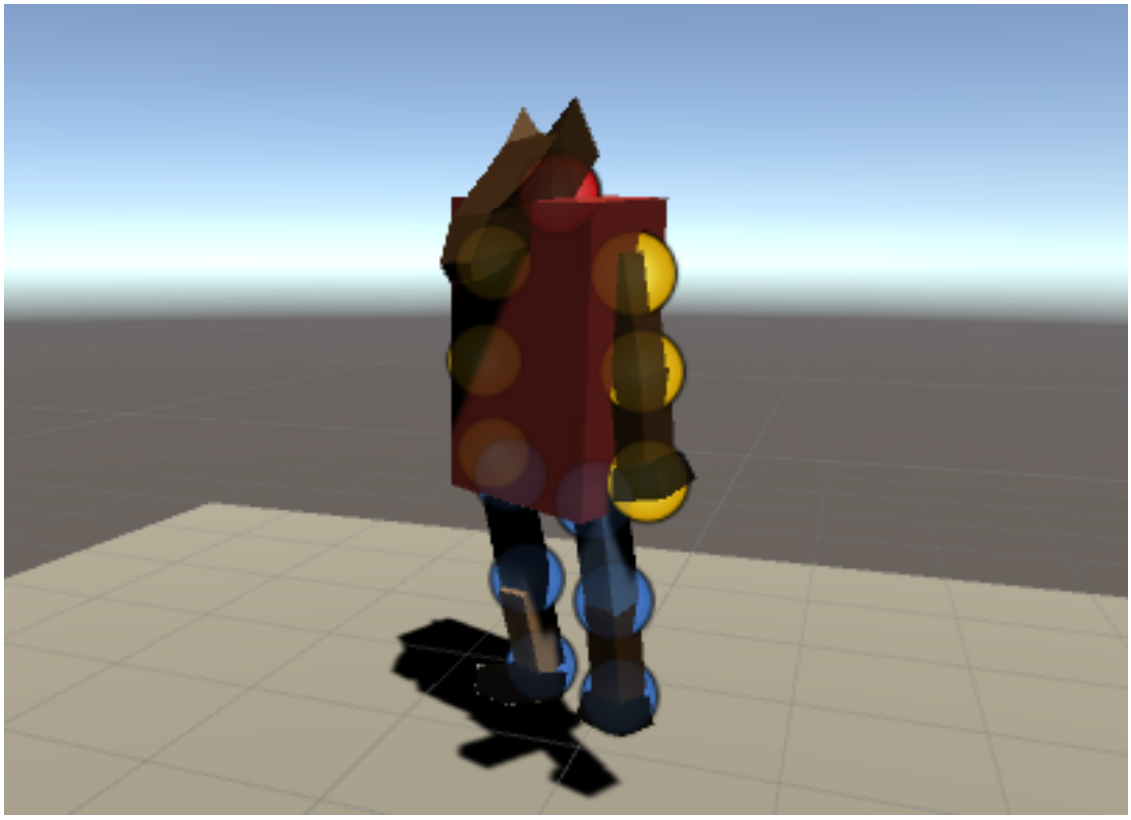
CHALMERS
UNIVERSITY OF TECHNOLOGY



A Video Game Using Procedural Animation

Bachelor's thesis in Information Technology

JOHAN ANDERSSON, LAGE BERGMAN, ERIK HILDINGE,
JOHAN IVERSEN, DANIEL JOHANSSON

CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Department of Information Technology
Gothenburg, Sweden 2017

Bachelor's thesis CIUX03-17-10

A Video Game Using Procedural Animation

Johan Andersson, Lage Bergman, Erik Hildinge, Johan Iversen, Daniel Johansson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Information Technology
Group: CIUX03-17-10
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

A Video Game Using Procedural Animation
Johan Andersson, Lage Bergman, Erik Hildinge, Johan Iversen, Daniel Johansson

Supervisor: Marco Fratarcangeli, Department of Computer Science and Engineering
Examiners:
Arne Linde, Associate Professor in the Computer Engineering division, department of Computer Science and Engineering.
Niklas Broberg, Lecturer, Information Security division, Department of Computer Science and Engineering.

Bachelor's Thesis CIUX03-17-10
Department of Information Technology
Group: CIUX03-17-10
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Images and figures are captured or created by the authors, if nothing else is stated.

Cover: A randomized character with parts from both horse and man.

Typeset in LaTeX
Department of Information Technology
Gothenburg, Sweden 2017

Abstract

Video games are played more often and by more people every year. As the consumption rises, so does the expectations on the animations. In this report, the potential of procedural animation is investigated through comparisons of different techniques within the field. These techniques include, but are not limited to, PID-controllers, genetic algorithms and contact invariant optimization. Different methods to balance, walk, fight and destroy objects are compared.

The end result is a fighting game where two controllable characters face off against each other in an arena where parts of the environment are destructible.

Sammandrag

TV- och datorspel spelas oftare och av allt fler människor varje år. Samtidigt som marknaden växer ökar förväntningarna på animationerna. Potentialen för procedurell animation undersöks i denna rapport tillsammans med en jämförelse av olika tekniker för att uppnå olika mål inom området för procedurell animation. Dessa tekniker inkluderar, men är inte begränsade till, PID-regulatorer, genetiska algoritmer och kontaktinvariant optimering. Olika metoder för att balansera, gå, slåss och förstöra objekt jämförs.

Slutresultatet är ett spel där två kontrollerbara karaktärer slåss mot varandra i en arena med delvis förstörbar miljö.

Keywords: animation, procedural animation, physics-based, game, balance, fighting, destruction, optimization

Acknowledgements

We would like to thank our supervisor, Marco Fratarcangeli, for believing in us and guiding us through this project. We would also like to thank Patrick Andersson, Pontus Eriksson, Adam Ingmansson and Love Westlund Gotby, for proofreading and giving valuable feedback on the report.

Contents

List of Figures

1
Introduction

Procedural animation is the collective name for all techniques that create animations algorithmically rather than manually. A switch from traditional animation to procedural animation changes focus for animators from creating good animations to creating good animation algorithms and algorithm configurations.

The traditional way to create movements for animated characters in movies and video games is through key frame animation or by use of motion capture technology. While animated movies require every scene to be animated separately, in video games the objective is to animate every character action in a way that they can be performed seamlessly in sequence, without the player noticing the change from the animation of one action to the next.

The key frame approach is often tedious and time consuming. Especially if the animations are for a game where the characters can perform many different actions. In these cases, each action requires a separate number of frames. While motion capture is less time consuming than designing frames by hand, the problem with repeating the process for any possible action in a game remains. Also, it can be hard to map the movement of an actor to non-human characters in a smooth way. Alternative methods to create movements that seem realistic is constantly being developed, and procedural animation is one of them.

By creating animations algorithmically, the same animations can be used for different objects and characters. For example character animation algorithms could be used by multiple different characters and new animations would not have to be remade manually. The algorithms can in some cases even be applicable to both bipedal and quadrupedal characters and even to characters without legs, like fish.

Good procedural character animations are important, not only because they would relieve some of the work of game and movie creators, but because they can potentially be transferred to robots. In some war ridden areas, robots are sent in, for example to deliver medical supplies or ammunition, to avoid risking human lives. Because of the terrain in these areas, walking robots are sometimes a better option than robots using wheels. Algorithms developed for virtual characters could, if developed in a modular, adaptable way, be applied directly to these kinds of robots without much modification at all.

To animate virtual creatures in a realistic way, the movements of real living creatures has to be replicated. The movement of living creatures, however, is highly complex and replicating them is not trivial. One approach is to create simplified versions of the musculature system of the creatures to be animated, and have them act in physics based environments. Furthermore the parameters for these simplified systems also need to be defined, manually or somehow optimized.

There are also some limitations as to how the systems and algorithms should work. In the movie industry, animations are allowed to take a long time to compute and render, as long as the finished result looks good. In contrast, the game industry has a higher focus on visually plausible results that can be computed in real-time.

1.1 Purpose

The purpose of this thesis is to investigate and implement different techniques for creating procedural animation applicable to game development.

1.2 Scope

In order to narrow down the field of researched methods, a number of criteria are presented.

For characters, only balance, basic locomotion and punching will be researched. No complex behaviour or facial movement will be studied. Neither will any algorithms regarding mesh rendering or skinning.

The environment will contain only solids. No animation of gases or liquids will be investigated. The way they will be animated is limited to destruction only. Destruction, in this case, means parts of the environment breaking and falling apart. No deformation will be featured in the thesis.

The goal is not for the animations to be strictly realistic, but rather visually plausible. Implementations that do not result in highly unrealistic animations, such as limbs bending in incorrect ways or piercing other body parts, will be considered sufficient. Cheating, like inserting forces to counter gravity, is allowed in order to reach a pleasing result.

Finding the optimal parameters for an animation beforehand is allowed, but the animations themselves needs to be calculated in real-time, which is defined to be at least 15 fps[1], and able to adjust themselves based on the circumstances.

Animations that meet all these criteria will be implemented separately and the ones that are compatible with each other will be put together into a simple game scene with limited interactivity.

2
Theory

This section explains the theory which serves as the basis of the implementations made.

2.1 Springs

A spring is a device that stores potential energy. Most springs follow Hooke's Law, which states that the force required to drag or push a spring to a certain length, X, away from its rest length, X_0, is proportional to the length, X. This means there is some constant which, multiplied with the distance to the rest length, gives the force the spring exerts on an object attached to it. [2]

$$F = k * X$$

By attaching a spring with a small rest length to two objects, the objects will be pulled together. In a physics simulation, invisible springs can be used in this way in order to move objects to stationary points in space.

By using invisible springs that follow Hooke's law, a physics solver can calculate a force on an object and move it. This is a good way to move objects if a physically accurate representation of the way a spring would do it is required. However, if only visually plausible results are desired, there are other methods of achieving the same effect. One way is to move the object towards the desired position with a velocity directly proportional to the remaining distance.

$$v = k * d$$

Like Hooke's law, this creates a greater movement towards the goal the further away the object is but without the unnecessary complication of involving forces and mass.

In scenarios where the stored energy of a spring is to be used for rotation, a specific form of spring is preferred. An angular, or torsion spring is a spring which has a rest angle instead of a rest length, and exerts a torque instead of a force. With these springs, adjusting the rest angle rotates the connected bodies.

2.2 Balance and Locomotion

The animations of greatest importance are the ones concerning balance and locomotion. For a natural and dynamic looking result, a character model should preferably strive towards keeping its balance by using its body to compensate for gravity. This

means that it should react in a realistic way to regain its standard pose when for example getting pushed. Various cheats can be used to attain the illusion that the character is keeping its balance. For example an upwards force could be applied to its torso, like an invisible string holding a marionette. Another workaround is having only certain parts of the body affected by gravity, like the arms, while the lower body is weightless and animated in a conventional way.

