DynamicsLab Design Review

3/16/2018

Gavin Fielder, Ryan Lynn, Matt Johnson, Josh Thurston, Alex Elson

DynamicsLab Architecture

Table of Contents

Architecture inherent to Unity

Unity most closely follows the Broker pattern of software architecture. It has three classes of note inherent to its architecture:

- GameObject : Any object that receives updates in the update cycle is a GameObject.
- Component : A GameObject has one or more Components that attach to it. Every Component of every GameObject receives updates in the update cycle. Components can be attached within the Unity development environment or during runtime through code. Notably, the functionality for rendering to the screen is handled with Components, and the functionality to run developer-defined code is handled with Components also.
  - MonoBehavior is a base class that can be attached as a Component to a GameObject. Object behavior scripts inherent from MonoBehavior. The two required functions are:
    - Start() : called once for initialization
    - Update() : called once every frame
- Scene : Contains GameObjects. Only one scene can be loaded and receive updates at a time.

There are two ways for GameObjects to be added to a scene:

- In design: objects can be added to a scene as single, statically allocated elements within Unity's graphical development environment. They can be configured within Unity with components and various settings.
- During runtime: objects can also be instantiated in code.

A typical Unity program will use both of these methods in eacdh scene as appropriate.

How we have built upon this

We have created two scenes so far:

- MainScene : For the spring-mass system simulation
- Motion3D : For 3D motion simulation

Since scenes are mutually exclusive, each of these scenes have their own architecture.

The math library exists separately from Unity architecture.

Overall Architecture between major modules

Description

MainScene (Spring-Mass) and Motion3D are mutually exclusive and independent: they share some common assets, but are not coupled in any applicable sense.

We have instituted a 2-layer architecture overall: Both scenes depend on the math library, but the math library depends on neither.

Analysis

This overall architecture is ideal for our purposes because:

- The math library being an independent lower layer means it is maintainable and understandable outside the context of everything else the software does.
- The scenes not affecting each other makes them both more maintainable. In addition to satisfying, in application, the separation of concerns and single responsibility principles, it makes the scenes themselves cleaner and more understandable in Unity's graphical development environment.
- It fits within Unity's inherent architecture – we would not want to try to bypass inherent features in how Unity software works.

MainScene (Spring-Mass System)

Description

As this scene was our product out of the first sprint, we did not adhere to any particular design patterns in the spring-mass system simulation. Since it is a relatively low-value part of the overall product, we have not redesigned or restructured it, either.

The work in the scene is split between three classes:

- ViewModel : handles user GUI input. It contains event handlers for user GUI interactions.
- CreateGraph : creates a static grid upon scene load.

- CreateLines : handles everything else, including:
    - Creating and managing a SpringMassSystem, a solver contained in the math library.
    - Updating the position of the mass based on data contained in SpringMassSystem
    - Altering the rendering parameters of the spring to track the mass
    - Updating peripheral graphics, such as text displays and the line curve background effects.

Analysis

Immediately, we see that CreateLines violates the single responsibility principle. It is also not ideal that this scene has architecture that is inconsistent with the other scene. Were we to put more priority into this feature of our product, we would redesign the architecture of the spring-mass scene to the modified MVC architecture we use in Motion3D so as to make it more understandable, maintainable, and extensible. Reusability may not be a large factor since spring-mass is a particular problem.

In terms of specific design principles other than single-responsibility:

- the current architecture of the spring-mass scene does not need to worry about separation of concerns since it is largely one class.
- It is non-ideal in terms of the least-knowledge principle as ViewModel must read and write specific variables within CreateLines.
- It has no similar code in multiple blocks, and as such adheres appropriately to DRY principle.
- It is very minimal in up-front design, and so adheres remarkably well to YAGNI principle.

