University of Miskolc

Faculty of Mechanical Engineering and Informatics
Institute of Automation and Infocommunication

TDK thesis

Emergent Animation Systems: from interpolation to procedural motion

Author:
Konrád Soma Kiss
BSc Student in Computer Engineering

Supervisor:
Dr. Attila Károly Varga
Associate Professor

Miskolc, 2025

Contents

1 Introduction

When the term “animation” or “motion graphics” is mentioned, people often think of the process of creating animated films or cartoons. This usually refers to hand-drawn animations or, more recently, computer-generated imagery (CGI) and visual effects. However, the terms "animation" and "motion graphics" have much broader meanings.

As Adobe, an industry leader in digital media, defines it: "Motion graphics are essentially ’graphics with movement’." [1] This definition aligns more closely with the theme of this thesis. What are procedural animations? What systems can be used to create them? What are rule-based animation systems, and what emergent behaviors can be achieved that are unforeseen by the designer?

The premise is that everything that moves on the screen is animation, and everything that is animated can be considered motion graphics. This thesis will delve into the different algorithms, aspects, use cases, and applications of these systems, and how they are used throughout the digital world, even though we might not notice them at first glance.

The basic hypothesis is that we can create complex movements, behaviors, and animations by defining simple rules and systems. The less artistic control we have over the final result, the more we can rely on the system to generate interesting, natural-looking motion.

2 Background

To understand the concepts discussed in this thesis, it is essential to have a basic understanding of vectors, matrices, affine transformations, and some principles of computer graphics. An excellent resource for this can be found in the book "Learn OpenGL - Graphics Programming" by Joey de Vries [10]. This book provides a comprehensive introduction to the mathematical foundations of computer graphics, which are crucial for understanding procedural animations and motion graphics.

My recommendation is to read sections 8 through 10 of the book, which covers all the necessary mathematical elements required to proceed.

In addition, it is beneficial to have a basic understanding of programming, especially in TypeScript, as this thesis will include code examples and implementations in this language. I will try to explain everything in mathematical terms first, and then provide the code examples to illustrate the concepts. This approach will help bridge the gap between theory and practice, making it easier to grasp the underlying principles of procedural animations and motion graphics.

3 Methodology

There are some major measurement methods that I will use to evaluate the results of different techniques and algorithms. These methods will help in understanding the performance, repetition, and overall "feel" of the animations created by the systems. While some of these measurements are subjective, they will provide a good basis for comparison.

3.1 Deviation

Deviation ($\bar{D}$), as defined by me in these measurements, is a measure of how much the actual motion deviates from the expected or desired motion. It can be quantified by calculating the difference between the actual position of an animated object and its expected position at a given time.

I will define 1 as the maximum deviation, where the motion has no correlation with the expected motion, and 0 as the minimum deviation, where the motion is the exact same as the expected motion. I will not provide an exact formula here, as it will depend on the specific use case.

The expected position will usually be a linear interpolation between two keyframes, while the actual position will be determined by the algorithm used to generate the motion. This means that easing functions, procedural noise and any other factors that affect the motion will be taken into account. This statistic may be a bit biased towards stiff, robotic motions, as they will have a lower deviation from the expected motion. However, it is still a useful metric to evaluate the overall smoothness and naturalness of the motion.

3.2 Repetition

Repetition (R) is a measurement of how prone an algorithm is to repetition. This will be a value between 0 and 1, where 0 means the algorithm is purely chaotic, while 1 means the algorithm is completely deterministic and will always produce the same motion for the same input. Of course, due to the pseudorandomness of computers, we could achieve deterministic results for every algorithm; therefore, as an extra rule, I will always use a different seed for each run of the algorithm where randomness is involved.

3.3 Control

Control (C) is a measure of how much control the designer has over the final result of the animation. This is a subjective measurement, with these possible values: "none", "low", "medium", "high" and "total". This represents how much influence the designer has over the final result of the animation. A stiff motion from point A to point B would have a "total" control, while a physics simulation would have "low" control, as the designer can only influence the initial conditions and the parameters of the simulation, but cannot control the final result.

3.4 Feel

Feel (F) is another subjective measurement, which will be based on my personal experience and the feedback of others. It will be a value between 0 and 1, where 0 means the motion feels unnatural and robotic, while 1 means the motion feels natural and fluid. This will be based on my personal experience with the animation, as well as the feedback of others who have seen the animation. The pool will not be large, but I will try to get feedback from as many people as possible. This will be a good indicator of how well the algorithm works in practice, and how well it can create natural-looking motion.