While workarounds like external forces might give a plausible result in many cases, it might also clash with the rest of the game world, if there is an overarching focus on procedural animation and realistic physics. In the case of an entirely procedurally animated game, having a pushed character remain static or bounce back like a spring would make it lose some of its main appeal and interesting gameplay.

Making a character fully affected by physics demands that is has a means of moving all its individual limbs in a synchronized fashion. Furthermore, all forces applied to it should be natural, i.e. any force should have a counter force to avoid getting a positive force resultant where energy is created in the system.

2.2.1 Muscles and Torques

One way of having a character model exert force while always making sure there is a counter force, is to only use torques. A symmetrical torque, by definition, always has a resultant force of zero. Any applied force in the form of torques thus has to make contact with other objects and the environment to move the character. This means that to walk, the character has to push against a surface. Using torques with no external object to exert force to, the only resulting behaviour is rotation with no net force. Thus, non-physical and unwanted behaviour like making a character fly by applying too much force becomes impossible.

On a computer, torques can be applied directly to limbs and other objects. In the real world however, this needs to be done indirectly. In the case of a human, torque is created by the contraction of muscles. One method of abstracting muscles is to let each muscle consist of a pair of points symbolizing the muscle's attachments. By applying forces of opposite directions on these two points, the effect will be that of a contracting muscle or an extending muscle depending on the directions of the forces, as seen in figure 2.1.

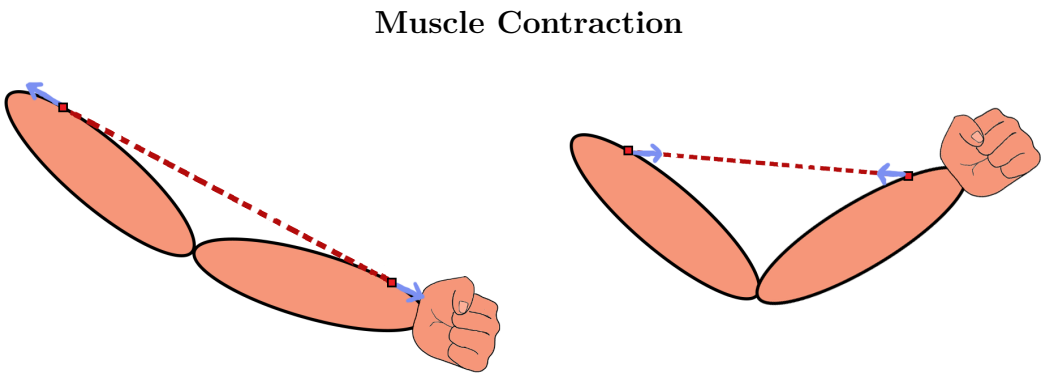


Figure 2.1: The figure shows an application of an arm contracting and expanding by the use of muscle points. This is an alternative way of applying a torque.

An advantage of using muscle representation, is that it gives a more intuitive interface for the implementation, compared to working directly with torques. Muscles also grants a natural range of motion, ensuring that joints are not twisted too far in any direction. Furthermore, it makes for an easier way of manipulating the lever and fulcrum dynamics which affects the resulting limb movements.

One aspect of the game is the generation of new characters from a set of predefined body parts. Having muscle attachments included on each separate body part makes the composition of joints relatively simple.

Joint Connection

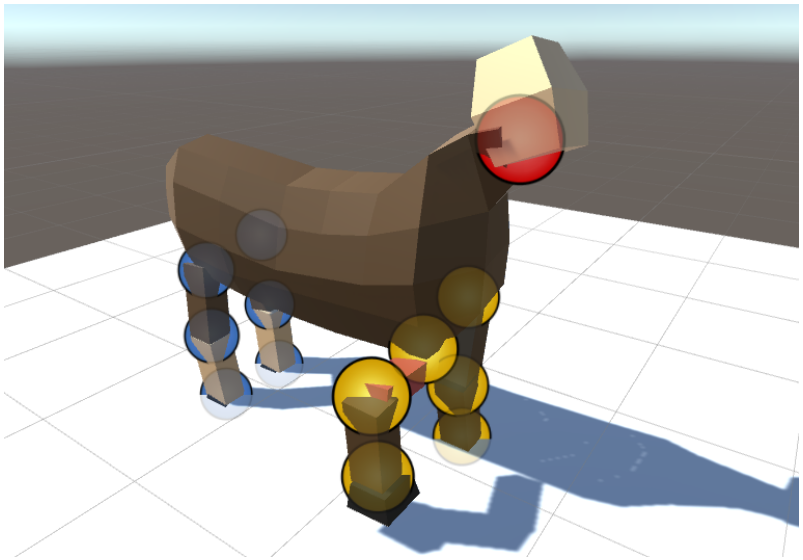


Figure 2.2: All parts of the figure are randomized and connected between the coloured dots. The picture contains a character with a human head, the body of a horse and a combination of limbs for legs.

2.2.2 Inverse Kinematics

One method of calculating the limb positions of an animated character or a physical robot, is to use inverse kinematics (IK). In this method, a target position of an end effector, i.e. an object placed at the end of a chain of limbs, e.g. a hand or a foot, is first defined. Changing the end effector's position also requires the positions and angles of the parent limbs to be recalculated. [3]

The main subject of IK within the world of animation is to make different end effectors react to the surroundings in the world. One instance of this is for the feet of a character to be parallel to a crooked floor.

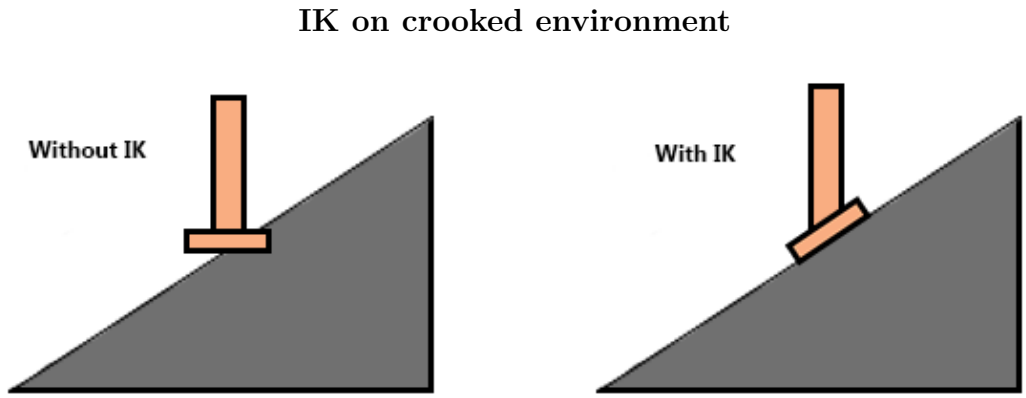


Figure 2.3: The image shows an example of how an end effector could adjust itself with the help of IK.

IK is also used to make characters fixate on, for example, eyesight in such a way that the head follows the object that the animator designates. Another example is to put the hands of a character on the edge of a wall when climbing. The fixation aspect from IK can include everything from the range of reaching towards the object as it gets close, to gentle panning of the head. IK could therefore also be used as a way for characters to punch. When the "punch button" is pressed, the character's fist could potentially register the opponent as the end goal.

As the name suggest, IK differ from traditional forward kinematics in the order that positions and rotations of the limbs are calculated. Using the movement of an arm as example; forward kinematics starts at the shoulder and ends at the hand, while IK starts at the hand and then finds the required positions and rotations of the previous joints, i.e. in the reverse order.

Unity has a built-in method for IK using the Mecanim[1] animation tool. This method however only works for humanoid characters using a properly configured avatar and thus was not applicable for the randomly generated characters.

2.3 Finding Solutions by Optimization

Whenever a character in a game should perform some task, such as walking or moving a limb to a certain position, it should be done in a realistic way to increase the realism of the game. The most realistic way is often the most efficient since our bodies have been tuned through many years of evolution to conserve energy. Therefore it is interesting to optimize, with respect to time and energy consumption, the movement of animated characters to reach a desired location or pose. This has been tried and tested with good results. T. Geijtenbeek et al. present a muscle-based control method for simulated bipeds.[4] The method uses a large number of parameters, including some representing muscle physiology, muscle geometry as

[1] `https://docs.unity3d.com/Manual/AnimationOverview.html`

well as control parameters. For optimizing this large set the method uses Covariance Matrix Adaptation[5], which is a form of evolutionary algorithm.

An evolutionary algorithm is an algorithm that draws its inspiration from natural phenomena. Here, two different types of evolutionary algorithms will be investigated, namely genetic algorithms and particle swarm optimization.

2.3.1 Genetic Algorithms

Genetic algorithms draw their inspiration from natural evolution. One can view evolution as an ongoing optimization of species' genomes because of the combinatorial nature of DNA. This is the reason why genetic algorithms are so good at solving combinatorial problems. They do this by simulating evolution on a population of particles. Each particle has some fitness score, calculated by a well defined fitness function. The fitness score states how good that particle's values are compared to those of other particles. In biology, the fitness function simply calculates how high chances an individual has of producing offspring, whereas in a genetic algorithm it can be defined to calculate how good the particles are at any task chosen. Between every iteration pairs of highly weighted particles are crossed over to create children. The children will have some values from one parent and some from the other. The children might also have mutations, which alter some of the values. In theory, after some amount of iterations the combinations have an increased amount of fitness and will then be good solutions to the problem [6].

Walking, running and jumping are often very similar movements every time they are acted out for any given character. Therefore these movements can be found and predefined using genetic algorithms. Basically these movements can be defined as a combination of variables, such as wavelength, frequency and phase of some sine function for every given limb's torque. Defining the fitness function as how long time the character can run or jump while maintaining balance makes finding these movements and predefining them well suited problems for genetic algorithms.

These types of algorithms are poorly suited for real-time calculations since they need to iterate through some amount of generations which can often be computationally heavy. However, for the creation of predefined animations, genetic algorithms can be very powerful, as it then only needs to be done once in order to be reused in real time application.

2.3.2 Particle Swarm Optimization

Particle swarm optimization (PSO) is an evolutionary computation scheme that has drawn its inspiration from flocks of birds[7]. It solves optimization problems within given search spaces by using swarm intelligence. Swarm intelligence is the collective cooperation between independent systems. Particle swarm optimization is faster than many other optimization algorithms, but the solution is not always the global maxima [8].

The algorithm initializes some amount of particles and spreads them out throughout the search space in some way. Then the particles calculate how good the solution which they are at is and communicates this to all the other particles. The meaning

of good here needs to take into consideration the problem being solved, particularly a particle's fitness needs to be well defined. After this there is usually one or a few particles which has the best solution out of the currently known solutions. All the other particles take some step, depending on their velocity, which is directly proportional to their distance from the best currently known solution, towards the best currently known solution. They then recalculate their new solutions' fitness. It goes on like this until all particles have converged at some solution.

