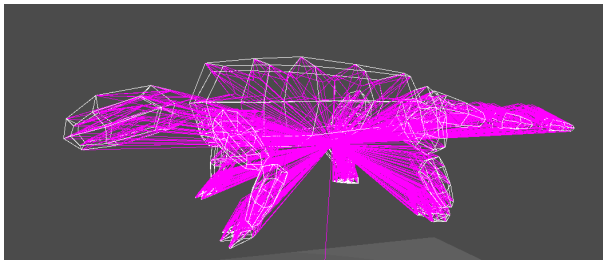
First, the center point of the mesh is determined, by calculating the average position of all vertex coordinates. This new point then gets connected to all other points, which make up its position. (Img.38.b) The resulting subdivision creates a set of pyramids, which each consist of one of the original mesh faces as the base and the center point as the tip. (Img.38.c) From there on out, the new volumes could get recursively divided again in the same way if needed, until enough stability is achieved.

While this procedure does not necessarily place the internal springs in the most strategically advantageous way for each unique mesh shape, it is universally applicable, with the same algorithm producing a working scaffolding for a variety of shapes, as can be seen in Img.39 for a cube and low-poly sphere.



Img.39: same internal structure algorithm applied to lowpoly sphere (left) and cube (right)

However, while this method works better than the ones previously tried, this approach still poses some issues when facing complex geometry. While a high face count inadvertently leads to an excess of internal springs, which in turn impacts performance, they still remain decently stable. However, a character mesh, which includes separate limbs, is going to be concave, leading to the center point being outside of the mesh, similarly as can be seen in Img.40, thus leading to higher inconsistency in stability with growing distance from the surfaces to the center.



Img.40: concave character mesh with central point located on the outside

Instead of trying to develop an even more complex approach for internal mesh support, which likely would have impacted the performance even more, inspiration was taken from the way ragdoll physics are created. Instead of simulating the entire body as one physics object, it gets divided into multiple, connected parts, each of which are far easier and faster to simulate. Unlike with ragdolls, however, the bones of the original rig would not follow the simplified body sections, but instead a complete character mesh would be laid over the SoftMesh objects, which vertices would then follow their closest SoftMesh vertex exactly. While it had been apparent that, due to time constraints, this concept was unlikely to be fully implemented, the separately-simulated-sections approach has been further pursued from this point on.

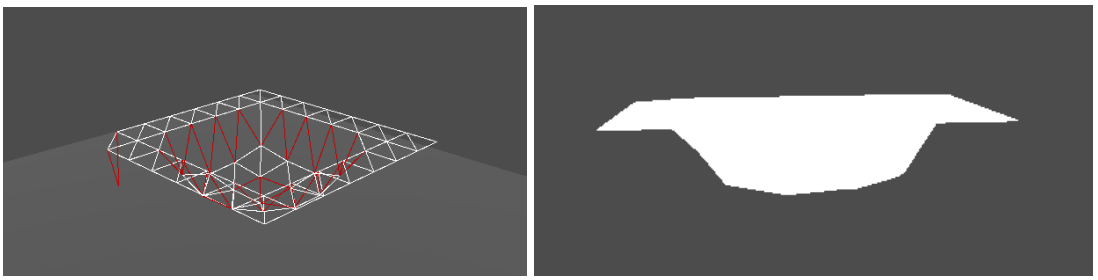
4.4 Rig Attachment

Vertex Pinning

As previously detailed, to attach a mesh to a rig, each vertex is assigned to one or more bones, which movements it then follows. (p.5) In Blender, this data is organized in a structure called Vertex Groups, which creates a separate group for every bone, and then saves a weight value for each vertex.

The initial idea for skinning the custom SoftMeshes to a rig was to use the bone deformation as a baseline and then apply the soft deformation on top, which would require knowing which vertices are pinned to which bones exactly. However, since .obj files are only used for exporting 3D meshes without additional rig and animation data, they are unable to export the Vertex Groups. The only comparable feature which they do include, however, are *Face Groups.* These function similarly to Vertex Groups, except that they assign the weight value per face, instead of per vertex. Since physics implementation does not simulate faces by themselves, but only the vertices they are made out of, this data was converted by pinning every vertex that belongs to at least one face that is part of a pinned Face Group.

In the example in Img.41, a plane with the outer ring of faces added to a pinned group was imported. For test purposes, these faces were not attached to a rig or any other structure yet, but instead for each pinned vertex the position calculation simply got skipped, keeping them statically in the place they were initialized in. Additionally,the spring constraint handler was modified, to detect whether one of the spring sides is pinned, and thus moves only the other side towards it, instead of distributing the movement on both vertices equally.



Img.41: Cloth mesh with pinned faces

However, the results of this pinning method were not as expected. As can be seen in Img.41, due to the entire faces getting pinned, the affected areas become unnaturally stiff with a very harsh transition towards the simulated sections of the mesh. This results in the sheet looking

less like its edges have been attached to something, and more like a piece of fabric, covering up a hole on an otherwise flat surface it is resting on.

While other file types do include Vertex Group data, switching file types would have required changing the way the mesh data is read in and converted completely. Alternatively, a custom Blender addon was briefly considered, that saves the mesh data into a text based file, imitating the .obj exports format, but that additionally includes the Vertex Group data.

In the end, both of these ideas were kept as back-up plans, and instead, a different approach was pursued.

Internal Pinning

Since pinning part of the outside mesh did not work like envisioned, the idea was to instead only pin the vertices inside the model. Since several new internal points are added during the internal subdivision step of the mesh initialization, there had to be no additional data imported to differentiate between pinned and free vertices. At this point, the decision to separate the full model into separately simulated sections had already been made, therefore, each body section would correspond to exactly one bone of the rig, making it possible to attach all pinned points of a mesh to the same object, without any additional pinning differentiation needed.

Another advantage of this approach was that this would also ensure that the mesh would move from the inside out, with the internal vertices acting as the ‘meat’, connecting the skin to the bones, and dragging it along with it.

The SoftMesh’s previous ‘*Verts*’ list got separated into ‘*ExternalVerts*’ and ‘*InternalVerts*’, making an easy separation of which points have to be simulated with SB physics and which points need to follow the transform of the connected bone.

```
public class Vertex
{
    public Vector3 Position;
    public Vector3 PrevPos;
    public readonly Dictionary<Vertex, float> Neighbors;
    public readonly Dictionary<Vertex, float> Shear;
    public readonly Dictionary<Vertex, float> Structure;
    public bool Pin;
```

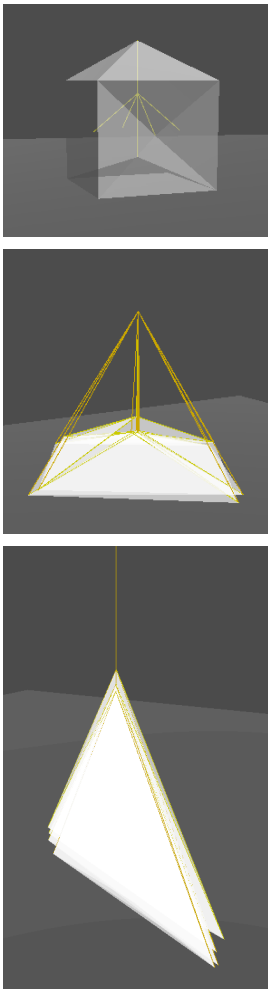
Img.42: extended Vertex class properties

```
public class SoftMesh
{
    public readonly string MeshName;
    public readonly List<Vertex> ExternalVerts;
    public readonly List<Vertex> InternalVerts;
    public readonly List<int[]> Faces;
```

Img.43: extended SoftMesh class properties

The first test runs, regarding the internal structure, were done without a connected rig yet, to see if the mesh would be able to keep the shape of the outer shell while suspended in the air.

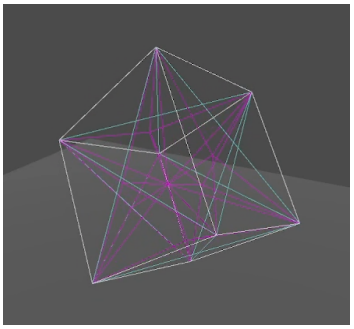
At this point, the spring setup was not fully finalized yet, so the outer springs (neighbors and shears) were still being split to create more access points for the internal structure springs to attach to. However, this resulted in the internal springs being able to be pushed too far out of the mesh, and the object folding into itself. While this posed only minor issues during the falling-on-ground tests, it completely collapsed the meshes during the hanging tests, due to the internal vertices not being able to move along. Thus, the entire internal structure setup had to be re-worked again, before finally reaching the process explained earlier. (see p.53)



Img.44: failed pinning attempts

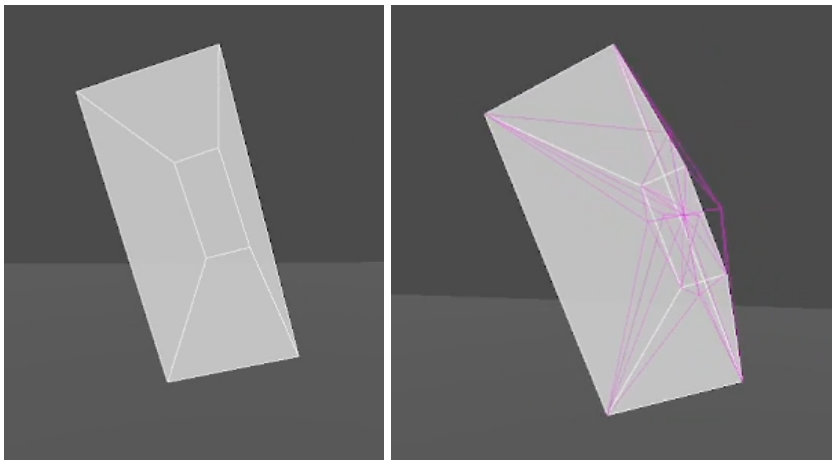
With the improved internal structure, and the air suspensions test being successful, next the pinned vertex handling had to be expanded to follow the movements of an attached root object. Unfortunately though, since the vertex movement calculation was written using *local* positions, the entire mesh followed the parent object rigidly, without any of the SoftBody jiggle. Thus, the entire class had to be re-worked to work with global positions instead.

To make the internal vertices follow the bone, the mesh class additionally saves the bone's global position from the previous frame, to be able to accurately determine the change in location, and apply the exact same movement to all pinned vertices.



