Geometry and Groups - “A Stereographic Projection Experience”

Tal Rastopchin

Advised by Christopher French

May 25, 2021

1 Introduction

Mathematician Grant Sanderson explains that one can view the field of group theory as being all about “codifying the idea of symmetry.” One important application of group theory is studying and classifying the symmetries of geometries. In my independent study I learned about how mathematicians gain insight into geometric structures by considering the collections of transformations preserving fundamental properties of geometric objects and spaces. In the first half of the independent study I worked through about half of Dr. Keith Carne’s reading notes for a course entitled *Geometry and Groups* which explores symmetry groups. Carne motivates and builds up to the construction of hyperbolic geometry through an investigation of metric spaces and their isometries. In the second half of the independent study I reflected on what I learned to design and build a game experience that immerses a player in some of the concepts I read about. The game experience, which I call “A Stereographic Projection Experience,” interactively guides the player through the construction of stereographic projection and some properties of Möbius transformations. A major design goal of the experience was to go beyond interactive visualization by directly engaging with important definitions and proofs.

In this paper I will reflect on and outline the design process of creating “A Stereographic Projection Experience.” In the first section I outline what background research I did on previous visualizations and interactive experiences that illustrate concepts in hyperbolic geometry. In the second section I outline my game experience and use the previous investigation to motivate design choices that I made. In the third section I reflect on important programming and computational problems I had to tackle in order to create the experience. I end this paper with some concluding thoughts about the project.

2 Previous work

What can we learn from previous visualizations and interactive experiences that illustrate concepts in hyperbolic geometry? In this section I will look at a small handful of the many wonderful experiences investigating hyperbolic geometry.

2.1 Seeing and moving through hyperbolic space

A large class of visualizations of hyperbolic geometry begin and end with using some tool to illustrate what hyperbolic spaces look like. These visualizations emphasize seeing and moving through hyperbolic space using various tessellations by different objects. We see this in art such as M. C. Escher's Circle Limit III and in @TilingBot's procedurally generated hyperbolic tilings. Coulon, Matsumoto, Segerman, and Trettel's Ray-Marching Thurston Geometries dives further into photorealism and uses advanced computer graphics to create physically-inspired renders of tilings of Thurston geometries. While these visualizations are important and can give meaningful context and insight to the subject of hyperbolic geometry, they often require additional explanation to understand what exactly is being illustrated and why we should care about it.

2.2 Constructing hyperbolic space

Many other visualizations of hyperbolic geometry go one step further to illustrate how hyperbolic space is constructed from Euclidean space. McClure's online interactive version of Arnold and Rogness' Möbius Transformations Revealed animation allows players to position and orient a Riemann sphere used to project a grid onto the xy plane and hence define their own Möbius transformations. Segerman and Schleimer's Illuminating Hyperbolic Geometry goes a step further to illustrate the construction of hyperbolic geometry by using physical light sources and 3D printed tessellations of the Riemann sphere. In the video they use these physical constructions to visualize tessellations in various models of hyperbolic geometry and in doing so provide real-world motivation and construction for how we arrive at our artwork and renders of hyperbolic tessellations. In her talk Crocheting hyperbolic planes, Daina Taimina pushes this motivation for construction even closer to reality by creating interactive tactile models that capture important tessellation and geodesic behavior of hyperbolic geometry. Taimina even uses these models to directly illustrate certain geometrical and topological proofs—something that many of these hyperbolic geometry visualizations don't do well.

2.3 Takeaways

One important takeaway from this previous work is that visualizing stereographic projection can be a useful tool in motivating the construction of hyperbolic space. McClure, Arnold, Rogness, Segerman, and Schleimer's work demonstrates how visualizing stereographic projection works well to bridge the gap between Euclidean geometry and hyperbolic geometry. Their work strongly influenced the design choice of centering the entire game experience around visualizing stereographic projection.

