TNM094 — Media Technology - Bachelor Project
2. Design Modeling

2025 Version 1.0

1 Introduction

There are many ways to convert an idea into a final implementation in code. One important and powerful approach, in its full or part, is through modeling. The modeling approach uses graphical models to design and define the software prior to implementation. The models become means for communicating ideas, testing concepts and protocols, exploring the use of interfaces, defining parts and classes, and even generating code. Examples of software development methodologies having modeling at its core are Model Driven Development, Agile Modeling, or Story-driven modeling.

1.1 Purpose

In this exercise, you will touch upon UML modeling to structure and understand your program before writing a single line of code. The purpose of experimenting with this kind of tool is to see how you can explore programming concepts, through UML models at differently detailed levels, and create a natural progression into details, and even analyze possible behavior already at an abstract level. This will allow you to familiarize yourself with some of the different diagrams and their strengths and weaknesses.

This exercise also requires programming and program design skills acquired in previous courses. Therefore, it can be regarded as a test that you can fully benefit from what you have learned. Take an idea or concept, model it, and then finally implement it.

1.2 Prerequisites

It is necessary to understand the fundamental aspects of object-oriented programming and object-oriented design, to be able to perform and understand all parts of this exercise. Before taking this exercise, review earlier material on abstraction, interfaces, polymorphism, how to select between inheritance and composition, and how to select and define object relationships.

1.3 Modeling Software

There are many different UML editors available for various platforms. They provide different features and different levels of strictness. Some are even capable of exporting the final diagram as source code. The more advanced UML editors can export the structure to at least one programming language, and many support a multitude.

However, using a full-featured UML editor takes years to master, so for this lab, we suggest a simpler modeler diagrams.net (https://diagrams.net). This is a free web-based model drawing tool that requires no installation. It should provide sufficient functionality for this exercise.

1.4 Documentation

This is not a manual, so you are expected to search for information in manuals, tool documentation, and command-line help. The following sources are useful for this exercise.

- The Unified Modeling Language Reference Manual
  https://www.utdallas.edu/ chung/Fujitsu/UML_2.0/Rumbaugh--UML_2.0_Reference_CD.pdf
- Tutorial on Object Oriented Analysis and Design
  http://www.tutorialspoint.com/object_oriented_analysis_design/
- Introduction to CMake https://cliutils.gitlab.io/modern-cmake/chapters/basics.html

1.5 Examination

After finishing all tasks in this exercise, show your results to a lab assistant and explain what you have done and what conclusions you have come to.

For a successful examination, please make sure that you fully understand what you have done and how the different diagrams and your implementation relate to each other.

To complete this lab, you need to show the following:

- A use case diagram reflecting the usage scenarios of your system.
- An activity diagram reflecting the high-level flow of your system.
- A class diagram reflecting the objects used in your system.
- A working implementation of a particle system fulfilling the requirement specification (see section 2).

Note that the exercise is an open-ended problem, where it is up to you to design and implement one out of many possible solutions for a particle system.

2 System to Design and Implement

In this exercise, you will design and implement a 2D particle system based on the code you retrieved in Lab 1. A particle system consists of a collection of particles. Each particle can have attributes affecting its behavior, such as velocity, lifetime, and appearance, for example, color. Particles are often represented graphically by points, but this is not a requirement.

Particle systems are often used in games because of their visually impressive effect with a relatively small amount of code. For an example of a 2D particle system, see http://jsoverson.github.io/JavaScript-Particle-System/. This example includes, in addition to the particle simulation itself, a graphical user interface to adjust properties and pretty nice graphics.

A typical particle system often goes through the following three phases:

- Generation — Particles are spawned.
- Dynamics — Particles are moved.
- Extinction — Particles are removed, e.g., those exceeding their lifetime.

A modular particle system provides different particle emitters for spawning new particles and various particle effects to control the dynamics of particles.

More details on particle systems can be found in Appendix A

2.1 Requirements

You will design and implement your own particle system. How you design and implement your particle system will be up to you, but it must meet the following requirements:

- The particle system should be designed as a reusable library that can be used by another developer.
- Support more than one type of emitter, which spawns new particles within the system.
- Support more than one type of effect, which acts upon the particles by modifying their state.
- Support user-defined types of emitters and effects without modifying the particle system library.