Vid.22: shaking cube
https://youtu.be/HBSxtQ-USHc

While this resulted in the internal structure following the bone correctly, the outer vertices started to shake frantically, (Vid.22) due to the implicitly calculated vertex movement causing overshoot of the intended target position. To combat this, a fourth step was added to the SoftMesh simulation: after applying external forces, spring constraints and collisions, an additional damping step needed to happen. A simple damping method determines the entire movement that was calculated for this frame, and reduces that value slightly before applying it to the actual vertex, to help the system settle. While this damping factor is also adjustable as one of the SoftMesh parameters, a value of around 10% dampening delivered the best results during testing.

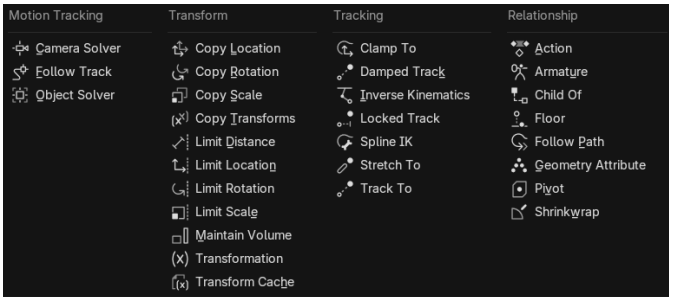


Vid.23: successfully jiggly trapezoid simulation, with and without internal spring overlay
https://youtu.be/IxhuARBK81U

4.5 Walk Automation

While there are many different possibilities of applying procedural elements to an animated character, for the scope of this thesis, a focus on an automatic waking animation has been chosen. The goal was to only control the forwards/backwards movement and turning using the WASD keys, while the body reacts to these inputs fully automatically, without the use of a keyframe base. Instead of tackling a bipedal humanoid, however, a lizard-shaped character was chosen instead, since less issues with upright body balancing and realism expectations were anticipated here. Additionally, a lizard includes a few interesting features, e.g., both a forwards and a backwards facing set of knees, as well as a tail, making it a diverse and interesting testing subject for different components.

Depending on the software used, there are different types of tools available to help implement these modifications easier. However, there is no software that reliably transforms a static character rig to a procedural one instantaneously.



Img.45: bone constraints available in Blender

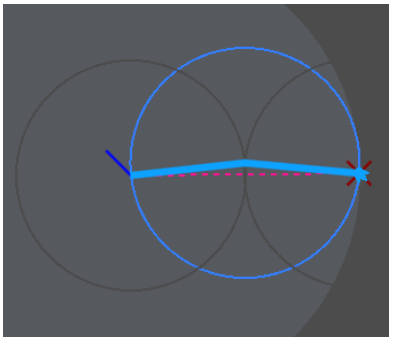
Blender offers a larger array of different bone constraints, seen in Img.45, with which complex relations or automatic triggered behaviors can be set up. While this can be helpful to quickly create procedural rig prototypes or try out certain concepts, these kinds of rig automatisations cannot be directly exported, since game engines lack matching equivalents of these tools.
While Godot does not offer many procedural rig tools, besides IK chains, Unreal and Unity do include a few more options, such as movement damping, multi-parenting or object tracking. [50]

Even with these utensils, however, building a procedurally adaptive rig requires a lot of manual configuration, as well as custom scripts to drive the behavior of bones correctly.

Custom IK

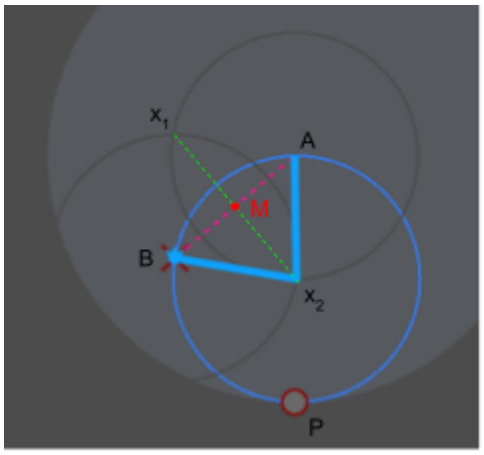
To figure out the best way to create an automatic walking algorithm, a 2D prototype was first developed, to be able to work on the step rhythm logic in isolation, before adding more complex features like placement in a 3D space or the attached SoftBody mesh.

However, since Godot has no 2D IK feature, custom leg logic was required. Since the focus had already been put on only lizard characters for the scope of this project, the IK chains would not require more than 2 bones each to accurately portray the legs. Additionally, to imitate a more natural range of motion, the legs were prevented from extending or bending at a maximal angle, by repositioning the end effector's target positioning whenever it was at maximum reach or beyond. With these limitations, all possible cases for which there are not exactly 2 possible solutions were eliminated. Adding a pole object, however, which dictated the direction in which the knee is supposed to bend, would ensure that there is always exactly one correct solution. Hence, instead of the iterative solutions addressed before, (see pp.13-16) a simple geometric approach was chosen to calculate the concrete positions for the IK chain segments.



Img.46: maximal possible leg extension

The idea behind this approach is that both leg segments each have a clearly defined range which they can reach, depending on the segments' lengths. By drawing in the first radius around the chain's origin point, and the second one around the foot target, the two possible knee positions become visible as the intersection points of these two ranges. (Img.47, x_1 and x_2) These points then simply need to be evaluated as to which is closer to the pole target point, to determine the one correct solution.



Img.47: 2D geometrical IK solution

(a) $ratio = \frac{R1^2 - R2^2 + 1}{2}$

(b) $M = A + \overline{AB} * ratio$

(c) $r = \sqrt{R1^2 - \left(\frac{R1^2 - R2^2 + 1}{2}\right)^2} * distance(AB)$

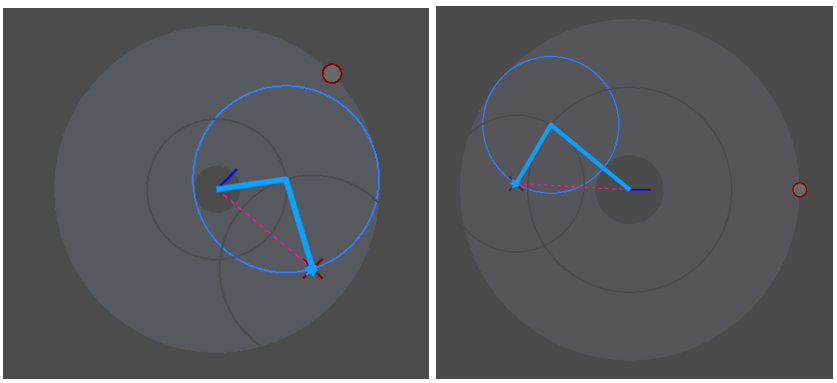
(d) $x_1, x_2 = M \pm (r).Rotated(90°)$

Img.48: 3D IK calculation

Above, the process of calculating the exact position of these two intersection points is illustrated, along with the mathematical formulas required. Point A is the chain's origin point, which will later be attached to the lizard body, Point B is the end effector where the foot is located. In this test demo, both the target position for the foot as well as the Pole Target P that controls the knee direction can be moved by clicking and dragging them around with the mouse.

The centerpoint M on the vector $\overline{AB}$ (Img.47, red dashed line) is located exactly in the middle for two equally sized bone sections. For cases like the two examples of Img.49, in which both bones are of different lengths, line (a) of Img.48 determines the ratio of the two radii, with which Point M can be found (Img.48.b). Next, line (c) finds the distance r between M and both of the wanted x-points, resulting in a vector that is parallel to $\overline{AB}$ and half the length of vector $\overline{x_1 x_2}$.

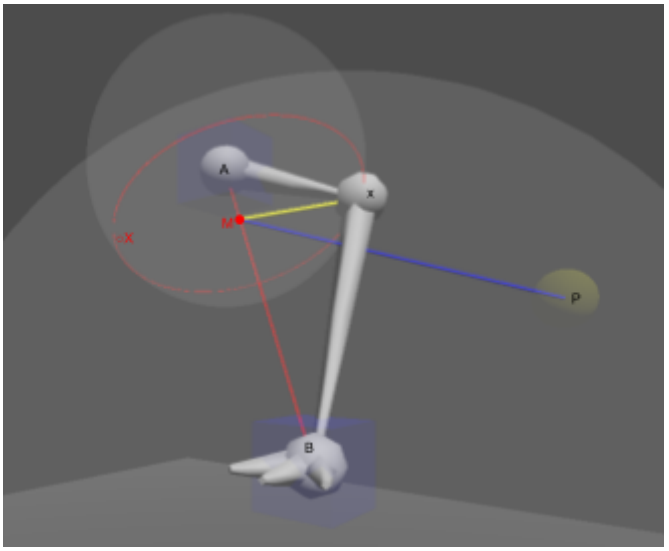
If full leg extension would be possible, r would be 0, and M would be the only possible knee position. Since these cases had already been prevented from happening, any extra checks for $r = 0$ are not necessary here.



Img.49: variable leg segment lengths

Originally, it was only planned to use this custom IK solution for the 2D prototype. However, after moving on to working in 3D, it was discovered that Godot did not include IK bones in 3D either. While this later changed with the update to GDv4.6, the adaptation of the 2D IK solution into 3D, had already been finished before this update was released.

The testing scene works similarly to its 2D counterpart: a debug view visualizes the ranges of each bone as well as various lines and points that are relevant for the knee placement calculation. Instead of being able to reposition the leg through mouse interactions though, the chain's origin and end effector each have a clickable hitbox (visualized as the semi-transparent blue cubes in Img.50) while the Pole Target is represented by a spherical hitbox. Once one of these hitboxes is clicked, it turns active (visualized in semi-transparent yellow, like the Pole sphere in Img.50) and can be moved around the scene using the W & S keys for movement on the Y axis, A & D for the X axis, and Q & E for the Z axis.