Another important takeaway from this previous work is that even though most of it requires additional explanation to understand what exactly is being illustrated and why we should care about it, interactive experiences can in fact illustrate formal mathematical proofs. The versatility and affordances of hyperbolic crochet allow it to do many things advanced computer graphics technologies have not yet done well. Daina Taimina's work strongly influenced the design choice of creating an experience that goes beyond interactive visualization by directly engaging with important definitions and proofs.

3 "A Stereographic Projection Experience"

3.1 Designing The Experience

Going into designing the game experience I knew I wanted to center visualizing stereographic projection and prioritize directly engaging with important definitions and proofs. My first design choice was to create a game experience that bootstrapped McClure's online interactive version of the Möbius transformations revealed animation. I wanted to add additional machinery and context in order to directly engage with important definitions and proofs. McClure's work was a great starting point because it demonstrated how well an interactive stereographic projection visualization illustrates the construction of Möbius transformations. Furthermore, McClure's experience is simple, looks good, and feels good to interact with.

After deciding I wanted to create a bootstrapped version of McClure's project I thought about what concepts I would want my game experience to explore as well as what axioms a player would need to understand and engage with those concepts. Towards the beginning of the design process I wanted the experience to explore the following concepts.

1. Stereographic projections preserve angles.
2. Stereographic projections send generalized circles to generalized circles.
3. Möbius transformations preserve angles.
4. Möbius transformations send generalized circles to generalized circles.
5. Möbius transformations can send any triple of points to any other triple of points.
6. Möbius transformations can send any generalized circle to any generalized circle.

I discussed this list with with my advisor and we realized that it's very hard to design a game experience that walks a player through proofs of these concepts.

Proofs of these concepts rely on arbitrary objects (points, circles, curves, etc.) and sometimes break down into specific computations. Because of this, we decided that it would be more useful for the experience to focus on definitions and geometric constructions. On the development side, I also quickly ran into issues with managing the point at infinity (which I'll discuss a bit later). Through discussion and prototyping, we decided that the experience would explore the following revised list of concepts.

1. The definition of stereographic projection as a geometric construction.
2. A definition of Möbius transformations as a geometric construction built using stereographic projections and rigid transformations of the Riemann sphere.
3. We can geometrically construct a Möbius transformation that takes a circle to another circle.
4. We can geometrically construct a Möbius transformation that takes a triple of points to another triple of points.

Shifting from illustrating a proof to focusing on a specific geometric construction would allow a player to solve a specific instance of a problem and hopefully gain an intuition about how one might generalize their solution. In fact, working through an example or two before proving a general theorem is something we do in mathematics all the time. This revised list of concepts became the main goals of the game experience.

3.2 A Description of the Experience

In "A Stereographic Projection Experience" a player is guided through a sequence of tasks which interactively engage them with the previously outlined mathematical concepts. The experience starts by placing the player as an observer on the real plane. Throughout the experience the observer is prompted with a couple slides of text explaining what to do and giving them a task. The slides of text are followed by a task which is often followed by more slides of text. The game experience essentially guides the player through the following tasks (in the order they are presented).

1. To teach the player how to rotate the camera, the player is instructed to left-click and drag to find a glowing blue sphere behind them. The player must left-click on the glowing blue sphere to complete the task. The player is also instructed to scroll their wheel to learn how to move around.
2. The player is told that the blue sphere in front of them is called a Riemann sphere and that the small metallic ball on top of the Riemann sphere marks its north pole. The player is then given a definition of stereographic projection and is asked to right-click to place points on the Riemann sphere. The player then sees an animation visualizing the stereographic projection of the point they placed and has to confirm they see the result of the

projection by left-clicking the resulting point. After placing and following their first point, the player is prompted to repeat this sequence three more times, experimenting with where they place the point on the sphere.

3. The player is next given a definition of the inverse stereographic projection and is asked to right-click to place points on the plane. The player then sees an animation visualizing the inverse stereographic projection of the point they placed and has to confirm they see the result of the projection by left-clicking the resulting point. After placing and following their first point, the player is prompted to repeat this sequence three more times, experimenting with where they place the point on the plane.

