Project 4: Better interfacing of Blender and MDAnalysis

Project proposal for GSoC 2025

- Introduction
- Relevant background and experiences
- Problem to be solved
  - Background
  - Overarching question
  - Why this is an interesting / difficult problem
  - Overall goal
- Approach
  - Design considerations
    - Ease of use / Usability
    - Consistency
    - Data
    - Extensibility
    - Leveraging Blender
  - Design choices
    - Blender properties for data
    - Universe centric approach
    - Annotations (Text / Lines / Objects)
    - Blender Timeline
    - Camera, Lighting and Background
    - MDAnalysis selection phrases and AtomGroups
    - Named entities
  - APIs
    - APIs for styling representations
    - APIs for annotations
    - APIs for components
    - APIs for rendering
  - GUI
- Measurable outcomes
- Timeline
- Conclusion
- References

Introduction

This document is a project proposal for “Project 4: Better interfacing of Blender and MDAnalysis” as part of GSoC 2025. More details about this project can be found on the MDAnalysis wiki under GSoC 2025 Project Ideas.

Relevant background and experiences

I am Pardhav Maradani, a first year undergraduate student from India (class of ‘28), pursuing a Bachelors in Computer Science and Engineering from Vellore Institute of Technology and a parallel BS in Data Science and Applications from IIT Madras.

I have a keen interest in Visualization as my programming journey started with interactive graphics programming with Python and turtle graphics several years back at the beginning of my highschool (around 9th grade). All the code and interactive examples from that time can be found on my blog - Pardhav’s blog. My exposure to Blender started with a 3D-printing hobby and due to my familiarity with Python at that point, it led to some basic scripting and simple add-on programming within Blender to address some custom use cases. I have since used Blender extensively for my various 3D modelling needs.

Though I am new to this space, I have been playing with MDAnalysis and Molecular Nodes (MN) since early December 2024 when I started exploring some aspects of this project out of curiosity. This led me to initiating a discussion on Visualization of MDAnalysis results in Blender, which showcased some very early prototyping and potential possibilities. At that time I wasn’t sure if there was still interest around this idea because it had been a previous GSoC project idea and there seemed to be no recent updates. It was Brady who mentioned that this is being actively explored and pointed to the ggmolvis effort.

I have since taken part in some discussions and contributed to both Molecular Nodes (Issues / Pull requests) and ggmolvis (Issues / Pull requests) by sharing my experiences and learnings from exploring and prototyping some aspects of this project over the past few months whenever I found some time. I have also had a few PRs merged into MN for issues that I ran into during this exploratory process.

Problem to be solved

This section describes the background of the problem and formulates an overarching question that needs to be answered as part of this project.

Background

As outlined in the project description, Blender is an industry-leading 3D modelling and animation software that is completely open source. Molecular Nodes is a very popular Blender extension that already supports importing MDAnalysis universes into Blender and provides a lot of tools through Blender's Geometry Nodes interface to style, animate and render very cool visualizations of these universes and trajectories. Blender's Geometry Nodes interface, though very powerful, is not necessarily very user friendly for beginners. Though some aspects of Molecular Nodes can already be scripted, it lacks a formal API for scripting at the moment and this is currently in the process of being developed. GGMolVis is a project that started more recently with a goal to provide a high-level interface to Molecular Nodes that makes visualization tasks specific to MDAnalysis easily programmable from Jupyter notebooks and other similar environments. The project is in its early stages and only an initial set of APIs are defined at this point. This project is also being actively developed. The work done as part of this GSoC 2025 project will help move both the projects forward by helping define, prototype and implement several APIs and features that provide a powerful visualization solution for MDAnalysis results.

With this background, the overarching question could be stated as follows:

Overarching question

How do we build a visualization tool for MDAnalysis that is easy to use, intuitive, programmable and extensible that will allow users to explore, visualize and analyze MDAnalysis results?

Why this is an interesting / difficult problem

Blender is by far the best open source choice for implementing any new or custom visualizations of complex stuff like molecular dynamics. It has a very rich programmable interface via Python that also includes infrastructure to build custom GUI elements that can be made part of its overall interface. Molecular Nodes already provides a great platform to visualize MDAnalysis universes. By providing a rich scripting API and an integrated GUI within Blender that work hand in hand, we could address several visualization use cases both via a headless mode, where everything can be scripted through the API and a GUI mode that allows for a lot more interactivity, exploration and analysis. The endless possibilities of visualization that this opens up is what makes this an interesting and exciting problem for me. Blender however is a very complex piece of software. There are several caveats and inner workings that one needs to be familiar with to program and leverage all the great stuff that Blender has to offer. This is what makes it a difficult problem as well.

Overall goal

My overall goal as part of this project is to help define, prototype and build the visualization API along with the corresponding GUI within Blender that can address current and future MDAnalysis visualization use cases which can be integrated seamlessly into an existing project like GGMolVis, Molecular Nodes or a standalone project.

Approach

This section outlines the approach I intend to use to answer the overarching question stated above and reach the specified goal.

Design considerations

There are several design considerations we have to take into account to achieve our overall goal. This will ensure that we can address all current and potential future use cases while keeping everything consistent and leveraging all the features that Blender offers.

Here is a list of such design considerations:

Ease of use / Usability

We should not expect users to be familiar with Blender or Blender's Python API. All of the visualization APIs or the GUI provided should be self sufficient for users to accomplish their MDAnalysis visualization use cases. APIs should be intuitive and follow a consistent theme of other visualization tools that MDAnalysis users might already be familiar with, wherever possible. Blender provides a very extensive python API of its own where a lot of what can be done through its GUI can be scripted as well. We should not attempt to expose all of that API but instead provide well defined interfaces that will address the most common use cases with sensible defaults.

Consistency

MDAnalysis users can interface with Blender in one of two possible ways: 1. A pure headless mode, where there is no Blender GUI involved and 2. A GUI mode where Blender GUI is available alongside their common programming environments (like Jupyter Notebooks). These modes are not mutually exclusive. Users could be using one or the other depending on their constraints and sharing Notebooks with others who could have different preferences. Whatever we build should be consistent across both these modes from the very beginning. A GUI mode obviously provides a lot more interactivity and ease of use for most of the use cases, but similar to Blender (which has a GUI and a background mode as well),

we should have both working in sync. The consistency consideration applies across the board to a lot of other areas as well.

Data

There should be a single source of truth for all data used by our APIs and GUI. APIs and GUI should also be completely in sync with each other through this underlying data layer. Any changes made through API should instantly reflect in the GUI and any interactive customizations made through the GUI should be exportable back as complete API calls or available from the APIs directly. This will ensure repeatability and reproducibility. This is something we should consider right from the very beginning. Without this, these could deviate significantly very soon and add to additional layers of complexity in the future. Blender also has certain constraints on how the custom GUI elements are built (because it gets embedded within Blender's own interface), which dictates the way underlying data is stored within Blender. Factoring this in while designing the APIs will keep them in sync.