Img.50: 3D geometrical IK solution

(a) $ratio = \frac{R1^2 - R2^2 + 1}{2}$

(b) $M = A + \overline{AB} * ratio$

(c) $r(X) = \sqrt{R1^2 - \left(\frac{R1^2 - R2^2 + 1}{2}\right)^2} * dista$

(d) $\overline{MP}\downarrow_{\overline{AB}} = \frac{\overline{MP} \cdot \overline{AB}}{distance(AB)^2} * \overline{AB}$

(e) $\overline{MP}\downarrow_X = \overline{MP} - \overline{MP}\downarrow_{\overline{AB}}$

(f) $x = M + \overline{MP}\downarrow_X.Normalized * r(X)$

Img.51: 3D IK calculation

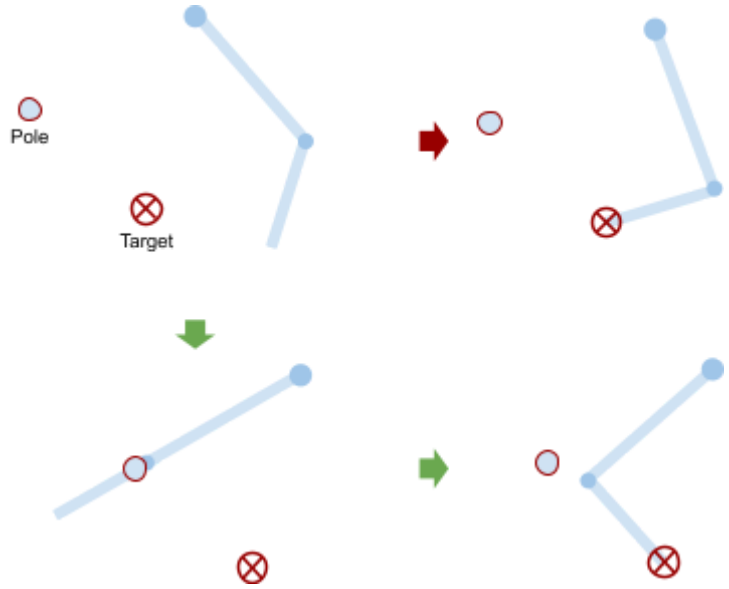
The majority of the calculation is the same as in 2D. However, the difference is that instead of having two possible solutions, the two spheres, representing each bone's range, form an intersection circle, labelled as ○X in Img.50, any point on which could be a possible knee placement.

Besides this, the calculation starts in the same way, with line (a) to (c) of Img.51 determining the ratio of the two bone lengths, as well as the middle point and radius of their intersection ○X.

Since comparing the infinite points on this circle is not possible, the closest point is determined by projecting the vector $\overline{MP}$ (Img.50, blue line) onto the intersection plane X. To project a vector

onto a plane, however, the vector first needs to be projected onto the plane's normal vector, which is already known as $\overline{AB}$. (Img.50, red line) Once projected onto the plane, the vector only needs to be scaled to fit the circle's radius, resulting in the yellow line in Img.50, which points directly towards the final position of the knee.

As an alternative to the geometrically accurate method, a CCD algorithm has additionally been implemented for posing the IK chain. While that worked without a problem for free posing of the knee, the base CCD does not take the influence of the Pole Target into account. However, since iterative approaches, like the CCD algorithm, always finds the IK solution which is the closest to the previous position, one way to include the Pole target is by first rotating each bone in the direction of the Pole point and then performing the regular CCD rotations. (Img.52)

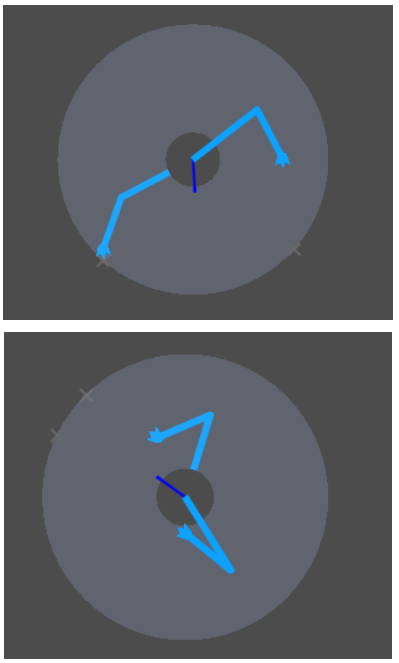


Img.52: approach for including a pole target into CCD IK

2D Rig

To figure out the best way to create the automatic walking algorithm, a simplified test rig was first made in 2D, without any SoftBody attached to it yet. Once one individual IK leg worked correctly, as described in the last chapter, two legs were combined together to create one 'Bug' creature. By adjusting the parameters of the Bug class, the two legs could be created with either forwards and backwards facing knees that automatically adapt to its current walking direction. Then, the Bug can be controlled to walk around by clicking on any spot on the screen, prompting it to automatically walk towards it.

The foot target position is saved in global space, to keep the feet rigidly in place even when the bug body moves forwards. Whenever a foot is moved too far away from the body to reach, the target position is moved to a new step position in front of the body. The new step points, visualized in Img.53 by the two grey X's at the outer edge of the range circle, are determined by taking the maximum position in the look/move direction of the bug, (visualized by the dark blue 'nose' of the bug) and then moving outwards in each direction based on the adjustable step width parameter.



Img.53: wide step width (top) vs. narrow step width (bottom)

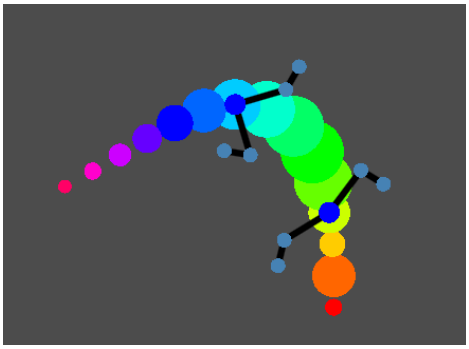
To synchronize the steps of the two legs, it was first attempted to add a separate step timer for each leg, to trigger steps. Both timers were shifted slightly to enable the alternating leg movement at fixed intervals. As long as the step timer and the walking speed were coordinated accurately to one another, and the bug walked straight forward, this approach worked fine. However, once its path included any sharper turns, this behavior led to problems, since both sides of the bug were passing by different amounts of space.

Vid.24: turning around one leg
https://youtu.be/XAeW4VStgOA

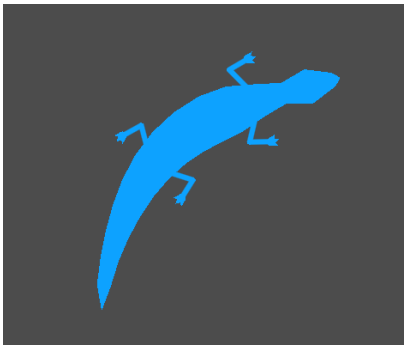
An attempted fix allowed for unscheduled steps whenever the leg required one, due to being out of the body's range, and additionally delaying the next scheduled step of the opposite leg, to ensure they are still in sync. While this was already better, this modification alone still resulted in fairly unnatural behavior in some cases, where both legs were still in an acceptable range, but due to the walking path of the bug it looked like the wrong foot was moved. Instead, the timer logic was changed to be controlled centrally on the Bug class, and always trigger the step process for the foot that is currently the furthest away from the body. This allowed the bug to make tight turns around one anchored foot, and only move the other leg around, as can be seen in Vid.24.

Lastly, to create the body of a lizard, a snake body was built by taking a set of differently scaled circles, imitating the typical lizard shape, and chaining them together by distance constraint to

keep them together as one compound whenever the creature moved around. (Img.54) This then received the same control logic as the Bug, to always move towards the point that was clicked on the screen. Due to the circle segments being dragged in a chain formation, the body displayed a very fluid and flexible behavior, with the head turning to the target location first, and the rest of the body up until the tail tip slowly following.



Img.54: debug view of 2D lizard



Img.55: lizard rest position

Since the step logic was already handled by the Bug script, two Bugs simply needed to be attached at specified points on the body, with the upper one's knees facing backwards, and the latter facing forward, to transform the snake to a fully functioning lizard.

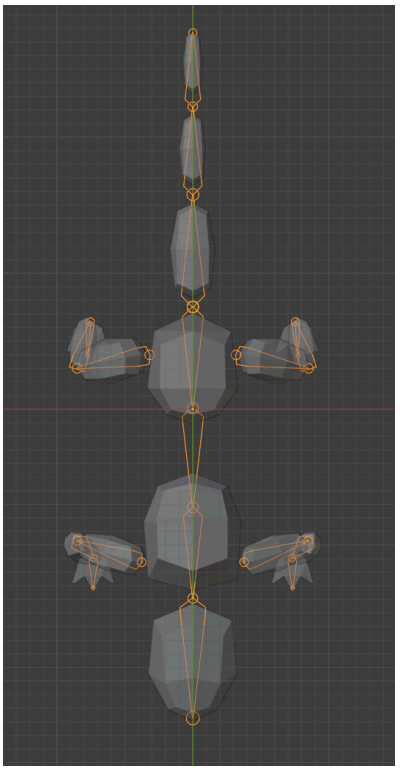
Additionally, a detection of the lizard's movement speed was introduced, to identify when it stops moving around. In this case, the legs should not stick out widely in whatever position the last step left them, but instead be neatly and symmetrically tucked besides the body in a rest position. To achieve this, the step position, which previously put the feet far in front of the bug body in preparation for the next step, was modified to be close beside the body instead, whenever the lizard stops moving forwards. (Img.55)

Vid.25: 2D walking lizard demo
https://youtu.be/t5wbhbq86NY

GDv4.6 Rig

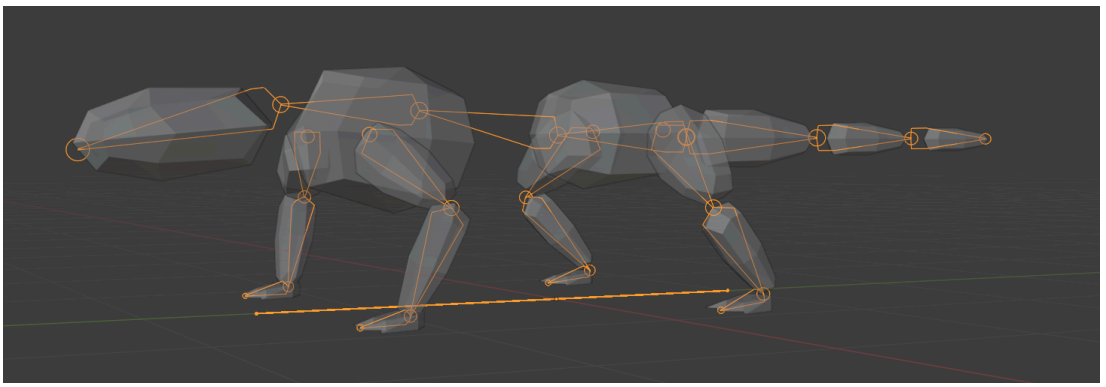
As mentioned before at several points in this paper, Godot released a new major update in January of 2026, which included several features relevant to this work, and thus caused a slight shift in focus. The biggest relevance amongst these new tools was the overhaul of the rig system, which now included several pre-implemented IK solutions, rendering the previous efforts to implement a custom 3D IK alternative pointless.

Besides a simplified two bone IK, which works similarly to the custom version implemented, the new tools include the previously analyzed CCD, FABRIK and Jacobian IK types. Additionally, Godot also offers a Spline IK option, which uses highly customizable and posable spline objects as a reference, to determine the smooth curvature of the bone chain.



