Athens Journal of Technology and Engineering - Volume 11, Issue 4,
December 2024 – Pages 251-266

Composite Command Pattern for Managing Spherical Geometry Constructions

By Hans Dulimarta[*] & William Dickinson[±]

Spherical Easel (https://easelgeo.app) is a free web application for researching, teaching, and learning spherical geometry written in Typescript. The authors have created an open-source code base in VueJS v3 that combines several design patterns, global data structures, and cloud-based storage for storing and retrieving spherical constructions. Spherical constructions are made up of interdependent geometric objects (like lines and circles). Interdependencies means that while being moved, updates to a single object must propagate to other dependent objects. The application stores the dependency structure using a directed acyclic graph to allow these updates to propagate correctly. In this paper, we describe how Spherical Easel employs the Command and Composite design patterns (fully integrated with various VueJS supporting libraries) to maintain a history of construction edits, allowing users to undo and redo edits, and to store a final construction script in a cloud database. To guarantee correct rendering of the spherical objects when loading a construction, the stored script also preserves the structure of the directed acyclic graph. The construction script parser built is also designed to take advantage of the (Composite) Command design pattern. Using this innovative design, adding new object types does not require a major redesign of the script parser.

Keywords: *Spherical geometry, software design patterns, web application*

Introduction

Spherical geometry is the earth's geometry, surrounds us all, and is obviously important for many large-scale applications. For example, spherical lines, like the equator and longitudes (but not latitudes), are the paths that orbiting satellites' earthly shadows follow. Despite its obvious applied nature, spherical geometry is rarely taught at the secondary level in the United States. This contrasts with Euclidean geometry, which is always taught at the secondary level. Changing this is one of the goals of our spherical geometry modeling application; one ultimate ideal use of Spherical Easel would be to have students at all levels (secondary and beyond) and in all countries be exposed to spherical geometry using this app whenever they are taught Euclidean geometry.

To effectively learn about, teach, and conduct research in spherical geometry, students, teachers, and researchers must be able to easily interact with and explore it. While Euclidean geometry is easily modeled on paper or a white board, spherical geometry is more challenging to physically model. Tennis balls are a cheap but ineffective option and Lénárt Spheres (large clear plastic spheres one can draw on)

[*]Professor, School of Computing, Grand Valley State University, USA.
[±]Professor, Department of Mathematics, Grand Valley State University, USA.

https://doi.org/10.30958/ajte.11-4-1 doi=10.30958/ajte.11-4-1

are expensive. This leaves dynamic digital tools for such explorations. While there are multiple digital tools for exploring Euclidean geometry including GeoGebra and Desmos and a few tools for exploring hyperbolic geometry like NonEuclid, *Spherical Easel is the only dedicated free tool for dynamically modeling spherical (or double elliptic) geometry that is currently available.*

Design Patterns

Software developers who employ the object-oriented paradigm in their development are very likely to have read the *de facto* standard design pattern book by Gamma et al. (1995). The book includes a catalogue of 23 commonly used design patterns in object-oriented programming. Since then, many other books/resources have been published, and the catalog items have been greatly expanded. For example, some of them are specific to a particular language (Smalltalk (Back 1997), Java Design Patterns), others are targeted for specific programming modes (concurrent programming (Schmidt et al. 2000), reactive programming (Oliveira Marum et al. 2024)), and others are specific to an application domain (game programming (Nystrom 2014), block chain smart contracts (Górski 2024)).

The focus of this paper is an extensive exposition of how the Command and Composite design patterns are incorporated into Spherical Easel. In Gamma et al. (1995) the intent of each design pattern is given as:

> **Command**: *encapsulate a request as an object*, thereby letting you parameterize clients with different requests, queue, or log requests, and *support undoable operations*.
> **Composite:** compose objects into tree structures to represent part-whole hierarchy. Composite lets *clients treat individual objects and compositions of objects uniformly*.

The *italics* in each quote indicate the part of each pattern that Spherical Easel uses extensively and will be the focus of this paper.

