Pull request : dynamicParameterSystem

Luciano Casanova

luciano.casanova@femto-st.fr

February 10, 2026

1 Dynamic Parameters

We implemented an option inside the scene json that allows to prescribe changes in all available physical parameters from TimeStep, Viscosity and SurfaceTension methods. It should be straightforward to implement the same system for other type of methods (Elasticity, Vorticity, XSPH). The syntax inside the scene is as follows:

```
{
    "Configuration" : {...},
    "Materials" : [{...}],
    "FluidModels" : [{...}],
    "RigidBodies" : {[...]},
    "DynamicParameters": [
                {
                   "fluidId": "inner",
                   "parameterName": "viscosity",
                   "defaultValue": 0.0,
                   "expression" : "10.0 + 2.0 * sin(2.0 * t)"
                },
                {
                   "fluidId": "inner",
                   "parameterName": "viscosityBoundary",
                   "defaultValue": 0.0,
                   "stepFunction": false,
                   "timelineTimes":  "0.5, 1.0",
                   "timelineValues": "1.0, 2.0"
                },
                {
                   "fluidId": "inner",
                   "parameterName": "surfaceTensionBoundary",
                   "defaultValue": 50.0,
                   "stepFunction": true,
                   "timelineTimes":  "1.0, 1.5,  2.0,   2.5,   3.0",
                   "timelineValues": "0.0, 50.0, 100.0, 200.0, 300.0"
                }
        ]
}
```

Each **Dynamic Parameter** is linked to one *fluidId*, controls only one parameter (via its *parameterName*), has a *defaultValue* and can change in two different ways. The first one is as a timeline, which can be changed abruptly or be smoothly interpolated between the given *timelineValues* at their respective *timelineTimes*. The other possibility is via a mathematical expression that is parsed using *tinyexpr*.

1.1 DynamicParameterSystem

The **DynamicParameterSystem** allows simulation parameters to change dynamically during runtime according to predefined schedules. This improves experiment repeatability by setting parameter changes directly in the scene JSON file rather than requiring manual intervention.

The **ParameterSchedule** struct encapsulates all information needed to manage a single parameter's temporal evolution:

```
struct ParameterSchedule {
    std::string fluidId;                // Target fluid identifier
    std::string parameter_name;         // Name of parameter to control
    std::string expression;             // Mathematical expression
    std::vector<std::pair<Real, Real>> timeline;  // Time-value pairs
    int current_timeline_index;         // Tracker
    bool use_expression;                // Expression mode flag
    bool use_step_function;             // Step vs. interpolation flag
    te_expr* compiled_expr;             // Compiled expression object
    Real current_value;                 // Current parameter value
    Real default_value;                 // Initial value
    bool active;                        // Enable/disable flag
};
```

The **DynamicParameterSystem** maintains a collection of schedules and provides the interface for adding, updating, and applying parameter changes. The system supports two types of parameter schedules.

The first one is the **timeline** schedule. It can be used for gradual parameter transitions (Interpolation mode, stepFunction = false) or for discrete parameter jumps (Step Function mode, stepFunction = true).

```
bool addTimelineSchedule(const std::string& fluidId,
                         const std::string& paramName,
                         const std::vector<std::pair<Real, Real>>& timeline,
                         Real defaultValue,
                         bool stepFunction);
```

This function automatically orders the given timeline.

The second type of schedule is **expression-based**.

```
bool addExpressionSchedule(const std::string& fluidId,
                           const std::string& paramName,
                           const std::string& expression,
                           Real defaultValue);
```

When this function is called, the expression is parsed and compiled using *tinyexpr*:

```
bool compileExpression(ParameterSchedule& schedule) {
    te_variable vars[] = {
        {"t", &m_t_double},   // Current simulation time
        {"dt", &m_dt_double}  // Current timestep
    };
    int err;
    schedule.compiled_expr = te_compile(schedule.expression.c_str(),
                                        vars, 2, &err);
    return (schedule.compiled_expr != nullptr && err == 0);
}
```