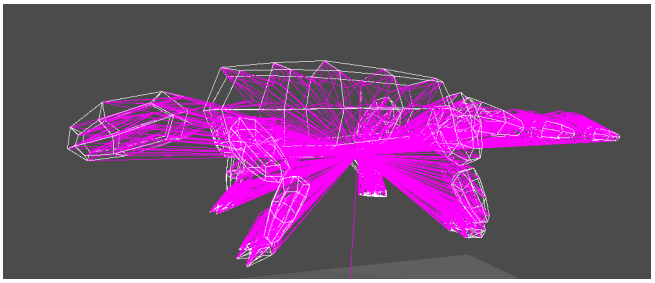
Img.56: test lizard character, top view

Since it was now possible to use the regular rig structure to animate the lizard character, it also enabled easy rig import from Blender, which made it possible to use the modelling and rigging tools from Blender for the character creation process. The created lizard consisted of the separated main body section, as explained earlier, each of which was built to be attached to exactly one bone, which allowed the continued use of the simple rig attachment, where all internal verts simply copy the position of the bone.



Img.57: test lizard character, side view

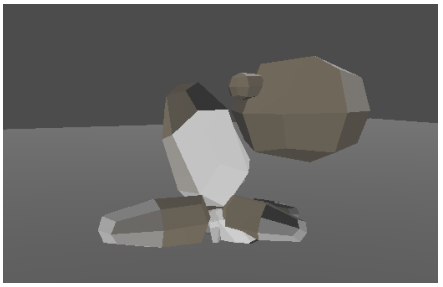
Once imported into the GD scene, the standard mesh had to be replaced with the custom SoftMesh objects. While it is possible to save multiple disconnected objects into one .obj file together, the read-in method did not differentiate between the individual meshes yet, causing all objects to be interconnected as one mesh, making the internal structure overly complex. (Img.58)



Img.58: separated meshes being constructed as one SoftMesh

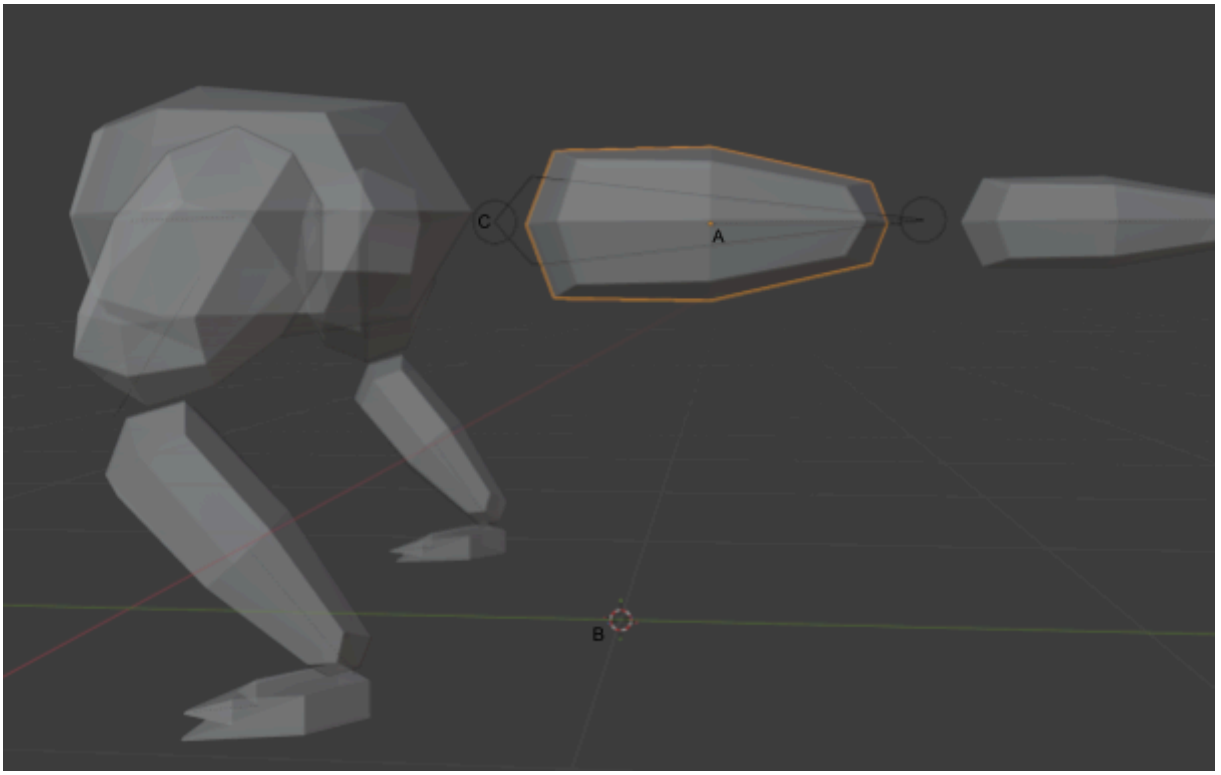
While rewriting the .obj reader class to separate the meshes into distinct objects was not difficult, assigning the meshes to their correct bone attachment posed a greater challenge, since the .obj data did not include any rig or skinning information. This was solved by making sure each mesh name was identical to the name of the bone it needed to be connected to, which made it possible to search for the correct connection by matching the names.

Connecting the meshes to the correct bone, however, was not enough to also arrange them correctly in the scene. While imported meshes that are skinned to a rig are automatically placed at the correct location to move together with their corresponding bone, the manually instantiated mesh placements varied, depended on multiple factors, e.g., where the origin point of the meshes were set, and whether local or global coordinates were used. At first the origin points of the meshes were each set to the center of their geometry (Img.61.A), causing them to clump together (Img.59). Moving the origin to the root location instead, (Img.61.B) impacted the meshes' movements, as they now pivoted around the wrong point. Eventually, each mesh's origin point was manually adjusted in Blender, to match the root position of their respective bone. (Img.61.C)



Img.59: body segments clumping around shared root positon

Img.60: mesh placement result of mismatches local-global coordinate calculations

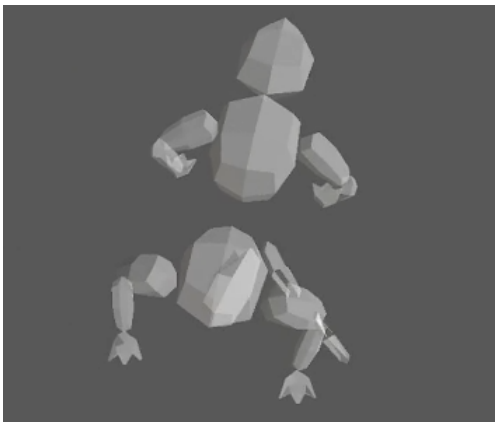


Img.61: possible body segment origin points

However, while the meshes now moved along with the rig correctly, the SoftMesh implementation had not accounted for rotation as well, since it only added the change in location to each vertex position. Manually adding the rotation to the internal vertices as well would have been possible, but would have required more complex matrix transformation, further complicated by the shifted origin point around which the structure needs to be rotated.

Instead this issue was resolved by restructuring the local vs. global coordinate system the SoftMeshes work with again, by introducing an even stronger separation of internal and external vertex handling. While the external vertex calculations remained in global space, unaffected by the influence of the rig, and only pulled by the spring connection of the inside, the pinned vertices were returned to their previous state: remaining motionless in the same local coordinates. This ensured that any transformation of the bones was passed down directly onto the internal vertices unchanged, while the outer ones remained unrestrained and instead fully relied on their physical simulation.

Another rotation-based issue that emerged, affected the rotations of the bones. Due to the way Godot converts Euler Rotations and Quaternions, none of the bones could be rotated past the positive Z-axis, since this caused a Pi-wrapping issue, which resulted in the bones performing one full rotation in the opposite direction to reach the desired angle. (Vid.26) While a similar issue had briefly been encountered during the 2D walking rig, it was solved quite simply, by adding or subtracting a full rotation from the desired angle. (Img.62) The same approach was not as straight-forward in 3D however. Instead, it took multiple failed attempts before a working solution was found, to normalize the Pi value on all three axes. (Img.63)



Vid.26: effects of the Pi-Overflow issue on lizard tail rotation
https://youtu.be/e9Z_UVZqWZE

```
// adjust for Pi wrap
if (angle > Math.PI) angle -= float.Tau;
if (angle < -Math.PI) angle += float.Tau;
```

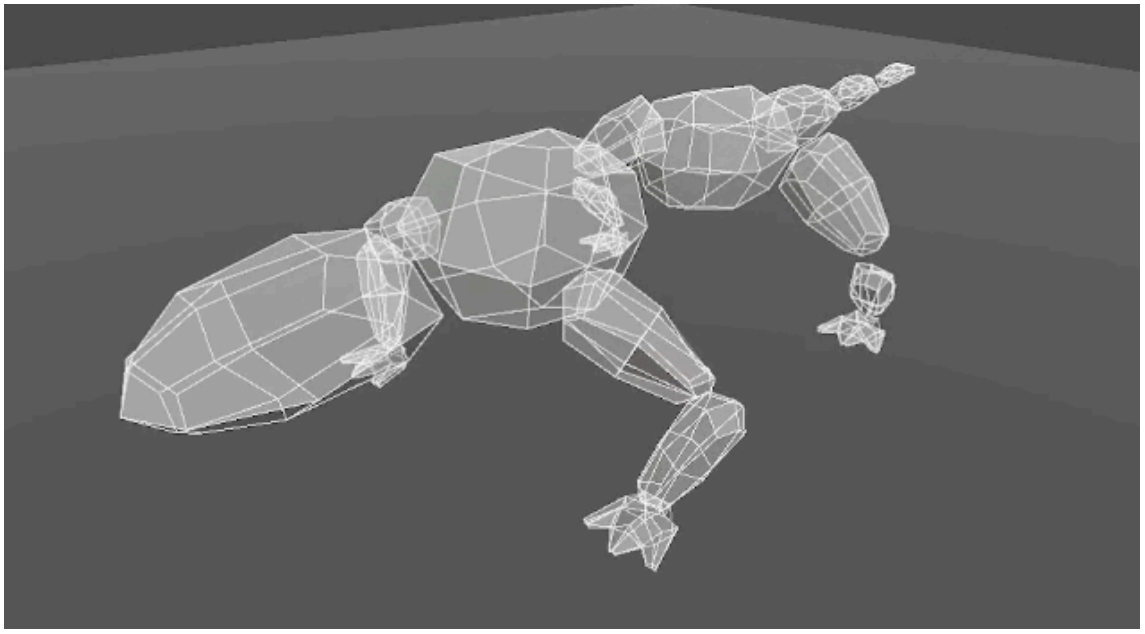
Img.62: code excerpt of 2D Pi-wrap fix

```
// PI WRAPPING
var a :int = angle.X > Math.PI ? 1 : angle.X < -Math.PI ? -1 : 0;
var b :int = angle.Y > Math.PI ? 1 : angle.Y < -Math.PI ? -1 : 0;
var c :int = angle.Z > Math.PI ? 1 : angle.Z < -Math.PI ? -1 : 0;

rot += new Vector3(a*float.Tau, b*float.Tau, c*float.Tau);
```

