ThedyxEngine

Rostyslav Liapkin

# Introduction

ThedyxEngine is a 2D physics engine designed to simulate heat transfer across different materials using a visually intuitive approach. The engine supports various forms of heat transfer mechanisms including conduction, convection, and radiation, presenting them in a visually engaging manner that changes color based on the temperature of the objects.

A screenshot of a computer

AI-generated content may be incorrect.

1. Features Conduction: Simulate heat transfer through direct contact.
2. Convection: Model the transfer of heat through fluids and gases.
3. Radiation: Represent the emission of heat through electromagnetic waves.

It uses widely used finite difference method to calculate heat transfer between adjacent squares by considering factors such as temperature differences, the thermal conductivity of the material, and the simulation timestep.

# Motivation

## Bringing Theory and Practice together

Traditional thermodynamics and heat transfer studies are often presented in an abstract and theoretical manner. While equations and formulas provide a strong mathematical foundation, they do not always offer an intuitive understanding of how heat behaves in different environments and ThedyxEngine aims to bridge this gap by offering a visually understandable simulation that allows users to see and experiment with heat transfer in real-time.

## User-Friendly Simulation

Many existing simulation tools are highly sophisticated but come with steep learning curves, making them inaccessible to non-experts and these complex tools are often designed for research institutions and industrial applications, requiring significant training to operate effectively. ThedyxEngine, on the other hand, provides an intuitive and user-friendly interface, allowing users of all skill levels to simulate and observe heat transfer without needing deep technical expertise.

## **Accessible Education**

One of the core goals of ThedyxEngine is to serve as a free and accessible platform for learning thermodynamics by providing real-time, interactive visualizations of heat conduction, convection, and radiation and it enables students, educators, and enthusiasts to experiment with different materials and scenarios.

## **Open-Source**

As an **open-source project**, ThedyxEngine is built on a foundation of **community-maintained framework in cont**inuous development and it invites collaboration and continuous improvement from the community.

# Novelty

## Lightweight and Cross-Platform

ThedyxEngine is optimized to run efficiently on a wide range of hardware, even on the oldest laptops, making it accessible for everyone. It’s also multiplatform, so users can use it both on Windows and MacOS.

## Community-Maintained & Open Source

Developed with a .NET 9.0 (and fully compatible with .NET 8.0) with Microsoft Application UI Framework and some elements from MAUI Community Toolkit, so it’s designed for accessibility and collaboration. As on open-source project, ThedyxEngine is available for everyone and easy to maintain.

## High Precision

By dividing objects into granular elements, ThedyxEngine delivers a **detailed and accurate** simulation of heat transfer. It models conduction, convection, and radiation with precision, making it a powerful tool for both educational and research applications.

#### MacOS

A screenshot of a computer

AI-generated content may be incorrect.

Windows

A screenshot of a computer

AI-generated content may be incorrect.

# Methodology

ThedyxEngine employs the **Finite Difference Method (FDM)** to simulate heat transfer with high accuracy. By discretizing objects into a grid of small squares, the engine applies numerical methods to approximate heat diffusion over time and this approach allows for precise calculations while maintaining computational efficiency. Timestep of the simulation can be changed and precision changes with a timestep.

A diagram of a diagram

AI-generated content may be incorrect.

Pic 1.

Each object in the simulation is divided into a structured grid, enabling localized temperature calculations. Heat is transferred only between adjacent squares, ensuring an accurate representation of conduction, convection, and radiation. This granular approach allows for a realistic and stable simulation of energy flow.

The simulation operates in discrete time steps, calculating the amount of energy transferred in each interval. This step-by-step approach ensures a smooth and incremental evolution of temperature changes, making it possible to visualize thermal dynamics in real time.

# Engine

Engine orchestrates heat transfer calculations with multiple threads and this design allows simulation to be effective for multi-core processors, really improving performance with large number of objects (described in chapter **Benchmarking: Multicore efficiency**)

## Multi-Threaded Engine Architecture

A diagram of a machine

AI-generated content may be incorrect.

Pic 2.

### Central coordination

Engine serves as a central coordinator of the simulation. It maintains the global state of the Engine (Stopped, Running, Paused) and controls the simulation loop. Before any calculation steps program tries to determine the number of the processor cores and create (number of threads that equals to number of processor cores – 2 to have some reserve for operating system and other processes that can run in the system). This chunking of tasks prevents any single thread from becoming a bottleneck and ensures an even distribution of computational effort.