The **step()** method is called each simulation timestep, which routes to the appropriate evaluation method based on the schedule type:

```
void step() {
    const Real currentTime = TimeManager::getCurrent()->getTime();
    const Real dt = TimeManager::getCurrent()->getTimeStepSize();

    m_t_double = currentTime;  // Updates for expression evaluation
    m_dt_double = dt;

    for (auto& schedule : m_schedules) {
        if (!schedule.active) continue;

        Real new_value;

        // Determine value based on schedule type
        if (schedule.use_expression) {
            new_value = te_eval(schedule.compiled_expr);
        } else {
            if (schedule.use_step_function)
                new_value = stepTimeline(...);
            else
                new_value = interpolateTimeline(...);
        }

        schedule.current_value = new_value;
        applyParameterValue(schedule.fluidId, schedule.parameter_name, new_value);
        }
    }
}
```

The **applyParameterValue()** method routes parameter changes to the appropriate simulation components.

1.2 Additions to SPlisHSPlasH files

In **Simulation**, we have to add a *m_ dynamicParameterSystem* member variable, as well as a *updateDynamicParameters()* method that calls the inner *step()* function.

Inside every **TimeStep**, at the end of the *step()* method, we add the line

```
sim->updateDynamicParameters();
```

In the **Utilities** subfolder we have to add some lines to the **SceneParameterObjects** and the **SceneLoader** to add the Dynamic Parameters to the scene parsing workflow.

In **SceneParameterObjects** we create a **DynamicParameterObject** that stores the entries from the scene json and we initialize its parameters.

By doing that, we can add to the **Scene** struct in **SceneLoader** a vector of DynamicParameterObjects. Finally, we add some lines in the **readScene** function:

```
////////////////////////////////////////////////////////////////////////
// read dynamic parameters
////////////////////////////////////////////////////////////////////////
if (m_jsonData.find("DynamicParameters") != m_jsonData.end())
{
        nlohmann::json dynParams = m_jsonData["DynamicParameters"];
        for (auto& param : dynParams)
        {
                DynamicParameterObject* data = new DynamicParameterObject();
                data->initParameters();
                readParameterObject(param, data);
                scene.dynamicParameters.push_back(data);
        }
}
```

1.3 Additions to Simulator files

Finally, in **SimulatorBase**, we add the **createDynamicParameters()** method, which we call from the **buildModel()** function. This is where we link the parsed scene to the simulation.

```
void SimulatorBase::createDynamicParameters() {
        Simulation* sim = Simulation::getCurrent();
        const Utilities::SceneLoader::Scene& scene =
        ↪  SceneConfiguration::getCurrent()->getScene();

        ////////////////////////////////////////////////////////////////////////
        // Dynamic parameters
        ////////////////////////////////////////////////////////////////////////

        if (scene.dynamicParameters.empty())
                return;

        if (sim->getDynamicParameterSystem() == nullptr)
        {
                DynamicParameterSystem* dps = new DynamicParameterSystem();
                sim->setDynamicParameterSystem(dps);
        }

        DynamicParameterSystem* dps = sim->getDynamicParameterSystem();
        for (unsigned int i = 0; i < scene.dynamicParameters.size(); i++)
        {
                DynamicParameterObject* data = scene.dynamicParameters[i];

                const std::string& fluidId = data->fluidId;
                const std::string& paramName = data->parameterName;
                Real defaultValue = data->defaultValue;
```

```
                if (!data->expression.empty())
                {
                        dps->addExpressionSchedule(fluidId, paramName,
                        ↪  data->expression, defaultValue);
                }
                else if (!data->timelineTimes.empty() &&
                ↪  !data->timelineValues.empty())
                {
                        std::vector<Real> times =
                        ↪  parseCommaSeparatedString(data->timelineTimes);
                        std::vector<Real> values =
                        ↪  parseCommaSeparatedString(data->timelineValues);

                        if (times.size() != values.size()) {
                                LOG_WARN << "Timeline times and values have
                                ↪  different sizes for parameter: " << paramName;
                                continue;
                        }

                        std::vector<std::pair<Real, Real>> timeline;
                        for (size_t i = 0; i < times.size(); ++i) {
                                timeline.push_back({ times[i], values[i] });
                        }

                        dps->addTimelineSchedule(fluidId, paramName, timeline,
                        ↪  defaultValue, data->stepFunction);
                }
        }
}
```