PSO can be used to solve complex initial problems. This means that no simplification of the problem is needed, it only needs to be well defined. Since optimizing character movement is a complex problem which can be well defined, PSO can be used for this.

2.4 PID Controller

A proportional-integral-derivative controller, or PID controller for short, is a type of controller commonly used to control industrial machines and robots.

The purpose of a PID controller is to reduce the difference between some input value and a desired goal value. By creating an error function $e(t)$, describing this difference, and summing up the proportional, integral and derivative of the function multiplied with some constant each, a PID controller is able to reduce the error at a good pace without overshooting.

The following equation is the most common description of a classic PID controller function.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{2.1}$$

The PID controller is a control feedback mechanism, meaning it uses the current state of the application it is controlling in order to calculate its next input. The process can be visualized by a diagram as can be seen in figure 2.4.

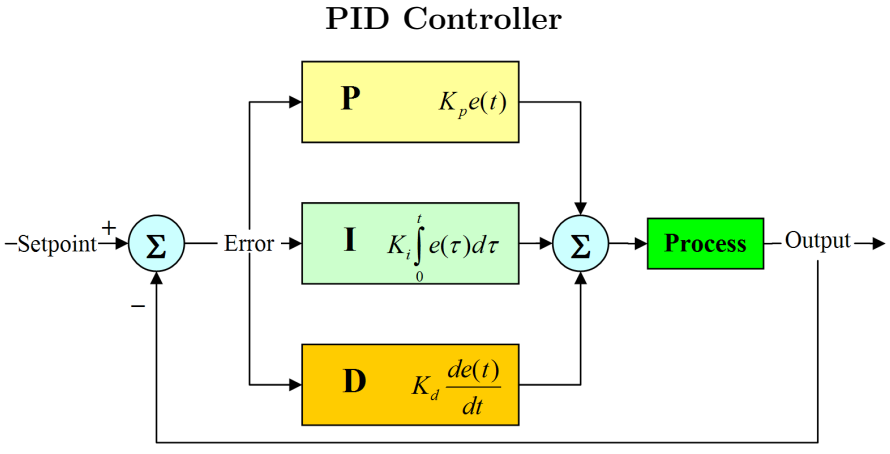


Figure 2.4: Traditional PID controller. The three terms of the controller are calculated and summed to produce an output. The difference between the output and the goal is then inserted into the function again, creating the desired feedback loop.

Each of the three different terms of the controller has a specific function. The proportional term gives an adequate starting value to increase or decrease the input with and reach the goal. The derivative accounts for reducing the frequency of the function, minimizing the risk of approaching the goal too fast and possibly overshooting. The integral term removes any small long term constant error that accumulates over time[9]. Each of these functions are of good use when working with balance.

If one of the functions of the integral or derivative is not desired or necessary, either can be removed to create a PI or PD controller. Both of these variations are as common as the regular, complete PID controller.

3
Related Work

3.1 Position Based Dynamics

Position based dynamics (PBD) is a method for simulation of physics phenomenon. The aim of the method is to produce visually plausible results while focusing on robustness and speed, and it is therefor highly suitable for game development[10]. The method works by omitting forces, accelerations and velocities and instead modifies positions directly.

PBD is known for being a fast and stable way to simulate everything from cloth movement to a characters' forward and inverse kinematics[11].

A good way to describe the mathematical and physical approach to the PBD method is presented by J. Bender et al.: "Classical methods in this field discretize Newton's second law and determine different forces to simulate various effects like stretching, shearing, and bending of deformable bodies or pressure and viscosity of fluids, to mention just a few. Given these forces, velocities and finally positions are determined by a numerical integration of the resulting accelerations."[12]

One of the problems in using PBD is, since it is physics-based, that collision detection is hard to implement[11].

3.2 Contact Invariant Optimization

One way of creating more complex behaviour with procedural animation was suggested at the 2012 SIGGRAPH conference[13]. The method involves splitting each requested behaviour into a fixed number of equally long phases in which the number of end effectors in contact remains invariant. This means the number of points on which forces can be exerted to create character momentum is limited within each phase.

By splitting the optimization process into different steps as well, you can limit the number of possible outcomes, speeding up the optimization process. In a first step, the placement of the end effectors in each phase is optimized. In the second step the forces on each end effector, and in turn the movement of the rest of the body, to reach the next set of contact points is optimized. Lastly a set of physics and collision terms are added to further the realism of the movement.

The objective function from the 2012 SIGGRAPH conference is described as following:

$$L(s) = L_{CI}(s) + L_{Physics}(s) + L_{Task}(s) + L_{Hint}(s) \tag{3.1}$$

As the author explains the equation: "L_{CI} is a novel contact-invariant cost introduced here. $L_{Physics}$ penalizes physics violations; we enforce physical consistency using a soft cost rather than a hard constraint because this enables powerful continuation methods. L_{Task} specifies the task objectives, and is the only term that needs to be modified in order to synthesize a novel behavior. L_{Hint} is optional and can be used to provide hints [...] in the early phases of optimization."[13]

Each term in the function represents some desired behaviour of the optimized movement. If the $L_{Physics}$ term is great, it means there are a lot of violations to the physics, such as limbs moving through each other or disobeying gravity. A high value of L_{Task} means the movement results in an end position and stance that is far from the desired outcome or that the goal is reached in a long time. These two directly counter each other as a physically correct movement is not the fastest. The character could fly to it's end goal but it would result in a huge value of $L_{Physics}$. This means the result of the optimization will always be some movement which reaches the end goal and stance in a relatively low amount of time while obeying the laws of physics to some extent.

The problem with this approach when it comes to gaming is the optimization is not done in real-time. Each scenario contains a fixed goal and the movements required to reach that goal are computed beforehand. In a real-time game, the player character should respond immediately to user input. This means the objective of the optimization could change at high frequency, not giving enough time for the algorithm to optimize the movements.

There is potential to adapt the optimization for real-time by switching some of the terms of the objective function to simpler versions or by switching the optimizing algorithm altogether to some algorithm which is guaranteed to reach a result in a short time without focus on finding the best possible result. There was, for example, an idea to replace the $L_{Physics}$ term with some value calculated through Position Based Dynamics, as it is a very fast approach to optimizing over basic physics constraints. This approach was not investigated further, however, because the complexity of the implementation would exceed the scope of this project.

3.3 Overgrowth

Overgrowth[1] is a game that uses procedural animation.

The basics of this game are about fighting other characters in armed and unarmed combat while also manoeuvring the terrain by running, jumping and climbing. It utilizes a combination of physical simulation, such as ragdolls, and procedural animation, using methods like interpolating between target poses.

3.4 Sumotori Dreams

Sumotori Dreams[2] is a wrestling game between two or more players, where the win condition is to make the other players fall over before you do. The game is using

[1] Overgrowth, `http://www.wolfire.com/overgrowth`

[2] Sumotori Dreams, `http://www.gravitysensation.com/sumotori/`

procedural animation in the standing and walking animations while using predefined animations for standing up again.

The way our game is built up is similar to Sumotori. The inspirations taken from this game are the cartoon looks and build up of the arena. The upper body regulation and pushing is also similar. The main differences lies in the win condition as well as the balancing of the lower body.

4

Methods

4.1 Finding Solutions by Optimization

Genetic algorithms were experimented with to find an optimal way for a quadruped to walk. This was done by using the muscle points together with periodic functions. Walking, for both bipedal and quadrupedal creatures, involves repeating the same motions over and over. Therefore these motions can be modelled with periodic functions, resulting in periodic motion.

In the experiments each muscle, together with its opposing muscle, had a periodic function modeling their motion. The periodic function was used to calculate the force with which the muscles would contract. Whenever the value of the function was positive one muscle contracted, with a force equaling that value, and when the value became negative the other muscle contracted with a force equaling the absolute value of that value. The periodic function for muscle n, and its opposite muscle, can be seen in equation 4.1. The parameter a is affecting the amplitude, ω is affecting the period, f is affecting the phase and t is the time lapsed since the start of this iteration. Note that every quadruped has specific parameters for each of its muscles, and their opposites.

$$force_n = a_n sin(\omega_n t + f_n) \tag{4.1}$$

Every iteration 1000 quadrupeds were created and given some parameters based on the previous iteration, see figure 4.1. Some amount of the quadrupeds that got the furthest during the time frame of the iteration were selected as parents for the quadrupeds of the next iteration. The parameters of two random parents were crossed-over randomly when creating a child. There was also a small amount of randomization added to the crossing-over to simulate mutation. This can be seen in figure 4.2. The reason why mutations are important is that they can create new possibilities for parameters even though they did not exist at the start. The parameters of all the quadrupeds in the first iteration were selected completely at random.

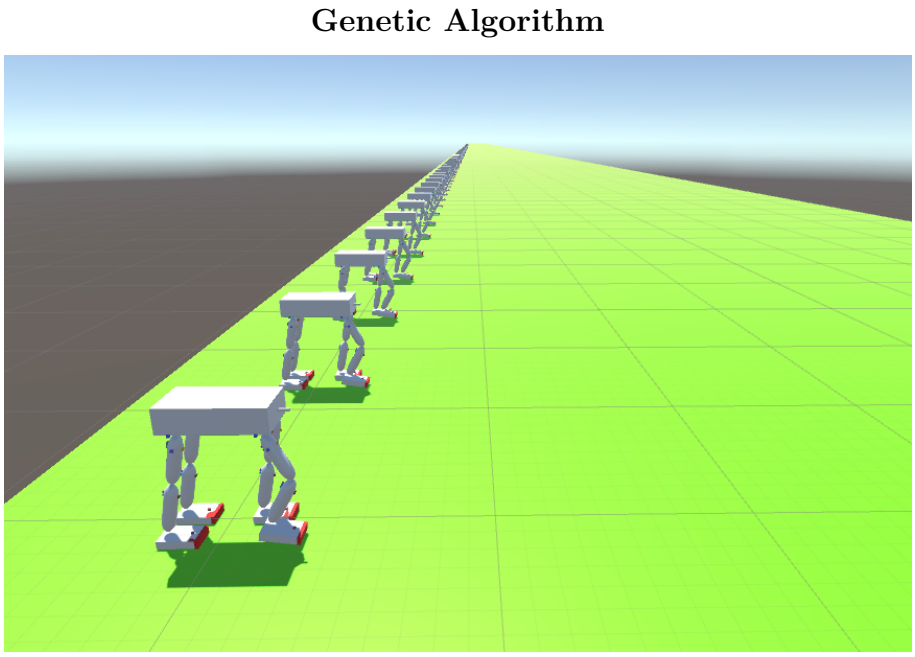


Figure 4.1: Many quadrupeds attempt to walk using different combinations of variables for many periodic functions.