3.5 Performance

Lastly, a purely objective measurement: performance ($P(...)$). This will be measured in ms, and will be the time the algorithm takes to calculate the position and rotation of the objects for the next frame. Every algorithm mentioned here will be tuned for real-time performance, so the goal is to achieve a frame rate of at least 60 FPS, which means the algorithm should take no more than 16.67 ms per frame (with leaving some room for rendering).

If other parameters, like the number of objects and such heavily influence this parameter, I will define an extrapolated function from many measurements. It is technically the "Big O" notation. It will always be a function defined by me, that tries to estimate the time it takes to run the algorithm for a given input.

3.6 Summary

Measurement	**Range**	**Example**	**Description**
Deviation	0 to 1	$\bar{D} = 0.2$	How much the actual motion deviates from the expected motion
Repetition	0 to 1	$R = 0.9$	How prone the algorithm is to repetition
Control	...	$C =$"high"	How much control the designer has over the final result
Feel	0 to 1	$F = 0.2$	How natural the motion feels
Performance	$P(...)$	$P(n) = O(n \cdot k)$	The time the algorithm takes to calculate the next frame (where n is the number of objects and k is a constant cost)

Table 1: Summary of measurement methods

I will provide a similar table for every relevant algorithm. There will be some algorithms that are included for the sake of building up the complexity, but cannot/will not be measured, as they are not relevant to the final results. In these cases, I will state that the measurement is not applicable (N/A).

4 Implementation

Before diving into the algorithms and techniques, it is essential to have a basic understanding of the implementations used in this thesis.

A really big point of this thesis is to show that we can create complex animations and behaviors with simple rules and systems. For this statement to be true, I recommend looking at every example as its own separate tiny implementation, disregarding my framework. The tools I created for these animations can be found in many other libraries and frameworks; the abstract concepts are the important part.

Note 1

My implementation is unnecessarily complex for most of the algorithms discussed in this thesis. It is designed to be a general-purpose framework that can handle various types of procedural animations and motion graphics. The complexity arises from the need to accommodate different algorithms, techniques, and use cases, as well as to provide a flexible and extensible architecture.

From this point on, I recommend having access to a device with a larger screen, preferably a computer or laptop. I also recommend reading the thesis in pdf form instead of printed form, as I will provide links to the `examples` on my website.

Motion graphics are inherently visual, and having the ability to interact with the examples and see them in action will greatly enhance your understanding of the concepts discussed.

Note 2

The code snippets provided here will be simplified, but the full implementations are available on my GitHub `repository` .

The implementation is built on top of the HTML5 Canvas API [7], which is purely used for rendering, dynamic frame timing [1], and some basic input handling. No external libraries are used for the examples. The website itself is built using Svelte [9], but that is not relevant to the thesis.

[1] A common practice in real-time rendering is to use the time between the current and the previous frame to calculate the delta time. Multiplying with the delta time allows us to create framerate-independent animations.

4.1 Architectural Overview

The implementation is structured as a modular system, reminiscent of a game engine. While it does not adhere to a strict design pattern, it is best described as a component-based, data-driven architecture utilizing imperative sequencing[2] for animation and rendering logic.

4.1.1 Canvas Wrapper

The canvas wrapper is a big, monolithic class that handles the rendering, input, timing, and some other animation related tasks. It in itself does not contain anything related to any algorithm, but provides an even blank slate for every implementation.

It defines a World Space using affine transformations, which allows easy positioning and scaling of objects with hard-coded values. It has a **resolution** property with default values of 1280x720, but it can be changed to any other resolution. It also has **scale** and **offset** properties, which are automatically calculated using the actual size of the canvas element and the **resolution** property. This allows us to always have a consistent coordinate system, regardless of the actual size and aspect ratio of the canvas.

The class also provides simple debugging tools like a frame rate graph and some information about the canvas. It stores and passes a reference to the canvas element and the rendering context to the algorithms, so any default canvas functionality can be used without any additional setup.

It also provides a simple UI object that allows us to create settings and panels without the need for HTML.