### Threads

Once engine gets information about the environment and the number of workers it needs, it uses them for a created list of tasks. Each thread is assigned to a subset of the simulated objects in our program, which is determined both by the number of the objects and number of the threads).

Type of the task that can be assigned:

1. **Optimization objects before the simulation.** Simulation is static, so the things like Raytracing and finding the closest and adjacent squared can be done once before the simulation. It described in chapter *OptimizationManager.*
2. **Parallel Heat Calculation.** In each time steps of simulation, threads concurrently invoke three managers to calculate heat transfers: ConductionManager, ConvectionManager and RadiationManager for own subsets of object. Each handle its own type of mechanism of heat transfer and contributes partial energy deltas that engine aggregates later.

### Heat Transfer Managers

Under the hood, thermodynamics can be decomposed into multiple mechanisms of energy exchange: conduction (heat transfer through direct contact), convection (heat transfer through fluid or gas movement), and radiation (heat transfer via electromagnetic waves). Rather than mixing these concerns into a single monolithic function, ThedyxEngine uses three separate managers. This modular approach both clarifies the code and facilitates potential expansions or customizations.

Each manager computes a partial energy value (*energyDelta*) for the objects or squares under its control, reflecting how much heat is gained or lost. By separating these calculations into distinct modules, the system remains flexible and extensible. If we want to introduce a new type of heat transfer or override existing logic, we can modify or replace one manager without disturbing the entire engine.

Each of managers has its own chapter that explains mechanisms and things that happens for each of them.

### EngineObjects and Energy Application

Once the three managers have finished computing partial energy changes, each EngineObject (or its smaller subunits) aggregates these deltas. However, to avoid inconsistencies, the engine applies these updates after all threads have signaled, they are done with their portion of the work. This strategy exits to prevent one manager from using partially updated temperatures that another manager has just modified.

### Stepped execution and Real-Time Constraints

The loop typically simulates the run of every 1/60th of a second (or another chosen interval). Because engine is really optimized, it’s able to run up to 20 seconds of simulation in one real-time seconds in a simple scene, so user can allow this in settings.

On each iteration:

1. Threads do radiation, conduction, and convection on their assigned objects.
2. The engine aggregates the results and updates object temperatures.
3. A short delay ensures the simulation does not exceed the desired frame rate.

If the computations exceed the time budget (e.g., it takes too long to finish one frame), the engine logs a warning. In high-load scenarios, the user might reduce detail or reduce the frequency of updates to maintain real-time performance.

## Benchmarking: MultiCore efficiency

To test MultiCore efficiency, we were using different number of EngineRectangles with a height and width of 10. Engine was allowed to use different number of cores.

Test configuration:

1. Number of objects: 400,225,100
2. 5 iterations
3. Hardware: Apple M3 Pro
4. Rendering image wasn’t happening.

As a benchmarking metric we were using number of timesteps that engine was able to calculate in one minute:

|  |  |  |  |
| --- | --- | --- | --- |
| Num. Cores | 400 objects | 225 objects | 100 objects |
| 1 | 533 | 972 | 2133 |
| 2 | 1036 | 1877 | 4027 |
| 3 | 1527 | 2729 | 6058 |
| 4 | 1837 | 3011 | 6294 |
| 5 | 2022 | 3150 | 6415 |
| 6 | 2115 | 3197 | 6743 |
| 7 | 2055 | 3150 | 6639 |
| 8 | 2091 | 3174 | 6524 |
| 9 | 2074 | 3116 | 6337 |
| 10 | 2038 | 3024 | 6034 |
| 11 | 2042 | 3072 | 5772 |
| 12 | 1973 | 2974 | 5569 |

Table 1.

Plot:

Pic 3.

*Pic 3* proofs that increasing the number of CPU cores generally boosts simulation throughput more that **4 times**. The data shows that we can achieve significant speedups with parallel processing up to an optimal number of cores.

Once we exceed a certain of cores, overheads start to outweigh the benefits of the parallelism, causing the performance even to become a little bit worse, so it’s exactly the reason why we don’t use all the CPU cores that are available in the system.

Additionally, we can see that with fewer number of objects the performance scaling saturates faster, because there is less work per timestep. And for heavier workloads we can see increase in performance from bigger number of cores for a bit longer, because we start from a lower baseline.