Tetrobot

John Russell Strauss

CS 6491 Fall 2019

Phase 1

For phase one of this project, I committed myself to using a different technology than the provided Processing example and included methods provided by Jarek. In the first 2 weeks my main priority was to determine if this was a viable route, and I decided that while potentially risky, being forced to develop my own methods to perform many geometrical transformations would promote deeper understanding and strengthen my Computer Graphics muscles. The majority of the time thus far has been spent learning how to display, translate, and rotate geometries and the bare basics of learning how to work with the Three.js framework. I have spent time writing custom methods to create shapes, draw points, display and label axes, calculate midpoints, calculate triangle and tetrahedral centroids, create vectors, get distances, and various other basic functions. All relevant Computer Graphics related code is contained within assets\js\components\scene.js. All code is original.

You will see settings at the top and the rest divided logically in the functions below them. The majority of all geometric logic stems from the addTetrahedron method. This is where the tetrahedron is added and initially rotated to be placed flat on the floor centered on the origin. After rotating the tetrahedron so that its bottom face is parallel with the floor, it is oriented below the ground due to this rotation, so I perform some calculations to move it upward. I calculate the centroid (source below) and use that value to move the tetrahedron up to be flat on the floor. The centroid of the faces and the tetrahedron can be calculated by calculating the midpoints of the respective geometry that you are looking for, whether a triangle face or the whole tetrahedron, the only difference being whether you average 3 or 4 vertices. The math for this calculation is contained within the method called calculateCentroidLocation. Once aligned, I can begin making calculations for the tetrahedral movement. I created a method that will let me define a rotation axis by any two points and then rotate any geometry around the axis created by those points by any angle of my input. I have gotten parts of it working, but there are problems with this method that I am still trying to locate. I was able to get the correct rotations by changing the inputs, but I think it is causing compounding issues in the rotations and therefore preventing my program from functioning correctly. For example,

every roll is correct on the first iteration. But after the first iteration, only alternating left and right directions function correctly. I believe this rotation function (threeStepRotation) is to blame, and this is the fist thing I need to fix before proceeding further. To see an example of an error-less working path, use LRLRLRLRL as your inputs to demonstrate the basic functionality.

The logic of this rotation method is as follows:

1. Take rotation axis $\overrightarrow{V}$ and move it to the origin. Perform same translation on object being rotated.
2. Rotate$\overrightarrow{V}$onto the YZ plane. Measure this rotation as angle -θ. You can calculate the value of the angle with arctan($\frac{Vx}{Vz}$).
3. Rotate $\overrightarrow{V}$ to be co-linear with the Z-axis. Measure this angle as Phi. You can calculate this angle with arctan($\frac{Vy}{\sqrt{Vx^2+Vz^2}}$). These are the singular components of V using Pythagorean theorem.
4. Now rotate object by this angle Phi. Since our rotation axis is now co-linear with the Z-axis, we can now rotate our object by our original desired angle. The remaining step involved performing the remaining 3 steps in reverse: rotate the object around the X-axis by -φ, rotate around Y by θ, and then perform the same translation in step 1, but in the negative direction.

Using this rotation and defining my rotation axis as the edge of one of the bottom faces of the tetrahedron (pivot), I can begin rotating the entire object to make it move from step to step. Since the equilateral triangle has interior angles of 60°, I can rotate it 120° (because 180° - 60°), or $\frac{2\Pi}{3}$. I then merely create a method for applying each direction, update the location, and pass the updated geometry location into the same method to take further steps.