Motion3D

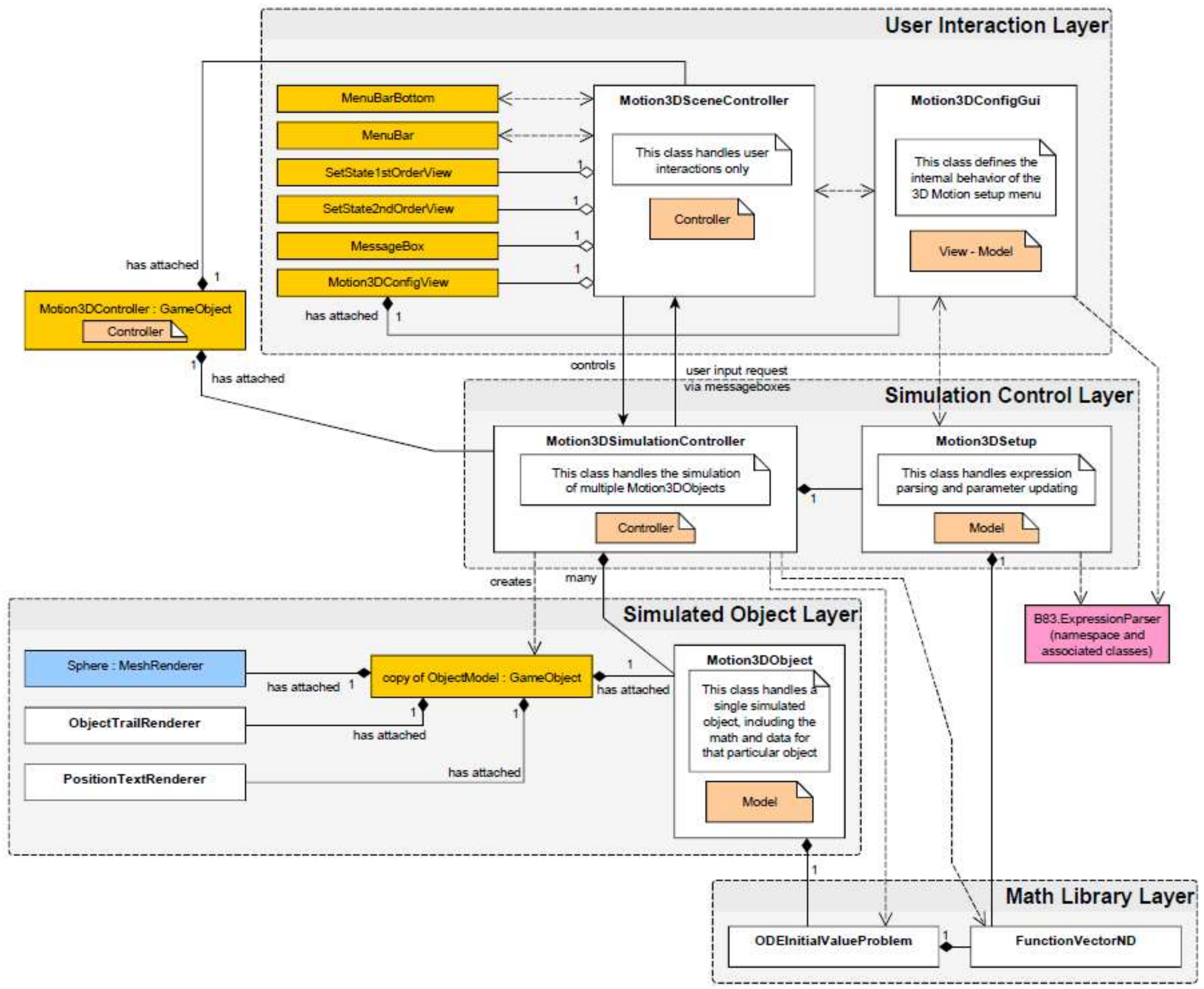
Description

For the 3D Motion scene, we implemented a layered adaptation of MVC design pattern.

- Controllers : Both are attached to a statically allocated Motion3DController object in the scene.
    - Motion3DSceneController : A MonoBehavior script which handles user interactions
    - Motion3DSimulationController : A MonoBehavior script which accepts commands from Motion3DSceneController to manage and control the simulation. Contains and manages the equations and multiple Motion3DObjects.
- Models
    - Motion3DObject : A MonoBehavior script which controls the motion of an object according to an internal ODEInitialValueProblem.
- Views : All of these are Unity UI structures that contains buttons, input fields, text, etc.
    - MenuBar
    - MenuBarBottom
    - SetState1stOrderView
    - SetState2ndOrderView
    - Motion3DConfigView
        - Has its own MonoBehavior script for internal behavior
    - MessageBox

While we would consider graphics rendering (such as MeshRenderer components on GameObjects) to be Views as well, this functionality is inherent to Unity. In many cases, it's sensible to not make much distinction between models and views in Unity, but it is sensible in the context of GUI menus.