While the canvas wrapper is a big class, it in itself does basically nothing. The actual functionality is provided by the **canvas scene** class. Every example will have its own scene, which is a collection of objects and algorithms that work together to create the desired animation.

4.1.2 Canvas Scene

For starters, I will show the bare minimum you need to create a scene:

```
1  /* Imports */
2
3  export class MinimumScene extends CanvasScene {
4      // Define
5      override textures;
6      override objects;
7      override sequencers;
8
```

[2]This approach requires explicit definition of animation sequences and rendering logic/order to ensure correct functionality.

```
    constructor(
        wrapper: CanvasWrapper,
        context: CanvasRenderingContext2D,
        time: Readonly<Time>
    ) {
        super(wrapper, context, time); // Stored as protected fields

        // Initialize
        this.textures   = {};
        this.objects    = {};
        this.sequencers = {};
    }

    override* sequence(): Generator<void> {
        yield* [];
    }

    override render(): void {
        return;
    }
}
```

In its current form, it is completely empty, but it has clear definitions. First, you define all the textures, objects, and animators you want to use in the scene. In the **constructor** you initialize these objects and call the parent **constructor** (super). In the **sequence** generator function, you may define an animation sequence. In the **render** function, you can define the rendering logic, which will be called every frame. This usually involves calling `this.renderInOrder([...])`. This is another imperative part of the implementation, as you have to define the order in which the objects are rendered. The library has no automatic z-indexing or depth-sorting, so explicit rendering order is used.

Note 3

In this animation system, **generator functions** are used to define time-dependent sequences in a linear, readable way without introducing threads. A generator is a special kind of function that can be paused and resumed: when it executes a **yield**, control returns to the caller, but the function's local variables and execution position are preserved. On the next call, execution continues exactly where it left off.

This is ideal for animations in a single-threaded rendering loop. Each **yield** means "wait until the next frame before continuing", while **yield*** runs another generator as part of the sequence. The result is that complex animations can be written as straightforward step-by-step scripts, yet they still advance in perfect sync with the rendering loop and remain framerate-independent.

Alternative approaches like **async/await** would decouple the animation timing from the render loop, introducing unpredictability, while multithreading is not available in this environment. Generators provide a lightweight, synchronous, and deterministic solution.

While this syntax is not often seen or known, and may be too hard to read at first, it makes creating animation sequences in order much more straightforward. It does come with its own set of problems, like the fact that you cannot easily play two sequences at the same time, but I do provide some helper functions to make things easier.

4.1.3 Canvas Object

The rendering layer in this library is built around the `CanvasObject` class, which represents a drawable entity in a hierarchical scene graph. Each `CanvasObject` stores its own **position**, **scale**, **rotation**, **pivot point**[3], and **rendering function**, and may contain child `CanvasObject` instances.

Note 4

All angles/rotations in this implementation are in radians.

Note 5

This class is a prime example why this implementation does not follow a strict design pattern like Object-Oriented Programming (OOP). It's a data oriented design, with functional elements. The class is not meant to be extended, but rather used as a base for creating specific objects with their own rendering logic.

The `render` method applies the object's local transformation to the rendering context, invokes its rendering function, and then recursively renders its children. This enables relative positioning and transformations: when a parent is moved, scaled, or rotated, its children are affected accordingly. (Local Space transformations are handled correctly and implicitly.)

Rendering behavior is defined by a `RenderFunction`, which receives the object and the canvas context. This function can be supplied at construction time to customize how the object is drawn; common shapes (e.g., ellipses, rectangles) are provided in the `Objects` utility class.

The system also includes optional debugging visualizations (pivot markers and bounding boxes) and a deep-copy method for duplicating objects with or without their children.

This design provides three major benefits:

- **Modularity:** Any drawable entity is just a `CanvasObject` with a specific render function.
- **Hierarchical transformations:** Complex objects can be built from multiple parts with parent-child relationships.
- **Flexibility:** The drawing logic is decoupled from object state, making it trivial to reuse or swap render functions.

[3]The pivot point is the point around which the object is rotated and scaled. It is usually the center of the object, but it can be set to anywhere. This is useful for animations where we want to rotate or scale the object around a specific point, like a character's feet or hands.

5 Techniques

5.1 Functional animation

Functional animation treats an animation as a pure function from time to state. In this example we will create a simple animation of an object following a circular path.