Img.63: code excerpt of more elaborated 3D Pi-wrap fix

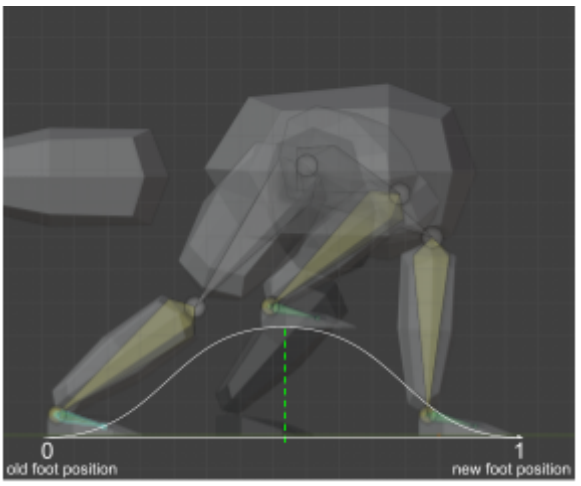
Finally, the results were a fully assembled lizard that could be posed through the rig, and whose outer skin deformed elastically based on gravitational and inertial influences. To move the rig around, a simple controller class was introduced, limited to only forwards/backwards movements as well as turning around to the left or right. This was all the direct input that had been decided to give the animation, while all other motions were supposed to be created procedurally.

As long as the body parts are in continuous motion, the deformations remain appropriately subtle, while being more clearly visible during any sudden changes in velocity. Vid.27 exaggerates this behavior for demonstration purposes, by repeatedly hitting the forwards button, causing the lizard to start and stop movement in quick succession, highlighting its jigglyness.



Vid.27: jiggly lizard, not yet with any procedural animation elements.
https://youtu.be/7kjEUOlNWd0

The main concepts of the procedural walk adaptations were able to be adapted from the 2D lizard prototype, including the forwards/backwards knee orientation, the new step position calculation with adjustable step width, the timer logic to trigger the movement of the furthest leg, and the rest step pose when the lizard stops moving.

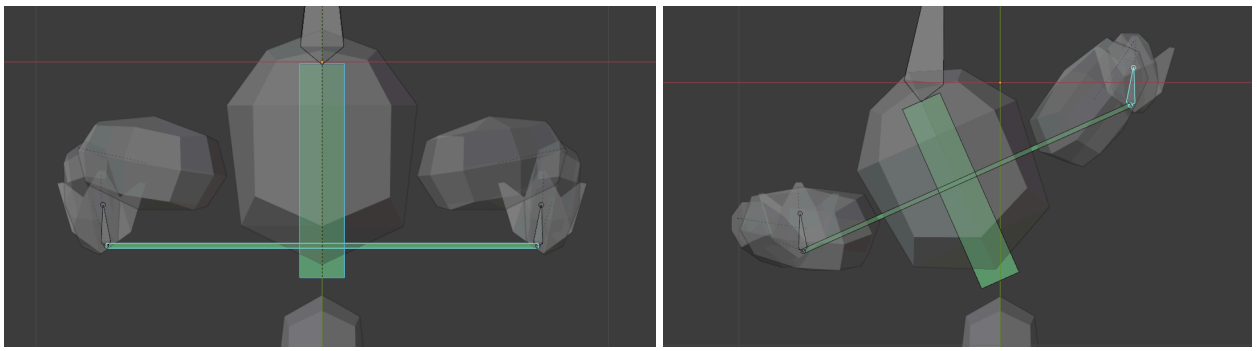


Img.64: Sine curve sampling to lift feet up during steps

However, there were also unique challenges that the new spatial dimension added. For example, the feet could no longer be moved on a direct path to their new step positions, but instead needed to be lifted up slightly while on their way there. This could be achieved with a sine curve, using the current foot position relative in between the last step position and the next target step position, to sample the curve for the height of the foot. This curve could additionally be scaled or otherwise modified, depending on the desired step height, to customize the walk further to the character, or the sampled elevation of the terrain, making it reactable to the environment.

While the IK controller drove the movement of the leg, the hips were currently still completely unaffected, no matter in which positions the feet were located. In a natural walk, however, the hips will automatically tilt along with the legs, to allow the entire leg more rotation space to move forward. To replicate this behavior, the hips should match the angle in which the feet are posed to one another. While this feature could not be fully implemented in Godot for the time being, the theoretical concept of how it could be realised was developed, and a prototype was created in Blender.

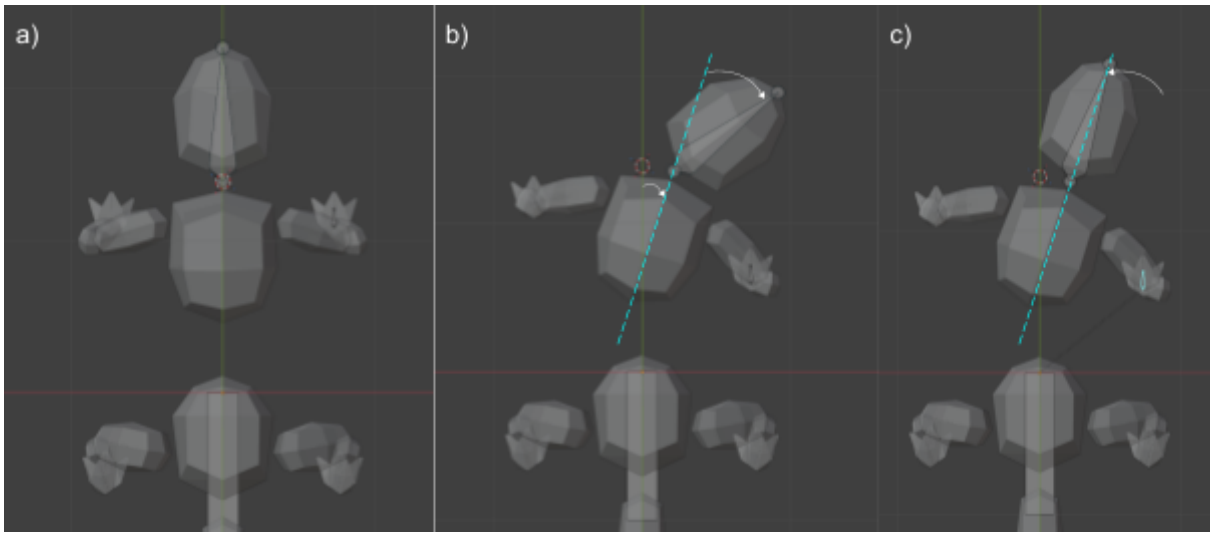
The prototype works with a newly added reference bone, connecting the two feet, to evaluate the angle in between them. In a normal rest pose, this reference line is perfectly perpendicular to the hip bone, viewed from above. When one of the feet moves forward, however, the orientation of the reference line between the feet shifts. By constraining the hips rotation to always remain consistent to the reference line, the hips can be rotated along while only moving the feet controller.



Vid.28: lizard hip tilt prototype
https://youtu.be/FP34aSTA0rg

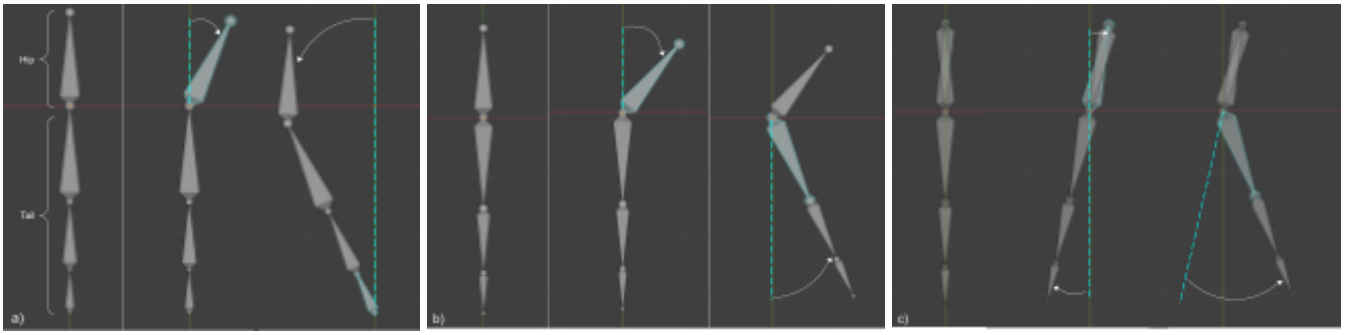
While the 2D lizard's movement was led from the head on backwards, causing it to always automatically orient its head towards their turning direction first, this does not automatically work for the 3D lizard, due to the different setup. Either a whole body rotation will pivot around the root position, which is located underneath the hips, or the upper body is moved independently, rotating the center connection between hip and shoulder bone. In either way, the head is firmly attached to the torso, and follows its rotation exactly, resulting in a stiff and unnatural look.

The proposed concept to recreate the predictive head turning works by measuring the change in rotation the torso does since the last frame, and adding it to the local rotation of the head. (Img.65.b) Since this would cause the head to turn twice as far as the torso, the head needs to be limited by a maximal rotation, to avoid it from turning too far with longer turns. Afterwards, once the torso stops moving, the head then needs to slowly return to its rest position, in line with the torso rotation. (Img.65.c)



Img.65: predictive head turn example

Finally, with the hips rotating along dynamically, the attached tail of course needs to be affected by this movement as well. Based on the way the rig has been set up, the tail will exhibit different kinds of behaviors. If the line from the tail along the spine up to the head is treated as one connected line, (Img.66.a) the origin point of the entire rig would be at the very tip of the tail. With this setup the hips are independently posable from the tail, however, any adjustment of the tail would also affect the hips, as well as the entire rest of the lizard.