To demonstrate the use of the library, the lab files proved contain an example application, using random particles. You should replace the random system with your new particle system library.

2.2 Examples of emitters

Uniform Spawn particles from a specified point at a fixed rate (number of particles / unit time) and scatter them with a uniform distribution in all directions.

Directional Spawns particles from a specified point at a fixed rate in a certain direction.

Explosion Instantaneously spawns a large number of particles from a specified point with high velocity and random directions.

2.3 Examples of effects

Gravity Well A small point with a large mass (black hole) that attracts particles through gravity.

Wind A force at a given point and a given direction affecting (nearby) particles.

Plane A plane (line) on which a particle will bounce.

3 Tasks

The following tasks will try and guide you to design the system *top down*. You will start at a high level of abstraction and progressively add more details to the design.

- First, you will describe the possible things that the client, i.e. the application code (basically the main file in the example code given), might want to do with your system by using a *Use Case diagram*.
- Then, you will create different *Activity diagrams* to identify and explore the dynamic structure of the system, first from the perspective of the client code.
- Then you will expand those exploring *Activity diagrams* with the inner workings of your system, still at an abstract level.
- Through these diagrams, you will be able to identify key components that you can then solidify the design of in a *Class diagram*.
- Lastly, you will implement your design.

Note that you will most likely iterate through these steps, so do not feel that you need to have a final version of the first diagram before moving on to the next.

Task 1 — Understand the Current Diagrams:
Examine the diagrams in Figures 1, 2, and 3. Make sure that you understand the relationship between the diagrams at the different levels of abstraction.

There is no explicit hierarchy or order between the UML diagrams, but one way to express *order* is by the level of abstraction, from high to low: Use Case diagram → Activity diagram → Class diagram. The class diagram provides descriptions that can be translated into actual code and is the lowest abstraction level.

You will start by defining *what* your particle system should do for the client code. Some of these use cases may be represented by a method in the final implementation, but that is not known or is not of importance at this abstraction level.

[Note: A diagram does not need to be fully complete before moving on to the next diagram. You should fill in as many details as you can at that stage and move on to the next. You will gain more understanding and knowledge for each diagram and most often it is required to go back and forth between diagrams in order to come up with a coherent design. You may also need to go back and revise your diagrams when implementing your code.]

Task 2 — Use Case Diagram:
First, think about the system from the perspective of the user, in this case, the client code. What use cases can you identify given the set of requirements in section 2? What parts should be directly exposed to the user? Recreate and extend the use case diagram in Figure 1 to reflect your set of identified use cases.

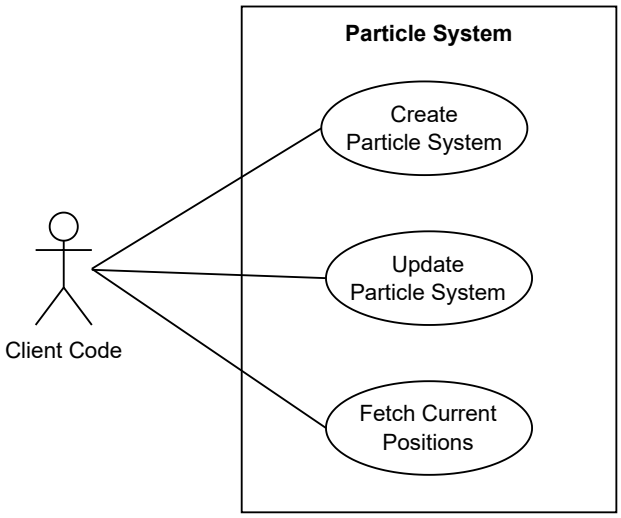

Figure 1: A simplistic use case diagram illustrating examples of operations that can be exposed to the user, in this case, another piece of software, *client code* from the particle system library's perspective.