Sources

- Calculate the centroid of a triangle and tetrahedron
  - https://youtu.be/Infxzuqd_F4
- Rotation Around an Arbitrary Axis – UC Davis Academics
  - https://www.youtube.com/watch?v=gRVxv8kWl0Q&t=1224s

Phase 2

For phase two, I have built on the functionality of the first phase by adding new methods and refactoring my faulty rotations from phase one. First off, I

have added reset functionality (esc key) in order to reset the scene, camera, and position of the robot. This helps the developer test more easily and the user to try again. It is still a work in progress with known issues, however, with my rotation logic working correctly, I can now accurately move the robot. This logic is contained within rotatePointAboutLine(), line 345. I have refactored the logic to calculate the opposite edge midpoint as the robot moves by calculating shared vertices from the current robot position to the previous robot position. Then in combination with the top vertex of the robot, I can calculate a direction parallel to the floor to use as my direction vector. The current issue is with the angle of rotation with each step. My next requirement is to correctly calculate this rotation angle as the robot moves so that he can stay along the floor. The logic of this rotation method is as follows.

Apply the following transformations to both your rotation axis and your point or object in space:

1. Move your rotation axis so that one endpoint lies on origin.
2. Rotate space along the X-axis so that the rotation axis lies on the XZ-plane.
3. Rotation space along the Y-axis so that the rotation axis lies perfectly along the Z-axis.
4. Now perform your original desired rotation along the Z-axis.
5. Reverse the Y-axis rotation previously applied.
6. Reverse the X-axis rotation previously applied.
7. Undo the translation to the origin.
8. Your point is now rotated through 3D space.

I then wrote a method (rotateGeometryAboutLine(), line 411) that took a list of points and applied the rotations to all vertices in the geometry input along the desired axis. This resulted in the entire object rotating through space about any axis of choice defined by two points. From this point, I will be poised to rotate the robot correctly on each step and write easing functions to modify acceleration during the animation loop as a function of time.

Sources

- http://paulbourke.net/geometry/rotate/

Phase 3

For this phase of the project, I have implement two functions that are crucial to my calculations for tetrahedral rotation. The first is a matter of

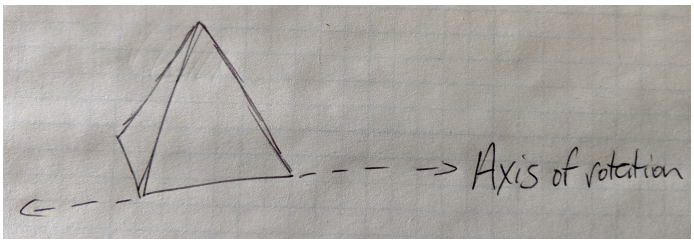

Figure 1: Axis of rotation

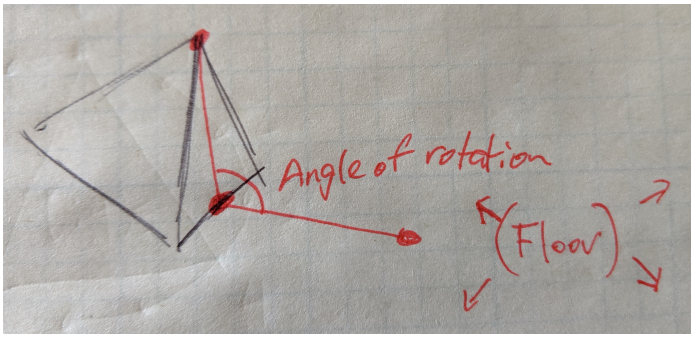

Figure 2: Angle of rotation

calculating the interior angle between two vectors. In the previous phase, I could not calculate the angle correctly, and thus could not accurately know how much to rotate the tetrahedron about its axis of rotation for each roll. We can create an axis of rotation using the two vertices of the edge in the direction we would like to roll.

```
getAngleBetweenVectors: function(vector1, vector2) {
        let dot = vector1.dot(vector2);
        let length1 = vector1.length();
        let length2 = vector2.length();
        let angle = Math.acos(dot / (length1 * length2));
        return angle;
}
```

The first step is to calculate the dot product of these two vectors. Then, divide by the products of the vectors length, and run this value through the arcosine function to get the desired angle value.

Generate vector $\overrightarrow{A}$ by starting at the midpoint of the edge in the direction of desired travel and ending at the highest vertex in the tetrahedron geometry.