Example can be found here

`https://kisskonraduni.github.io/emergent-animations/examples/functional-animations?tab=0`

I have defined two simple functions that set the value of an object's position over time using these parameters:

r : radius	α : starting angle
t : time	ω : angular speed (rad/s)

Assuming our object is rotated around the origin, or if necessary offset using the object's pivot point, or a parent object, the position $\vec{p}$ of the object at time t can be calculated as follows:

$$
\begin{aligned}
x(t) &= r\cos(\alpha + \omega t) \\
y(t) &= r\sin(\alpha + \omega t) \\
\vec{p}(t) &= [x(t), y(t)]
\end{aligned}
$$

Positive ω will rotate the object counter-clockwise, while negative ω will rotate it clockwise. Using these simple equations, we can create a simple animation for a clearly defined path.

Here is the code that implements this exact animation:

```
export class FunctionalAnimation extends CanvasScene {
    override objects = {...};

    constructor(...) {
        // ...
        this.objects = {
            parent: new CanvasObject(
                () => {},              // No render function
                new Vector2f(350, 360) // Position (animation center)
            ),
            circle: new CanvasObject(
                Objects.ellipse("red"),
                Vector2f.zero(),       // Position (relative to parent)
                new Vector2f(50, 50)   // Size (width, height)
            )
        };
        this.objects.parent.append(this.objects.circle);
    }

```

```
    params = { radius: 200, alpha: 0, omega: 2 * Math.PI };

    override render(): void {
        this.objects.circle.position = new Vector2f(
            this.params.radius * Math.cos(this.params.alpha + this.params.omega *
            ↪ this.time.now),
            this.params.radius * Math.sin(this.params.alpha + this.params.omega *
            ↪ this.time.now)
        );
        this.objects.parent.render(this.context);
    }
}
```

While animating an object along a circular path using these equations is straightforward, this approach has significant limitations. It is difficult to construct complex or precise paths, the resulting animation lacks adaptability, and synchronizing its timing with other animations is challenging. Nevertheless, this method can be extended to address some of these issues.

Note 6

As a sidenote, this animation can also be achieved using the parenting system. Instead of setting the object's position directly, its initial position is set to the edge of the circle, and the parent object is rotated. This produces a similar animation, although the entire circle will rotate as well.

Deviation	Repetition	Control	Feel	Performance
0.0	1.0	Total	0.15	$O(1)$

Table 2: Measurements for the functional animation algorithm.

5.2 Interpolation based animation

A more advanced and widely used approach in animation is keyframe interpolation[4]. In this method, the start and end values of the animation, as well as its duration, are defined. A simple equation then calculates the object's position at any given time. These animations can be started, stopped, and reversed at any time, and will always produce the same result for identical input parameters. This predictability makes them easy to control.

[4]Interpolation is a method of constructing new data points within the range of a discrete set of known data points. In animation, it is used to create smooth transitions between keyframes by calculating intermediate frames based on the defined keyframes.

The basic equation for **linear** interpolation is:

$$\begin{aligned} a &: \text{start value} & t_0 &: \text{start time} \\ b &: \text{end value} & t_1 &: \text{end time} \\ r(t) &: \text{result} & t &: \text{current time} \end{aligned}$$

$$\begin{aligned} \Delta t &= t_1 - t_0 \\ r(t) &= a + (b - a) \cdot \frac{t - t_0}{\Delta t} \end{aligned}$$

This equation produces a smooth transition from a to b over the interval $[t_0, t_1]$. It should be evaluated once per frame.

Note 7

In my implementation, I have created a simple **Interpolator** class that handles the interpolation for you. It takes care of the timing and the calculation of the result, so you can just use it to get the current value of the animation. Also, you just have to start it in the **sequence** generator function, and it will automatically update the value every frame.

A common question arises: how can we make the animation appear less mechanical? This can be achieved by using **easing functions**, which modify the timing of linear interpolations to create more natural motion. Easing functions can introduce acceleration, deceleration, and other effects that make the animation feel more fluid.

Note 8

I have implemented a simple **Easing** class that provides a set of common easing functions, like **ease-in**, **ease-out**, and **ease-in-out**. The basic syntax is really simple, so you can define your own easing functions easily. There are some basic rules to follow when creating valid easing functions: the function should take a value between 0 and 1 as input, and return 0 at 0 and 1 at 1. An excellent resource for common easing functions is [3].