Spherical Easel

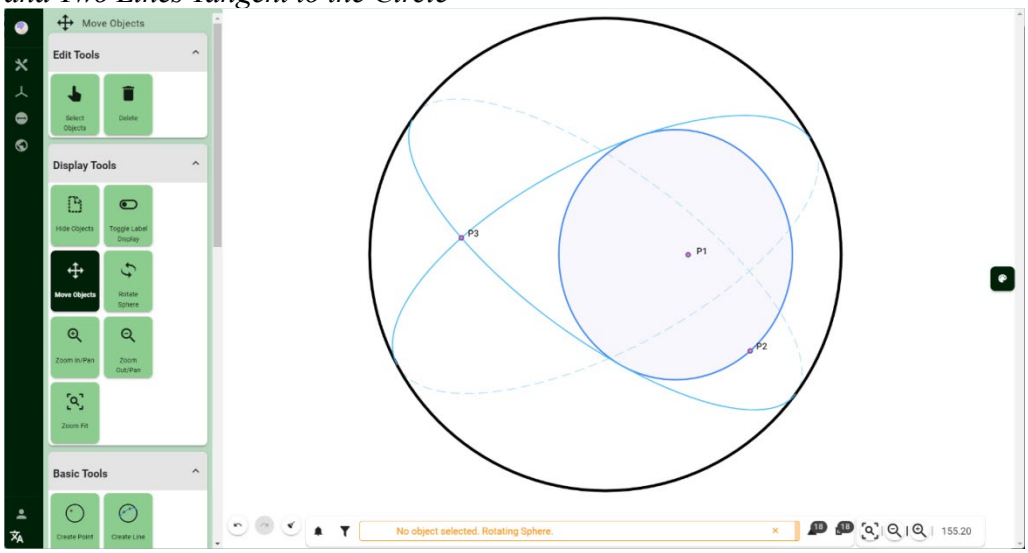
Spherical Easel is a web-based application offering an abundant suite of tools which enable users to create constructions of spherical objects including points, lines, segments, circles[1], ellipses and parametric curves. Intersection points between these objects are automatically created and dynamically updated as a user moves construction elements. Users can also transform the geometric objects using spherical rotations, translations, reflections, and inversions. In addition, there are tools for constructing tangents, antipodal points, perpendiculars, bisections, and polar lines among many others.

[1]Spherical circles are the intersection of a sphere with a plane. When that plane contains the center of the sphere, a spherical line or great circle or geodesic is formed, like the equator and longitudes on the Earth. Spherical lines are locally paths of shortest distance. When the plane does not contain the center of the sphere, a small circle is formed, like latitudes on the Earth.

As of summer 2024, Spherical Easel provides 32 tools, grouped under the following eight categories:

1. **Display Tools**: Show/Hide Labels, Show/Hide Objects, Zoom in/out/fit, Rotate Sphere, and Move Objects
2. **Basic Tools**: Create Lines, Line Segments, Points, and Circles
3. **Construction Tools**: Polar, Antipodal, Midpoint, Angle Bisectors, Tangent Lines, Intersections, Perpendicular Lines, and Points on Objects
4. **Measurement Tools**: Coordinates, Angle, Triangle Area, and Polygon Area
5. **Measured Objects**: Measured Circle
6. **Advanced Tools**: Three Point Circle, N-sect Segments, and N-sect Angle
7. **Conic Tool**: Create Ellipse
8. **Transformation Tools**: Reflection, Point Reflection, Rotation, Inversion, Translation, and Apply Transformation

Figure 1. *The Main Page of Spherical Easel at https://easelgeo.app Showing a Circle and Two Lines Tangent to the Circle*



In addition, it provides a calculation facility for computing mathematical functions of measured objects (like area, length, distance, or angles) and a limited functionality for plotting and interacting with simple spherical parametric curves. To the best of our knowledge, Spherical Easel is the first web-based application for **spherical geometry computations** that offers such extensive tools. We plan to implement a suite of about 55 tools; The goal is to implement the analog of any tool found in the digital Euclidean applications previously mentioned or any idea that is mentioned in the historical texts (such as Todhunter (1859)) used during the late 19th century heyday of spherical geometry, like radical axis and centers of similitude.

Among the main problems we attempt to solve in managing spherical constructions are:

- Enabling the user to redo and undo (graphical) edits on the canvas. These user actions must be synchronized with updating the directed acyclic graph (DAG).
- Generating a script that reflects the sequence of object constructions performed by the user on the graphical canvas in a format which can be saved on persistent storage.
- Reloading and interpreting construction scripts from persistent storage and rebuilding the DAG.