4. The player is next given a brief overview of how Möbius transformations are constructed using stereographic projection interspersed with translation and rotation of the Riemann sphere. The player is then prompted to learn how to translate the Riemann sphere and given the task of moving the sphere using one of the arrow gizmos.

5. The player is next given the task of rotating the sphere using one of the arrow gizmos.

6. The player is then taught how to geometrically construct a Möbius transformation. Thee player is first presented with a black player circle on the plane and is instructed to use the Q key to inverse stereographically project it onto the Riemann sphere. The player must then translate and rotate the Riemann sphere by some significant amount. Lastly, the player is instructed to use the W key to stereographically project the circle back onto the plane and finish constructing their first Möbius transformation.

7. The player is next given a black player circle and a yellow target circle. They are instructed to use some sequence of stereographic projections interspersed with translations and rotations of the Riemann sphere to construct a Möbius transformation that takes the black player circle to the yellow target circle.

8. The player is then asked to repeat the previous task except with an additional red point stuck to both the black player circle and the yellow target circle. The player has the additional goal that their constructed Möbius transformation must also send the red player point to the red target point.

9. The player is then asked to repeat the previous task except with an additional green point stuck to both the black player circle and the yellow target circle. The player has the additional goal that their constructed Möbius transformation must also send the green player point to the green target point.

10. The player is then asked to repeat the previous task except with an additional blue point stuck to both the black player circle and the yellow target circle. The player has the additional goal that their constructed

Möbius transformation must also send the blue player point to the blue target point.

11. At this point the player has completed the experience. The experience explains to the player that it turns out that given any six points of the plane we can construct a Möbius transformation that takes the first three points to the last three points. The experience thanks the player for engaging and ends.

Game Programming and Computational Design

In this section I reflect on and outline three important programming and computational problems I had to tackle in order to create the experience.

3.3 Circle Representations

The game experience heavily relies on the visualization, manipulation, and transformation of circles. The game experience must be able to model and render circles that are subject to translation, rotation, and stereographic projection. I ended up using the most straightforward technique of tessellating the circle into a finite number of vertices and applying transformations point-wise to the circle tessellation. A scientist might point out that this technique is computationally expensive and fundamentally lossy, and they would be right. I in fact was worried about the technique being non-performant and lossy, and so I spent some time trying to represent circles analytically by using three points to define a circle. Because you can reverse-engineer a circle from three points, one could apply a transformation to a circle by applying it point-wise to its three defining points and then reverse-engineering the image circle from the image points. I tried to implement this and found it was a horribly numerically unstable technique. There's probably a smarter way to robustly model and render a circle subject to transformations, but it turned out that a direct tessellation with 2^{10} points looked great and did not slow down the program in any significant way. (There is one caveat-if the circle gets too close to the Riemann sphere's north pole, numerical issues make the circle start to fall apart. I was not sure how to solve this and did not deal with it in my implementation of the experience.)

3.4 Gizmos

The game experience also heavily relies on the player being able to translate and rotate the Riemann sphere. The game experience must provide an intuitive interface that allows the player to perform these operations. Most interactive 3D software allows users to manipulate objects using transformation gizmos. Transformation gizmos are shapes that you can click-and-drag in order to apply a transformation to a 3D object. Directly drawing from Unity's gizmos, I knew that I wanted to create arrow gizmos to allow the player to translate the

Riemann sphere and circle gizmos to allow the player to rotate the Riemann sphere.

One significant problem I had to solve was that of raycasting circle gizmos. Whereas with the arrow gizmo we can just raycast a capsule collider containing the arrow, Unity doesn't have a torus collider we could use to contain and raycast a circle. I chose to tackle the problem by combining the results from a plane and sphere raycast. If I had more time I might have gone back and implemented it using Inigo Quilez's ray-torus intersection code. The big idea behind my solution is that if we raycast both the plane and the sphere containing the circle we have enough information to say whether or not the raycast was close enough to the circle. This is the approach that I took.