The way the architecture is layered is as follows:


User Interaction Layer
MenuBarBottom
MenuBar
SetState1stOrderView
SetState2ndOrderView
MessageBox
Motion3DConfigView
Motion3DSceneController
This class handles user interactions only
Controller
Motion3DConfigGui
This class defines the internal behavior of the 3D Motion setup menu
View - Model
has attached
Motion3DController : GameObject
Controller
controls
user input request via messageboxes
Simulation Control Layer
Motion3DSimulationController
This class handles the simulation of multiple Motion3DObjects
Controller
Motion3DSetup
This class handles expression parsing and parameter updating
Model
creates
many
B83.ExpressionParser (namespace and associated classes)
Simulated Object Layer
Sphere : MeshRenderer
ObjectTrailRenderer
PositionTextRenderer
copy of ObjectModel : GameObject
Motion3DObject
This class handles a single simulated object, including the math and data for that particular object
Model
Math Library Layer
ODEInitialValueProblem
FunctionVectorND

The user only interacts with the top layer, and the user interaction layer then in turns outputs to the simulation control layer. In turn, the simulation control layer outputs to the simulated object layer. The only exception to this is when the simulation controller layer needs user input through a messagebox, and so then must send a messagebox request back up to the Motion3DSceneController. Both simulation layers are built onto the math library layer.

Analysis

The layered architecture produces good separation of concerns: there is minimal interaction between classes on different layers, and with the exception of message box functionality, the interactions flow in only one direction.

Classes do not need to know extensive details of the inner workings of other classes, and for the most part layers don't need to have knowledge of the details of layers that are more than one layer separated. The exception here is that both simulation layers need to have some knowledge of both the math library and the type of math in general so as to at least understand the basic principles of what is required to solve a problem (ie, knowing that solutions require initial conditions).

The classes are appropriately split for good single-responsibility design, and each class has an important, specific role.

The object-oriented design and implementation makes good use of DRY principle—for example, ObjectTrailRenderer being its own Component enables it to be used identically on each object.

It does not have extraneous features, and so adheres to YAGNI principle.

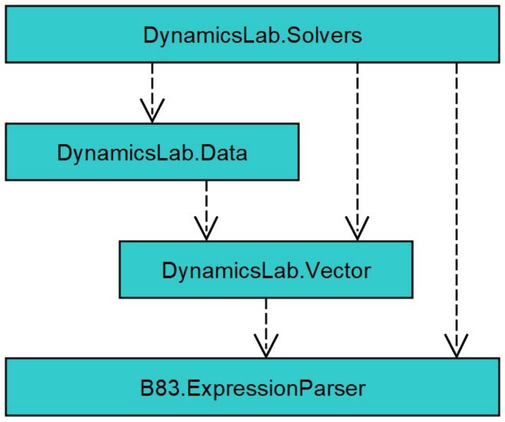
For the most part, each class is as simplistic as possible while maintaining control over its responsibility, and so adheres to KISS. The messagebox functionality stands out as the more complicated feature of interaction between the user interaction layer and the simulation control layer, but likely could not be simplified further while maintaining that only the user interaction layer interacts with the user.

Overall, the architecture for 3D motion is likely already in its ideal form. The one point for possible improvement is in the simulation control layer requesting user input, though it's possible this particular need may not have a better solution than to allow some reverse-flow interaction.

Math Library

Description

The math library is designed to be independent from Unity architecture. It follows a layered design pattern in different namespaces based on area of responsibilities. These namespaces are layered as following, with higher layers being dependent on lower layers:



- DynamicsLab.Solvers
    - ODEInitialValueProblem – Handles an initial value problem and stores solution data in a DataSet.
    - ODEIVPSolver – Solves an initial value problem.
    - SpringMassSystem – Wraps an ODEInitialValueProblem for a spring-mass problem.
- DynamicsLab.Data

    - DataSet – Handles a dynamic set of data consisting of n-dimensional numerical vectors. Contains any number of DataChunks. All members are public because the class is internal to the assembly and should not need to be used outside the math library.
    - DataChunk – Handles an array of 1000 VectorND's (essentially a 1000 x n table of doubles). Also an internal class, so all members are public.
- DynamicsLab.Vector
    - VectorND – a basic n-dimensional numerical vector. Has no dependencies.
    - FunctionVectorND – an n-dimensional vector function using an array of delegates.
- B83.ExpressionParser : this is an external asset not of our design.