Application Architecture and Design Patterns

A more elaborate description of the application architecture and various design patterns used in our application can be found in Dulimarta and Dickinson (2023). In this paper, we focus on the two design patterns which are heavily used in managing saved spherical constructions: the Command and the Composite design patterns.

Directed Acyclic Graph

Spherical objects may be constructed with one or more dependencies. For instance:

- A circle created from a center point and a point on its perimeter is dependent on the two points.
- Each intersection point between two circles depends on the two circles.
- A line might depend on the intersection points of two circles.

Our application stores the dependencies using an important data structure: a directed acyclic graph (DAG) of spherical objects. In the DAG, all the dependencies are stored as parent/child relationships. In Figure 2, the relationships are recorded in two separate arrays: `_parents and _kids`.

Figure 2. *SENodule Abstract Class*

```
abstract class SENodule {
  protected _parents: SENodule[] =
[]
  protected _kids: SENodule[] = []
  protected _outOfDate = false

  public abstract update();

  public canUpdateNow(): boolean {
    return !_parents.some(item =>
        item.isOutOfDate()
    )
  }

  public updateKids() {
    const updateNow: number[] = []
    _kids.forEach((item,index) =>
      if (item.canUpdateNow())
        updateNow.push(index))
    updateNow.forEach((kid:number)
=> {
      this._kids[kid].update()
    })
  }
```

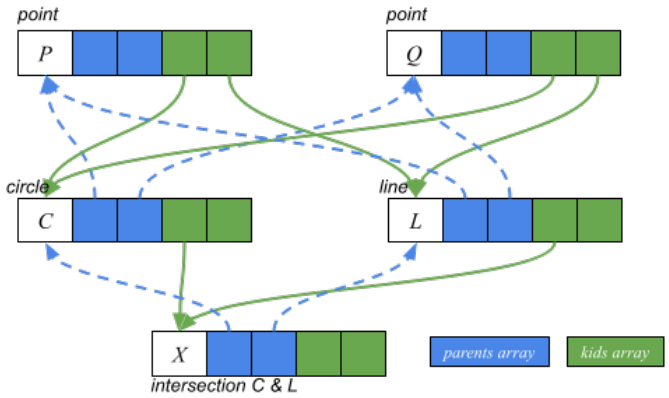
```
public addKid(n: SENodule) {
    this._kids.push(n)
  }


  public addParent(n:SENodule) {
    this._parents.push(n)
  }

  public
registerChild(n:SENodule) {
    this.addKid(n);
    n.addParent(this);
  }
}
```

For instance, when creating a circle C between with a center point *P* and point on the perimeter *Q*, the circle C becomes the child of both *P* and *Q*, and *P* (and *Q*) becomes the parent of C. Moving one of the points (*P* or *Q*) to a new position will update the details of circle C. If later the user creates a line *L* through *P* and *Q*, and its intersection *X* with the circle *C*, moving either *P* or *Q* to a new position will also update the position of the intersection point. A snapshot of the DAG representing this construction is shown in Figure 3.

Figure 3. *Directed Acyclic Graph*



Independent objects (such as *P* and *Q* at the top of the diagram) become root nodes who have no parental dependencies on other objects. These top-level objects can be directly manipulated by users. In a typical construction, we will find a forest

of DAGs each associated with a distinct root node or nodes. In general, any updates to a single object which is also a parent of other objects will propagate to the children, grandchildren, grand-grandchildren, and so on. Before updating a particular object in the DAG, we must check to make sure that all its parents have been updated.

The internal structure of the DAG is implied in the design of the parent abstract class `SENodule` inherited by all the other geometric object classes. The methods `addKid`, `addParent`, and `registerChild` allow the creation of the DAG and before updating a particular object in the DAG, the variable `_outOfDate` is checked and updated with the method `canUpdateNow.` For example, in Figure 3, updates to *X* can be performed only after *L* and *C* are updated. The `updateKids` method allows changes to propagate to the rest of the DAG.