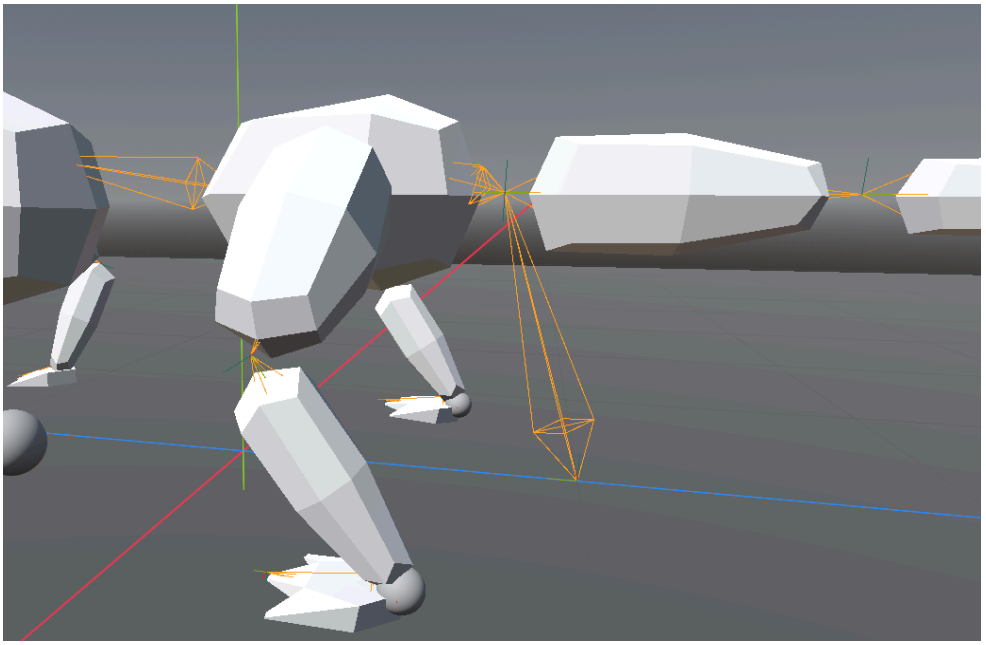


Img.66: different tail-hip rig setups

Alternatively, the spine and tail can be modeled as two separate chains, which flow in opposite directions, with the spine starting at the hips and going forward up to the head, while the tail goes from the hips backwards to the tail tip.(Img.66.b) This way both chains are completely disconnected from one another and can be posed individually. However, this also means the tail will not be able to follow the position of the hips, and could potentially become disconnected.

To make the tail follow the rotation of the hips, the bones need to be connected to them, even though the hip bone points in the wrong direction. This can be achieved by duplicating and reversing the hip bone, to act as a connection between the hip and tail base. (Img.66.c)

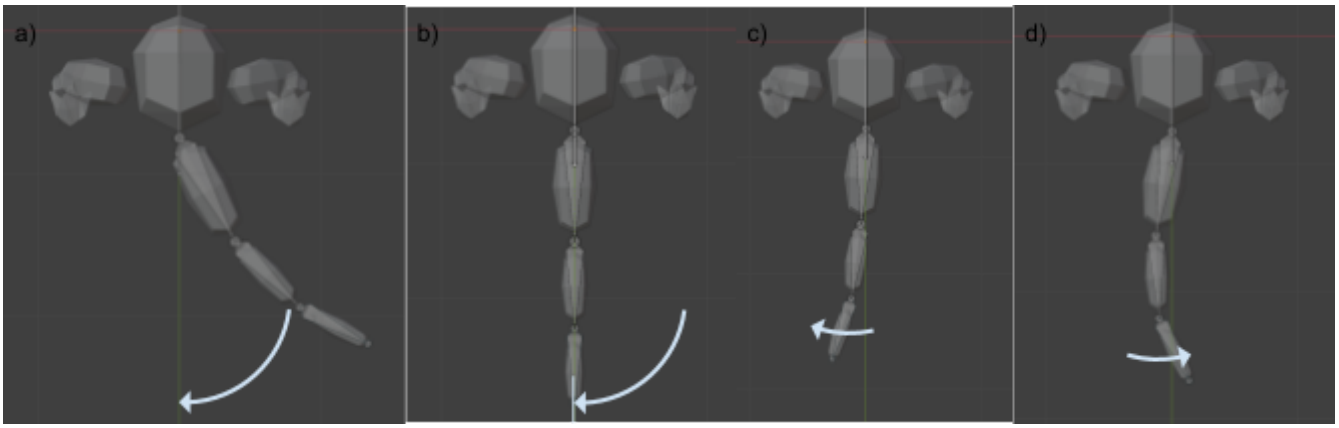
To allow for detailed tail adjustments, the procedural rig required full control over the tail rotation, without being pre-constrained by the hips. However, it still needed to be avoided that the tail disconnects from the body.[4] Thus, a variation of the connections from Img.66 was implemented, which introduces an additional root bone, to which both the hip and tail bone are parented, preventing them from being moved around, while still allowing them to be rotated freely.



Img.67: lizard root bone

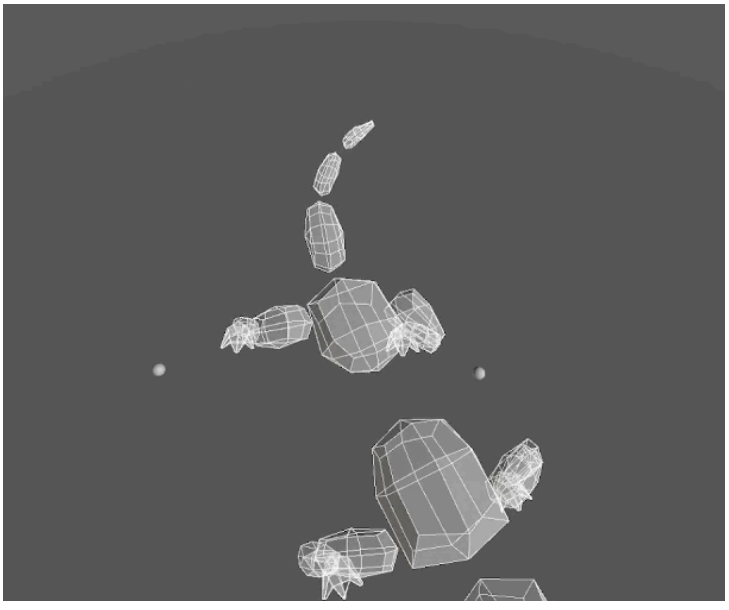
[4] after all, lizards only do that when in great distress

With the tail freely poseable, next it had to follow the hip rotation again. However, instead of exactly copying the rotation, (Img.68.b) each bone segment needed to gradually assume the previous' rotation, while being limited by a maximal possible rotation per frame. This lead to a damped chain that behaves similarly to the 2D lizard tail.



Img.68: (a) tail rotate to match hip orientation (b) stopping exactly at target orientation leads to unnatural movements (c) more natural alternative with overshooting motion (d) damped swing back to target orientation

For an even more natural appearance, this behavior was expanded to allow for some overshoot of the tail movement, by using the same Verlet integration and dampening approach, previously used for the SoftBody vertex behavior. (p.51) This way, instead of moving directly towards its target rotation, the tail wags back and forth a few times before settling in the final pose. (Vid.29)

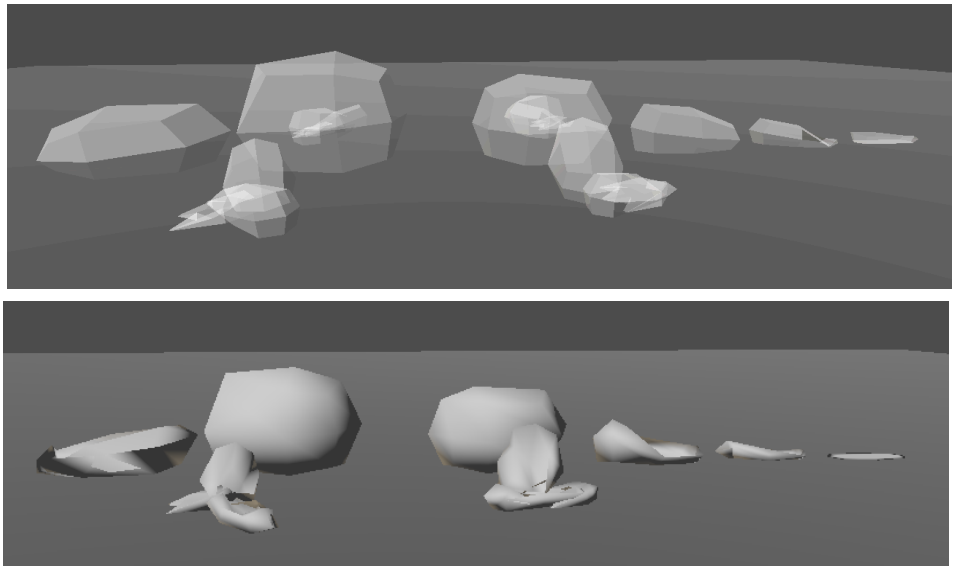


Vid.29: tail lag demo with IK-restrained hindlegs
https://youtu.be/-6fck7u33yA

5 Conclusions

5.1 Evaluation and Comparisons

The goal for this thesis was to use the open-source game engine Godot, to find a way to combine the topic of SoftBody physics with Procedural Animation. While the tools that are currently included in Godot are not fully suitable to be used in this fashion, a new approach to strategically abstract and simplify the physics aspect has been presented, and a custom SoftBody physics class that is specifically optimized for use in combination with rig based animation was created. The split of the complete character mesh into separate body sections brings many advantages, one of which is the improved performance due to the simplified shapes requiring less spring calculations to achieve the same internal strength as more complex, concave meshes would. This also allows for more internal subdivision to be made, without having a noticeable influence on the performance, improving the stability of the individual meshes. Img.69 shows a direct comparison of dropping the segmented lizard mesh onto a collision ground plane. The upper mesh, simulated by the custom SoftMesh class, shows slight collapse in the smaller tail sections, but overall maintains the shape of the individual meshes well. Meanwhile the mesh on the bottom, which is simulated using GD SoftBody presets, includes major deflation in nearly all objects, besides the two biggest main sections of the torso. These collapses persisted even after attempting to improve the meshes stability by adjusting the SB’s parameters regarding stiffness and pressure.



Img.69: custom SoftMesh lizard (top) and GD physics SB lizard (bottom) falling on a ground collision plane

Additionally, due to the rig attachment having been one of the main requirements while developing the SB class, the resulting SoftMeshes imitate the movement of real bodies in a more natural fashion, with the outer layers of skin getting only indirectly moved from the inside out through their attachment to the bones.

Besides this, this thesis details multiple aspects in which a character rig can be modified to include different areas of predictive or delayed motions, as well as how these can work together to achieve a unified walking animation that is able to procedurally react to user input and environmental influences.

In summary, this thesis evaluated the different tools that Godot and Blender provide for the creation of Procedural Animations as well as SoftBody physics, determined which of these can work in cooperation with one another, and developed custom tools, where the provided ones failed to match the expectations.