Extensibility

Similar to MDAnalysis, we want the visualization tool to also be extensible. Though we will provide some of the most common visualizations, we do want users to be able to extend the tool for their own custom analysis needs as part of their research. We would still not expect such users to be familiar with Blender's Python API and hence provide all the basic building block APIs that they can build their custom visualization analyses on. Our own MDAnalysis visualizations will be built on top of these building blocks (for example annotating text / drawings / objects in 3D space, 2D space, styling different representations, etc) and will serve as examples for how they can be extended. This requires the API layer to be simple and user-friendly for both end users and other MDAnalysis developers equally and hence needs abstracting out Blender internals as much as possible for the sake of extensibility.

Leveraging Blender

As mentioned in one of the sections above, Blender is extremely powerful in what it offers - all of it via a nice programming interface. If we look at it from the perspective of just generating simple images and videos as outputs, we could be limiting ourselves - even for our use cases. Blender has a really nice animation subsystem, a compositing subsystem, constraints and tracking mechanisms for camera and several others that could be leveraged for our use cases as well to go beyond the ordinary and provide complex visualizations similar to Molywood, but all within a single unified system, thanks to Blender. This however requires us to design keeping these in mind so they can be leveraged. For example, only non ID properties ("bpy.props.*") can be keyframed and hence used in the animation subsystem. Blender also provides a way to expose inputs from Geometry Node trees directly in the GUI (also making them available to the animation system as mentioned before) without us having to write an intermediary layer in between. Building an API by taking all of these into account will help us leverage Blender to the fullest and minimize the need for additional complexity.

Design choices

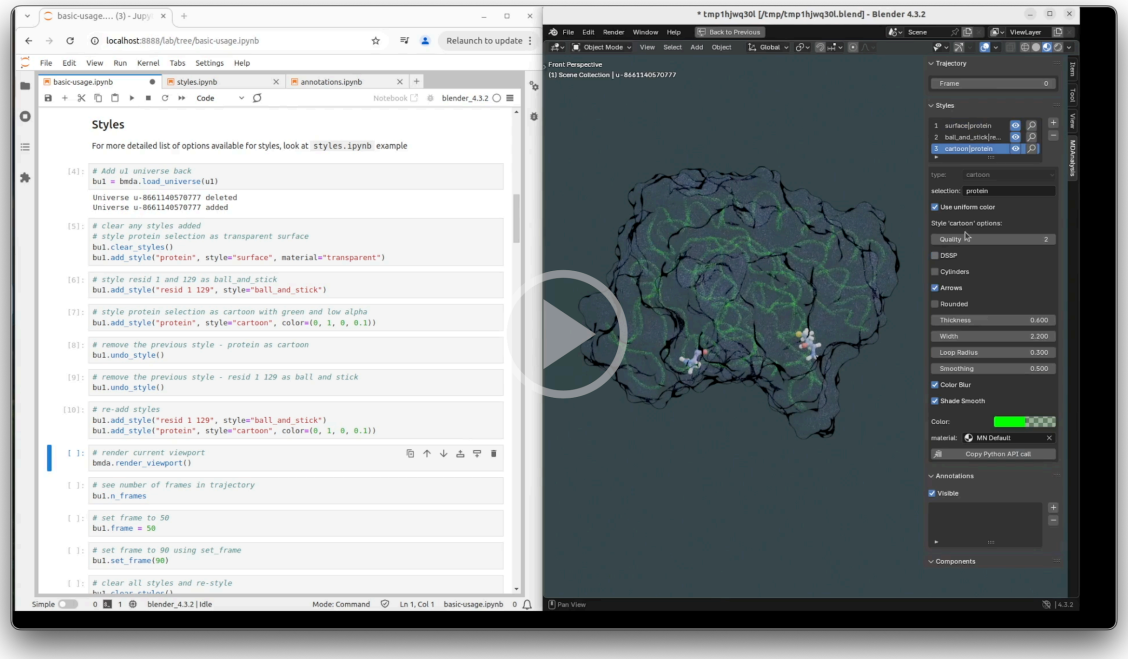
This section outlines some of the design choices we can make based on the design considerations from the previous section.

Blender properties for data

By using Blender properties (bpy.props) for data, we address several of the design considerations above.

Here is an example video that shows the APIs and GUI being in sync with this approach:

(Note that when images are hyperlinked to videos in this document, the same direct link is also inserted below the image for convenience)



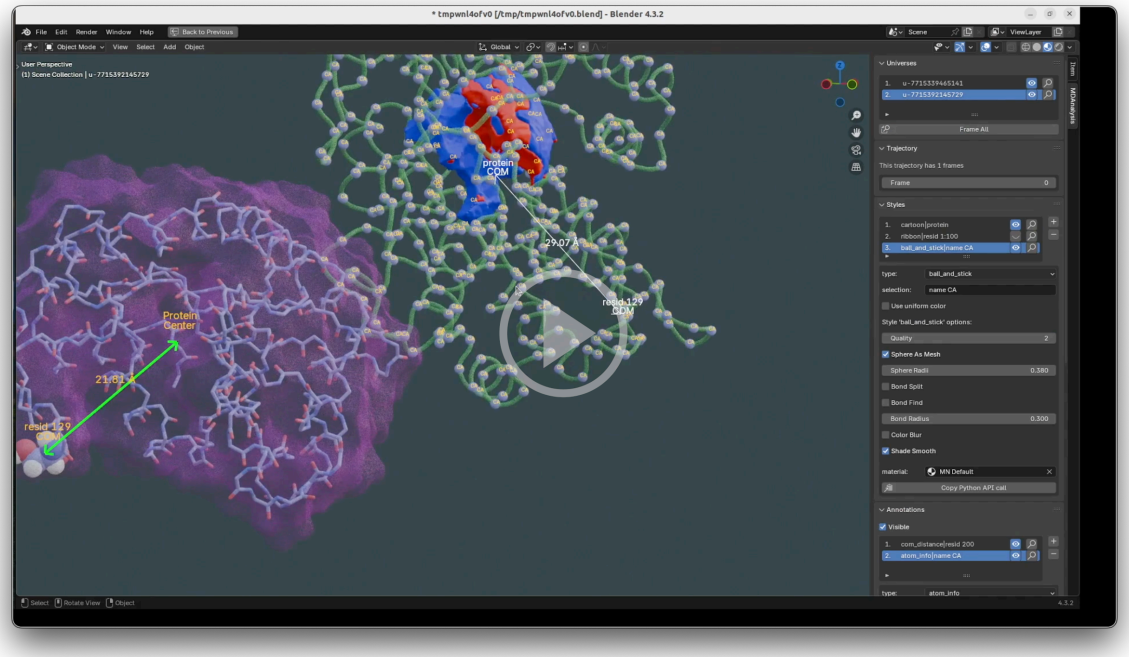
Direct video link:
https://drive.google.com/file/d/1QIU0Hwz2iDSPNIzr9Y7fOddsDPHB93Ro/view

The above is in a GUI mode setup. On the left, we can see the API calls made from within a Jupyter Notebook and on the right we can see the same reflected in the GUI real time allowing interactive customizations. The “Copy Python API call” button in the GUI would generate the API call that matches the customizations made so that they can be pasted back into the Notebook cells. Reading the corresponding entity from the API will also show the updated values from the GUI.