The method that performs the **actual update** is abstract because sometimes the information needed to update an object is not contained in the parent objects. For example, when a line segment connects a pair of antipodal points (like a longitude connecting the north and south poles) the line segment is not uniquely determined by the endpoints, and the update of that line segment depends on a normal vector which is independent of the parent antipodal points. Also, to properly undo a user's action certain pieces of information (like a line segment's normal vector) must be stored and passed along during update propagation.

Handling user Interactions

Users create spherical objects by first selecting one of the tools pictured in the lefthand side of Figure 1. When a selected tool becomes active, Spherical Easel switches to a specific mode associated with that tool. Operating under this mode, user actions to create spherical objects are mapped as a specific Command object or a group of Command objects.

The Command Pattern

Each user action of editing the graphical canvas (creating a new object or modifying/removing an existing object) is recorded as a Command object. When a new object is created, we know that all the objects it depends on and their dependencies must have been already created. We take advantage of this obvious fact by recording the sequence of user actions in a list of commands. Those objects created expand the DAG with new nodes and dependencies. Using the example of creating a circle C with center point *P* and perimeter point *Q* translates to the following three simple commands:

- Create a new point *P* (with no dependencies).
- Create a new point *Q* (with no dependencies).
- Create a new circle C with *P* and *Q* as its parent.

In the command history the above three commands are recorded in the order shown, and the third command unifies the two independent DAGs (each consisting

of a single point) into one DAG. Our implementation of the Command pattern revolves around the following abstract class shown in Figure 4.

Figure 4. *Executable Command*

```
static CMD_HISTORY: Command[] = [];

abstract class Command {
    abstract restoreState(): void;
    abstract saveState(): void;
    abstract do(): void;

    execute(): void {
        CMD_HISTORY.push(this);
        this.saveState();
        this.do();
    }
}
```

Using the Command pattern makes it easy to support undoing (and redoing) of user actions. The `saveState()` call facilitates storing any necessary information that may be needed to restore the data structure when this particular command is being undone. By pushing every executed command to a stack (CMD_HISTORY), undoing user actions can be implemented as a static method that pops the most recent command and invoking its `restoreState()` function as shown in Figure 5.

Figure 5. *Command Undo*

```
abstract class Command {
    static undo(): void {
        if (CMD_HISTORY.length === 0)
            return;
        const lastAction: Command | undefined = CMD_HISTORY.pop();
        if (lastAction)
            lastAction.restoreState();
```

Practically, the `restoreState()` function performs the opposite of the `do()` function and also *in reverse order*. For instance, if the user wants to add a circle C (with label L_c) that depends on existing points P (center) and Q (on the perimeter), in the implementation of `AddCircleCommand`, the sequence of operation of its `do()` and `restoreState()` are shown in Table 1.

Table 1. *Sequence of Operation in AddSegmentCommand*

`do()`	`restoreState()`
1. Add *C* as a child of *P* *2.* Add *C* as a child of *Q* *3.* Add label L_c as a child of *C* 4. Add circle *C* to the list of circle objects 5. Add label L_c to the list of label objects	1. Remove label L_c from the list of label objects 2. Remove circle *C* from the list of circle objects *3.* Remove L_c from the children of *C* 4. Remove *S* from the children of *Q* 5. Remove *S* from the children of *P*

It is important to note that the user action of removing an existing object from the graphical canvas is also recorded as a command object. As a result, the graphical canvas may show fewer objects, but the command history will continue to grow longer.

Composite Pattern

For convenience, our graphical editor allows the user to create objects in several ways. For instance, to create a circle *C* as shown in the previous example, the user can perform one of the following actions with using the Point Tool and Circle Tool or both just the Circle Tool:

- Initiate three separate actions: With the Point Tool, first create the first point *P*, then create the second point *Q.* Then with the Circle Tool create a circle with center *P* and containing *Q* by clicking on *P* and dragging to release on *Q* or
- Using only the Circle Tool, in a single action drag and release the mouse from the location of the points (without creating them first).

Although the final visual outcome for both methods is the same, to remove the circle and two points from the canvas the first method requires three calls to `undo()`, while the second method requires only one. To support these features correctly, we apply the Composite design pattern shown in Figure 6.