The use case diagram provides a rough and abstract overview of the system(s) and how the user interacts with them. Being abstract, it should leave out the details behind each operation and the sequence in which they occur. This leads us to the activity diagram, which aims to reduce the level of abstraction and provide clarity. The activity diagram helps us visualize what needs to be done and in what order.

Any non-trivial system would have to be modeled using many activity diagrams, each describing a single use case or other functionality. Here we will minimize the number of diagrams and focus only on the *update particle system* use case.

Task 3 — Update particle system activities:
Focus on the *update particle system* step in the clients activity diagram Figure 2. Add all the activities that you believe are necessary for the particle system to complete the task of updating all parts of the system (see the three phases of a typical particle system in Section 2). Consider in which order the different steps need to happen. Are there any dependencies?

We can now start reasoning about how the activities can be divided between different components so that each is concerned with a specific aspect of the system. It can be beneficial to use *swimlanes*, to move activities to *components* in the system, as you identify the need for such components and determine which component is most fittingly responsible for each activity. There is a template for *Flowchart diagram* in diagrams.net that you can use, or you can search for *lane* among the shapes to find individual shapes.

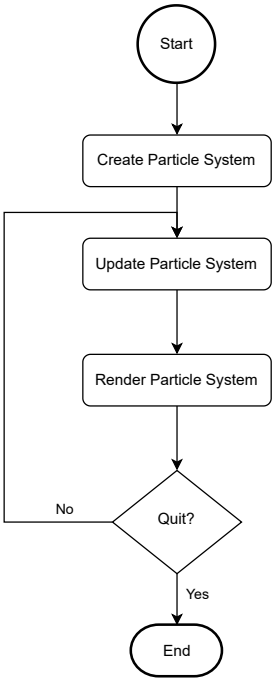

Figure 2: An example of a coarse-level activity diagram representing the flow of operations within the users' application code.

Task 4 — Identify system components:
Divide the activities in the precious diagram into lanes. Start with two lanes (or leave the others empty if you have more), one for the client code and one for the particle system. Create new lanes or use empty lanes, name them, and move activities there when you realize what component you want to be responsible for those activities.

Now that we have a better understanding of which parts the system should consist of and how these parts will make use of each other to fulfill the common goal, we can begin to concretize the design in classes. The swimlanes in the Activity Diagram indicate the need for classes or components, and the activity sequences crossing the boundaries between such swimlanes indicate some association between these classes or components. These associations may require method calls later on, however, now we just specify associations.

Task 5 — Class Diagram of the Concrete Design:
Create a basic class diagram containing the classes that will be involved in the system and specify associations. Do not focus on functions and attributes just yet, just on the basic structures.

Now that we have an idea of which classes our system should include, we can reason in more detail about *how* each class in the system should work and communicate (i.e. their interfaces). As you have learned more about the system based on your different views, through your diagrams, it is a good time to go back to your other diagrams and see if they need to be revised. When you are satisfied with the design, it is time to add more details.

Task 6 — Class Contents:
Add the most important members to the classes — both functions, to allow the classes to communicate, and attributes that control their internal states and aggregations. Go back and adjust your earlier diagrams as necessary.

As a final step before we start implementing, we should review our design.

Task 7 — Design review:
Consider the system requirements in section 2. Especially think over how your system will handle each and every requirement in subsection 2.1. Go back and update the design as needed if you identify any requirements that are not handled. If you are unsure about any of the requirements, you can discuss with one of the supervisors.

When you feel satisfied with your design and see that it fulfills the requirements move on to the implementation.

[Note: To build the project, follow the instructions in *README.md*.]

[Note: To add new header and source files, create them in the folders (probably *include/particlesystem* or *src/particlesystem*). Then add these in *CMakeLists.txt*, search for SOURCE_FILES and HEADER_FILES. Your IDE should run CMake and update the project automatically when you rebuild.]

Task 8 — Implementation:
Implement your system with the provided code as a starting point. Note: Ensure that your fork is based on the latest commit in the original repository. Use your class diagrams as blueprints for the implementation. Remember to go back and update the design if you notice the the implementation deviates from it. You will have to describe to the supervisor how the code relates to the diagrams.