Universe centric approach

This document proposes a universe centric approach where one universe corresponds to one object within Blender. This differs from the approach within ggmolvis as of this writing and is discussed in issue #5 and issue #11. This has a lot of advantages as outlined in issue #5 (comment) and issue #11 (comment). This also keeps Blender's outliner clean, reduces memory footprint and helps with everything downstream as it organizes the universe around a single object helping the GUI, the data layer etc. This requires the ability from Molecular Nodes to allow styling different representations of the same object.

Here is an example video showing how the data organized at an object level helps with the overall usability in the GUI mode when multiple universes are involved with multiple representations, annotations and components involved:



Direct video link:
https://drive.google.com/file/d/1VZ4VrCDDWnxPOsj2EdOHYrHY_8JkzlwE/view

The above shows a couple of universes added through the API and customized from the GUI with various representations and annotations.

Annotations (Text / Lines / Objects)

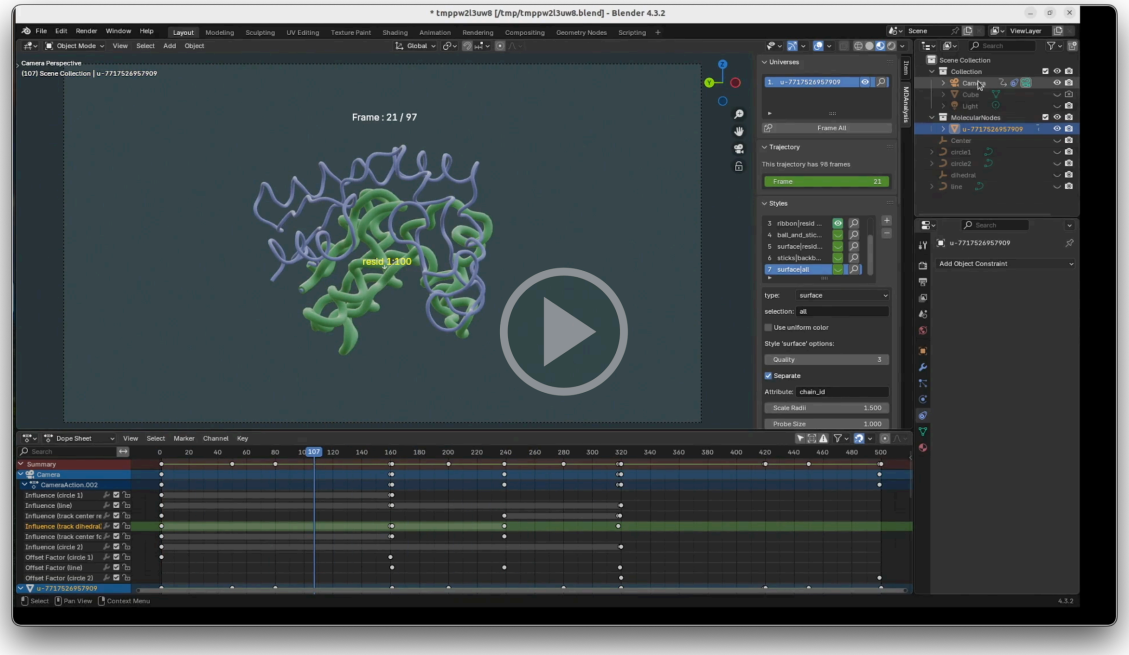
Annotations are an integral part of visualization. Several MDAnalysis results can be annotated onto the universe using text, lines and objects providing very useful context for deeper analysis. Blender provides a few different ways to represent these. The text related options are described in issue #27 along with their pros and cons. Blender's blf and gpu modules provide a common approach that several popular tools (including Blender itself)

take to achieve overlays in Blender's 3D viewport. This document proposes using these mainly along with regular Blender Text and primitive objects as options available for some specific use cases. There are known issues due to the nature of the 'blf' and 'gpu' modules that have to be worked around to get them to show up in renders. A prototype of these working can be see in [issue #27 (comment)].

Blender Timeline

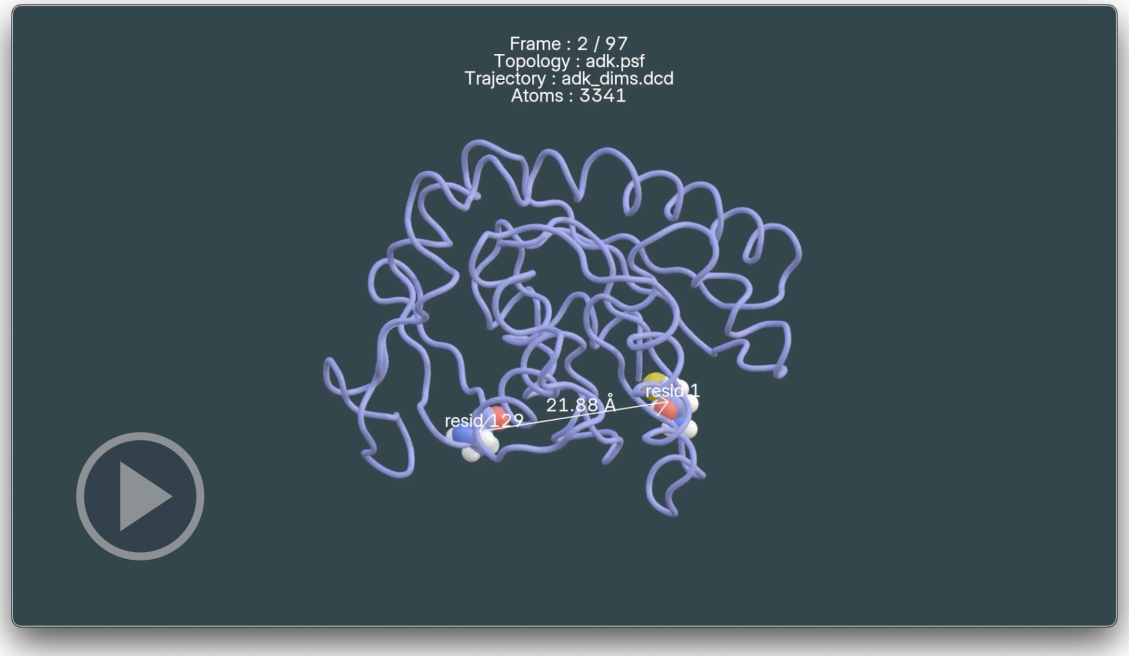
Blender has a single timeline that is used in a variety of ways to create animations. The current approach of ggmolvis and MN as of this writing ties the universe trajectory frames to this timeline. We should not explicitly tie our universe(s) frames to Blender's timeline given that we can have multiple universes with varying trajectory lengths and requirements to keep some selections / universes static etc. MN already has a Update with Scene option and individual nodes for animation. Keeping these decoupled, but with an ability to link / unlink virtually (by keyframing) will provide the maximum flexibility to render a variety of different animations. Building the API with this in mind from the beginning will let us achieve this goal.