5.2 Outlook

While in this thesis a general workflow was developed, and multiple concepts were introduced that can be used to create a fully SB simulated character model with a procedurally reactive rig, no final polished end result was achieved. This was partly due to the time constraint of the project and partly because, at multiple points, the decision had to be made to abandon pre-existing tools that did not work for the intended purpose. Instead the time had to be invested in building more fitting custom alternatives. Additionally, this thesis began without a clear ’end goal’ in mind, but was rather an experimental approach to combining two complex topics. It led to exploring different alternative paths, moving from the simpler 2D procedurally walking lizard to a 3D SoftBody, figuring out what concepts work together well, what can be done, and what can be achieved within the scope of this thesis.

The following is a collection of various aspects continuing on this work that could still be addressed in future works.

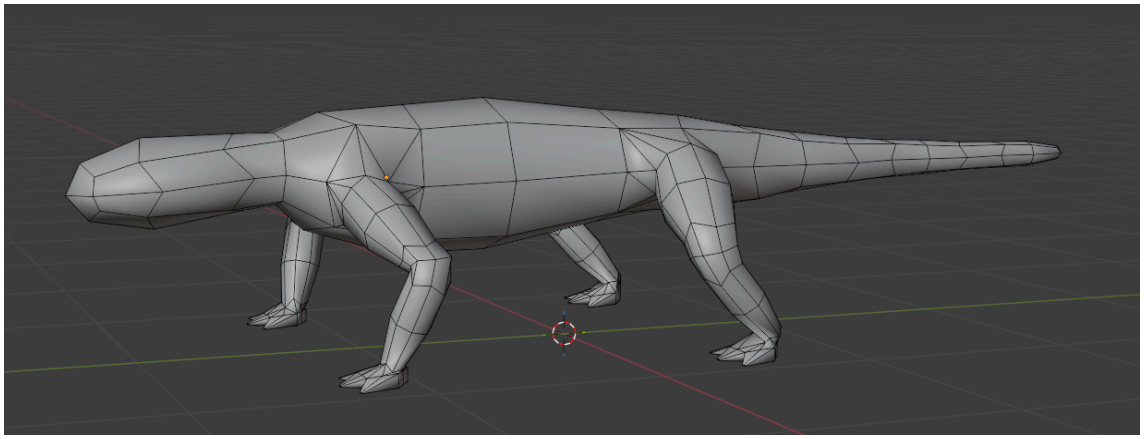
Finalization

The current state of the lizard rig is not considered fully finished. While the results are satisfactory, the implementation of the predictive head turn as well as the hips orientation following the foot placement currently remain primarily theoretical features. Additionally, an effective way to synchronize and place the feet’s IK controller to achieve the reactive steps was developed, but not yet adapted from the 2D lizard to the final 3D rig.

Nevertheless, several of the unique obstacles the 3D rig manipulation posed, like the complex bone hierarchy and accessibility, were successfully handled. Thus new additional features can be implemented in the future with greater ease and expertise.

Secondly, there is the aspect of the disconnected body sections that was left unfinished. While, in theory, it should be easy to add one final skin layer over the body parts, to achieve a finalized, whole mesh, the experimentation or research for this subtopic could not be focused on deeply during this thesis. However, a quick mockup in Blender was created, where all body sections were manually joined to fill the gaps between them. (Img.70) While, due to the concave shape of the extruding legs, it was not possible to use Blender’s shrinkwrap function to unify the body fully automatically, the use of the Bridge function, which fills in gaps between two specified edges, created a semi-automatic approach for this issue. Nevertheless, this process

required enough manual decision making, e.g., deciding which faces of these section objects needed to be deleted, or which edges needed to be bridged towards which other edge, that it prevented the process from being entirely automated.



Img.70: unified lizard mesh

Collision

Currently, neither the custom made SoftBodies, nor the character rig, possess any form of collision detection that would enable it to interact with obstacles. While possible approaches for collision handling with SBs or simulated rigs were researched and explained in this work, they have not been integrated yet. The only collision-adjacent feature that was implemented is a brute force approach, that stopped any of the SB vertices from surpassing a Y-coordinate of 0, creating a hardwired ground floor for the meshes to land on. While this feature was used to test the behavior of the SBs collision during their development phase, this constraint was later disabled again, since it interfered with the free motion of the rig-attached SoftBodies.

Automation

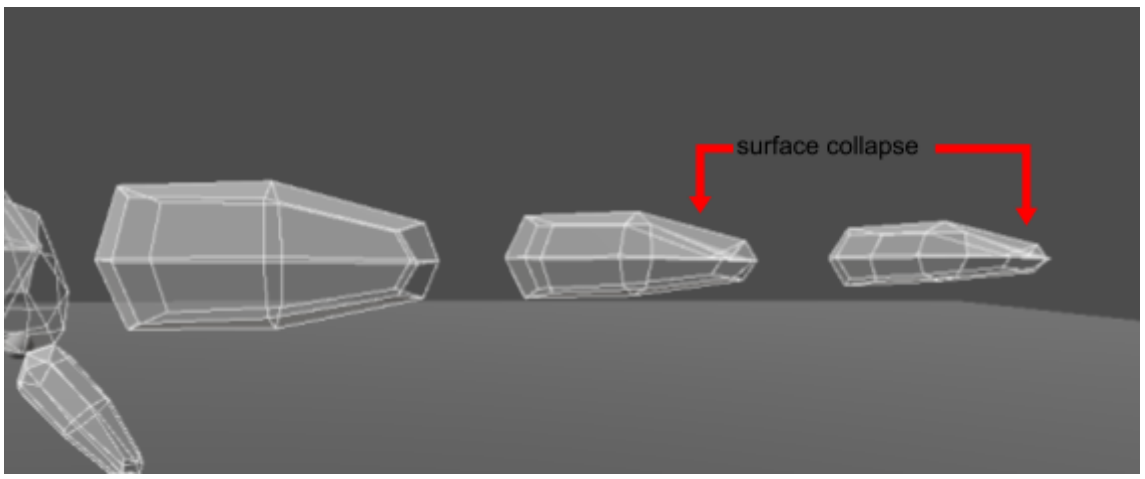
The way the test lizard mesh was created included manually modeling each body section, and importing the rig and mesh separately into Godot. Initially, an attempt was made to import both objects together, by using Godot's *MeshDataTool* to scan the imported mesh, and replace it with the custom SoftMesh. This approach had to be abandoned, since Godot automatically converted the model's topology into exclusively triangles, and thus interfered with the generation of additional springs for added support. While reading in mesh- and rig-data separately creates no issues by itself, it is redundant work to select the correct object twice,

which also poses a potential error source if different objects or incompatible versions of the object are accidentally being selected.

Another idea, which could not be pursued further yet, was finding a way to take a whole character mesh and automatically split it into simplified body sections, based on the bone structure of the rig, as opposed to modeling them manually. This approach would also make applying a final skin layer over the simulated SBs much easier, since the original mesh data would already be existent, and thus would not need generating.

Stability

Due to all SoftMeshes of the separated lizard mesh getting created from the same .obj file, they are all instantiated from the same class and are thus passed down the same variables for spring stiffness and dampening. While this does not cause big enough problems, which would have warranted sacrificing more time for trying to rework this aspect, it does lead to smaller body sections being noticeably less stable than bigger ones. This becomes especially apparent when looking at the tail, which is made up of the same repeated mesh shape, which get continuously smaller towards the tail tip. While the base section of the tail shows no stability issues, the latter pieces caves in slightly. (Img.71) This issue could be addressed by determining each object's volume during the initialization phase, and scaling the spring constants inversibly to it, providing additional stiffness to the smallest objects.



Img.71: intentionally lowered stiffness values to demonstrate surface collapse on smaller body section

Besides this aspect, no other major stability issues were left noticeable with the finished SoftMesh implementation. That being said, there were also no extensive stress tests done,

which could have revealed any more problems. Theoretically, however, there are certain facets that could prove problematic under specific circumstances.

For one, there are no recursive checks of already solved constraints, as previously addressed. (p.50) Whenever a vertex gets moved by multiple springs, these springs will get adjusted one after the other. If the constraint of later springs would stretch out one of the earlier finished springs beyond reach, it would not get adjusted again until the next frame update, which can cause issues with extreme deformations. Additionally, since the vertices are always iterated through in the same order, the same springs are likely to be overextended again in the next update as well. Another non-recursive approach to address this issue could be to randomize the order in which the vertices get worked on, to avoid the same mistakes compounding on the same springs at every frame. However, since the created SoftMeshes did not display any noticeable deformation issues, no extra time was dedicated to implement any of these possible solutions.

Optimization

Finally, during test runs, no major performance issues had become apparent. However, this is likely due to the simplified body segments' shapes, and the omitting of intensive collision checks. For applications in a more complex scene however, a multithreading approach could be implemented to parallelize the calculations of the separate body sections. Currently, all code is simply put into the update method, to let Godot handle the sequence in which each object gets worked on, instead of manually ensuring that each object gets parallelized in the most effective way.

References

[1] A. O. Staff, “When Disney Went Digital,” Animation Obsessive. [Online]. Available:https://animationobsessive.substack.com/p/when-disney-went-digital

[2] Jayofthetrees [@jayofthetrees], “The vehicles in 101 Dalmatians are actually 3d models that are then rotoscoped(Filmed and then animated over) into the film itself. The thick, black outlines are actually crucial for this process. This movie also just turned 62. https://t.co/HtkPDcoCjP,” Twitter. [Online]. Available:https://x.com/jayofthetrees/status/1618474276021235713

[3] T. J. Pierce, “Drawn to Imagination: Behind the Magic: 101 Dalmatians,” drawn to Imagination. [Online]. Available:http://www.drawntoimagination.com/2017/03/the-snowy-truth.html

[4] N. Magnenat-Thalmann and D. Thalmann, “The Direction of Synthetic Actors in the Film Rendez-Vous a Montreal,” *IEEE Computer Graphics and Applications*, vol. 7, no. 12, pp. 9–19, Dec. 1987, doi: 10.1109/MCG.1987.276934.

[5] R. Wareham and J. Lasenby, “Bone Glow: An Improved Method for the Assignment of Weights for Mesh Deformation,” in *Articulated Motion and Deformable Objects*, F. J. Perales and R. B. Fisher, Eds., Berlin, Heidelberg: Springer, 2008, pp. 63–71. doi: 10.1007/978-3-540-70517-8_7.