The correct method to integrate easing functions into the interpolation is as follows:

$$\begin{aligned} \epsilon(t) &: \text{easing function} \\ t &: \text{current time} \end{aligned}$$

$$r(t) = a + (b - a) \cdot \epsilon\left(\frac{t - t_0}{\Delta t}\right)$$

Example can be found here

`https://kisskonraduni.github.io/emergent-animations/examples/functional-animations?tab=1`

Interpolation-based animation is one of the most widely used methods in computer graphics and user interface design. It is straightforward to implement, offers precise control, and produces smooth transitions. This technique forms the foundation of most animation software, game engines, and even simple UI frameworks. Furthermore, it serves as the basis for many advanced animation techniques.

Deviation	Repetition	Control	Feel	Performance
Defined below	1.0	High	0.8	$O(1)$

Table 3: Measurements for the interpolation based animation algorithm.

Depends on the chosen easing function The deviation for this algorithm is defined as the integral of the absolute difference between the linear function and the chosen easing function:

$$\begin{aligned} l(t) = t &: \text{linear function} \\ \epsilon(t) &: \text{easing function} \\ \bar{D} &= \int_{t=0}^{1} |l(t) - \epsilon(t)| \, dt \end{aligned}$$

5.3 Timelines and sequencing

Many practical animations are compositions of multiple steps rather than a single, fixed-duration function. Timelines and sequencing provide a simple abstraction for assembling such compositions.

A **timeline** is an ordered collection of animations (or sequences) that are played one after another. Timelines may themselves be composed and can be executed in parallel with other timelines so that multiple objects are animated independently. A **sequence** is a single animation described by a set of keyframes or an interpolation procedure; it represents one contiguous part of a timeline.

Note 9

In this implementation, each timeline is represented as a generator function, following the same pattern as the **sequence** generator. To run multiple timelines concurrently, you create separate generator functions for each timeline and pass their generator instances to a parallel runner. For example, you can yield the parallel helper (represented here as `Animator.runParallel(...)`) from the main **sequence** generator to execute multiple timelines at the same time.

Note 10

When defining sequences within a timeline, the provided **InterpolationSequence** class uses a per-keyframe **duration** convention: each keyframe specifies the time interval from the previous keyframe to itself, rather than an absolute timestamp. This approach streamlines both implementation and editing, as you directly control the duration of each segment. However, it differs from traditional absolute-time timelines, where each keyframe has a specific time value. If needed, you can easily convert between these representations: cumulative sums of durations yield absolute times, while differences between absolute times yield per-segment durations. This system was chosen for its simplicity and efficiency, but it remains flexible enough to support absolute-time timelines if required.

An example sequence can be found here

`https://kisskonraduni.github.io/emergent-animations/examples/functional-animations?tab=2`

and a more complex example can be found here

`https://kisskonraduni.github.io/emergent-animations/examples/functional-animations?tab=3`

5.4 Data-driven animations

There are animation techniques that rely on pre-recorded data rather than procedural generation. These methods can produce highly realistic results, as they are based on real-world motion capture or other data sources. However, they also have limitations, such as the need for large amounts of data and the difficulty of adapting the animation to different contexts.

A commonly used data-driven technique is motion capture, where the movements of a real person or animal are recorded and then applied to a digital character. This can produce highly realistic animations, but it requires specialized equipment and can be time-consuming and expensive. If the actor's size or proportions differ significantly from the digital character, the animation may look unnatural and may require a lot of manual adjustment.

Statistics are omitted for this algorithm, as it builds upon the interpolation-based animation technique discussed earlier. The only difference is that the keyframes are derived from external data rather than being manually defined.

5.5 Blend trees and directional blending

Blend trees and directional blending are techniques used to combine multiple animations based on certain parameters, such as the direction of movement or the speed of an object. These methods allow for smooth transitions between different animations, creating a more natural and fluid motion.

These techniques are commonly used in video games and simulations, where we cannot predict the exact movements of the player or other objects. By blending between different animations based on the current state of the object, we can create a more dynamic and responsive animation system.

A common example of directional blending is a character that can walk in any direction. Instead of creating a separate animation for each possible direction, we can create a set of animations for the cardinal directions (up, down, left, right) and then blend between them based on the character's current movement direction.