Here is an example video of a setup of a complex animation from the GUI mode where various properties are keyframed onto Blender's timeline:



Direct video link:
https://drive.google.com/file/d/1klHtubVDEo5coqgqg_PcjpQxbpCaYQ_W/view

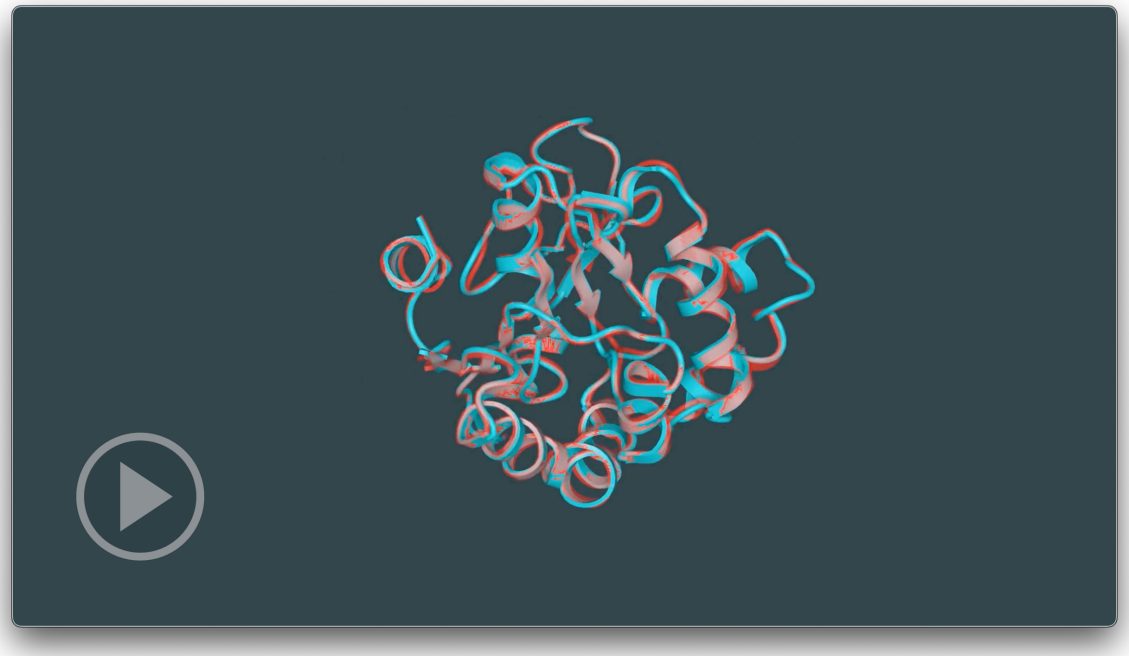
Here is the output animation video from the above setup:



Direct video link:
https://drive.google.com/file/d/194cJ8yK1S4wVKrszZ4Ucs-27GqrOpMbb/view

The above is also a great example to show how building APIs and GUI the right way can directly leverage Blender's animation system as well for complex use cases. Though the above requires some Blender knowledge, we could expose simpler and common animation options as described in later sections.

Here is a video of the trajectory alignment example from the user guide showing static and dynamic universes on the same timeline:



Direct video link:
https://drive.google.com/file/d/1h1TQt-MSj0TCvqi8H-nQ9Q66zhbKU3Yv/view

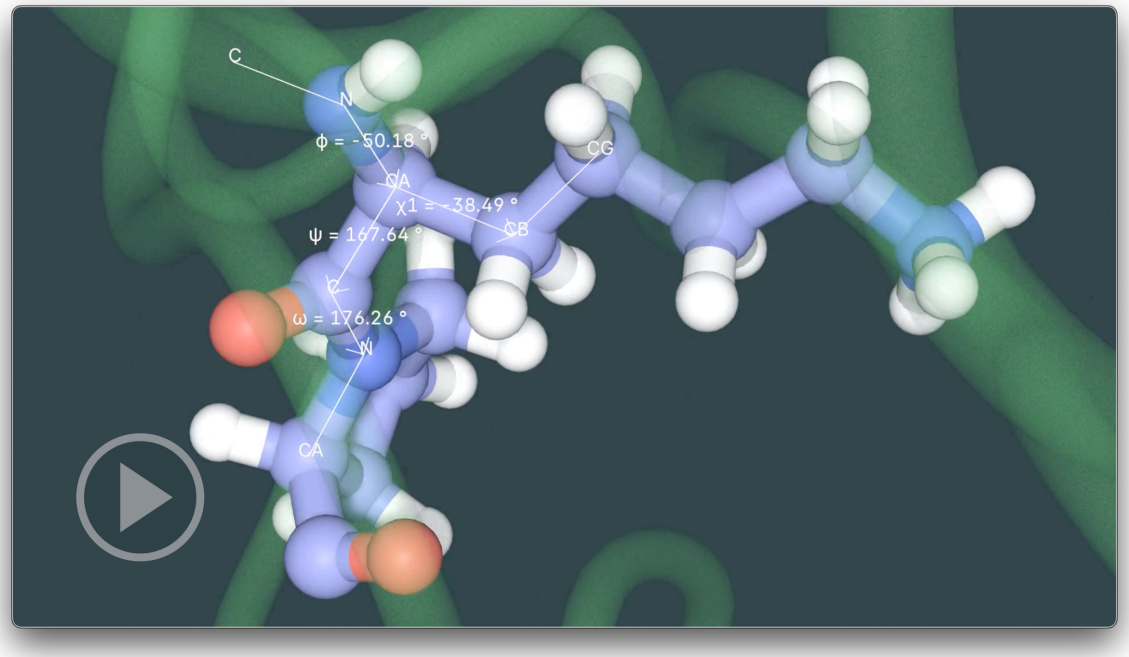
Camera, Lighting and Background

This document proposes to keep a few things constant to achieve the consistency goal. We should not need to change the camera focal length for most of our use cases. Having a fixed value with the appropriate clipping planes will suffice. Zooming in would be equivalent to positioning the camera appropriately and not having to change the focal length. The reason for this requirement is that Blender uses a virtual camera for the 3D viewport (with its own focal length and clipping planes) and to get what the viewport shows to be consistent with the render outputs these have to be synchronized with the corresponding actual camera values and the camera sensor size. We lose no functionality by keeping this constant.