Then generate vector $\overrightarrow{B}$ in two steps:

1. Get the vector from the centroid of the base of the tetrahedron to the midpoint on the travel edge.

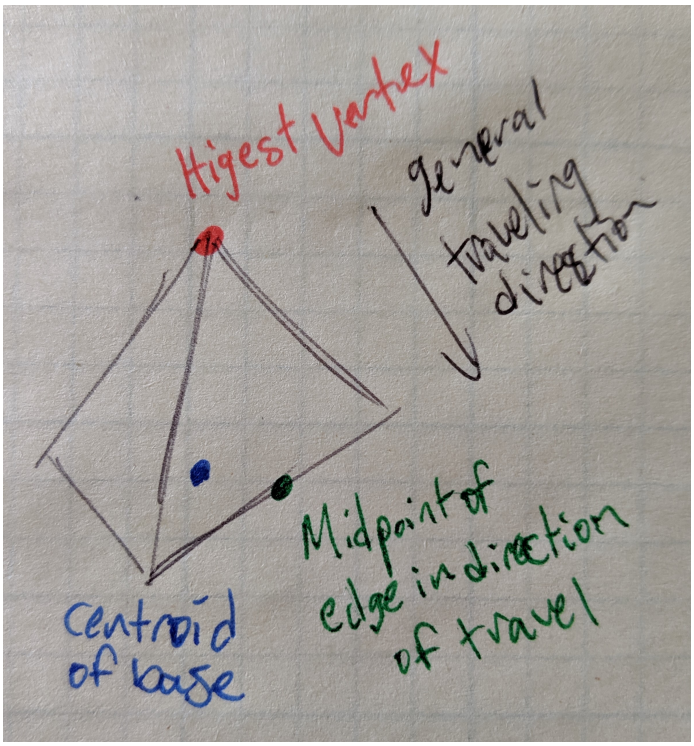

Figure 3: Finding vector $\overrightarrow{A}$

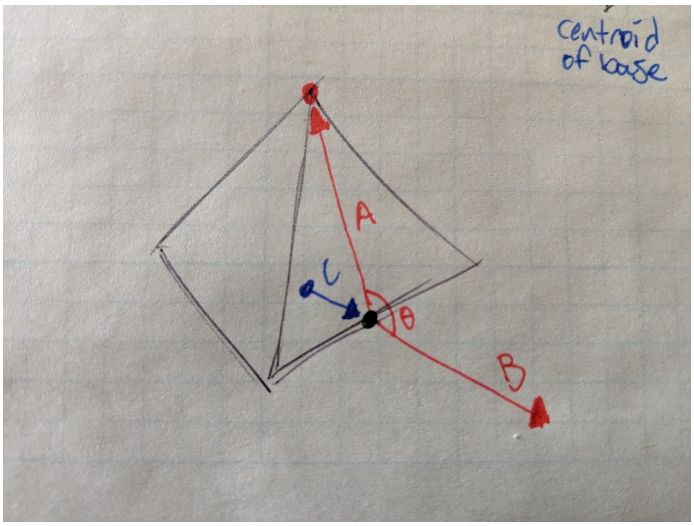

Figure 4: Searching for angle θ

2. Project distance $\overrightarrow{A}$ on vector $\overrightarrow{C}$ from the starting point of the midpoint. Now we have the two vectors used to determine the angle of the roll.