Figure 6. *Command Group*

```
class CommandGroup extends Command {
    public subCommands: Command[] = [];
     addCommand(c: Command): Command {
        this.subCommands.push(c);
        return this;
    }
    restoreState(): void {
        // Restore state should be done in REVERSE order
        for (let kIdx = subCommands.length - 1; kIdx >= 0; kIdx--) {
            this.subCommands[kIdx].restoreState();
        }
    }
    saveState(): void {
        this.subCommands.forEach(x => { x.saveState() });
    }
    do(): void {
        this.subCommands.forEach(x => { x.do() });
    }
}
```

Using the above definition of `CommandGroup`, creating a circle from two points can be implemented using code snippet as shown in Figure 7.

Figure 7. *Creating a Circle from Two Points*

```
const circleGroup = new CommandGroup();
const createCenter = new AddPointCommand(/*args*/);
const createPeripheral = new AddPointCommand(/*args*/)
const createCircle = new AddCircleCommand(/*args*/);
circleGroup.addCommand(createCenter);
circleGroup.addCommand(createPeripheral);
circleGroup.addCommand(createCircle);
circleGroup.execute();
```

Cloud Database

To provide persistent storage, authenticated users of Spherical Easel are given access to a cloud database for storing their geometric constructions. Specifically, Spherical Easel keeps the user data on the Google Firebase Firestore which stores two different kinds of user data:

- Most user data (user profile, constructions details) which can be encoded as "text format" is stored on Firebase Firestore.
- Binary data related to the constructions (such as construction preview images, compressed textual data) are stored in Firebase Storage.

Incorporating the Firebase Cloud data store into Spherical Easel allows its users to easily share their work with other users. These features, for instance, allow a teacher to prepare several geometric constructions that can be easily shared and built upon by their students or a researcher can easily share and build on a spherical construction.

Generating Geometric Construction Scripts

```
abstract class Command {
    abstract toOpcode(): null | string | Array<string>

    static dumpOpcode(): string {
       const out = CMD_HISTORY.map(c => c.toOpcode())
           .filter(z => z !== null)
       return JSON.stringify(out)
    }
}
```

The Command history data structure, along with the Command Pattern used in Spherical Easel, enables the app to save geometric constructions in the cloud database as a textual script in JSON (JavaScript Object Notation). Every `Command` subclass implements a method to dump its execution context.

Invoking `dumpOpcode()` will iterate over the current sequence of command objects and each command will produce its own JSON representation of its executable context via its implementation of the `toOpcode()` method. For flexibility, this method has multiple return types:

- `null`: when the command does not actually create/delete spherical objects, but instead it mainly modifies the visual property of spherical objects on the canvas. For instance, moving the position of a label, changing the visibility of spherical objects, changing the display style of labels fall under this category. Commands under this category do not have to be saved to the cloud database, hence they are filtered out from being converted to the final JSON string.
- A single string: when the command executes as a standalone, and not part of a group.
- An array of strings: when several related commands must execute as a group.

Since the implementation of `opCode()` is specific to each command, it is declared as abstract. It is up to the individual command, to generate its own execution context. For instance, the `AddCircleCommand` includes the following details in its generated execution context:

- The circle identifier
- The center point identifier and an identifier for the point on the perimeter of the circle
- Boolean flags (exists, showing)
- Foreground & background rendering styles
- Label details: position vector, flags (exists, showing), style and label object identifier

A partial output of `AddCircleCommand`'s opcode method is shown as a string of key-value pairs delimited by '&' below:

```
AddCircle&objectName=C8&objectExists=true&objectShowing=true&
        circleCenterPointName=P2&circlePointOnCircleName=P8
```

The `opcode()` method must also be implemented by the `CommandGroup` class, so all the individual commands in a group produce their execution context. For instance, the opcode output a group of commands that create a circle with a center is an array of strings such as shown below:

```
    ["AddPoint&objectName=P2&pointVector=(0.12333,0.82231,0.25833)",
     "AddPoint&objectName=P3&pointVector=(0.87225,0.01672,0.62244)",
"AddCircle&objectName=C2&circleCenterPointName=P2&circlePointOnCircleNam
                                    e=P3"]
```