For consistent lighting, a Sun (with a low power of 3-4W - configurable) that is positioned at the base of the camera pointed straight ahead and parented to the camera achieves the best results. Because the light is a Sun, it is uniform and doesn't vary with the distance to the object and we get consistent results by pointing the camera anywhere in the 3D space around our universes.

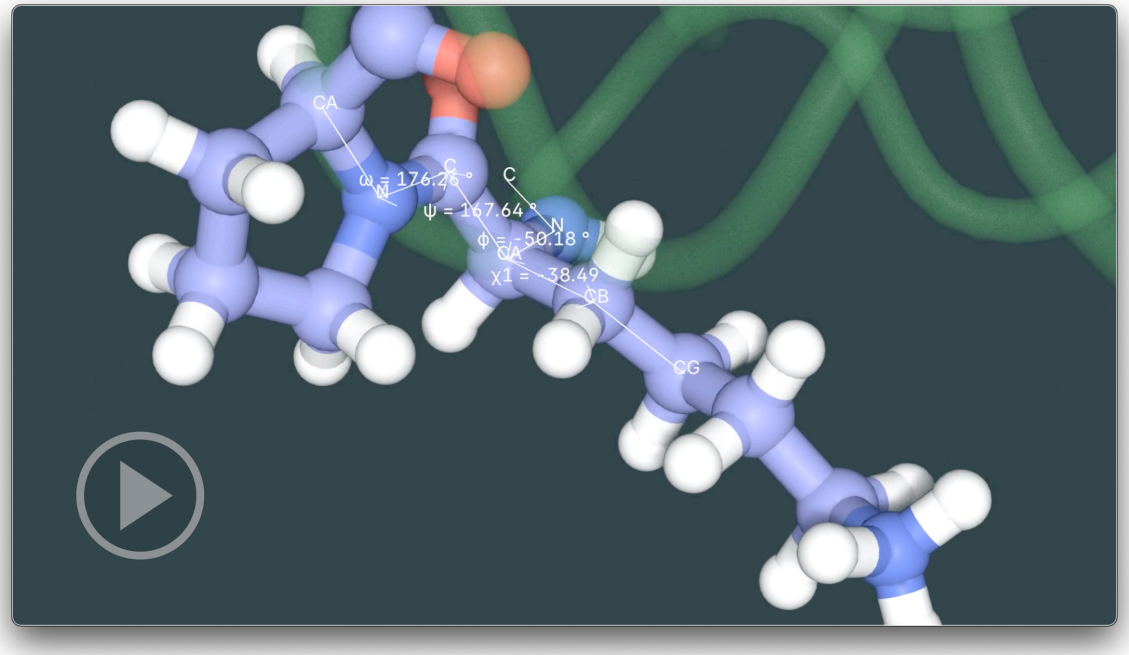
Here are a couple of example videos that show a yaw and pitch around a residue with some annotations that show the canonical dihedral angles (EEVEE renders):

Yaw video:



Direct video link:
https://drive.google.com/file/d/14ND4-LDEx0D2M3KjPY777XaCG2jlhqfc/view

Pitch video:



Direct video link:
https://drive.google.com/file/d/1ORCvxUnyBn4aBBNSg3ir7Ths23gn7Iv9/view

As we can see from the above examples, it is the camera that moves around the region of interest and the lighting remains consistent as it is parented to the camera. These also show a couple of simple animations around regions without necessarily requiring trajectory frame changes.

To achieve consistent and simple background colors that match viewport renders (the really fast renders) and regular renders (EEVEE, Cycles, etc) we could use a solid background color for the world settings. Though the 3D viewport background and the world solid background colors are two different settings, it is easy to keep them in sync. This keeps the fast viewport renders in sync with the high quality EEVEE or Cycles renders.

MDAnalysis selection phrases and AtomGroups

MDAnalysis has a rich Atom selection language based on selection phrases which are strings. A lot of MDAnalysis use cases center around AtomGroup where a user-provided selection phrase might not be available (except for an ‘UpdatingAtomGroup’). Using selection phrases for the GUI is very natural. For the API we would need to support both. Given that any AtomGroup could be represented using the ‘index’ and ‘bynum’ keyworks of the Atom selection language, we could bridge this gap between the API and the GUI and make this seamless. We will be able to handle updating atom groups and periodic boundary conditions as well. MN today uses selection phrases for selection attributes in its GUI.

Named entities

Almost everything in Blender has a name. Every Blender property too has an implicit name attribute. This makes looking up entities from the underlying Blender data structures very accessible and opens up a lot of use cases. This is something we could carry into our API design as well. Every entity (style, annotation, component etc) will have a name (implicit, but can be explicitly set / modified). This will allow them to be easily looked up and referenced. For example: In a simple use case like - turn off visibility of style1 in universe u1 after 3 seconds, this becomes straightforward in both our internal API specifications and in a higher level specification like a Molywood script.

APIs

With the design considerations taken into account and the relevant design choices outlined above, we can specify the requirements for the overall APIs. As designing the actual API is part of the project (along with prototyping and implementation), this section outlines all the key requirements that will be considered. These APIs can be broken down into four main categories:

APIs for styling representations

These APIs are responsible for styling different representations of a universe. They will use Molecular Nodes internally and support all the style types that are currently possible through the MN Geometry Nodes interface. These APIs will support the following features:

- Add a style for a specific selections of the universe
- Undo the most recently added style (pop)
- List all the styles added to a universe (showing index, entity name and style details)
- Get a style object based on index or entity name
- Remove a style entity based on index or entity name
- Show / Hide a style entity based on index or entity name
    - This will internally unmute / mute the “named attribute” selection node in the corresponding node frame
- Configuration of all style related properties using a single dictionary (for exporting a single complete style API call from the GUI and during initial setup) as well as each individual style specific property (allowing auto-complete in programming environments) using a returned style object
    - This includes nested style configuration elements like material - material name and all corresponding material properties (Principled BSDF inputs)
- Uniform colors and grouping of colors via a color theme to override defaults
- Dynamic change of the style type of a style entity (eg: spheres to cartoon)
- Dynamic change of the selection phrase of a style entity
- Focus (viewport and camera) on a particular style entity

