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 [4]. 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. There will be a lot of techniques where this will be omitted entirely, because the motion is purely procedural and we cannot define an expected motion. In these cases, we will state that the deviation metric is 1.

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 similar to 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	$P(...)$	$P(n) = n \cdot k$	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. I will provide still images for some relevant examples, but they will not be as informative as the actual animations.

> **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 [2], 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 [3], 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:

```
/* Imports */

export class MinimumScene extends CanvasScene {
    // Define
    override textures;
    override objects;
    override sequencers;

```

[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 Algorithms

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

`http://kisskonraduni.github.com/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:

$$
\begin{aligned}
r &: \text{radius} & \alpha &: \text{starting angle} \\
t &: \text{time} & \omega &: \text{angular speed (rad/s)}
\end{aligned}
$$

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.

6 Results

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

7 Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

References

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

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

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

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