1. Let e be the ray's origin and et d be the ray's direction so that the ray equation is $e + td$ for $t \in [0, \infty)$. Let p be the center of the circle and let n be the normal of the plane containing circle.

2. The first thing we do is try to raycast the plane containing the circle. If the ray intersects the plane containing the circle, let q be the ray-plane intersection point. Now we can determine if q is close enough to the circle (by some threshold $\varepsilon > 0$) by plugging it into the equation for the sphere containing the circle we are raycasting. If q is close enough to that sphere, because it is also on the plane containing the circle, we can say that the ray intersects the circle and return true.

3. Suppose that the ray did not intersect the plane containing the circle, or it did but the intersection point was too far away from the surface of the sphere containing the circle. In this case we can raycast the sphere containing the circle. If the ray intersected the sphere we have two possible intersection points (one for each "side") of the sphere. Similar to before, we now check how far away these intersection points are to the plane containing the circle. If an intersection point is close enough to the plane and is on the sphere then it is close enough to the circle. If at least one of the intersection points is close enough, we return true.

4. If we've reached this point then our ray did not intersect the sphere or the intersection points were not close enough to the circle. At this point we can return false.

3.5 Managing Input

The game experience uses mouse input such as clicks and movement in order to allow the player to move to a next text slide, rotate the camera, and manipulate the Riemann sphere. When multiple `MonoBehaviour`s are using `Update()` and coroutine updates to query for mouse input, it can be difficult to ensure that events happen mutually exlusively and execute in desired order. For example both the `CameraController` and `Transformable` scripts need to determine if the mouse is clicked in order to perform a specific action. The `CameraController` might check if the left mouse button is clicked down in order to start rotating the camera and the `Transformable` might also check if the left mouse button is clicked down in order to start translating the Riemann sphere.

If we don't specify that the operations need to be mutually exclusive, we could result in the case that clicking on and dragging an arrow gizmo not only translates the Riemann sphere but also rotates the camera. It would be very disorienting to try to move something as we are rotating, and so we don't want that to happen. A first step at solving this problem would be to use some sort of global control variable that tracks whether or not some other script has already requested and used input from the mouse. At the beginning of each frame this global control variable could be reset to false. To attempt to process input, each script would then do something along the lines of the following each frame.

1. Check the global control variable to determine whether or not a different script has already processed input. If another script has already processed input, the script would give up.

2. If another script has not already processed input, the script would alter the global control variable to indicate a script has processed input and then proceed to process input.

In this way we can ensure that only one script at a time processes input. In fact this works! It solves the problem of mutual exclusivity, however, it does not solve the problem of executing the scripts in the desired order. Now when the player hovers their mouse over an arrow gizmo, left-clicks, and drags their mouse, the camera rotates and the Riemann sphere stays in place. That is not the desired behaviour because we want the player to be able to translate the Riemann sphere using the arrow gizmos.

According to Unity's documentation on the order of execution for event functions, it looks like if we tell Unity that the `Transformable` script should always execute before the `CameraController` script, we can ensure that the `Transformable` script can get control of the global control variable before the `CameraController` script would. I implemented this and experimentally it seemed to work for the first time I translated the Riemann sphere, but the second time I tried to translate the Riemann sphere I ended up rotating the camera once again. Either my implementation was incorrect or I could not ensure that certain scripts' coroutine updates executed before other scripts' coroutine updates.

All of this lead me to design an `InputManager` script that solved the problems of ensuring mutual exclusivity and correct sequence of execution. My solution drew inspiration from operating system scheduler algorithms and assigned priorities to each script requesting to hangle input. Each frame the `InputManager` script lazily computes a list of scripts that are requesting to handle input the current frame. It only computes this list once some other script signals that it is requesting to handle input. After compiling a list of all the scripts requesting to handle input that frame, the `InputManager` sorts them by priority and only allows the requesting script with highest priority to handle input.