An extra procedural animation can be applied to the character's upper body using a blend tree, allowing the player to aim in any direction while moving. This is a common technique in third-person shooter games, where the character's movement and aiming are controlled independently.

This technique was not implemented here, as it is quite complex and requires substantial data to work effectively. However, the concept is straightforward and builds upon the interpolation-based animation technique discussed earlier. Statistically, it would be almost identical to interpolation-based animation, with a slightly more natural feel due to the blending of multiple animations.

5.6 Frame animations

Frame animations are a common technique in computer graphics, particularly in 2D animation. They're based on the original concepts of filmmaking, where a series of still images are displayed in rapid succession to create the illusion of motion. In computer graphics, this is often implemented as a sequence of images or frames that are displayed one after another. Compression algorithms, sprite sheets, and similar techniques may cause the implementation to differ slightly, but the basic principle remains the same.

Note 11

In this implementation, I utilize spritesheets to store the frames of the animation, and a function to control which frame is shown at any given time. To use this, you need to define a regular **CanvasObject** and use the `Objects.sprite(...)` drawing function.

An example frame animation can be found here

`https://kisskonraduni.github.io/emergent-animations/examples/frame-animations?tab=0`

While frame animations are simple in concept, they can be combined with other techniques to create more complex results. For example, you can use frame animations to create a character that moves around the screen, while using interpolation to control the speed and direction of movement. This allows for a wide range of possibilities and can lead to interesting results.

I have created an example[5] to show that these techniques are not mutually exclusive, and can be combined to create more complex animations. The example is a simple "pendulum" with three cats, each using its own frame animation.

Example can be found here

`https://kisskonraduni.github.io/emergent-animations/examples/frame-animations?tab=1`

Deviation	Repetition	Control	Feel	Performance
0.0	1.0	Total	N/A	$O(1)$

Table 4: Measurements for the frame animation algorithm.

The reason for omitting the feel measurement is that frame animations are not inherently tied to any specific timing or easing function. They simply display a sequence of images at a fixed rate, which can be adjusted independently of the animation itself. The feel of the animation is determined by the choice of frames and their timing, rather than the algorithm itself.

5.7 Double buffering

Double buffering is a technique used to reduce flickering and improve animation smoothness. It uses two buffers: while one is displayed, the other is drawn to in the background. When drawing is complete, the buffers are swapped, showing the new frame.

Although not strictly an animation technique, double buffering is often used alongside other methods to improve quality, especially for complex or slow rendering. This section serves as a segue, since many upcoming techniques rely on double buffering.

Note 12

Double buffering can be applied to data beyond images, such as simulation states or object positions in physics engines. Without it, execution order can cause inconsistencies. It also enables some multithreading, though this project does not use it due to language constraints.

[5]This is not intended as a serious example; I created it while experimenting with the implementation. Surprisingly, it demonstrates how frame animations can be combined with other techniques, such as interpolation and parenting, to create more complex animations. It also shows how the implementation can be used to create animations with minimal effort.

5.8 Rule-based animation systems

Rule-based animation systems use a set of predefined rules to determine how an object should move or change over time. These rules can be based on various factors, such as the object's current state, its environment, or user input. Rule-based systems are often used in games and simulations to create more dynamic and responsive animations.

A simple example is a basic implementation of Conway's Game of Life [8]. Each cell in a grid can be either alive or dead, and its state is determined by a set of rules based on the states of its neighboring cells.

My implementation can be found here

`https://kisskonraduni.github.io/emergent-animations/examples/rule-based-animations?tab=0`

This serves as a demonstration of how rule-based systems can be implemented and used to create interesting animations. It is not intended for direct comparison with the other algorithms, but rather to illustrate the fundamental concepts.

5.9 Boids

Boids is a popular algorithm for simulating the flocking behavior of birds or fish. It was developed by Craig Reynolds in 1986 [5] and is based on three simple rules:

1. **Separation**: steer to avoid crowding local flockmates
2. **Alignment**: steer towards the average heading of local flockmates
3. **Cohesion**: steer to move toward the average position of local flockmates

Each boid (bird-like object) in the simulation follows these rules to determine its movement. The result is realistic flocking behavior that emerges from the simple interactions of individual boids.

Note 13

I have made some modifications to the original algorithm, including an extra rule to steer away from the edges of the screen, and the strength of most rules varies cubically with distance. The core concepts remain the same, and the result is similar to the original algorithm, but it is adapted to my preferences.