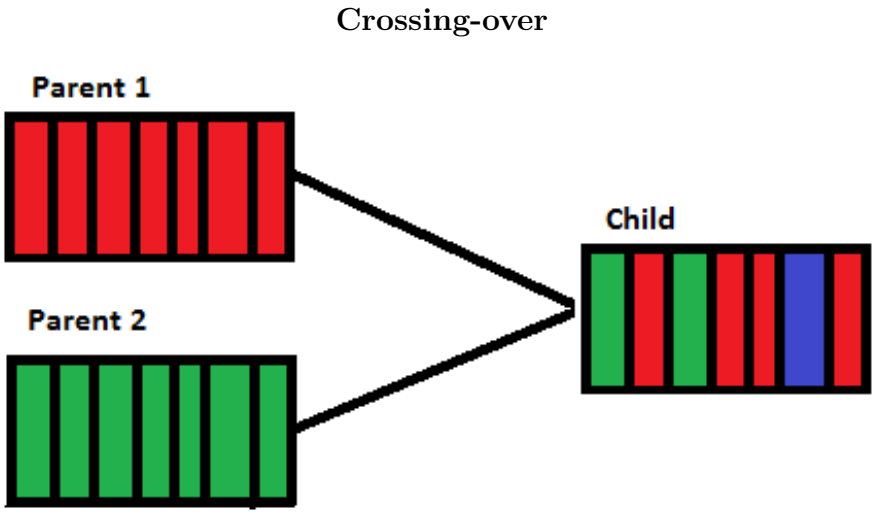


Figure 4.2: The parameters of two parents are crossed-over to create the parameters for the child. Note that the blue parameter does not exist in either parent for it is the cause of a mutation.

4.2 Punching

In order to make the characters punch, two methods were investigated: regular springs and angular springs. Torques were not investigated as a method for punching because it makes it possible for the upper and lower arm to rotate too much, creating unnatural behaviour. This is not a problem when using springs.

4.2.1 Springs

As described in chapter 2.1, regular springs can be used to move an object by attaching it to some other stationary object. This technique was implemented to keep a character's fist in one position. By placing an invisible object with collision disabled just in front of the character's shoulder, and creating an invisible spring from the object to the character's fist, the fist is kept in a stance resembling that of a boxer.

Another object is then placed at an arms length from the character's shoulder. By creating a new spring from the fists to this new object and removing the spring keeping the fist in place, the arm moves towards the target. When the fist touches the object an arms length away, the spring dragging it there is replaced with one between the fist and object at its original position. The process can be viewed as a cycle, as seen in figure 4.3.

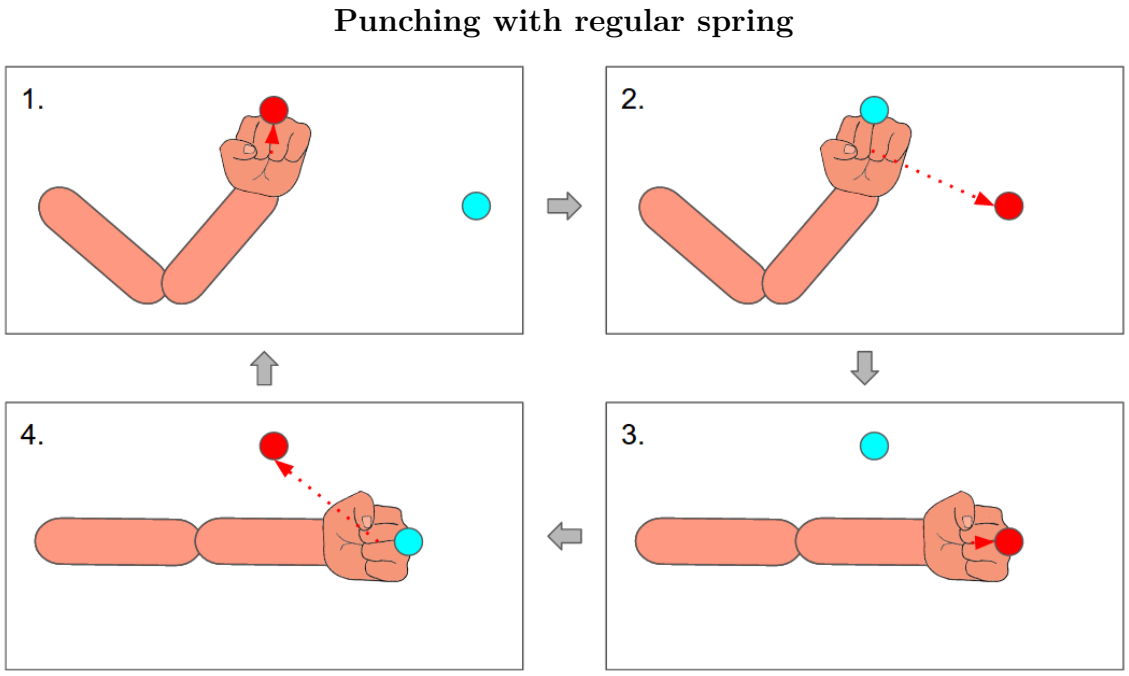


Figure 4.3: The cycle of a punch using springs. The spring's attachment changes from the close position to the one at an arms length away. When the position is reached it is changed back, resulting in a return to the starting position.

4.2.2 Angular Springs

Angular springs in unity are achieved using hinge joints with joint spring objects attached. By connecting two objects with this kind of joint, they are dragged towards a specific angle relative to each other.

Angular springs can be used to achieve a simplified form of inverse kinematics, with less precise results. With angular springs in the shoulder, elbow and wrist of a character, the hand can be made to reach a specific location with some tweaking.

In order to achieve punching using this technique, an experiment was conducted where the rest angle of the angular springs in shoulder, elbow and wrist of a character

were rotated away from each other. This moves the fist away from the character. By summing the real-time passed between each frame, a timer was constructed which told the character when to reverse the process in order to contract the arm again. Like with the regular spring implementation, this creates a cycle, as seen in figure 4.4.

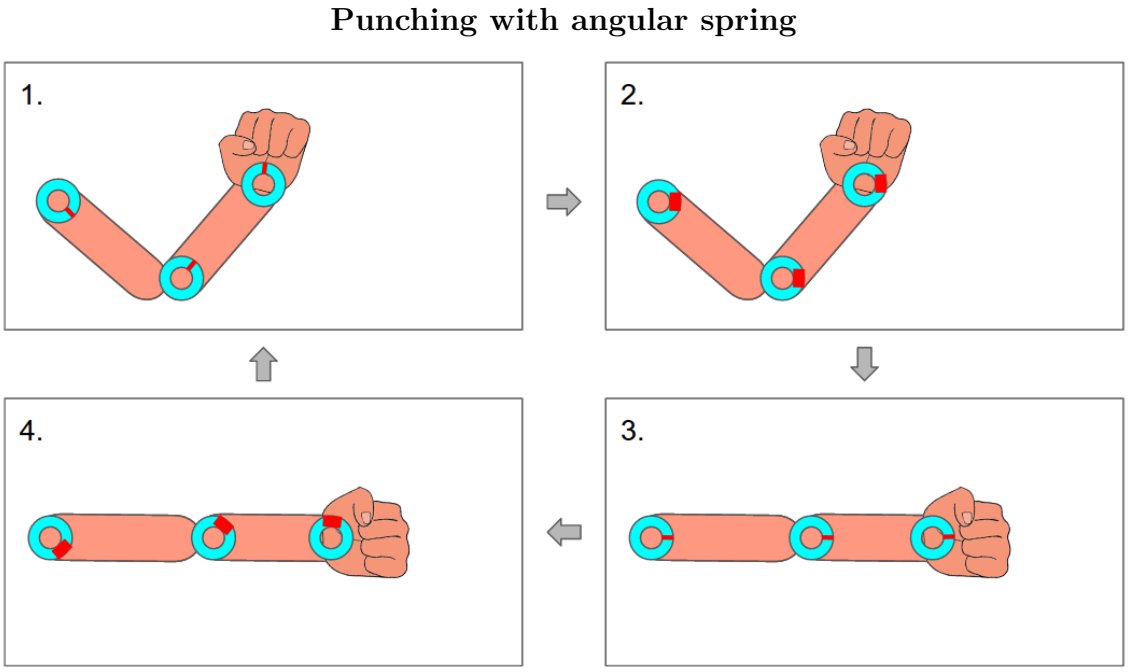


Figure 4.4: The cycle of a punch using angular springs. The springs' rest angles changes, making the arm go from bent to fully stretched. After it has stretched, the angles reset, making the arm return to its original pose.

4.3 Balance and Locomotion

As described in the Theory chapter, balance and locomotion are important parts of physics-based procedural animation. In order to achieve both of these aspects, a number of prototypes were constructed and experimented with. The implementation of these are described here.

4.3.1 Muscles

In one prototype, inspiration was taken from biomechanics to implement the muscles for moving a character. Each character is equipped with muscle points on its limbs. Each muscle point has a corresponding muscle point on an adjacent limb, and these pairs of points constitute the abstractions of muscles. We define a muscle as a class with muscle points and muscle force as parameters. This muscle have methods for contraction and extension. To contract the muscle, a force is applied to both muscle points towards each other, as shown in figure 4.5. To extend a muscle, the forces are instead directed away from each other. This ensures that each applied force is accompanied by a counter-force in the other direction, and so no net force is ever

applied to the system. This is important in order to avoid unnatural behaviour where a limb could be drawn away by an outward force. To balance a character's leg, a PD function is first evaluated, as described in section 4.3.2. To reach the target position and angles of the leg, micro-adjustments are done by contracting or extending muscles in every time step until a satisfactory stance has been achieved. The desired stance in this case is one where the rotation of every part of each leg is zero, meaning the thighs and shins are completely vertical while the feet rest flat on the ground. This was first done with a single limb, a shin, planted on a rectangular foot. After this was implemented satisfactory, a second limb, the thigh, was attached above the shin. The thigh had the same script and muscle points corresponding to those of the shin. Finally, a torso and a second two part leg was added to create a simple bipedal creature. If more dynamic movements are desired, for example the lifting of a leg to move forward, the muscle with points on the front hip and front upper thigh can be contracted. Since the contraction will put two equal and opposing forces on the two connected bodies, the body with the lowest mass will move more towards the body of higher mass than vice versa. This means that to get the desired behaviour of a lifted leg, it is important that the upper body has a significantly larger mass than that of the leg.