The `InputManager` script provides a public API that allows other scripts to add themselves to the list of possible scripts that are requesting input through the `AddInputHandler(IInput inputHandler, int priority)` method. You'll notice that I've also created an interface `IInputHandler` that requires scripts participating scripts to implement a `RequestingToHandleInput()` method to signal whether or not scripts are requesting to handle input the current frame. After adding themselves to be managed by the `InputManager`, scripts can request to manage input by calling the public

```
bool RequestToHandleInput(IInputHandler requestingInputHandler);
```

method. Internally, this method works as follows.

1. If the `requestingInputHandler` is not being managed by the `InputManager`, we return false.
2. Otherwise, we check if we have already computed the list of of all which of the managed input handlers are requesting to handle input this frame. If we haven't, we iterate through of the input handlers managed by `InputManager`, and add them to the list of input handlers requesting to handle input this frame if they might be requesting to handle input (if `RequestingToHandleInput()` returns true). Because this list contains every script that might request to handle input this frame, we can determine which input handler gets to handle input by sorting the compiled list according to the priority assigned to each input handler. The `InputManager` internally stores this input handler as the `controllingInputHandler` for the duration of the frame.
3. At this point the `InputManager` knows which input handler is the `controllingInputHandler`. The method can return `true` if the `requestingInputHandler == controllingInputHandler` and return `false` otherwise.

In this fashion, we can assign a priority to every script that adds itself to be managed by the `InputManager`, and each script can use the the `RequestToHandleInput(...)` method to determine whether or not to proceed with a particular operation. To solve our problem, what's left is to

1. Make sure the `CameraController` and `Transformable` components add themselves to the `InputManager` at the beginning of their lifetime. When we add them to the `InputManager` we need to make sure to assign `Transformable` a highe priority than `CameraController`.
2. Implement a `RequestingToHandleInput()` method for the `CameraController` script. This method returns true if the left mouse button is clicked (as well as if the scroll wheel has moved).
3. Implement a `RequestingToHandleInput()` method for the `Transformable` script. This is a more complicated method than that of the `CameraController`. We have to return true if any one of the arrow or circle gizmos is behind the mouse.

At this point we've done a lot of work to set up a `InputManager` and subscribe scripts that are handling input to it, but our problem is solved. By manually specifying that the `Transformable` has a higher priority than the `CameraController` we ensure that whenever we click on and drag any of the `Transformable`'s gizmos, the correct transformation takes place. Whenever we click on and drag anywhere else, the `CameraController` rotates the camera. What's great about the `InputManager` is that we can use the same functionality to ensure that whenever text is displayed neither the `CameraController` or `Transformable` handle input.

A concerned scientist might point out that this solution is computationally expensive. Particularly, calling every input handling script's `RequestingToHandleInput()` method each frame can be expensive when a `Transformable`'s implementation relies on at least 6 raycasts (the circle gizmo raycasts actually use 3 per gizmo). Just like with the case of the circle representations, there's probably a smarter way to do this, but this solution worked just fine and did not slow down the program in any significant way.

Conclusion

I've used the indefinite article "a" instead of the definite article "the" in naming the game "A Stereographic Projection Experience" with the intention of recognizing that the game is only one of many ways to experience stereographic projection and Möbius transformations. My independent study concluded with finishing "A Stereographic Projection Experience" and writing this accompanying explanatory paper. For what it is I think that the game experience works well to go beyond interactive virtualization by directly engaging with important definitions and proofs. Namely, the experience interactively guides the player through the geometric construction of stereographic projection and Möbius transformations. The experience also guides the player through constructing Möbius transformations that send particular circles to particular circles. The experience built up to the sharply 3-transitive property of Möbius transformations by concluding with a task where the player had to construct a Möbius transformations that mapped a particular circle plus a set of three points to a corresponding circle plus a set of three points. I'm really proud of what the game experience accomplishes.