Figure 8. *toOpcode() Implementation in CommandGroup*

```
abstract class CommandGroup extends Command {

    toOpcode(): null | string | Array<string> {
        const group: Array<String> = []
        this.subCommand.forEach((cmd:Command) => {
            const converted = cmd.toOpcode() as null | string
            if (converted !== null) group.push(converted)
        }
        return group.length > 0 ? group : null
    }
}
```

Storing Geometric Construction Scripts

For a typical construction, the result of calling `dumpOpcode()` is a (huge) JSON array of the command script which can be saved as a string in the cloud database. When the string length is below the maximum limit allowed by Firebase Firestore, the JSON array is stored directly as string in a Firestore document. Otherwise, we apply the GZIP data compression algorithm on the JSON array and save it as binary data to Firebase Storage.

When a saved construction is loaded from the cloud database and a global parser inflates the JSON string and then determines which `Command` subclass to dispatch for parsing a specific command and thus replaying its execution context. Since the command history records the exact order how geometric objects were created, parent-child dependencies in the DAG of object tree are reconstructed correctly during the replay. The task of interpreting a construction script is implemented across several functions:

- The `run()` function initiates the entire process of command interpretation. The incoming argument to this function is an array. Each item in this array can be either a string (representing a single command) or an array of strings (representing a group of commands).
- The `interpret()` function executes each command either as a single execution or a group of executions.
- The `executeIndividual`() function's role is to dispatch the proper parser based on the command name, thus converting a string encoding of a command back to a Command object. Essentially, `executeIndividual()` performs the inverse of `toOpcode()` where it turns a Command to a JSON string.

Figure 9. *Parsing and Interpreting Geometric Construction Scripts*

```
type ConstructionScript = Array<string | Array<string>>

function run(script: ConstructionScript) {
    script.forEach ((s: string | Array<string>) => {
        interpret(s)
    })
}

function interpret (command: string | Array<string>) {
    if (typeof command === "string") {
        executeIndividual(command).execute()
    } else {
        const group = new CommandGroup()
        command.forEach((cmd: string) => {
            group.addCommand(executeIndividual(cmd))
        })
        group.execute()
    }
}

const NODULE_DICTIONARY = new Map<string,SENodule>()
function executeIndividual(command: string): Command {
  const tokens = command.split("&")
  switch (tokens[0]) {
    case "AddPoint":
      return AddPointCommand.parse(command, NODULE_DICTIONARY)
    case "AddCircle":
      return AddCircleCommand.parse(command, NODULE_DICTIONARY)
    // many more cases
  }
}
```

Most of the command subclasses either create or remove objects from the global state. When a command has one or more dependencies (to the existing objects), this command must be able to locate its dependent objects. When the `parse()` function is invoked a dictionary of objects created so far is passed (`NODULE_DICTIONARY`) as the second argument of the call. Each `parse()` function must be implemented as a static function since we cannot invoke it from an existing object. The snippet of the `parse()` function of `AddPointCommand` is shown in Figure 10.

Conclusions

Incorporating various design patterns into Spherical Easel has enabled us to be very productive in building a significantly large number of features (32 tools) within a relatively short period of time (three fulltime summer semesters since April 2020). Summer 2023 was mainly used for migrating VueJS from version 2.0 to 3.0. Encapsulating individual spherical construction as a Command object gives us the

capability to create the foundational building blocks for several important features in Spherical Easel, this includes:

- Undoing graphical edits without having to explicitly saving (and later restoring) the rendering state of the drawing canvas. Instead of saving graphical properties and values to restore, we save the "recipe" (actions) to restore them.
- The "recipes" can also be transcribed into a machine-readable format (JSON string) which can saved to persistent storage.
- Since saved "recipes" are machine readable, they can be interpreted, and graphical objects can be rebuilt.
- Adding new objects and tools to expand the "ingredients" for each recipe is easy. The Composite design pattern enables us to group several (primitive) actions which must be performed collectively together in a single execution context.