The requirements of the math library in general, up through 3D motion were:

- Be able to solve any first order ODE IVPs in 2 dimensions, 3 dimensions, and 6 dimensions.
- Be able to store as much data as required in a dynamically growing data set.
- To do both of the above reasonably quickly.
- To be able to track the interval of valid solution data.

Analysis

This architecture adheres well to the single-responsibility and separation of concerns design principles.

ODEInitialValueProblem may be too closely coupled with DataSet, as it accesses the members of DataSet directly rather than through a public interface. DataSet having all public members may not be the best design going forward—it mostly depends on what functionality is added onto the math library in the future.

The classes are as simple as possible while meeting the design requirements while also adhering to the DRY principle. DataSet is the most complicated and difficult to understand class, but it's a good solution to the design requirements. Ensuring that it has ample documentation and comments would be a better safeguard for maintainability, extensibility, and reusability than refactoring the class itself.

The math library has adhered to the YAGNI principle in two ways: during the first sprint, the math library was hard-coded to only implement the spring-mass system and was later expanded more generally to suit the more varying requirements of 3D motion. Nothing other than ODE IVPs have been implemented yet, as well, which is all that both spring mass and 3D motion require. The present math library is the minimum implementation it needs to be right now.

Class Design and Cohesion

All classes seem appropriately cohesive. Motion3DObject has an LCOM4 of 3 due to its Highlight and Dehighlight functions, which are appropriate to include in this class since it adheres to the architectural design. All other classes complex enough to consider have an LCOM4 of 1.

LCOM4 Results Summary

A full list of analyzed results is at the bottom of this document.

Class	**LCOM4**
Motion3DConfigGui	1
Motion3DSetup	1
Motion3DObject	3
Motion3DSimulationController	1
ObjectTrailRenderer	1
ODEInitialValueProblem	1
DataSet	1

Classes left un-analyzed for cohesive design:

- **Motion3DSceneController**, because it's a skinny controller whose main purpose is holding gui handlers. LCOM4 is not an appropriate metric for this class in particular.
- **VectorND**, because it's a small and basic class we feel is well-built for its purpose
- **FunctionVectorND**, same
- **DataChunk**, because it's only really an internal class for DataSet. It's small and simple.
- **SpringMassSystem**, because we ran out of time and it's a low priority for the overall product as far as we know
- **CreateLines**, because it's designed with a two big functions anyway and is bound to have an LCOM4 of 1, which is irrelevant for its design value.
- **CreateGraph**, because we may not even continue to use it
- Any extremely small and simple classes

Function Analysis

Most functions are simple in that they have three independent branches or fewer.

Most Complex Functions

These functions are identified as posing a potential problem for maintainability and extensibility:

File	Function	Complexity	Comment
Motion3DConfigGui	SeparateCSV	12	Fine for its purpose. It's a well-defined function anyway.
CreateLines	Update	9	Could be broken into multiple functions. Since this is a low priority part of the overall product, we may or may not redesign it.
Motion3DConfigGui	ValidateExpression	9	While this is fairly complex it’s needed in order to validate the expressions.
Motion3DSceneController	SetState2ndOrderView_OkButtonPress	8	This is all just input checking, it’s actually a simple function.
Motion3DConfigGui	UpdateStatusDisplay	7	The status display has a lot of things to check, so this complexity is expected.
Motion3DConfigGui	Update	6	Since this is modelled to detect changes and update only what needs updating, this complexity is expected.
Motion3DSceneController	Update	6	This function handles various cases of user input, so this complexity is expected.
dlab-ivp	InvalidateData	6	This function could use re-examination, especially since it's tightly coupled with DataSet
CreateLines	Start	5	This is small enough in complexity to be fine. This functions is meant to just initialize everything for CreateLines. This complexity is simple enough for that.
Motion3DConfigGui	PopulateFields	5	This is small enough in complexity to be fine.
Motion3DSceneController	SetState1stOrderView_OkButtonPress	5	This is small enough in complexity to be fine.

Code Smells Identified

The following code smells could be identified and may affect maintainability, reusability, and/or extensibility.