[6] M. Sharma, “Ultra Wideband Wearable Sensors for Motion Tracking Applications,” Ph.D. dissertation, School of Elect. Eng. and Comp. Science., Queen Mary Univ. of London, London, 2015. [Online]. Available:http://qmro.qmul.ac.uk/xmlui/handle/123456789/9859

[7] “Mimic Productions”, “Motion Capture Technology: 5 Different Types of Motion Capture,” Mimic Productions. [Online]. Available:https://www.mimicproductions.com/post/motion-capture-technology-types-of-motion-capture

[8] A. Bereznyak, *IK Rig: Procedural Pose Animation*, (2016). [GDC Festival of Gaming]. Available:https://www.youtube.com/watch?v=KLjTU0yKS00

[9] t3ssel8r, *Giving Personality to Procedural Animations using Math*, (Jun. 29, 2022). Accessed: Mar. 21, 2026. [Online Video]. Available:https://www.youtube.com/watch?v=KPoeNZZ6H4s

[10] S. Murray, *A Behind-The-Scenes Tour Of No Man's Sky's Technology - YouTube*, (Dec. 05, 2014). [Online Video]. Available:https://www.youtube.com/results?search_query=A%20Behind-The-Scenes%20Tour%20Of%20No%20Man%27s%20Sky%27s%20Technology%2C%202014

[11] J. Jakobsson and J. Therrien, *The Rain World Animation Process*, (2016). [GDC Festival of Gaming]. Available:https://www.youtube.com/watch?v=sVntwsrjNe4

[12] A. Aristidou and J. Lasenby, "Inverse Kinematics: a review of existing techniques and introduction of a new fast iterative solver," Cambridge University Engineering Department. Available:https://andreasaristidou.com/publications/papers/CUEDF-INFENG,%20TR-632.pdf

[13] R. Ye, "Inverse Kinematics for Game Programming," Medium. [Online]. Available: https://medium.com/@turtle50vp/inverse-kinematics-for-game-programming-5f9a408e24b2

[14] A. Zucconi, "Inverse Kinematics for Tentacles," Alan Zucconi. [Online]. Available:https://www.alanzucconi.com/2017/04/12/tentacles/

[15] T. Asfour, "Chapter 3 – Inverse Kinematics," *Introduction to Robotics*, [Online]. Available:https://pasta.place/Informatik/Robotik_1_[HIS]/Folien/WS_24-25/Robotics-I-Chapter-03-Inverse-Kinematics.pdf

[16] I. Herzog, *Citizencon 2017: Teaching a character how to walk on any terrain*, (2017). [Online Video]. Available:https://www.youtube.com/watch?v=PryJ3CpHcXQ

[17] D. Rosen, *Animation Bootcamp: An Indie Approach to Procedural Animation*, (2014). [GDC Festival of Gaming]. Available:https://www.youtube.com/watch?v=LNidsMesxSE

[18] S. Abreu, "How to make Active Ragdolls in Unity," Medium. [Online]. Available:https://sergioabreu-g.medium.com/how-to-make-active-ragdolls-in-unity-35347dcb952d

[19] K. Sugimori, H. Mitake, H. Sato, and S. Hasegawa, *Avatar Tracking Control with Featherstone's Algorithm and Newton-Euler Formulation for Inverse Dynamics*, (Nov. 2023). [Proceedings of the 16th ACM SIGGRAPH Conference on Motion, Interaction and Games]. Available:https://www.youtube.com/watch?v=GpNKBbl6OLk

[20] B. Kenwright, R. Davison, and G. Morgan, “Real-Time Deformable Soft-Body Simulation using Distributed Mass-Spring Approximations,” 2011. Available:https://www.semanticscholar.org/paper/Real-Time-Deformable-Soft-Body-Simulation-using-Kenwright-Davison/ef27ec3ae925849e43c7a1cca54cf026b3699679

[21] X. Provot, “Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior” [Online]. Available:https://www.semanticscholar.org/paper/Deformation-Constraints-in-a-Mass-Spring-Model-to-Provot/733f7ce98ff239f20e66447205b58395e1107384

[22] M. Matyka and M. Ollila, “Pressure Model of Soft Body Simulation,” Jul. 01, 2004, *arXiv*: arXiv:physics/0407003. doi: 10.48550/arXiv.physics/0407003.

[23] argonaut, *Simulating soft body animals*, (Mar. 09, 2025). [Online Video]. Available:https://www.youtube.com/watch?v=GXh0Vxg7AnQ

[24] K. Czapla and M. Pleszczyński, “Montecarlo Methods to Efficently Compute Volume of Solids,” in *Proceedings of the International Conference of Yearly Reports on Informatics, Mathematics and Engineering*, R. Fazzolari and A. A. Jaber, Eds., in CEUR Workshop Proceedings, vol. 3118. Online, July: CEUR, Jul. 2021, pp. 1–9. [Online]. Available:https://ceur-ws.org/Vol-3118/#p01

[25] M. G. Larson and F. Bengzon, *The Finite Element Method: Theory, Implementation, and Applications*, vol. 10. in Texts in Computational Science and Engineering, vol. 10. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-33287-6.

[26] Y. Shumak, “FEM in Structural Analysis: Workflow, Element Types, and Validation,” SDC Verifier. [Online]. Available:https://sdcverifier.com/structural-engineering-101/fem-in-structural-analysis/

[27] M. Müller, M. Teschner, and M. Gross, “Physically-based simulation of objects represented by surface meshes,” in *Proceedings Computer Graphics International, 2004.*, Crete, Greece: IEEE, 2004, pp. 26–33. doi: 10.1109/CGI.2004.1309189.

[28] D. L. James, J. Barbič, and C. D. Twigg, “Squashing cubes: automating deformable model construction for graphics,” in *ACM SIGGRAPH 2004 Sketches on - SIGGRAPH ’04*, Los Angeles, California: ACM Press, 2004, p. 38. doi: 10.1145/1186223.1186271.

[29] “Blender,” “Physics - Cloth - Introduction - Blender 5.1 Manual.” [Online]. Available:https://docs.blender.org/manual/en/latest/physics/cloth/introduction.html#workflow

[30] PiCode, *Godot FINALLY Got Better Softbody*, (Jun. 05, 2024). [Online Video]. Available:https://www.youtube.com/watch?v=2fP9uioaEaA

[31] J. Rouwe, *jrouwe/JoltPhysics*. (Mar. 21, 2026). C++. Available:https://github.com/jrouwe/JoltPhysics

[32] J. Rouwe, “Architecting Jolt Physics for ‘Horizon Forbidden West,’” GDC Vault. [Online]. Available:https://gdcvault.com/play/1027891/Architecting-Jolt-Physics-for-Horizon/?_mc=edit_gdcsf_gdcsf_le_x_19_x_2023

[33] “Guerrilla Games,” “We create awe-inspiring worlds - Guerrilla.” [Online]. Available:https://www.guerrilla-games.com/read/architecting-jolt-physics-for-horizon-forbidden-west

[34] J. Rouwe, “Soft body support · Issue #390 · jrouwe/JoltPhysics,” GitHub. [Online]. Available:https://github.com/jrouwe/JoltPhysics/issues/390

[35] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, “Position based dynamics,” *Journal of Visual Communication and Image Representation*, vol. 18, no. 2, pp. 109–118, Apr. 2007, doi: 10.1016/j.jvcir.2007.01.005.

[36] M. Macklin, M. Müller, and N. Chentanez, “XPBD: position-based simulation of compliant constrained dynamics,” in *Proceedings of the 9th International Conference on Motion in Games*, Burlingame California: ACM, Oct. 2016, pp. 49–54. doi: 10.1145/2994258.2994272.

[37] godotengine, “Add (or at least investigate) Jolt physics engine · godotengine/godot-proposals · Discussion #5161,” GitHub. [Online]. Available:https://github.com/godotengine/godot-proposals/discussions/5161

[38] Chevifier, *Godot Physics vs Jolt Physics Uncut*, (Oct. 15, 2023). [Online Video]. Available:https://www.youtube.com/watch?v=B3fEdhKw8mg

[39] rakun99, “Attempt of making custom vehicle: Jolt vs GodotPhysics,” r/godot. [Online]. Available:https://www.reddit.com/r/godot/comments/1hkod6c/attempt_of_making_custom_vehicle_jolt_vs/

[40] “Godot,” “Godot 4.6 Release: It’s all about your flow.” [Online]. Available:https://godotengine.org/releases/4.6/#section-highlights

[41] “Blender,” “Physics - Soft Body - Introduction - Blender 5.1 Manual.” [Online]. Available:https://docs.blender.org/manual/en/latest/physics/soft_body/introduction.html

[42] Rakenval, *3D Character Rigging Timelapse - Part 3, Legs and Tail*, (Mar. 27, 2025). [Online Video]. Available:https://www.youtube.com/watch?v=EPiHGtRFCR0

[43] Savallion, *how to do cloth physics on ACTUAL clothing ✨ Godot Softbody3D*, (Jan. 04, 2025). [Online Video]. Available:https://www.youtube.com/watch?v=7higlB3ueDk

[44] 최성수, *Godot engine - Softbody work on hair in 5 minutes*, (Jun. 09, 2021). Accessed: Mar. 21, 2026. [Online Video]. Available:https://www.youtube.com/watch?v=yttmpihjmEo

[45] abrasivetroop, “active soft bodies in my game :D,” r/godot. Accessed: Mar. 21, 2026. [Online]. Available:https://www.reddit.com/r/godot/comments/14kdqzx/active_soft_bodies_in_my_game_d/

[46] Godot,” “Vertex displacement with shaders — Godot Engine (3.0) documentation in English.” Accessed: Mar. 21, 2026. [Online]. Available: https://docs.godotengine.org/en/3.0/tutorials/3d/vertex_displacement_with_shaders.html

[47] “Godot,” “PrimitiveMesh,” Godot Engine documentation. [Online]. Available:https://docs.godotengine.org/en/stable/classes/class_primitivemesh.html

[48] “Godot,” “ArrayMesh,” Godot Engine documentation. [Online]. Available:https://docs.godotengine.org/en/stable/classes/class_arraymesh.html

[49] “Godot,” “Using ImmediateMesh,” Godot Engine documentation. [Online]. Available:https://docs.godotengine.org/en/stable/tutorials/3d/procedural_geometry/immediatemesh.html

[50] S. Jouda, “Procedural Animation in Unity.” [Online]. Available:https://saifjouda.github.io/SaifJouda/procanim

Appendix

Link to the Github Repository

https://github.com/chaotic-pan/SB-ProcAnim

List of Abbreviations

- GD - Godot
- SB - SoftBody
- RB - RigidBody
- BBone - Bendy Bone
- FK - Forward Kinematics
- IK - Inverse Kinematics
- CCD - Cyclic Coordinate Descent
- FABRIK - Forward And Backward Reaching Inverse Kinematics
- tris - 3 vertex face / triangle
- quad - 4 vertex face
- n-gon - face that consists of more than 4 vertices
- pos - position
- velo - velocity