    - This will internally use the bounding box of the selection returned by the corresponding selection phrase for the style - see the focus APIs in the rendering section below
- Renaming of the style entity name
- Clear all styles entities added to a universe
- All the style properties will be keyframable (for the animation system) both from the animation APIs and the GUI
- All these APIs will be in sync with the GUI and point to the same data

APIs for annotations

These APIs are responsible for adding and managing annotations to a universe. Annotations can be in the form of text, drawings (lines, arrows, circles, arcs, etc) and primitive Blender shapes (spheres, cylinders, curves, text objects, etc). These APIs will internally use Blender's "blf" and "gpu" modules to draw on the 3D viewport (for non Blender shapes) and PIL to draw them onto an image that will be composited for final renders (both images and video). These APIs will support the following features:

- Common annotations like atom info (along with residue and segment details), bond angles, center of mass, center of geometry, distance between entities, canonical and regular dihedral angles, universe info (like topology, trajectory and frame details)
- Arbitrary text and drawings anywhere in the 3D space with respect to the universe geometry (both direct and computed coordinates)
    - They will be automatically projected onto the 2D space depending on the viewport / camera position and always face the user and retain their specified size at all times
    - Some of these APIs will require a normal vector to determine the 3D plane for the drawings
- Arbitrary text and drawings in the 2D space with respect to the viewport and camera views. These are with respect to the final 2D renders and hence retain their position independent of any camera position or movements
- Configuration of font, font size, font color, text alignment, text rotations and text positioning offsets (along the final 2D +X, -X, +Y, -Y)
- Configuration of line color, line width, arrow sizes / location, pointer lengths for all line based drawings (lines, circles, arcs, etc)
- Add a built-in or custom annotation to a universe
- Undo the most recently added annotation (pop)
- List all the annotations added to a universe (showing index, entity name and annotation details)
- Get a annotation object based on index or entity name
- Remove a annotation entity based on index or entity name
- Show / Hide an annotation entity based on index or entity name
- Show / Hide all annotations of a universe
- Dynamic change of the selection phrases used in an annotation entity
- Focus (viewport and camera) on a particular annotation entity
    - This will internally use the bounding box of the selection returned by the corresponding selection phrases for the annotation

- Renaming of the annotation entity name
- Clear all annotations entities added to a universe
- All the annotation properties will be keyframable (for the animation system) both from the animation APIs and the GUI
- All these APIs will be in sync with the GUI and point to the same data

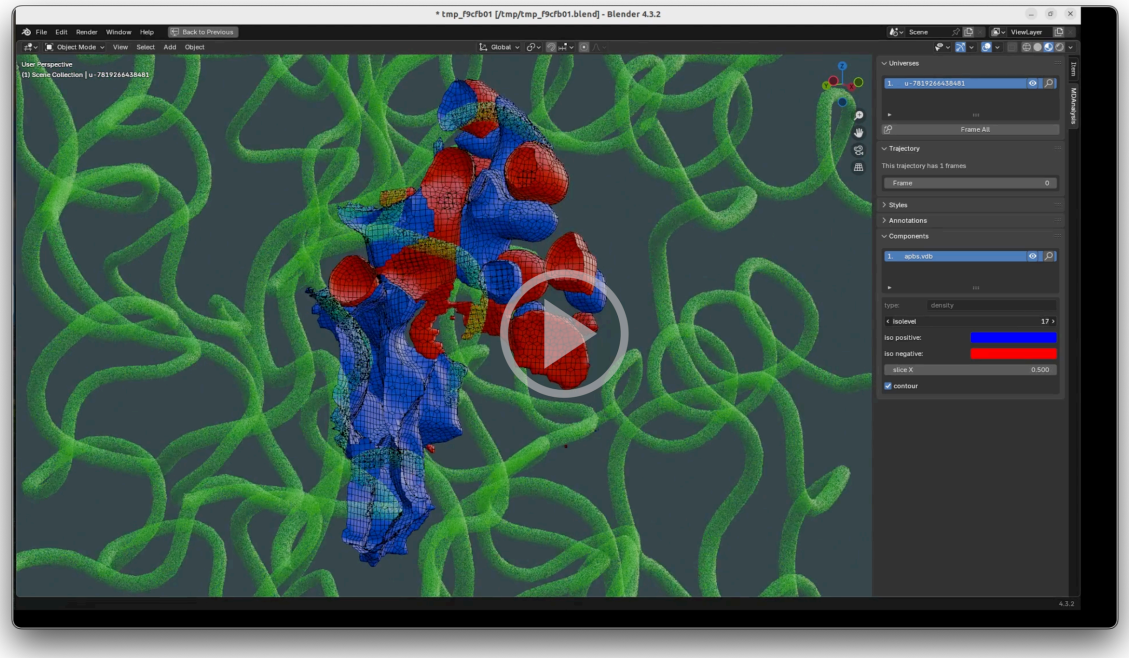
APIs for components

These APIs are responsible for adding and managing different components to a universe. A built-in density analysis component will be provided by default. A framework to manage the lifecycle of these components will be provided through these API. This will allow other advanced analysis components added by MDAnalysis or other users to be plugged in and managed the same way. These APIs will support the following features:

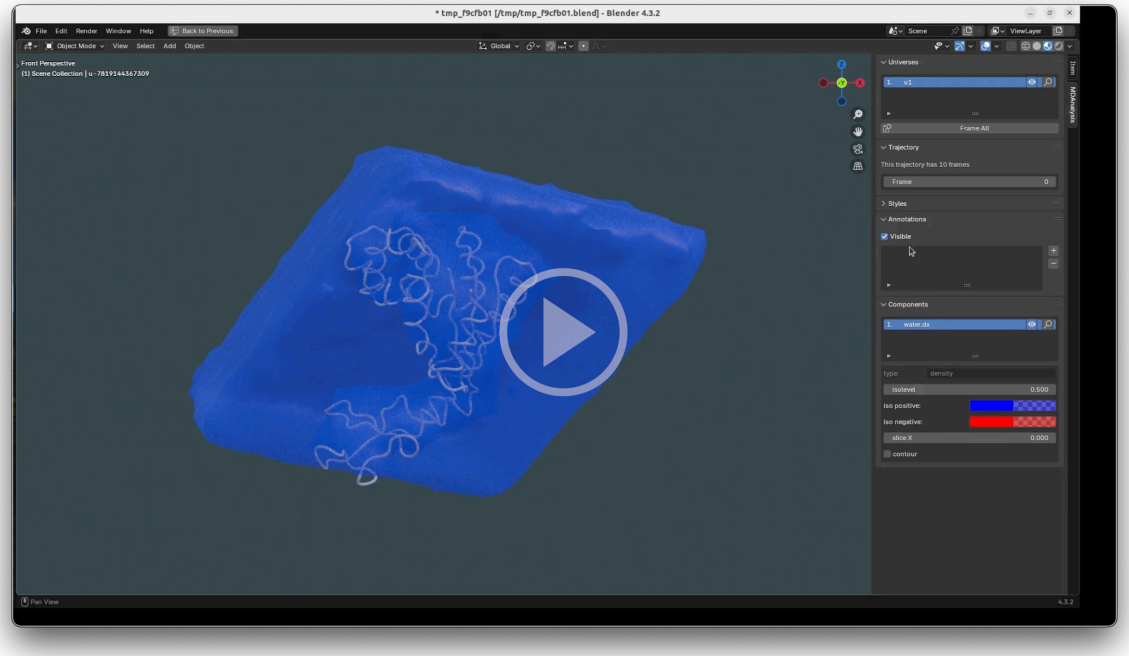
- A built-in density component that supports both ".dx" and ".vdb" formats
    - Configurable options will include iso value, colors for positive and negative iso values, option to show contours of the volume, slicing along +X, -X, +Y, -Y, +Z and -Z axes
- Add a component to a universe
- List all the components added to a universe (showing index, entity name and component details)
- Get a component object based on index or entity name
- Remove a component entity based on index or entity name
- Show / Hide a component entity based on index or entity name
- Focus (viewport and camera) on a particular component
- Renaming of the component entity name
- Clear all component entities added to a universe
- Customizable defaults and ranges for component property parameters

Here is a couple of examples that show two different use cases that use the same density component, but are set up with different defaults and ranges for the configurable values:

In the first example, the "isovalue" is an integer with a default value of 1 and a range from 1 to 100. The volume in this case is also from a ".vdb" file. The second example is from Calculating the solvent density around a protein in the MDAnalysis user guide. In this case, the "isovalue" is a float with a default value of 0.5 and a range from -1.5 to 1.5 and the same "water.dx" output file generated by MDAnalysis is used around the transformed universe.



Direct video link:
https://drive.google.com/file/d/1Tqm1iqHLMtBpw6Syszc5mN9KQLa9sTLM/view



Direct video link:
https://drive.google.com/file/d/1eDZ9nPdmpbgxb3_U4MssMrujqseccjTK/view

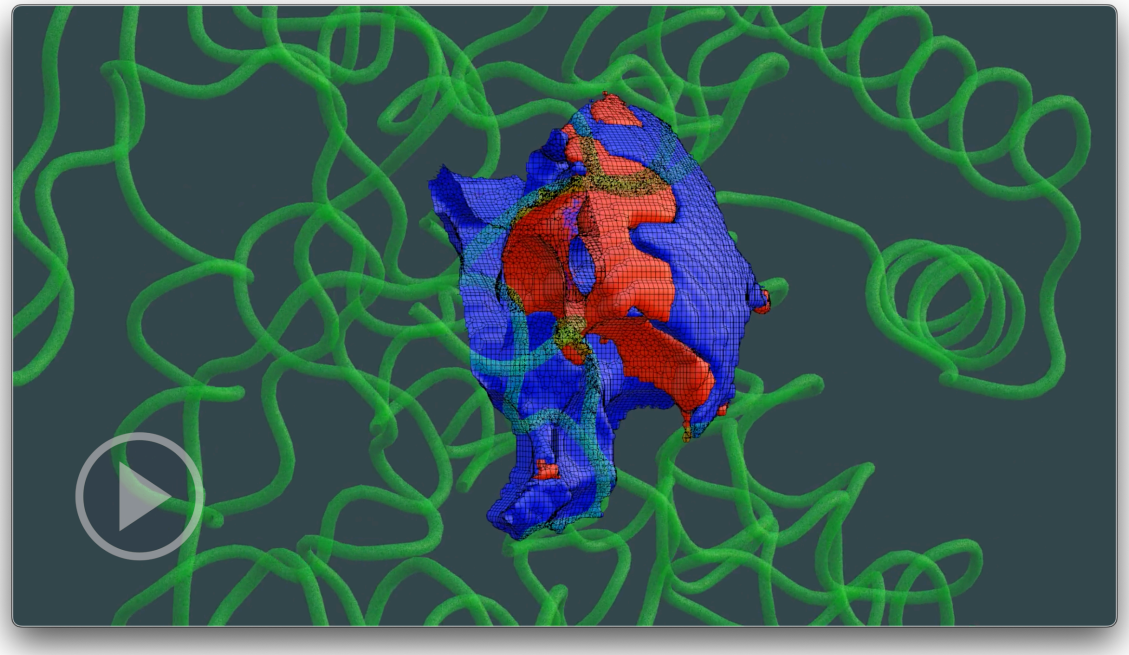
- All the component specific properties will be keyframable (for the animation system) both from the animation APIs and the GUI
- All these APIs will be in sync with the GUI and point to the same data

APIs for rendering

These APIs are responsible for configuring the camera, lighting, background, animation and various render related settings that will eventually generate images and movies as outputs from Blender. All these APIs will work in exactly the same way in both the headless mode and GUI modes. These API will support the following features:

- Ability to specify and focus on any region of interest across the 3D space (this updates the 3D viewport and the camera to match the viewport)
    - Focus from different standard viewpoints (front, back, top, bottom, left, right)
    - Focus on individual universes, selections within universes, selections across different universes - all based on MDAnalysis selection phrases
    - Focus on individual styles, annotations, components (of specific universes)
    - Dynamic focus based on updates owing to trajectory frame changes
    - A generic “lookAt” function that allows a standard way to specify arbitrary positioning of the camera
- Simple camera movements around focus areas - like yaw, pitch with customizable angles and durations when plugging into the animation system

The videos in the Camera, Lighting and Background section already show examples of how this will look like. Here is another example of a simple camera yaw around a density component:

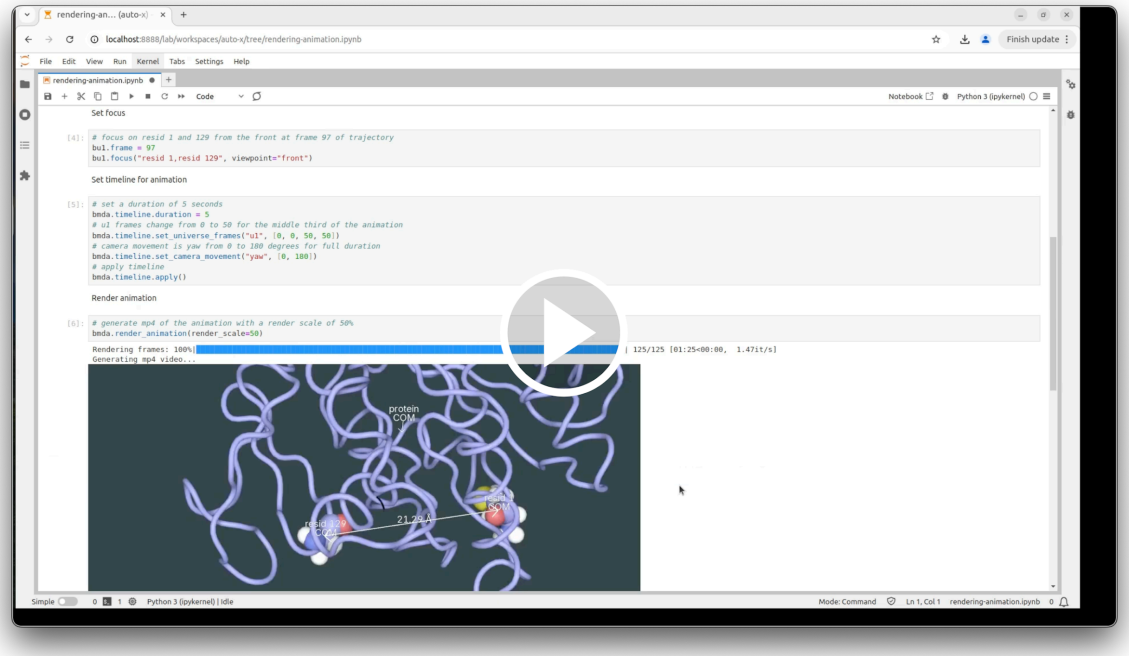


Direct video link:
https://drive.google.com/file/d/1wiCqK89RkJRuRV74Yn0z1WXcJqMIsGIA/view

- Configuration of the lighting intensity (a simple Sun with low power as outlined in the Camera, Lighting and Background section)

- Configuration of a solid background color that is consistent across 3D viewport and viewport renders and regular renders
- A simple animation framework that allows keyframing all our properties on a customizable timeline using entity names and property attributes, while being extensible for complex use cases in the future

Here is an example of a customizable timeline and how rendering animations could look like:



Direct video link:
https://drive.google.com/file/d/1uWCPM4TauPMD6M5aTnyWTQ9-ZGJK0kCE/view

- Configuration of several render related settings
  - Rendering engine, number of samples
  - Output image / video resolution and resolution scale
  - Common output image / video settings and filenames
- Viewport renders and full renders (including scaled renders using the resolution scale for faster evaluation) - both images and video

GUI

All the GUI for MDAnalysis will be in one single place - as a tab named “MDAnalysis” in the sidebar (n-panel), which is directly accessible from the 3D viewport. All the various sections like universes, trajectory details, styles, annotations, components, etc will be separate panels. The GUI will be built to be in sync with the API as described and illustrated in the Blender properties for data section before. The GUI will support the following features, all of which will be possible through the API as well:

- Support for multiple universes and selection / focusing on individual universes
- Ability to frame (focus) all universes
- Trajectory details and dynamic trajectory frame changes of a universe
- Addition / deletion of styles along with complete customization of styles (including nested configurations like material properties) in real time
- Dynamic change of style types and selection phrases for different representations
- Control the visibility and focus of individual style entities
- Addition / deletion of annotations - both built-in and custom annotations along with complete customization of each annotation
- Control the visibility and focus of individual annotation entities
- Control the visibility and focus of individual component entities
- Customization of component specific properties
- Ability to generate the Python API code based on the current set of customizations for each of the different entities (styles, annotations, components, etc)
- Error handling that shows details and resolution options
- Keyframability of all the properties for use in the animation system

Measurable outcomes

This section outlines some non-trivial outcomes that have to be achieved to reach the overall goal of this project.

1. Help design, prototype and implement the style APIs
2. Help design, prototype and implement the annotation APIs
3. Help design, prototype and implement the component APIs
4. Help design, prototype and implement the rendering APIs
5. Build a GUI within Blender that is in sync with all the APIs above

All the implementations above will include a clear documentation of the APIs, tests and example notebooks that illustrate all available features.

Timeline

This is a project that can be categorized as “Large” as per the GSoC requirements. I intend to complete this work as part of the standard 12 week coding period (June 2 - September 1). Most of this period (2 months of it) coincides with the summer break in my University and I will be able to commit full time on this project. My 3rd semester starts at the beginning of August, but given that it is early into the semester, I will still be able to commit the majority of my time to complete this project. I will also have some time after the end of my current semester before the official coding period starts to get an early start and ramp up.

Here is a detailed breakup of this timeline:

Before May 8: (Before contributor announcement)

- Continue exploration and prototyping to eliminate any unknowns
- Continue to take part in discussions

May 8 - June 1: (Before the official coding time)

- Discuss and finalize the design choices and API specifications with the mentors
- Document the above
- Setup the environment and overall project structure

June 2 - June 15: (Official coding period starts)

- Build the universe level abstraction and common utilities across all APIs and GUI
- Build the common data framework between APIs and GUI
- Have a working framework on which the rest of the APIs and GUI can be built

June 16 - June 29:

- Build the style APIs and corresponding GUI - code, document, test
- Start building the basic rendering APIs to see some early outputs

June 30 - July 13:

- Build the annotation APIs and corresponding GUI - code, document, test
- Continue work on the rendering APIs

JULY 18th: MID TERM EVALUATION

July 14 - July 27:

- Build the component APIs and corresponding GUI - code, document, test
- Continue work on the rendering APIs

July 28 - August 10:

- Complete the rendering APIs - code, document, test

August 11 - August 24:

- Complete coding any remaining pieces
- Complete any missing documentation and tests
- Create end to end example notebooks that illustrate all the various features

August 25 - September 1: (Final week)

- Tie up any loose ends
- Prepare for the final submission

SEPTEMBER 1st: SUBMIT FINAL WORK

Conclusion

This document proposed details about how "Project 4: Better interfacing of Blender and MDAnalysis" can be implemented as part of GSoC 2025. A detailed description of the problem, the approach to solve it based on early prototyping and explorations along with measurable outcomes as part of the GSoC 2025 timeline are provided.

References

1. [MDAnalysis GSoC 2025 Project Idea] Project 4: Better interfacing of Blender and MDAnalysis
2. [MDAnalysis Discussion] Visualization of MDAnalysis results in Blender
3. Learnings from explorations and prototyping
4. [ggmolvis] How we handle rendering
5. [ggmolvis] Selections / Representations of the same data
6. [ggmolvis] A discussion on on overall structure
7. [ggmolvis] Design choices for Text Annotations
8. MolecularNodes involvement
9. ggmolvis involvement