- Comments (too many comments, overly explained code)
    - TODO: where does it seem overly explained or too many comments
    - Dlab-vector.cs
        - Maybe a separate documentation file would be useful for explaining the components of this file
    - Dlab-ivp.cs
        - Separate documentation as there are over 100 lines of comments explaining how each function works
- Dead code (code that no longer does anything)
    - Dead code in CreateLines
        - velX, accel, oldmx and their accessing code are dead
    - MotionMaster is completely dead now. It's no longer used, granted.
    - Motion3DConfigGui - extra includes not needed
    - Motion3DSceneController - in function DisplayMessageBox, chunk of commented out code
    - To review: dim in Motion3DSetup
    - TODO: anything else we can find?
- Sequential coupling (class requires function calls in a certain order)
    - ODEInitialValueProblem has this
        - SetState needed after InvalidateData
        - SetState needed before SolveTo (can't really get around this though, it's a basic requirement of the math)
    - SpringMassSystem maybe has this
        - SetInitialCondition needed before Update (can't really get around this though, it's a basic requirement of the math)
- Diaper pattern (catching all exceptions)
    - this is in ValidateExpression in Motion3DConfigGui
- Duplicate code
    - CreateGraph.Start has very similar code blocks
- Long methods (number of lines)
    - CreateGraph.Start: 28 lines
    - Line.Update : 31 lines
    - Motion3DConfigGui.Update: 29 lines
    - Motion3DConfigGui.SeparateCSV: 35 lines
    - Motion3DConfigGui.ValidateExpression: 99 lines
    - Motion3DSimulationController.SetState: 41 lines
    - Motion3DSimulationController.SetState_callback: 27 lines
    - Motion3DSimulationController.AddObject: 27 lines
    - ObjectTrailRenderer.UpdateTrail : 55 lines
    - CreateLines.Start : 54 lines
    - CreateLines.Update : 71 lines
- Large classes (in number of fields or number of methods)
    - Motion3DConfigGui has 22 fields and 14 methods
    - Motion3DSceneController has 22 methods
    - Motion3DSimulationController has 19 fields and 19 methods
    - CreateLines has 23 fields
    -

- Temporary field
    - Motion3DSimulationController has tempsave_state and tempsave_time. It uses these in the messagebox callback functionality.
- Inappropriate intimacy
    - Motion3DSimulationController accesses ODEInitialValueProblem inside Motion3DObject
    - ODEInitialValueProblem accesses members of DataSet
    - ViewModel has lines where it grabs references to fields in CreateLines.
- Speculative Generality (things that aren't yet implemented)
    - ClickScript.cs - created skeleton class with no functionality
- Switch Statements
    - Motion3DConfigGui has large switch statement with multiple empty break switch cases.

Other Data Collected

LCOM4 Calculations and Notes

<u>Motion3DConfigGui</u>

1. parametersInput	7. equationLabelX	13. expressionYChanged	19. exprZ
2. expressionInputX	8. equationLabelY	14. expressionZChanged	20. variablesAvailable
3. expressionInputY	9. equationLabelZ	15. valid	21. parameters
4. expressionInputZ	10. orderToggle	16. order	22. initialized
5. variableDisplays	11. parametersChanged	17. exprX	
6. validationStatusDisplay	12. expressionXChanged	18. exprY	

Parameters

a. Start() b, h, 12, 13, 14, 15, 16, 21, 22	h. OnOrderSwitch() i, 5, 7, 8, 9, 10, 12, 13, 14, 15, 16, 20
b. hookObjectReferences() 1, 2, 3, 4, 5, 6, 7, 8, 9, 10	i. SeparateCSV()
c. Update() i, j, 11, 12, 13, 14, 21	j. ValidateExpression() k, l, 2, 3, 4, 6, 16, 20, 21
d. OnParametersChanged() 11	k. UpdateStatusDisplay() 2, 3, 4, 6, 15
e. OnExpressionXChanged() 12	l. CheckInvalidFlags() j, 15
f. OnExpressionYChanged() 13	m. exportForSetup() 2, 3, 4, 15, 16, 21
g. OnExpressionZChanged() 14	n. PopulateFields() a, h, j, 1, 2, 3, 4, 10

Functions

LCOM4 = 1.

<u>Motion3DSetup</u>

1. expressionX 2. expressionY	3. expressionZ 4. parameters	5. order 6. dim	7. parser 8. F_internal

Parameters

a. ChangeParameter() c, 4 b. GetParameterValue() 4 c. Parse() d, e, 5 d. Parse1stOrder() f, 1, 2, 3, 8	e. Parse2ndOrder() f, 1, 2, 3, 4 f. IncludeParameters() 7 g. Motion3DSetup() c, 1, 2, 3, 4, 5, 6, 7, 8

Functions

*Note: Motion3DSetup() is a constructor, however it is necessary to be included here because it is the only place that uses dim. dim may not need to be a member variable, so could be hardcoded into the constructor, in which case it wouldn't need to be listed here.