You are ready to show the exercise to a supervisor when your particle system design and implementation meet the requirements in section 2.

A Particle Systems

The purpose of this section is to serve as a complement and to provide more details on what defines a particle system and its different components.

A particle system is a set of particles which are spawned, move around and die under some set of rules. The rules apply to all particles within the system. The rules control how frequently, the direction and velocity of each spawned particle as well as their lifetime. The rules also govern forces acting upon the Particles during its lifetime, modifying their trajectories.

If we break this down, we have three concepts at play here.

- Particle,
- rules that dictate the spawning of Particles (Emitter), and
- rules that dictate the forces acting upon Particles (Effect)

A.1 Particle

According to Wikipedia the definition of a particle is:

> "In the physical sciences, a particle (or corpuscle in older texts) is a small localized object to which can be ascribed several physical or chemical properties such as volume, density or mass."

In the context of our Particle System, which properties do our Particles need? Well, they are certainly simulating physical particles, as we intend to move them around and have forces acting upon them. In order to move objects through forces, it is a good idea to review Newton's laws of motion (https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion).

A.1.1 Moving Particles

Newton's first law, often referred to as the law of inertia, states that an object in motion will stay in motion until it is acted upon by a net external force:

$$\sum \mathbf{F} = 0 \Leftrightarrow \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = 0 \tag{1}$$

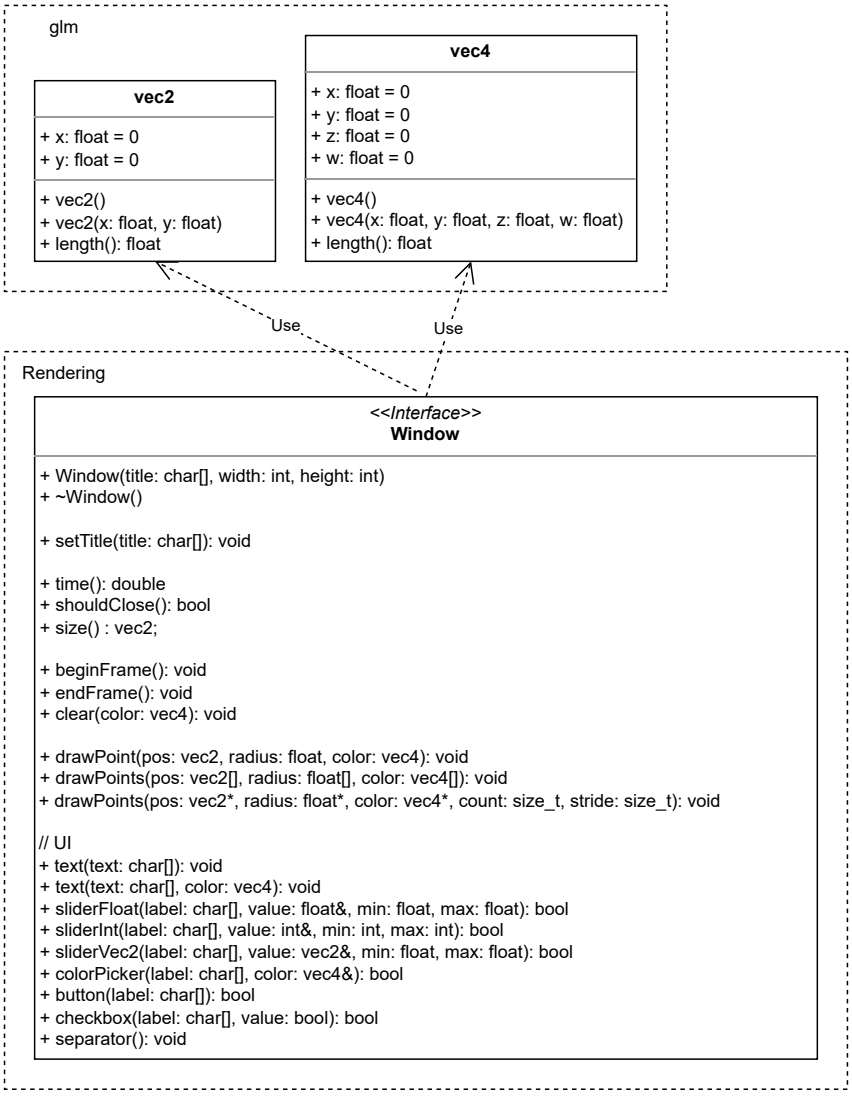

Figure 3: A class diagram depicting the structure of the provided code.

