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-aided visual effects. However, the terms "animation" and "motion graphics" have a much broader meaning.

As Adobe, an industry leader in digital media, defines it: "Motion graphics are essentially 'graphics with movement'." [1] This definition is much more in line 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 not foreseen by the designer?

The baseline 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 everywhere in the digital world, even though we might not notice them at first glance.

1.1 Important Mathematical Concepts

Before we begin, there are some essential mathematical concepts that will be used and referenced throughout this thesis. These concepts are fundamental to understanding the algorithms and systems discussed in the following sections.

Note 1

The mathematical definitions and operations presented here are simplified for clarity. In practice, especially in computer graphics, these operations are often implemented using matrices for efficiency and to handle more complex transformations. For more information, an excellent resource can be found at [2].

1.2 Vectors

The widely accepted definition is: "A vector is a mathematical object that has both a magnitude and a direction." A key point many people miss is that the numerical definition of vectors does not inherently include direction or magnitude. A vector is simply defined as: "An n-dimensional array of numbers." This can represent a direction, a position, a size, or a scale.

The context is crucial to understanding what a vector represents. I will talk more about the different contexts when we discuss affine transformations in Section 1.3. For now, let us discuss the vector operations that are important for our discussion.

- **Addition and subtraction:** Vectors can be added or subtracted component-wise.
- **Scalar multiplication:** A vector can be multiplied by a scalar, which scales its magnitude without changing its direction. Simply put, we multiply each component of the vector by the scalar.
- **Length:** The length (or magnitude) of a vector can be calculated using the Pythagorean theorem. For a vector $\vec{v} = [x_1, x_2, \ldots, x_n]$, the length is given by: $\|\vec{v}\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$.

- **Normalization:** To normalize a vector, we divide it by its length: $\vec{v}_{norm} = \frac{\vec{v}}{\|\vec{v}\|}$. This results in a unit vector with a length of 1. This operation is used when we only want to consider the direction of a vector without its magnitude.

- **Convert to rotation and back:** This is not a standard vector operation, but rather an essential concept used in many of the implementations discussed in this thesis. In 2D, this is straightforward, as we can represent a vector as an angle. Note that these operations discard the magnitude of the vector.

  The angle can be calculated using the arctangent function: $\theta = \text{atan}(\frac{y}{x})$ where our vector is $\vec{v} = [x, y]$.

  To convert back to a vector, we can use the cosine and sine functions: $\vec{v} = [\cos(\theta), \sin(\theta)]$.

  As it is not the focus of this thesis, I will not go into the details of 3D rotations using vectors, as Euler angles, quaternions, and rotation matrices deserve a whole discussion on their own.

1.3 Affine Transformations

Affine transformations are a set of operations that can be applied to vectors to change their position, orientation, and scale. They are fundamental in computer graphics. These transformations are commonly implemented using matrices, which allow for efficient computation and combination of multiple transformations.

The implementations in this thesis will primarily use 2D affine transformations, so I will not explain the 3D transformations and matrices in detail. However, the concepts are similar, and the same principles apply.

The main concept we need to understand is that every vector we define "lives" in a coordinate system that we can manipulate using these transformations. The base coordinate system that we use when drawing things on the screen depends on the implementation and the context. Most graphics APIs use a right-handed coordinate system, where the origin is at the center of the screen.

Because I wanted to keep the implementations simple, I used an older concept where we define the coordinate system in the top-left corner of the screen, with the positive y-axis pointing downwards. This was a common practice in early computer graphics due to the way old displays and systems worked. Although this clashes with the standard coordinate system used in mathematics, it is still widely used in many graphics libraries and game engines today. When we begin using it in the implementations, it will become clear that it is indeed a simple convention. The operations we will use are the following:

- **Translation:** This operation moves a vector by a certain amount in the x and y directions. For a vector $\vec{v} = [x, y]$ and a translation vector $\vec{t} = [t_x, t_y]$, the new vector after translation is given by: $\vec{v'} = \vec{v} + \vec{t}$.

- **Scaling:** This is not a standard vector operation, as there is no operation that multiplies a vector by a different scalar for each axis. Still, this is an immensely useful operation. This allows us to change the size of a vector unevenly in the x and y directions. For a vector $\vec{v} = [x, y]$ and a scaling factor $\vec{s} = [s_x, s_y]$, the new vector after scaling is given by: $\vec{v'} = [x \cdot s_x, y \cdot s_y]$. Depending on the order of operations, this can be used as a **shearing** transformation if used right after a rotation.