Muscle points

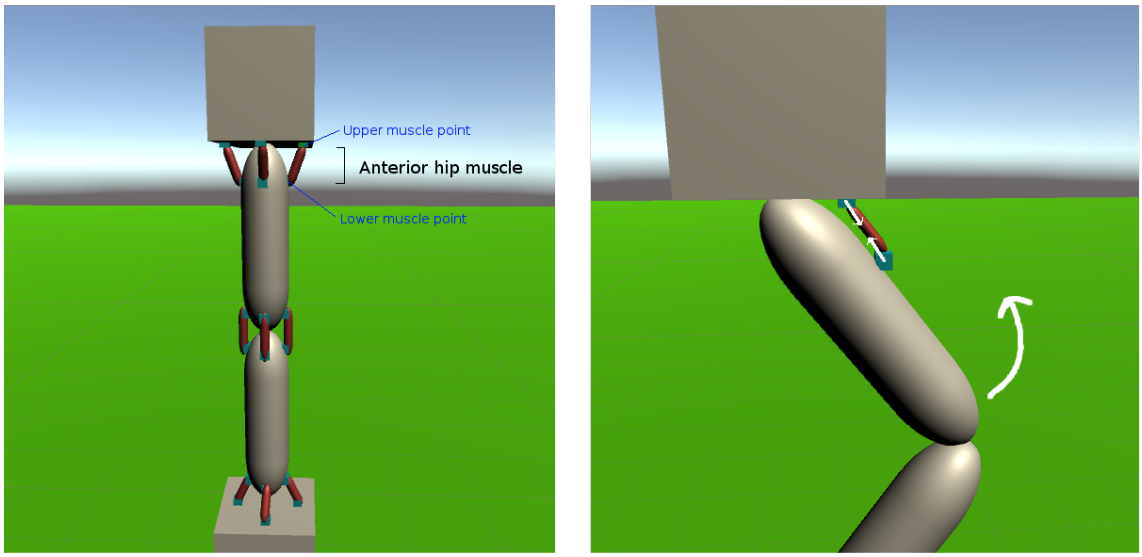


Figure 4.5: The left figure illustrates muscle points and the imagined muscles that go between them. The right figure illustrates the lifting of the thigh when its muscle points are pulled together to simulate the contraction of the muscle.

4.3.2 PD Controller

In order to know how much each muscle should pull in each direction one must know the state of the limb. At first this was done using an if-statement which simply checked if the difference of the limb's rotation and its target rotation, the angular error, exceeded a certain value. From there, the suitable contractions of the different muscles were calculated. As a potential improvement to this if-statement

a PID-controller was implemented and experimented with.

It was decided early on that the integral part was to be ignored in order to simplify the implementation of the controller. This meant that it was going to be a PD-controller.

The PD-controller was implemented using the angle as the only input. The derivative part was calculated by using the previous input and the time step. A sine function was used in order to lower the force as the angular error approached zero and it was continually used throughout the project while being improved upon. Most of the improvements made were parameter tuning, but another improvement was for it to work with any angle in 3D. The PD function that was used can be seen in equation 4.2. The first term is calculating the force that should be used depending on the angular error, θ. The second term is calculating how much the force should be lowered depending on the angular speed. These terms are both multiplied by constants $force_0$ and k_d which were tuned throughout the project. Note that the force can become negative if the angular speed is high. This just means that the opposite muscle should contract to lower the speed so that the limb does not overshoot its goal angle to much.

$$force_n = force_0 * sin(\theta_n) - k_d * abs(\theta_n - \theta_{n-1}) \tag{4.2}$$

The PD-controller also was experimented with to use with torques of limbs instead of muscles. This was done by switching the contraction of the muscle into a torque of the limb such that the limb rotated in the same direction as it would have with the muscle's contraction.

4.3.3 Character Movement

The center of mass of a character's body is constantly evaluated. To check whether the character is currently well balanced, the concept of a body's support polygon is taken into consideration. The support polygon in this case is the two-dimensional rectangle spanned by the edges of the character's feet. If the body's center of mass is contained within its support polygon, the character can rely on its PD-controller to retain or regain its balance. If the character's center of mass is pushed outside the support polygon, action must be taken to regain equilibrium. The two possible choices are to either move the character's center of mass using the PD-controller, or to adjust the support polygon itself. An unrealistic muscle strength would be needed to recover from having the center of mass too far outside the support polygon, so moving the center of mass is not the preferred alternative. Instead, we opt to move the feet to make a new support polygon which accommodates the displaced center of mass. As seen in figure 4.6, when the character's center of mass gets located outside the edges of the toes, a calculation is made of which foot is furthest away from the body's center. This foot gets a new target position calculated, which is of equal length from the body's center of mass as the other foot. The new target position is on the line that passes the back foot and the body's center of mass, so that the center of mass is now between the feet. To move the foot to this new position, the foot is connected with a spring joint to a point halfway towards the target position, with a higher Y-coordinate. When the foot reaches this halfway point, the joint

connection is removed, and the foot is allowed to fall. This brings the foot at or close to its target position, and the body's center of mass is now yet again inside the character's support polygon. If a constant forward force is applied to the character's body, it will keep taking steps to regain its balance, which will result in walking.

Walking method using center of mass.

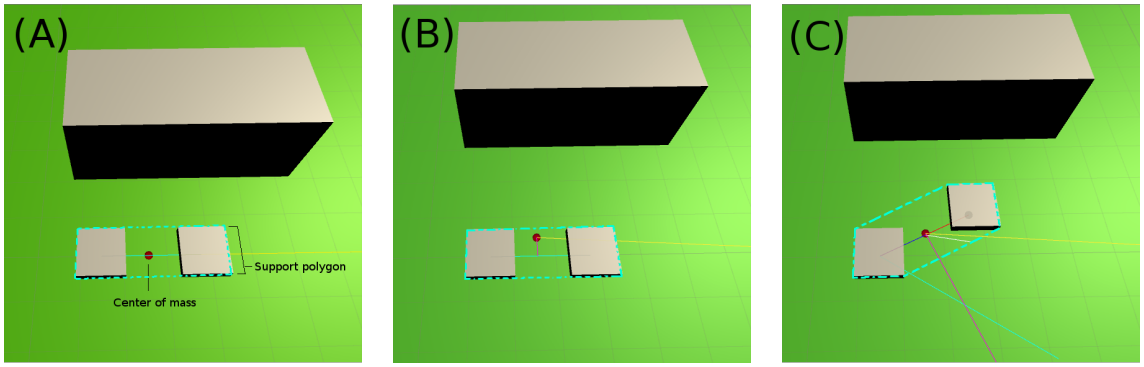


Figure 4.6: A depiction of the three different stages that are important during the progress of taking a step. (A) shows a balanced state where the center of mass is between the feet. (B) shows the center of mass approaching the edge of the support polygon. This is where a step is necessary to regain balance. In (C) the center of mass has been placed outside the support polygon, and a step has been taken to yet again include the center of mass inside the support polygon. The PD-controller can now be used to retain balance.

4.4 Randomized Characters

The player characters are generated by combining body parts from three custom made character models; a human, a horse and a T-Rex. These models provide somewhat different characteristics which makes the generated characters less predictable and requires the animation algorithm to be able to handle both bipedal and quadrupedal movement while also dealing with various imbalances. The models can be seen in figure 4.7.

Character Models

Figure 4.7: The three character models; a T-Rex, a horse, and a human.

The models are built up from several sub-models, one for each body part, which are connected with joints. In order to recreate the body parts at run-time with existing joints and with parameters like mass already adjusted, they are all saved as prefabricated objects. In Unity, a prefabricated object is called a prefab. Each body part prefab contains the mesh[1], joints, muscle points and sockets that represent the location of where to connect other body parts. Along with the mesh there is a Rigidbody component[2] which is used for physics simulation in Unity, a mesh collider, and a shader component.

The characters were all created using Blender and imported into Unity.

4.5 Destruction of Virtual Objects

In order to create destructible environments, objects need to be modified or replaced as they collide. Depending on the objects colliding, different results are more suitable. In some cases, shattering might be the best option, and in other cases some kind of slicing effect is preferred.

The way you define the objects that remain after destruction occurs can change the realism of the result substantially. One could just set a fixed number of arbitrary shapes and lay them out in a way so that they match the shape. In this case, as soon as the pieces fall apart, the flaws of the method become very clear as the pieces does not look as if they could be put back together to form the original object. To make sure this does not happen, the total volume of the pieces should be exactly that of the original object.

4.5.1 Cuboids

To achieve shattering of an object, an algorithm using cuboids was implemented. As cuboids are standard components available in unity, no custom models were required. The algorithm simply removes an object and creates a number of cuboids in its place as it collides with some other object.

An extension of the algorithm was created were the original is always replaced by exactly eight equal cuboids. The new cuboids are all instantiated with the same behaviour, making them fall apart further on collision. An exception was added, making sure the new cuboids colliding with each other does not trigger the algorithm. This creates a recursive behaviour where the original object breaks into smaller and smaller pieces, only in the specific local point where the collision happens.

In theory, cubes could be used to simulate the destruction of any shape, as long as the cubes are sufficiently small. This is proven by the advancements made in the field of computer graphics where so called voxels have been used to display a number of complicated shapes as far back as the 1990's[14].

[1] A mesh is a collection of polygons defining the surface of a 3D object

[2] `https://docs.unity3d.com/ScriptReference/Rigidbody.html`

4.5.2 Shattering

A way to make the destruction of any shape look realistic is to predefine the way it falls apart. By creating two separate models for each object, one that is intact and one that is not, all objects are guaranteed to fall apart in realistic ways.

This was achieved using the modeling software Blender. Blender can be used to break objects in more advanced ways. Some of these include soft body destruction, fracture destruction and molecular destruction[15]. None of these mentioned methods are meant for use in real-time, however, and have to be pre-calibrated in Blender. Blender also has build in algorithms for creating random shattering within existing meshes. One of these algorithms was used to create separate models for the same objects to switch between on destruction.

This shattering method is not purely procedural since there is only one predefined shattering, and the resulting shrapnel is going to look identical no matter what. The only thing that will separate the results is the effect of the friction on the ground and how hard the object got hit.

```
OnCollisionEnter{
        Instantiate prefab with same position and rotation;
        Destroy old game object;
}
```

The way that the code works is, as listed above, by instantiating a completely new prefabricated object, containing all of the destroyed pieces in the same shape as the previous one in the same position. The trigger of the event is any collision made to the object. At the same frame as the new prefab is created, the original game object is removed to create a seamless transition.

4.5.3 Slicing

Like with shattering, slicing of a soft object with a sharp one could be achieved by predefining meshes to replace the sliced object with. However, in this case, since there are only two new pieces, if the slice does not happen where the sharp objects hit, the realism is lost. To be able to replace the object with two predefined pieces when sliced, there would either have to be a large number of possible replacement pairs which are mapped to points and an angles of slicing, or a limited number of possible points and angles at which the soft object can be sliced.

To avoid either of these alternatives, we can instead calculate new meshes based on the original mesh, the point of collision and a plane parallel to the edge of the sharp object, as seen in figure 4.8. This creates two new meshes which can be applied to two separate copies of the original object, making all features of the original transfer to the halves. If any features are not supposed to be on both halves, the objects can easily be modified after the slice is performed. A common feature that needs modification after a slice has been made is the mass of each new object.