If the net force is zero, the change in velocity is also zero. So any particle with a certain velocity will continue along that direction until some force acts upon the particle.

Newton's second law states that the rate of change of momentum of a body over time is directly proportional to the force applied and occurs in the same direction as the applied force:

$$\mathbf{F} = \frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t} = \frac{\mathrm{d}(m\mathbf{v})}{\mathrm{d}t} = m\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = m\mathbf{a} \tag{2}$$

So, the net force, **F**, acting on a particle is equal to the particle's mass, m, times the particle's acceleration **a**.

This means that we can derive an expression for acceleration.

$$\mathbf{a} = \frac{\mathbf{F}}{m} \tag{3}$$

Newton's third law states that for every action, there is an equal and opposite reaction.

$$\mathbf{F}_A = -\mathbf{F}_b \tag{4}$$

This would mean that any body exerting some force upon another body would be affected by an equal and opposite force. An example would be the sun through gravity exerting a force on earth holding it in orbit. In doing so, the sun is also being pulled with an equal force toward the earth. However, the immense difference in mass renders this effect negligible.

Newton's third law is often ignored in the context of particle systems for games because it is computationally expensive to simulate particles affected by every other particle ($O(N^2)$). The additional required computations are fine for smaller systems where the number of particles N is small, but as N grows large the simulation time slows to a halt. There are tricks to reduce the complexity by grouping particles into bigger pseudo-particles which are good enough approximations for particles far away. But this requires efficient spatial acceleration structures and is beyond the scope of this text.

Integrating a new position

We start by making the assumption that we are going to simulate the particles in small discrete timesteps Δt and for the duration of each timestep we are going to assume that the forces acting on a body is constant. If the force **F** is constant, this means that the acceleration **a** is also constant. We also know that the relative change in velocity **v** with respect to time is acceleration **a** and the relative change in position **x** with regards to time is velocity **v**. Therefore, we have what we need to express the change in position **x** given a Force **F**:

$$\mathbf{a} = \frac{\sum \mathbf{F}}{m} \tag{5}$$

$$\mathbf{v'} = \mathbf{v} + \mathbf{a}\Delta t \tag{6}$$

$$\mathbf{x'} = \mathbf{x} + \mathbf{v}\Delta t \tag{7}$$

Where **v'** and **x'** are the new velocity and position vectors expressed with the current velocity and position vectors **v** and **x**. This technique is known as forward Euler integration and is the simplest form of integration. There are many different integration schemes, but this will suffice for our purposes and is simple to implement.

A.1.2 State of a Particle

Armed with the knowledge of Forward Euler integration, we can express the required state of a Particle.

Position: A vector describing its position in space.

Velocity: A vector describing the relative change in position over time.

Force: A vector describing the net force that is acting on the particle.

Mass: A scalar describing the mass of the particle (This could be constant for all particles within the system as we do not expect it to change, so this could be implicitly given from the system).

Lifetime: A scalar describing how long left the particle in seconds has to live.

A.2 Emitter and Effect

As stated before, an *Emitter* is a set of rules which dictate the spawning mechanics of particles. An example of an Emitter is a Directional Emitter, an emitter that spawns particles with a fixed direction or randomly within a cone. In essence, the purpose of an emitter is to provide a starting state for particles when they are created.

Effect is a rule that governs the forces acting on a particle. Usually, one effect models only one type of acting force, e.g. gravity or wind. In essence, the purpose of an effect is for a given particle state to produce a force vector as a result.

From an implementation perspective, one approach to achieve this polymorphic behavior is to provide a common interface for all *Emitters* and one common interface for all *Effects*. Then implement concrete entities (such as *DirectionalEmitter* or *GravityEffect*) which adhere to the interfaces.