Figure 10. *Parsing the AddPoint Command*

```
class AddPointCommand: Command {
  static parse(command: string, objMap: Map<string,SENodule>) {\
    const tokens = command.split("&")
    const propMap = new Map<string,string>()
    tokens.forEach((token,ind) => {
        if (ind == 0) return
        const parts = token.split("=").  // split the key-value
        propMap.put(parts[0], parts[1])
    })
    const sePointLocation = new Vector3()
    sePointLocation.from(propMap.get("pointVector"))
    const sePoint = new SEPoint()
    sePoint.locationVector.copy(sePointLocation)
    const pointName = propMap.get("objectName")
    objMap.put(pointName, sePoint).  // (*) Update the dictionary

    return new AddPointCommand(sePoint, /* more args */)
  }
}
```

Future Work

Spherical Easel is a work in progress, and we have a long list of features we are planning to implement including:

- More tools,
- A classroom mode where a teacher can view, comment, and pause all their students' work,
- An electronic system for vetting and distributing lessons plans for students of all ages,
- An earth mode which allows users to explore the geography of the earth,

- Building on the i18N package to simplify the process of translating it into more languages, and
- Implementing a more robust system of rendering and interacting with spherical parametric curves.

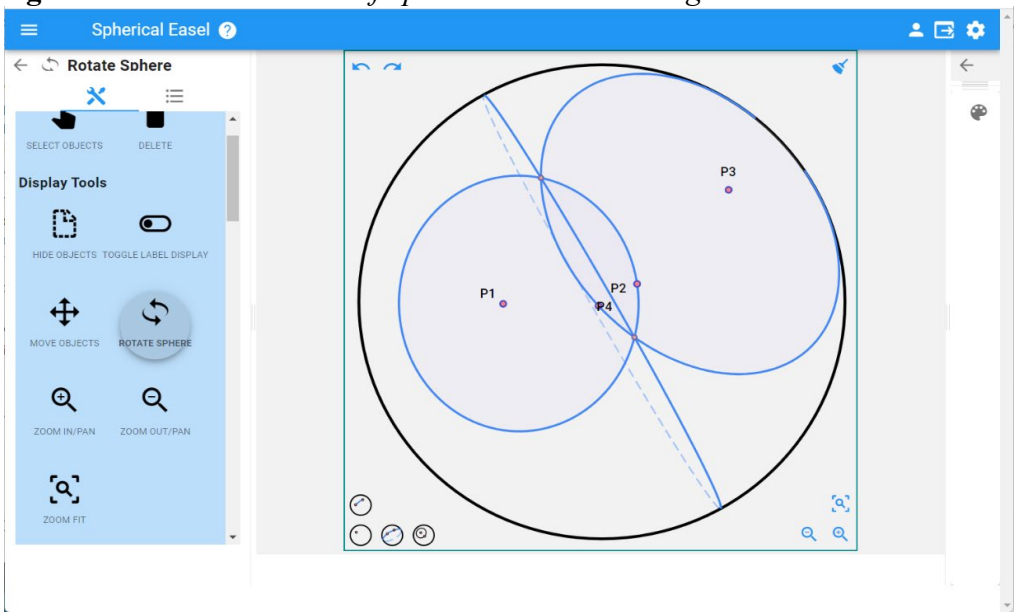
We have also involved students at several stages in the process of creating Spherical Easel. The lead authors are full time faculty members with a substantial teaching load and so time to work on the application during the academic year is very limited. In order to maintain continual progress throughout the academic year, we invite senior students in our Capstone course to contribute to the project. Additionally, in summer 2023, we were able to secure a research grant to fund a summer project which involves a collaborative team of faculty and students across three disciplines: mathematics, computer science, and visual & media arts to improve Spherical Easel usability and navigability as well as adding the new Earth Mode. The visual & media team started with the previous UI design (Figure 11) and made recommendations that were implemented by the computer science team that resulted in the current UI (Figure 1). In addition, the computer science team implemented an Earth Mode (Figure 12) that, for instance, allows the user to:

- Create new points associated with physical geographic locations on the earth.
- Measure distances between those locations.
- Measure angles among any three locations.