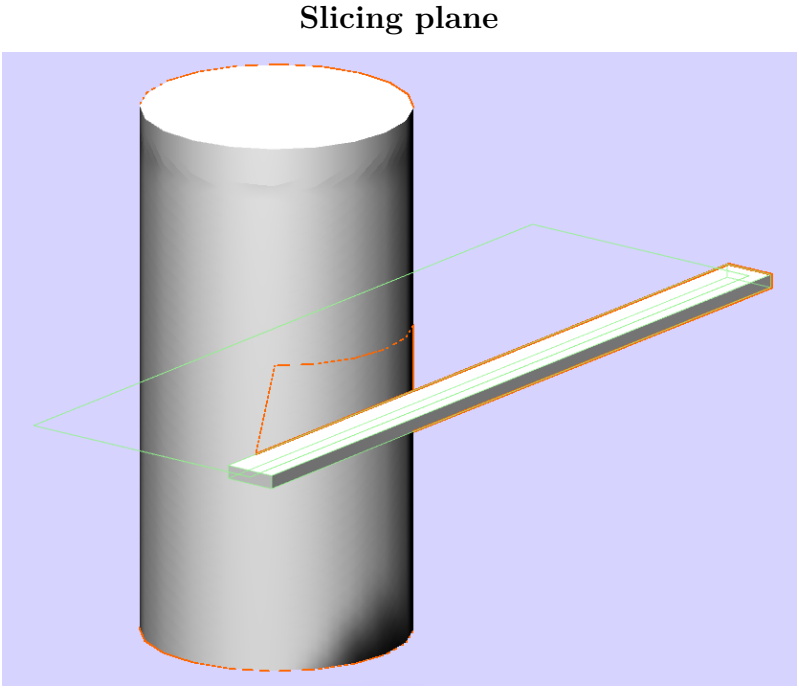


Figure 4.8: The new pieces of a sliced object can be calculated by searching a plane parallel to the edge of the slicing object. The plane, in this example, is marked in green.

Turbo Slicer 2 is a downloadable unity asset containing scripts for real-time slicing of meshes[3]. It works as described above. It looks for closed polygons within the cross section of the sliced area and fills them in with the material of the original sliced object. After a slice has been made, the outline of the two new objects, used for collision detection, can be recalculated if desired. Any other features can be modified after the slice by creating own components based on templates from the asset.

This method can be quite computationally heavy but Turbo Slicer solves this by detecting collisions on the unity main thread, converting the mesh to be sliced to its own internal working format and adds the slice to be made to a queue. Once per update, one slice is picked from the queue and executed. This way, only one slices is computed at a time, meaning some slices may be delayed but frame rate is never affected.

This kind of slicing, where the new models are calculated entirely based on situation, can be considered completely procedural, as both their generation and animation is procedural. Neither the created pieces nor the way they fall down will be the same between different executions.

4.5.4 Joint Removal

When it comes to character models, a well suited way for destruction is to destroy the joints holding the character together. This may not produce the most realistic

[3]Turbo Slicer 2, `http://noblemuffins.com/?page_id=452`

result as, for example a slice to the forearm removes the joint in the elbow instead of cutting the forearm itself in half, but it is a very efficient way to achieve a visually plausible result.

This method would also be well suited for objects such as brick walls, that are naturally constructed from smaller objects and joints, that are supposed to fall apart in a certain way but not all at once. The only problem with this becomes apparent is when it is implemented in unity. Joints in unity only goes in one direction, from one object to another. This means some bricks in the wall would not be able to break loose without violating object oriented principles stating that two objects should not contain references to each other.

5

Results

In this chapter, the results of the conducted experiments are presented as well as an overview of the final game scene implementation.

5.1 Genetic Algorithm

After running the genetic algorithm for approximately 3500 iterations, the performance of the quadrupeds improved marginally. The quadrupeds of the final iteration moved a little bit further than initially, but still not far enough to give a usable result.

5.2 Punching

There were two methods tested for implementation of punching in the game, regular springs and angular springs. With regular springs, no matter how stiff the spring holding the fist at its normal position was, the elbow was still very loose. Because of this, moving the body resulted in the elbow swinging and hitting the character's torso and sometimes its head. It did not produce a realistic result. In order to correct this, the stiffness of the elbow was increased, resulting in a slower punch. This implementation also resulted in the fist overshooting its desired positions.

With angular springs, the punching followed the same general motion as with the regular springs, but the loose elbow was no longer an issue. With relatively high spring constants, the arm was stable in its normal position, while still punching at realistic looking speeds. Some fine tuning of the rotation of the lower and upper arm, as well as the joints of the shoulder and the elbow, was required in order to achieve the best looking result, as shown in figure **??**. The final game scene uses this implementation.

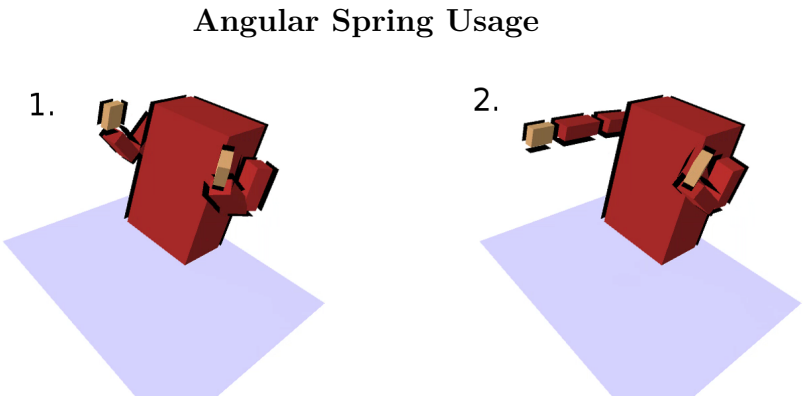


Figure 5.1: The right arm uses angular springs to move from its starting position in 1., to its extended position in a punch, in 2.

5.3 Balance and Locomotion

5.3.1 Muscle Points

Most of the time the muscle abstraction looked realistic. The muscle points made the characters unstable, however.

5.3.2 PD-Controller

Using a simple if-statement to check how to adjust the angular error worked, but was severely lacking in smoothness and did not look good visually.

Using a sine function to lower the force as the angular error approached zero yielded very good results right from the start.

Using torques instead of muscles together with the PD-controller yielded as good results as with the muscles.

5.3.3 Walking

The result reached when pushing a character and having it try to keep its balance, is similar to human walk. The character continues to put the next foot forward in its attempts to not fall. This result is utilizing the muscles and PD-controller together with knowledge of the character's center of gravity and how its legs should be placed in order to achieve a balanced pose.

This result is physics-based, but not entirely realistic since the start and end of the walking process together with the placement of the feet are using external forces. This is illustrated in figure 5.2

Taking step with spring joints

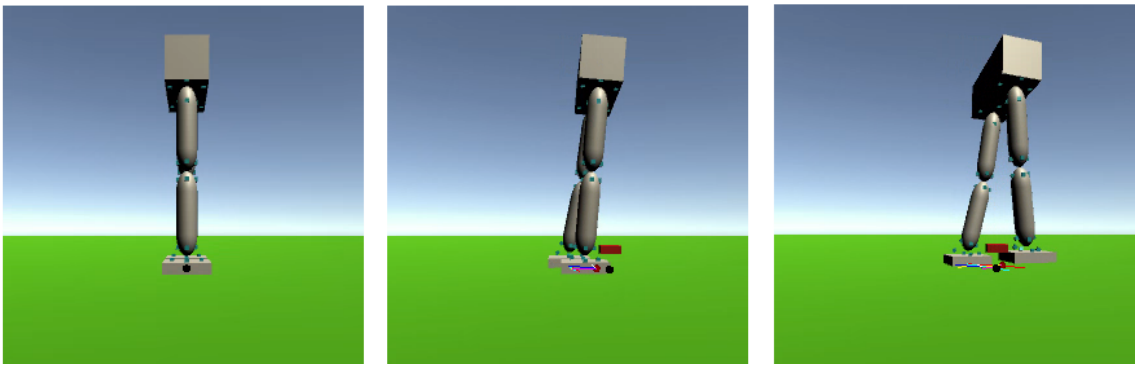


Figure 5.2: A force is applied to the bipedal's back (1). When its center of gravity is outside its support polygon, a new position is calculated for the right foot (2). The right foot is pulled towards the red marker halfway to its new position and is now taking a step (3).

5.4 Randomized Characters

Because the animation algorithms should hold for any character, randomization of characters was implemented. The randomization consists of an algorithm picking random parts from three different characters, a horse, a T-Rex and a human.

The three custom made characters all consist of the same set of body parts; head, chest, upper arms, lower arms, hands, upper legs, lower legs, and feet. In addition, the T-Rex and horse characters also include a tail.

The body parts are combined in a semi-random way to form the player characters. In order to find the places where joints need to be placed for the character to hold together, the joint positions were stored in the prefabricated body part objects. The joints were placed in the correct locations and the characters held together. The result can be seen in figure 5.3.

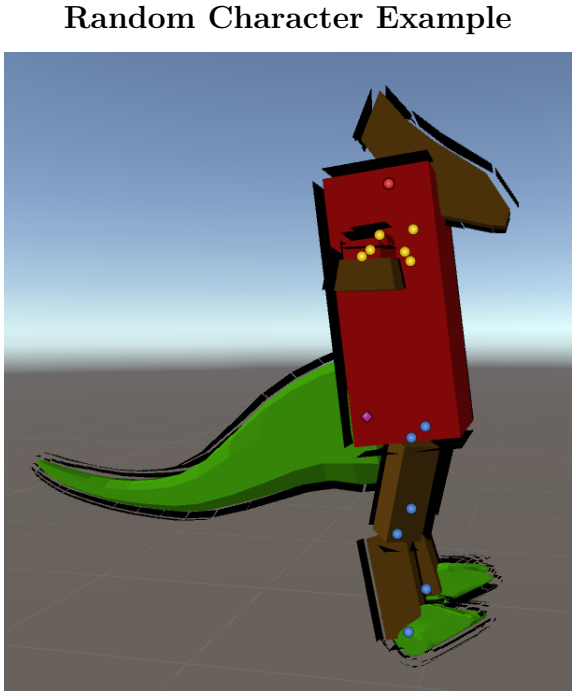


Figure 5.3: Example of a character put together using random body parts

5.5 Destruction

Some parts of the environments in the game are destructible. There is no single algorithm for destruction which is well suited for all possible scenarios but the method implemented in the final version of the game is swapping the object with a prefab containing the shattered version of the same mesh. Because the new meshes that spawn when the original is destroyed are effected by gravity and impulses from collisions, each individual destruction looks different from the next.