LCOM4 = 1.

Motion3DObject

1. ivp 2. xIndex	3. yIndex 4.zIndex	5. currentTime 6. forwardSolveTime	7. resetState 8. reseTime	9. trailRenderer

Parameters

a. Start() 9 b. Update() c c. UpdateObjectPosition() 1, 2, 3, 4, 5 d. SetAs1stOrder() 2, 3, 4 e. SetAs2ndOrder() 2, 3, 4 f. GetCurrentState(val) 1, 2, 3, 4, 5	g. GetCurrentState() f, 1 h. ResetState() 1, 5, 7, 8, 9 i. SetState() h, 7, 8 j. Highlight() k. Dehighlight() l. CurrentTime[Property] set 1, 5, 6

Functions

*Note: CurrentTime is a C# property, not a function, but it is the only place where the member variable forwardSolveState is mentioned aside from its own property, so I included it.

LCOM4 = 3.

Motion3DSimulationController

1. simulatedObjects 2. currentObjectIndex 3. simulationIsRunning 4. motion3DSetup	5. objectModel 6. currentTime 7. resetTime 8. speed	9. allObjectsDataLowerBound 10. allObjectsDataUpperBound 11. forwardSolveTime 12. sceneController	13. startStopButtonText 14. timeDisplaytext 15. state_tempsave 16. time_tempsave

Parameters

a. Start() 1, 4, 5, 11, 12, 13 b. Update() 1, 3, 4, 6, 8, 14 c. ApplyConfiguration() d, 1, 4 d. ApplyConfigurationToObject() 1, 4, 6	j. RemoveObject() 1, 2 k. SelectNextObject() 1, 2 l. SelectPrevObject() 1, 2 m. SelectObject() 1, 2

e. ChangeParameter() d, 1, 4 f. SetStateOnCurrentObject() g, p, 1, 2, 6, 7, 9, 10, 11, 12, 15, 16 g. SetStateOnCurrentObject_Callback() 1, 6, 7, 9, 10, 11 h. GetCurrentState() 1, 2 i. AddObject() d, 1, 2, 4, 5, 6, 11	n. StartStopToggle() o, p, 3 o. StartSimulation() 3, 13 p. StopSimulation() 3, 13 q. IncreaseSpeed() 8 r. DecreaseSpeed() 8

Functions

LCOM4 = 1.

<u>ObjectTrailRenderer</u>

1. c1 2. c2	3. lineX 4. lineY	5. lineZ 6. lineRenderer	7. lineNum 8. count

Parameters

a. Start() c, d b. Update() f c. InitializeTrail() d, 1, 2, 6, 7	d. NewTrail() 3, 4, 5, 7 e. resetTrail() d, 8 f. UpdateTrail() 3, 4, 5, 6, 7, 8

Functions

LCOM4 = 1.

<u>ODEInitialValueProblem</u>

1. F	2. solver	3. Solution

Parameters

a. GetDim() 3 b. GetH() 3 c. GetDataLowerBound() 3	d. GetDataUpperBound() 3 e. SolutionData() 3 f. SetH() 3	g. InvalidateData() 3 h. SolveTo() 1, 2, 3 i. SetState() 3

Functions

LCOM4 = 1.

<u>DataSet</u>

1. Data 2. dataLowerBound 3. dataUpperBound 4. maxIndex	5. minIndex 6. tSlope 7. tIntercept 8. h	9. dim

a. DataSet: all fields B. This[double]: E, 1 C. This[int]: E, F, G, 1 D. This[int, int]: 1	E. Lookup : 7, 8 F. ChunkIndex G. LocalIndex H. Tval: 6, 7	I. WriteState: 1 J. WriteNext: 1,K,F,G K. NewChunk L. RemoveFirstChunk: 1,7 M. RemoveLastChunk: 1,6,7

LCOM4 = 1

All cyclomatic complexity metrics for complexity >1

Spring mass system

CreateLines.cs

Start:4 if statements - V(G) = 5
This is small enough in complexity to be fine. This functions is meant to just initialize everything for CreateLines. This complexity is simple enough for that.

Update:8 if statements - V(G) = 9
This is small enough in complexity to be fine. When updating this script it needs to have a fairly large complexity in order to handle everything that the lines need to display.