Teacher Session Control

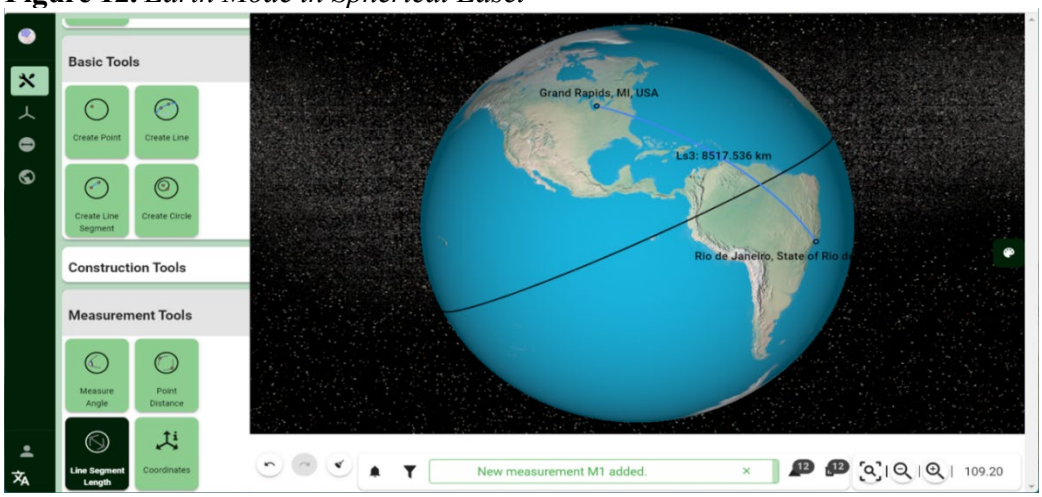
We also have experimented with implementing a feature where a teacher creates a "session" in Spherical Easel where students can join. Spherical constructions initiated by the teacher are broadcast and made visible on the students' spherical easel canvas. All these experimental features are heavily built on top of the Command pattern in conjunction with the Socket.IO library. Instead of "saving" the commands to persistent storage, the commands are transmitted to and reinterpreted by the students' spherical easel instance. The overall visual effect is the teacher's actions on his/her graphical canvas are replicated on each student canvas.

Figure 11. *The First Version of Spherical Easel UI Design*



The project source code is available on GitHub (https://github.com/dulimarta/spherical-easel) and has been deployed to Netlify. The web application itself can be accessed at https://easelgeo.app. One of the main reasons we keep our code on GitHub is to invite other collaborators to our project.

Figure 12. *Earth Mode in Spherical Easel*



Acknowledgments

We would like to recognize students Dat Nguyen, Hannah Cline and Prof. Vinicius Lima for their hard work in redesigning Spherical Easel and incorporating the new Earth Mode into the new UI design. Many capstone students and mentor Prof. Michelle Dowling also contributed to this project. Prof. David Austin was the main architect of the first version of this program published in 2003 and he was the

originator of the idea to store the relationships between geometric objects in a directed acyclic graph.

References

Back K (1997) *Smalltalk Best Practice Design Patterns*. Prentice-Hall.

Desmos Geometry. [Online.] Available at: https://www.desmos.com/geometry.

Dulimarta H, Dickinson W (2023) Spherical Easel: An Open-Source Web Application Modeling Spherical Geometry. In *III International Conference on Electrical, Computer, and Energy Technologies (ICECET 2023)*. Cape Town, South Africa.

Firebase. [Online.] Available at: https://firebase.google.com.

Gamma E, Helm R, Johnson R, Vlissides J (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Geogebra Geometry. [Online.] Available at: https://www.geogebra.org/geometry.

Górski T (2024) Smart Contract Design Pattern for Processing Logically Coherent Transaction Types. *Applied Science* 14(6): 2224.

Java Design Patterns. [Online.] Available at: https://java-design-patterns.com/patter ns/.

Oliveira Marum JP, Cunningham CH, Jones AJ, Liu Y (2024) Following the Writer's Path to the Dynamically Coalescing Reactive Chain Design Pattern. *Algorithm* 17(2): 56.

Node-GZIP. [Online.] Available at: https://npmjs.com/package/node-gzip.

NonEuclid - Hyperbolic Geometry. [Online.] Available at: https://www.cs.unm.edu/~joel/NonEuclid.

Nystrom R (2014) *Game Programming Patterns*. Genever Benning.

Schmidt D, Stal M, Rohnert H, Buschmann F (2000) *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons.

Todhunter I (1859) *Spherical Trigonometry, for the use of colleges and shcools.* London, England: Macmillan.