5.5.1 Cuboids

The cuboid shattering algorithm worked as intended. The first version instantiated a large number of cubes at once, resulting in some performance issues. The version that made eight cuboids at a time worked better in this regard. Because performance become an issue when there were too many shattered objects in a scene, the number of pieces were limited by removing the oldest ones when the total number of pieces reached a certain limit.The results of the second version can be seen in figure 5.4.

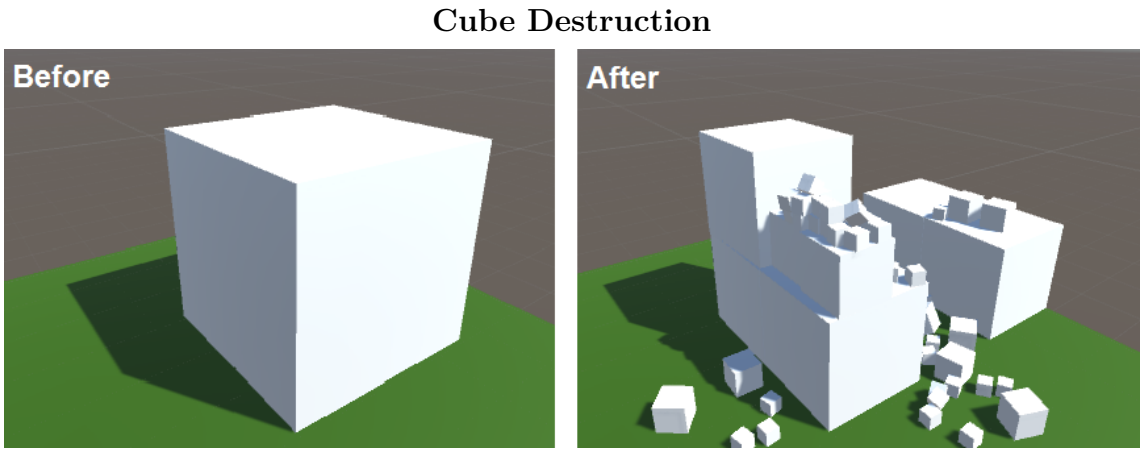


Figure 5.4: The cube on the left has not yet been hit with an object, while the right one shows how the cube breaks into smaller cubes depending on the proximity of the collision.

5.5.2 Shattering

The shattering algorithm worked as intended. No performance drop was experienced during the experiments. The results can be seen in figure 5.5.

In the game, when a character or other object collides with a destroyable object, The Unity physics engine takes into account the momentum of the colliding object and automatically applies an impulse at the point of collision. This causes the shattered pieces to spread in a physically realistic way. It is also possible to manually add and adjust additional impulses to achieve certain effects such as explosions.

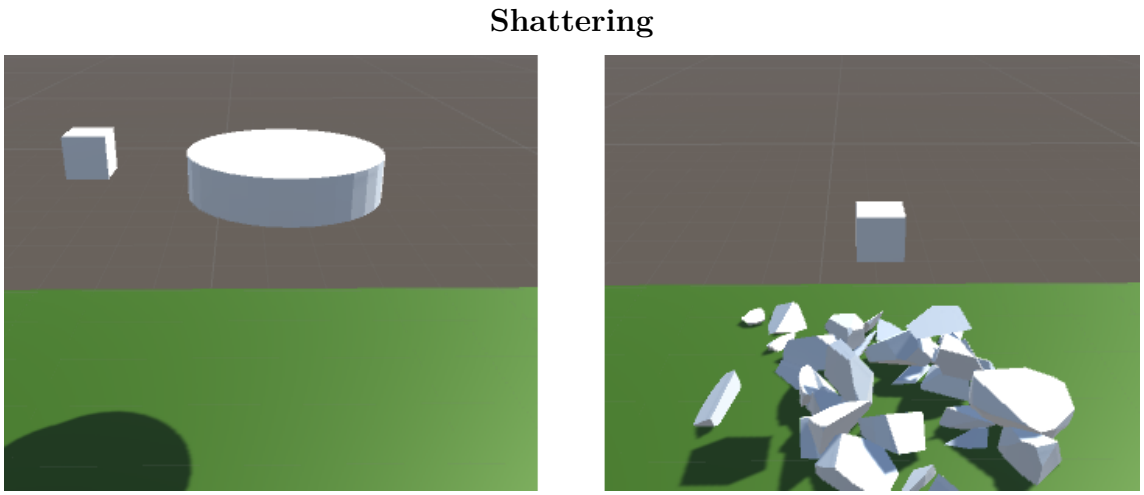


Figure 5.5: The left shape in the figure has not yet been subject to a collision, while the right one shows how the shapes can look if you predefine a shattering result.

5.5.3 Slicing

Slicing with Turbo Slicer 2 worked without noticeable performance drop. Some test where stationary cubes laying on the ground where sliced by a flat cuboid representing a knife were executed and worked without any problems. Other scenarios with spheres connected by joints to form a pendulum where also subjected to the blade. With an extension of the algorithms, even objects these objects that were connected to each other with joints could be sliced. The results of one of the slicing experiments can be seen in figure 5.6.

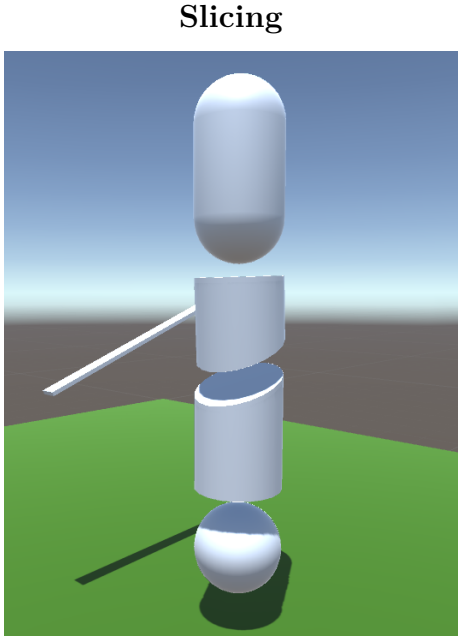


Figure 5.6: A pendulum after the execution of a slice. Both the joint connecting the upper part of the sliced cylinder to the top sphere, and the joint connecting the lower part to the bottom sphere are still intact.

5.5.4 Joint Removal

Because joints in Unity are defined to be from one character to another, the joint removal algorithm only worked when the body part on the correct side of a joint collided. In cases where this was desired, the algorithm worked well.

5.6 The Game

After all separate experiments had been conducted, a few of the investigated techniques were merged in a fighting game scene.

Walking in the game uses only kinematic animation while the upper body is completely physics-based. This is achieved by a connection from the upper body to an invisible box beneath it. The legs are not collision enabled at all, making them able to move freely within the box, while other objects cannot collide with them. By

fine tuning the friction constant of the ground and the box surrounding the lower body, a smooth movement is achieved.

Walking frames

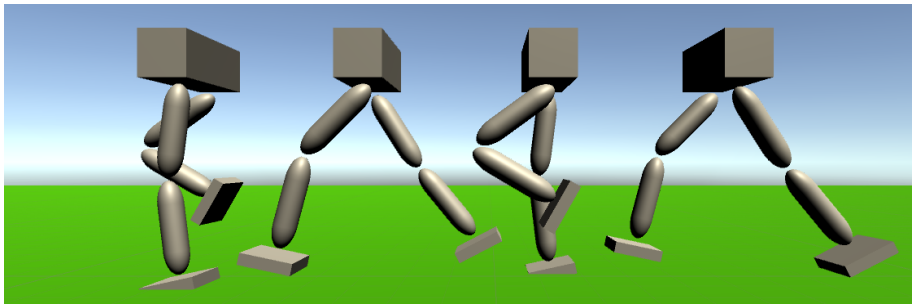


Figure 5.7: The walking frames of a character, placed side by side. These are interpolated between during run time.

Two randomly generated characters are spawned into an arena and needs to knock the other character to the ground in order to win. Once a character falls and its head touches the ground, the player loses control over the character which turns into an ragdoll. The character left standing is the winner.

Although the visual aspect of the game was not a priority, some work has been put into making the scenes look clear and the different assets easily distinguishable. The environment has a cartoon inspired look and for the characters a specific shader was used to better highlight the various body parts.[1]

For the skybox and surrounding landscape, meshes and materials from another asset[2] were used.

Arena Overview

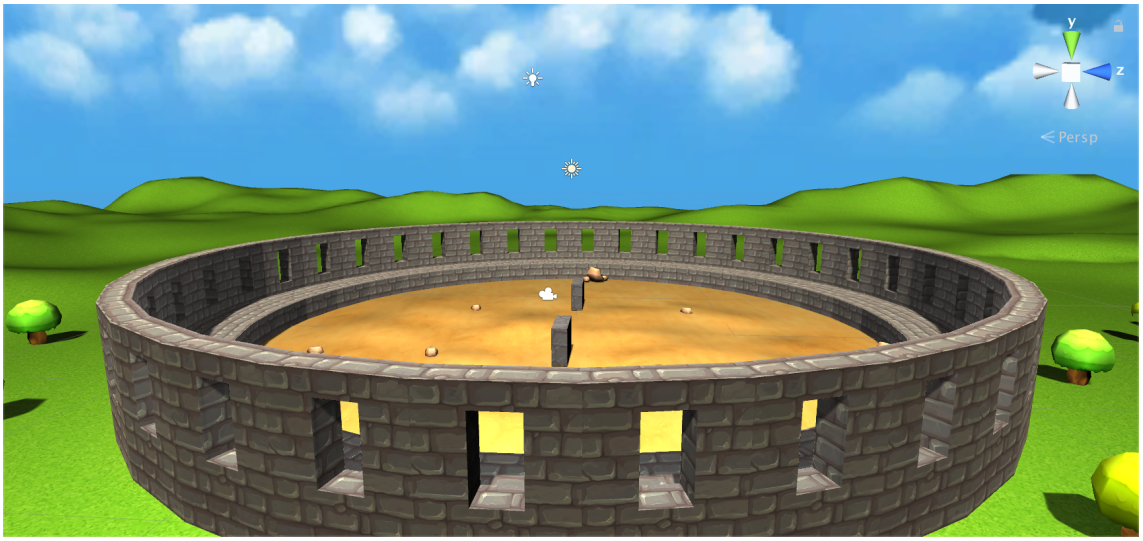


Figure 5.8: An overview of the arena and surrounding landscape in Unity

[1]Toon Shader Free (https://www.assetstore.unity3d.com/en/#!/content/21288)
[2]Fantasy Skybox FREE (https://www.assetstore.unity3d.com/en/#!/content/18353)

6
Discussion

A lot of new concepts were introduced during the duration of the project. The concept of muscle points contracting and extending from each other using forces were unexplored, according to our search of popular procedural methods, prior to our project. The concept of randomizing body parts in the extent that we are doing is also seldom used, and was an exciting area to explore.