- **Rotation:** This operation rotates a vector around the origin by a certain angle. For a vector $\vec{v} = [x, y]$ and an angle θ, the new vector after rotation is given by: $\vec{v'} = [x \cdot \cos(\theta) - y \cdot \sin(\theta), x \cdot \sin(\theta) + y \cdot \cos(\theta)]$ This transformation preserves the magnitude of the vector.

The main concept I want to convey with these operations is that the coordinate system we use can (and should) be manipulated to achieve specific effects. This is infinitely more powerful than it seems at first glance, and I will demonstrate this using a few examples:

- **I don't like the top-left origin:** We can simply create a scale transformation with $\vec{s} = [1, -1]$ and a translation transformation with $\vec{t} = [0, h]$ where h is the height of the screen. This will flip the y-axis and move the origin to the bottom-left corner of the screen. This transformation will likely be used in the implementations when dealing with physics simulations, as it is more intuitive to think of the ground as the bottom of the screen.

- **I want this object to stay locked to another object:** We can think of these transformations as a stack. If we want an object to stay locked to another, we simply push the parent object's transformations onto the stack before applying the child object's transformations, then we draw the child object. This way, the child object will always be drawn in the correct position relative to the parent object, regardless of how the parent is transformed. And the stack analogy is a good way to think about how these transformations are applied in practice.

  Note 2

  In this context, we usually differentiate between the local coordinate system of an object (Object space), the global coordinate system (World space), and the screen coordinate system (Screen space).

  Note 3

  These transformations can be reversed by applying thier inverse values. For example, to reverse a translation, we can use $\vec{t'} = [-t_x, -t_y]$. This is not necessary most of the time as we keep the stack of transformations in memory, but there can be cases when we need it.

- **I don't want to rotate an object around the origin:** The order of operations matters a lot. If we first rotate an object, then translate it, the object will rotate around its own origin. If we want to rotate it around a different point, we can simply translate the object to the desired point, apply the rotation, and then translate it back. First, this sounds complicated, but seeing it in practice makes it clear most of the time. The fact that these things can be done even on paper by hand often helps understand the concepts better.

Having established the mathematical foundations, we can now explore how these concepts are applied in practice. In the next section, we will begin with some visual examples that showcase different coordinate systems and transformations. From there, we will gradually build up to more complex techniques and applications.

2 Methods

Before diving into the complex implementations and algorithms, it is essential that we visualize the core concepts mentioned in the previous section.

Note 4

All of these visualizations are created using my own TypeScript implementation of a simple graphics library built on top of the HTML5 Canvas API. The code for this library is available in the repository of this thesis, and the visualizations can be viewed in the browser here: `https://kisskonraduni.github.io/emergent-animations/examples`.

2.1 Visualizing Coordinate Systems and Transformations

Understanding how objects are positioned and transformed requires a clear grasp of the different coordinate spaces used in graphics programming.

From here on, I recommend opening the interactive examples on a nearby device, as the visualizations will be referenced throughout the thesis. While still images are provided for most examples, seeing them in motion is ideal. Let's begin by examining the main coordinate spaces used in graphics programming.

Open interactive example

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

2.1.1 World Space

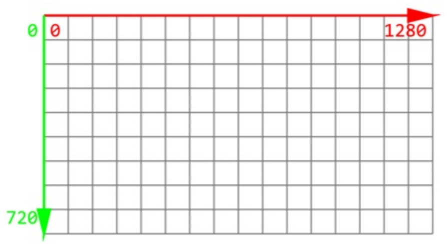

Figure 1: World Space Coordinate System **in my Graphics Library**

World Space is the base coordinate system in a graphics library or game engine. The origin is at the top-left corner *in my library*, with the x-axis pointing right and the y-axis pointing down. Almost all objects are positioned in this space, and transformations are applied relative to the origin.

In this graphics library, I define a $\vec{resolution} = [1280, 720]$ vector that represents the visible area.

Note 5

Technically speaking, the $\vec{resolution}$ vector represents the bounds of an imaginary camera spanning from $[0, 0]$ to $[1280, 720]$. This variable is often used to define positions in the examples, as it makes our calculations easy to follow. The library ensures that, for any resolution and aspect ratio, this view is centered and fully visible on the screen.