My implementation can be found here

`https://kisskonraduni.github.io/emergent-animations/examples/rule-based-animations?tab=1`

There are many parameters that can be adjusted to change the behavior of the flock. In my implementation, these controls are available in the bottom right corner of the interface. Some combinations of parameters may slow down the simulation significantly, as the algorithm is originally $O(n^2)$ in complexity. Although I have implemented optimizations using a spatial grid [11], it is still not efficient for very large numbers of boids.

Deviation	**Repetition**	**Control**	**Feel**	**Performance**
N/A	0.2	Medium	0.9	$O(n^2)$

Table 5: Measurements for the boids algorithm.

The deviation metric is not defined for this algorithm, as it is not deterministic. The performance is also not strictly $O(n^2)$ due to the spatial grid optimization; in theory, the best case achievable is $O(n \log n)$.

5.10 Inverse kinematics

Inverse kinematics is a technique often used to animate limbs or other articulated structures. It involves calculating the positions and orientations of the individual segments of a structure based on the desired position of the end effector (e.g., a hand or foot). This is often used in character animation to ensure that a character's hand reaches a specific point in space, such as when grabbing an object.

Its most common use case, seen in many video games, is to ensure a character's feet remain in contact with the ground while walking on uneven terrain. This can be achieved by using inverse kinematics to adjust the position of the character's legs based on the height of the ground at each foot. When game developers create stairs, they often use a sloped collision surface and apply inverse kinematics to the character's legs so that the feet visually align with the steps, even if the steps themselves are only graphical.

This technique was not implemented in this project, but an example could have been included if time permitted. The concept is straightforward, and there are many resources available online that explain how to implement it. A simple implementation could use the Cyclic Coordinate Descent (CCD) algorithm [4] or the Forward And Backward Reaching Inverse Kinematics (FABRIK) algorithm [2].

As this technique was not implemented, exact statistics cannot be provided. However, based on its use in many games, it likely achieves a high feel rating and at least decent control.

5.11 Physics simulation based systems

Physics simulation based systems use a physics engine to simulate the movement and interactions of objects in a virtual environment. This can be used to create realistic animations that respond to forces such as gravity, friction, and collisions.

These simulations are used in a wide range of applications, from video games to scientific simulations. They can be used to create realistic animations of objects falling, bouncing, or colliding with each other. Physics simulations can also be used to create more complex animations, such as cloth simulation or fluid dynamics.

These systems are usually quite complex and computationally expensive, so in games we usually only use crude approximations to achieve a good balance between realism and performance. We use more advanced simulations for pre-rendered animations and visual effects in movies, where we can afford to spend more time on the simulation.

This is a topic that is too broad to cover in detail here, as there are many different types of physics simulations and many different algorithms used to implement them. This technique was not implemented in this project thanks to this.

6 Results

In this section, we summarize and compare the animation techniques discussed previously. Each method has its own strengths, limitations, and areas of application. To provide a clear overview, I present a table with the main techniques and their measured properties, as defined in the previous sections.

Technique	Deviation	Repetition	Control	Feel	Performance
Functional animation	0.0	1.0	Total	0.15	$O(1)$
Interpolation-based	See in 5.2	1.0	High	0.8	$O(1)$
Frame animation	0.0	1.0	Total	N/A	$O(1)$
Boids	N/A	0.2	Medium	0.9	$O(n^2)$

Table 6: Comparison of main animation techniques and their measured properties.

Some techniques, such as data-driven animations, blend trees, inverse kinematics, physics simulation, and rule-based systems, are not included in the table above. This is because their properties are either highly context-dependent, not deterministic, or not directly comparable using the same metrics. For example, data-driven and blend tree techniques build upon interpolation, but their deviation and feel depend on the source data and blending parameters. Inverse kinematics and physics-based systems are typically evaluated qualitatively or by simulation accuracy, rather than by simple metrics. Rule-based systems, such as cellular automata, are often used for emergent or unpredictable behaviors, making standard measurements less meaningful.

These techniques are nevertheless important and widely used, but their evaluation requires different approaches, often tailored to the specific application or desired outcome. They are mentioned here for completeness and for a sense of continuity.

7 Discussion

In this section, I reflect on the results and techniques presented earlier. I also address a few questions that may arise from the thesis, and suggest directions for further exploration.