Many techniques were investigated and tested but only a few made it to the final implementation. Because the tested techniques in punching, balancing and destruction were implemented in a very simple environment parallel to the implementation of randomly created characters a lot of unnecessary complication appeared when everything was put together in a final scene.

Since not so many games are using procedural animation to a greater extent, this is a fairly unexplored area in the gaming community compared to traditional animation. There is academical research being done outside of the gaming industry but little of it is intended for use in real time.

While the task was clearly stated at the start of the project, we expanded in ambition to something that would be better suited for a master's thesis and had to revert to the initial project after about a month. This made us lose time and made the end result lacking in some aspects.

6.1 Genetic Algorithm

If the bad results of the genetic algorithm was because of the character model not being good enough, the genetic algorithm not being well implemented, bad starting parameters or something else, is not certain.

Since the movement of the characters improved marginally during a large time frame perhaps the movement of the characters would become regular walking after an even larger time of running the algorithm. This is not certain, however. Something that would significantly speed this up would be to increase the number of quadrupeds per iteration. This was not possible in our case because this would exceed the capabilities of the computer used. With a better computer this would be possible, though.

To transform the combinations of values, which were generated for every individual between each iteration, into movement turned out to be a rather difficult and complex problem. This coupled with the realization that optimal movement is not necessary to have a realistic motion and the weak results yielded by the testing of the genetic algorithm led the project away from trying optimization techniques.

Note that the two optimization methods that have been investigated, particle swarm optimization and genetic algorithms, can potentially be used to great effect in procedural animation of a character.

6.2 Punching

A punch delivered by one character, hitting another, forces the hit character back. Because both bodies are physics-based, the punch also pushes the punching character back a bit. The scenario looks adequate no matter at which distance the characters are from each other when the punch is delivered.

While working with the regular springs, one character punching another character in this implementation, means the spring drags the fist of one character towards a point inside or behind the other character. Because the spring is not removed until the fists hits its target rigid body, the other character is pushed away from the target. In most cases this looks good, but sometimes it causes problems as the region can be in some unreachable position. An example is the rare case where both characters are standing very close to each other and attempting to punch each other at the same time. They are then stuck in a position where neither is able to reach their attempted target but they are dragged towards each other.

The angular springs however, did not encounter big problems since they returned to the starting position after a certain time. Since you can not get stuck as easily, this is an objectively better way to apply punching.

6.3 Character Balance

The balancing of a predefined bipedal character was successful. However, using the same general method for all possible combination of randomized characters, did not yield desirable results. More work would need to be placed on optimizing the method.

A PD-controller was implemented, both using muscle points and torque. In its simplest implementation, torques gave a smoother result, but adapted to a more complex character, the forces of the contractions of the muscles were easier to work with and to tweak. This was the reason why muscles were used instead of torques together with the PD-controller.

6.4 Character Randomization

The randomizing of the characters different parts and adjusting their weight accordingly turned out to be more complex than expected. Since some of the legs were smaller than the other, for example the T-Rex and the Human, the PD-controllers would have to be optimized on the basis of this. For the randomized legs to work properly, several different sets of parameters for the PD-controller script would have to be implemented, and would be a project in itself. Since only a couple of combinations of legs actually worked, we settled for a single set of legs and randomized

only the upper body.

At first, all parts were chosen fully randomly and independently of each other but this lead to difficulties with balance and punching movements. For example trying to find the right parameters proved to be more difficult when the characters legs were of different length. To remedy these problems, only a certain combination of body parts are allowed in the final version of the game. For example, the left and corresponding right body parts are of the same type.

6.5 Cuboid Shattering

Both the cuboid shattering algorithm in its original form and the extended, recursive version were implemented and tested for various shapes. The result of a collision is not very realistic but has some artistic qualities. As for performance, the splitting works fine until you get to a cube count of around 4000. Because the shrapnel is never removed, and each cube is subject to gravity, the computations in the physics solver became too many at that point. This was solved by setting a limit to how small cubes can be split. Because it has do with the number of cubes and not the size of the cubes, this performance drop would be the same if a number of cubes were split a few times each as well. An alternative solution to the performance problems could be to start despawning cubes, or removing physical attributes like mass and collisions from them, when approaching the maximum count. However, making this seem realistic or plausible will also be a challenge.

An issue with this is that the number of cubes can become too many for the physics engine, causing lowered quality of the game. This number was around 4000 for our implementation of the algorithm.

6.6 Shattering

An argument against this method is that each model can only shatter in one specific way, no matter from which angel or where it is hit. This could be countered by making separate shattered models to match each potential scenario. At some point, though, this removes the benefits of procedural animation as you essentially have to model each potential scenario by hand, even the ones that may never happen.

To use this method on a shape that is not a cuboid could be superior in speed and looks, whereupon the weighing would be between realism or following the strictness of our goal to implement procedural animation.

Creating a lot of predefined scenarios for how an object can break will create the illusion of the object breaking at the correct spot but would take an immense amount of time to complete. This technique is great when used on small objects though, when the exact location of the scattering is not vital.

6.7 Muscle Points

Using simulated muscles to punch and walk is potentially a good idea, but eventually this turned out to make the characters unstable. It is possible this is due to how physics and collisions are handled in the Unity engine. When the physics solver does not find a solution that satisfies all the constraints given, it outputs a bad solution, causing instabilities.

There were a lot of muscle points, many of them trying to pull a limb in a certain direction. This created a lot of different constraints that had to be satisfied. A possible way to stop the instabilities is to make the physics constraints softer so that the physics engine always finds a solution. This would of course lower the realism of the game, but if it would look visually good it would absolutely be worth it.

These muscle abstractions continued to be used, however, because of a lack of time to implement an alternative to them.

6.8 Potential Improvements

The project could have been done in 2D instead of 3D. It is still not apparent if all of the procedural applications can be made in a 2D environment, but several of the problems such as turning the character would not have been an issue.

The muscle points were perhaps not necessary since they could easily be replaced with the pre-implemented hinge joints or angular springs. Our initial thoughts were that the only options of twisting a joint were to use forces or a torque. Since we did not want the body parts to rotate in the middle, but rather rotate on the joints, our new way of rotating seemed like a genius idea and took too much time, creating functionality that was already available.

6.9 Future Work

The field of procedural animation is very broad and contains a lot of interesting concepts. During our research we have touched upon many techniques that would have been very interesting to dig deeper into, if we had more time. One of the most interesting ones, is the technique which uses contact invariant optimization for animation of physics-based characters. In this project, the technique was never implemented as it is not meant to be used in real-time and is quite slow. However, the concept of focusing on foot placement for moving a character was used as inspiration for walking. There were ideas of combining the technique with the concept of Position Based Dynamics in order to make it run faster. The slow computation time of CIO is due to it's mathematical optimization of physics and collisions. Position Based Dynamics uses simple constraints based only on positions to make estimations of physics which can be computed very fast. A combination of these two techniques could solve the problems with the balancing of a character encountered in this project and also bring in more complex movements than just walking and punching.

7
Conclusion

Procedural animation is a good technique for avoiding manual labor within the field of animation. Good algorithms for movement of objects and characters can be reused on different characters eliminating the need to repeat the animation process.

In this thesis, a number of techniques for characters movement as well as for destruction of solid objects in a physics based environment have been investigated. The results were adequate but the implementation was more difficult than expected in most cases. Despite this, we see great potential in the field of procedural animation applied to video game development.

Bibliography

[1] T. Akenine-Möller, E. Haines, and N.Hoffman,Real-time rendering. CRCPress, 2008.

[2] B. Lautrup, "Hooke's Law" in Physics of continuous matter: exotic and everyday phenomena in the macroscopic world, 2nd ed. Hoboken, CRC Press, 2011.

[3] L. Barinka, R. Berka, "Inverse Kinematics - Basic Methods", 2002.

[4] T. Geijtenbeek, M. van de Panne, and A. F. van der Stappen, "Flexible muscle-based locomotion for bipedal creatures", ACM Transactions on Graphics, vol. 32, no. 6, 2013.

[5] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distribution in evolution strategies: The covariance matrix adaptation", in Proceedings of IEEE International Conference on Evolutionary Computation, May 1996, pp. 312–317. doi:10.1109/ICEC.1996.542381.

[6] Colin R. Reeves and Jonathan E. Rowe , "Genetic Algorithms—Principles and Perspectives", 2002

[7] J. Kennedy and R. Eberhart, "Particle Swarm Optimization, " in Proc. 1995 IEEE Int. Conf. on Neural Networks.

[8] C. Chiu et al, "Comparison of Particle Swarm Optimization and Genetic Algorithm for the Path Loss Reduction in an Urban Area," in Journal of Applied Science and Engineering, Vol. 15, No. 4, pp. 371-380, 2012.

[9] Kiam Heong Ang, G. Chong and Yun Li, "PID control system analysis, design, and technology," in IEEE Transactions on Control Systems Technology, vol. 13, no. 4, pp. 559-576, July 2005.

[10] M. Müller et al, "Position Based Dynamics," in Journal of Visual Communication and Image Representation Vol. 18, No. 2, pp. 109–118, April 2007.

[11] T. Jakobsen, “Advanced Character Physics,” 2003. Available: http://www.gamasutra.com/resource_guide/20030121/jacobson_pfv.htm

[12] J. Bender, M. Müller, and M. Macklin, “Tutorial: Position-Based Simulation Methods in Computer Graphics,” 2015.

[13] I. Mordatch et al, "Discovery of complex behaviors through contact-invariant optimization," Presented at SIGGRAPH Conference 2012 in Los Angeles, USA, 2012.

[14] A. Kaufman et al, "Volume Graphics," in IEEE Computer, Vol. 26, No. 7, pp. 51-64, July 1993.

[15] Gleb Alexandrov, "Blender Tutorial: 9 Ways to Destroy Things," YouTube, 2014. [Online]. Available: https://www.youtube.com/watch?v=wJFE2pb0Ri4&t=146s. [Accessed: 19- Apr- 2017].

[16] M. Fratarcangeli, V. Tibaldo and F. Pellacini et al, "Vivace: a Practical Gauss-Seidel Method for Stable Soft Body Dynamics," Presented at SIGGRAPH Conference 2016 in Macao, China, 2016.

[17] L. Bengt, "Dimensionering av PID-regulatorer," in Reglerteknikens grunder 4:9th ed. Lund, Sweden: Student literature, 2002, ch. 8.

[18] Blender HD, "Shattering Glass in Blender," YouTube, 2013. [Online]. Available: https://www.youtube.com/watch?v=zu6zbf7tdoU&t=766s (at 7:30). [Accessed: 19- Apr- 2017].

[19] D. Rosen, “Animation Bootcamp: An Indie Approach to Procedural Animation”, Available: http://www.gdcvault.com/play/1020583/Animation-Bootcamp-An-Indie-Approach, 2014.