Motion3D

Motion3DConfigGui.cs

Update:1 if statement, 3 else if statements, 1 for statement - V(G) = 6
This is small enough in complexity to be fine. Updating tends to take more complexity then a normal function in order to handle what's thrown at it.

OnOrderSwitch: 1 if statement - V(G) = 2
This is small enough in complexity to be fine. It's a relatively simple function for switching text and order.

SeparateCSV:1 Switch 9 cases, 1 if statement, 1 for loop- V(G) = 12
This function may be overly complex, but this is due to the switch statements required by all the key tokens used in string parsing. It cannot be further abstracted even if split into two functions.

ValidateExpression: 1 switch 3 cases, 3 if statements, 2 for loops - V(G) = 9
This is small enough in complexity to be fine. While this is fairly complex it's needed in order to validate the expressions.

UpdateStatusDisplay: 4 if statements, 2 &&'s - V(G) = 7
This is small enough in complexity to be fine.

CheckInvalidFlags: 6 if statements, 3 exits- V(G) = 4
This is small enough in complexity to be fine.

exportForSetup: 1 if statement, 2 &&'s, 1 exit - V(G) = 3
This is small enough in complexity to be fine.

PopulateFields: 3 if statements, 1 for loop - V(G) = 5
This is small enough in complexity to be fine.

Motion3DObjects.cs

UpdateObjectPosition: 1 if statement, 1 || - V(G) = 3
This is small enough in complexity to be fine.

GetCurrentState: 1 switch statement 8 cases, 7 exits - V(G) = 1
This is small enough in complexity to be fine.

Motion3DSceneController.cs

Update: 5 if statements - V(G) = 6
This is small enough in complexity to be fine. Updating tends to take more complexity then a normal function in order to handle what's thrown at it.

MenuBar_SetStateButtonPress: 1 if statement - V(G) = 2
This is small enough in complexity to be fine.

ConfigView_OkButtonPress: 1 if statement - V(G) = 2
This is small enough in complexity to be fine.

SetState1stOrderView_OkButtonPress: 4 if statements - V(G) = 5
This is small enough in complexity to be fine.

SetState2ndOrderView_OkButtonPress: 7 if statements - V(G) = 8
This is small enough in complexity to be fine.

Motion3DSetup.cs

Motion3DSetup: 1 statement - V(G) = 2
This is small enough in complexity to be fine.

Parse: 1 if statement - V(G) = 2
This is small enough in complexity to be fine.

Motion3DSimulationController.cs

Update: 1 if statement - V(G) = 2
This is small enough in complexity to be fine. Updating tends to take more complexity then a normal function in order to handle what's thrown at it.

ApplyConfiguration: 1 if statement - V(G) = 2
This is small enough in complexity to be fine.

ApplyConfigurationToObject: 2 if statements - V(G) = 3
This is small enough in complexity to be fine.

AddObject: 2 if statements - V(G) = 3
This is small enough in complexity to be fine.

RemoveObject: 1 if statement - V(G) = 2
This is small enough in complexity to be fine.

SelectNextObject: 3 if statements - V(G) = 4
This is small enough in complexity to be fine.

SelectPrevObject: 3 if statements - V(G) = 4
This is small enough in complexity to be fine.

SelectObject: 2 if statements - V(G) = 3
This is small enough in complexity to be fine.

StartStopToggle: 1 if statement - V(G) = 2
This is small enough in complexity to be fine.

ObjectTrailRenderer.cs

UpdateTrail: 2 if statements - V(G) = 3
This is small enough in complexity to be fine.

MathLibrary

dlab-data.cs

DataChunk: 1 for loop - V(G) = 2

WriteNext: 1 if statement - V(G) = 2

RemoveFirstChunck: 2 if statements - V(G) = 3

dlab-ivp.cs

InvalidateData: 2 if statements, 3 while loops - V(G) = 6

SolveTo: 1 while loop - V(G) = 2

ODEIVPSolver: 1 for loop - V(G) = 2

RKCallBuild: 1 for loop - V(G) = 2

dlab-vector.cs

VectorND: 1 for loop - V(G) = 2

VectorND operator +: 1 for loop - V(G) = 2

VectorND operator -: 1 for loop - V(G) = 2

VectorND operator *: 1 for loop - V(G) = 2

VectorND operator *: 1 for loop - V(G) = 2

FunctionVectorND: 1 if statement - V(G) = 2

Eval: 1 for loop - $V(G) = 2$