Why are some techniques not measured directly?
Some algorithms, such as physics simulation and rule-based systems, produce results that are highly context-dependent or emergent. Standard metrics like deviation or repetition do not capture their behavior meaningfully. Instead, qualitative analysis or application-specific metrics are more appropriate. These techniques are also heavily implementation-dependent, making it difficult to generalize measurements across different implementations.

Can techniques be combined?
Yes, many practical animation systems combine multiple techniques, it's very standard in the available tools as of writing. The latest Unreal Engine 5 [6] character animation system is a great example of a modern game engine that combines many of these techniques to create realistic and dynamic animations.

How important is performance?
Performance is crucial in real-time applications, such as games or interactive graphics. Algorithms with $O(1)$ or $O(n)$ complexity are preferred for smooth user experiences. More expensive techniques are often reserved for offline rendering or precomputed effects.

What determines the 'feel' of an animation?
The feel is influenced by timing, motion quality, and how closely the animation matches natural movement. Easing functions, blending, and data-driven approaches can all improve the feel, but the choice of technique should match the desired effect and application.

7.1 Further exploration

There are many directions that could be explored beyond the scope of this thesis:

- **Procedural generation of animation parameters:** Using machine learning or optimization algorithms to generate animation curves or blending weights automatically.
- **Real-time adaptation:** Developing systems that adapt animations dynamically based on user input, environment, or other factors.
- **Advanced physical simulation:** Exploring more complex simulations, such as soft body dynamics, fluids, or crowds, and their integration with traditional animation techniques.
- **User interfaces for animation control:** Designing intuitive tools for artists and developers to manipulate and combine animation techniques.
- **Cross-disciplinary applications:** Applying animation algorithms to fields such as scientific visualization, education, or art installations.

These topics offer exciting possibilities for future research and development. I believe that continued exploration in this area will lead to more powerful, flexible, and expressive animation systems.

8 Conclusion

In this thesis, I have explored a variety of animation techniques, ranging from simple functional and interpolation-based methods to more complex systems such as rule-based algorithms, blend trees, and physics simulations. While not every technique was measured in detail, I focused on representative examples to illustrate the strengths and limitations of each approach.

The results support the main hypothesis: techniques that are easily created and controlled, such as interpolation and functional animation, offer precise and predictable motion. In contrast, emergent systems—those based on rules, data, or simulation—tend to produce more natural and lifelike movement, often at the expense of direct control.

Ultimately, the choice of technique depends on the desired balance between control and realism. By understanding the properties and trade-offs of each method, it is possible to design animation systems that are both expressive and efficient, tailored to the needs of the application. The combination of multiple techniques often yields the best results, leveraging the strengths of each to create rich and dynamic animations.

References

[1] Adobe. *What Is Motion Graphics?* Accessed 05. 08. 2025. URL: https://www.adobe.com/uk/creativecloud/animation/discover/motion-graphics.html.

[2] Andreas Aristidou and Joan Lasenby. *FABRIK: A fast, iterative solver for the Inverse Kinematics problem*. URL: http://www.andreasaristidou.com/publications/papers/FABRIK.pdf.

[3] Andrey Sitnik and Ivan Solovev. *easings.net*. URL: https://easings.net/.

[4] Ben Kenwright. *Inverse Kinematics - Cyclic Coordinate Descent (CCD)*. URL: https://alogicalmind.com/res/inverse_kinematics_ccd/paper.pdf.

[5] Craig Reynolds. *Boids*. URL: https://www.red3d.com/cwr/boids/.

[6] Epic Games. *Unreal Engine 5 - Animation Workflow Guides and Examples*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/animation-workflow-guides-and-examples-in-unreal-engine.

[7] Mozilla. *Canvas API*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

[8] Nathaniel Johnston and Dave Greene. *Conway's Game of Life*. URL: https://conwaylife.com/book/.

[9] Svelte. *"web development for the rest of us"*. URL: https://svelte.dev/.

[10] Joey de Vries. *Learn OpenGL - Graphics Programming*. Kendall & Welling, 2020. URL: https://learnopengl.com/book/book_pdf.pdf.

[11] Wikipedia contributors. *Grid (spatial index)*. URL: https://en.wikipedia.org/wiki/Grid_(spatial_index).