Another important method development is to determine our right turn points so that we can orient ourselves in 3D space to understand which way a roll will be performed. First create vectors to and from the point to would like to measure to the other two. Measure the cross product of these two vectors. This dot product forms an orthogonal vector, which we can measure the sign of to determine true or false for a right turn. For my program, I assume a vector pointing toward the sky (the positive y-axis) determines the positivity of this vector.

```
isRightTurn: function(startingPoint, turningPoint, endingPoint) {
        let segment1 = graphics.createVector(startingPoint, turningPoint);
        let segment2 = graphics.createVector(turningPoint, endingPoint);
        let result = new THREE.Vector3();
        result.crossVectors(segment1, segment2);
        return result.y > 0;
}
```

Sources

- http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap35.htm

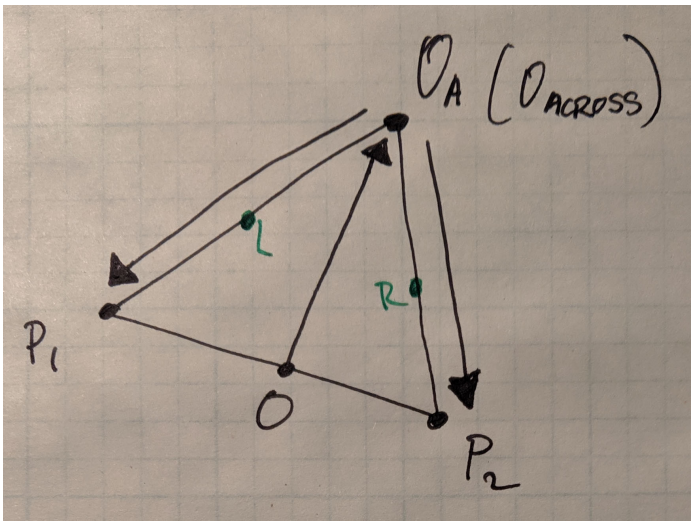

Figure 5: isRightTurn(O, O_A,P_1) returns false, therefore $P_1 = O_{Left}$; isRightTurn(O, O_A,P_2) returns false, therefore $P_2 = O_{Right}$

Phase 4

For the fourth phase of this project, I have fixed some of the fundamental problems that were prohibiting me from making progress and fulfilling the requirements of the previous phases. The first problem is that I was using hard-coded indices of my geometry vertices (i.e. performing operations and assuming that tetrahedronGeometry.vertices[i] is in a certain position/orientation). This led to too many edge cases by making assumptions about the orientation of the vertices. Eventually, I began to do geometric calculations to determine the orientation and perform operations on the vertices instead of looping through the vertices, and this solved many of my problems. First, I set the location of the very first opposite edge, calculated the midpoint of that edge (call it O), and then determined the left and right edges based on the location of that midpoint by using the rightTurn() method (see Figure 5). Once I determine the location of O, I can calculate the height, and form a vector normal to the opposite edge to move the point across from O. So if the vertex across from O is called Oa, then I can use the right turn method with O, Oa, and the remaining two tetrahedron vertices to determine which side is left or right:

```
isRightTurn(O, Oa, P[i])
```

where P[i] is either 2 vertices that share the opposite edge with O. If it returns true, we are looking at the right edge, else it is the left edge. Then I have all sides oriented to rotate the tetrahedron in whichever direction without looping through the tetrahedron's vertices and trying to assume they are in the location that I expect them to be by their vertex order in the geometry.

Then, I used the method laid out in the Phase 3 report to calculate the roll angle and rotate the entire tetrahedron around the axis of rotation (edge of the tetrahedron in the direction of desired rotation) to begin its rolling movements.

The second major technical roadblock that I was experiencing was the inability to update a mesh's orientation on the screen as the shape's geometry is updated. For example, three.js uses different object types to calculate and display 3D objects on the screen. The first type of object is a geometry, which stores all vertices and forms the structure of the shape. A material is configured and this material is applied to the geometry in the form of a mesh. While the geometry forms the structure, I am performing explicit geometrical operations on these geometries' vertices to move the object throughout space. The problem is that these operations do not update the mesh's position or orientation unless a very specific set of procedures is performed:

Once I began using the set() method instead of

```
geometry.vertices[i].x = newPointLocation;
```

to update the position and then ran

```
geometry.verticesNeedUpdate = true;
```

in every animation frame, the mesh's location began to update and the tetrahedron was now moving.