2.1.2 Object Space

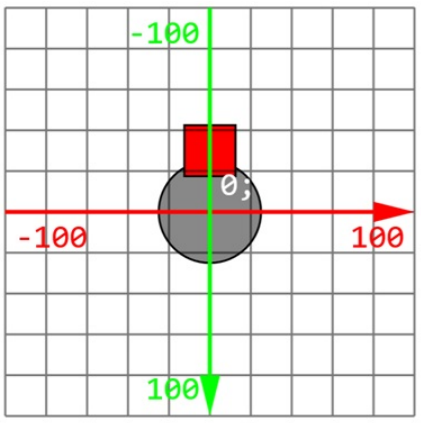

Figure 2: An example object with its Coordinate System visualized

Object Space is the local coordinate system of an object, with the origin typically at the center. This space is used when defining the object's geometry and rendering. Transformations in Object Space are applied relative to the object's pivot point.

The object has a tiny hat as a child element to visualize the local coordinate system. The hat's position is set relative to the circle's pivot point and moves with it. If we want the hat's position in World Space, we need to apply the object's transformations to the vector representing the hat's position.

Note 6

The pivot point is the reference for rotation and positioning within World Space. Correctly setting the pivot point is crucial: for UI elements, it allows easy anchoring; for game objects, such as a player character, the pivot is often set to the feet to ensure rotation occurs around the correct point.

2.1.3 Screen Space

In this graphics library, Screen Space typically matches World Space, with the origin at the top-left corner. In 3D applications, an additional Camera Space is introduced: World Space → Camera Space → Screen Space. Camera Space represents the camera's position and orientation, while Screen Space is where the camera's view is projected, either at the top-left or center of the screen. This separation is important, as game objects and UI elements often exist in different spaces or layers.

Note 7

Using these transformations, we can create interactions between different spaces. For example, dragging an object in Screen Space can be transformed into World Space to update its position. Most 3D editors use a similar approach. This is possible because transformations can be inverted, allowing us to transform a point from Screen Space to World Space, or from Object Space to World Space.

2.2 Understanding "Objects" in Graphics Programming

In graphics programming, the term "object" can refer to various entities, from simple shapes to complex models. The key to animating these is understanding their properties and how we can build our scenes using them.

Open interactive example

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

2.2.1 Basic Properties

There are a few properties that most implementations use to define objects:

- **Position:** The object's position, typically represented as a vector.
- **Rotation:** The object's orientation, often represented as an angle in 2D, as a 3D vector using Euler angles in three dimensions, or as a quaternion in more advanced implementations. See [2, Section 10.6] for more information.
- **Scale:** The object's scale is a vector that defines how much the object is stretched or compressed in each dimension. Note that scale does not directly represent the object's size, but rather its proportional transformation along each axis.
- **Parent:** The object's parent, if defined, is another object that serves as the basis for its transformations. Circular dependencies and self-references are not allowed.

Notably, these properties do not explicitly define a coordinate system; instead, they are always interpreted in the parent object's Object Space, or in World Space if no parent exists.

Open interactive example

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

An emergent behavior of these properties is that we can create complex animations with simple parent-child relationships.

2.2.2 Drawing Objects

Drawing an object typically involves rendering its geometry, which can be as simple as a circle or as complex as a 3D model. A key aspect of this is that we define a **size** property for the object, which is **not** the same as the scale property.

The size is used to determine how large the object appears on the screen, while the scale is used to stretch or compress the object in its Object Space. As an extra note, the way we position the object before drawing is also important, usually defined by another property called the **pivot point**. For an explanation, see 2.1.2

Note 8

The size property often can be an implied property from the data we use to define how we draw said object. For example, a 3D model would be defined by points in space around the origin, and the size would be the distance between the furthest points in each axis.

The exact implementation used to draw the object depends on the graphics library or engine, so I will not go into detail here, as it is not the focus of this thesis. However, I will outline the general steps taken, as it is important to understand how these objects are rendered:

- **Apply the transformation stack.** - Get all the elements in the transformation stack, and "apply" them in the order they were added.

  Note 9

  "Applying" the transformation stack can differ depending on the implementation. Most commonly, this means multiplying the current transformation matrix by the transformation matrix of the object. The starting "World Space" matrix is the identity matrix, which represents no transformation. The order of multiplication matters, as it determines how the transformations are applied.

- **Draw the object.** - Use the current transformation matrix to draw the object in its Object Space, using its size and pivot point.

- **Reset the transformations for the next object.** - After drawing the object, reset the transformations in order to avoid affecting subsequent objects.

Take this method and repeat it for every object in the scene. An extra caveat in 2D is that draw order matters (sometimes it matters in 3D as well, but that is not the focus of this thesis), so we need to make sure that the objects are drawn in the correct order. We can imagine them as stickers being placed on top of each other, where the last sticker drawn is the one on top. This is why we often use a "z-index" property to determine the order of drawing if we do not want to manually sort the objects.

3 Results

Here will be some results.

4 Conclusion

Here is a conclusion.

References

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

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