

# Modelling and Simulating Complex Projectile Motion

A. Joshi and S. Alam

## 1 Intercontinental Projectile Modelling

### 1.1 Tools to begin with

Before starting off with a project as such, it is imperative to take inventory of the tools we have at our disposal, and begin building a solution to the problem statement.

#### 1.1.1 SFML

As with our previous applications, we decided to use SFML as our primary graphics library. We decided to keep using SFML, as to finish this project faster, and prevent ourselves from getting bogged down in the details of learning a new library.

Since we were not using OpenGL directly, we will be rendering purely on the CPU side, and will not be lending the help of the GPU for rasterization. This is a trade-off we were willing to make, as we were not looking to make a game, but rather a simulation.

#### 1.1.2 ImGui

Like with SFML, we already had the code infrastructure to use ImGui, and we decided to use it for the same reasons as SFML.

#### 1.1.3 GLM

The only new addition to our toolset was GLM. We decided to use GLM for its vector and matrix operations. If we had written our own matrix and vector classes we would not have been able to optimise our code as well as GLM, which would have lead to inefficiencies, especially when we aren't using the GPU for these calculations.

## 1.2 Algorithm

### 1.2.1 Motivation

Our intention is to create a projectile launcher, that works on a model of the Earth. We can decompose this problem into the following steps:

1. Be able to render a sphere
2. Be able to map a texture on to the sphere
3. Be able to draw a point on the sphere (as the launch point)
4. Be able to compute and draw the trajectory of the projectile
5. Be able to account for the rotation of the Earth
6. Be able to animate the projectile

### 1.2.2 Rendering a Sphere

To render a sphere, we must note that we have no way to render a sphere directly (since we do not have access to OpenGL, and could use `gluSphere()`). Note that SFML does provide a method for us to render polygons, so we thought of rendering a sphere in terms of polygons.

To do so, we must first introduce the idea triangle subdivision. The idea is to take a triangle and divide it into smaller triangles (as the name would suggest). There are multiple ways of doing this (see [3] for more), and the general reasoning behind this that we can have a better more refined representation of any polygon, without having to store extra information<sup>1</sup>. See in figure 1 how by subdividing a tetrahedron, we can approximate a smooth surface. Usually a few iterations of this process is suffice to give a good approximation of the limit surface.

The second idea we must introduce is of normalisation, with respect to a set distance. Normally, normalisation preserves the direction of a vector, but scales it such that its magnitude is 1. Our normalisation is a bit different, however, because we don't end up with magnitude 1, but rather a magnitude of a set distance.

Here is a two-dimensional example of normalisation with respect to a distance: 3 shows two points,  $A$  and  $B$ , and the line drawn between them. Currently, the distance between  $A$  and  $B$  is 6 units, however if one were tasked to find a point on the line  $AB$  that is 12 units away from  $A$  (see figure 4 as point  $C$ ).

More generally, we can say that this point  $C$  will always be colinear with  $A$  and

$B$ , but isn't necessarily on the line segment  $AB$ .

Staying with the two dimension story, if we were to draw a set point of points  $P$  that were all on a straight line not going through the origin, and we were to normalise them in reference to the origin, with a certain distance  $d$ , we would construct an arc of a circle with radius  $d$ , since all this exercise is, is drawing a set of points on a circle with radius  $d$ . It is then trivial to prove that the same would hold in three dimensions<sup>2</sup>.

The reason to even go through such an exercise, it to realise that we can start of a octahedron, and then subdivide it, yielding us the points on a straight line. Then we can normalise these points to get the points on a sphere, and obviously we can also control the radius of such a sphere. To keep things simple, we use an octahedron, because it is a comprised of 8 equilateral triangles, which are trivial to subdivide.

Now that we have our points that we can render, we need to somehow convert these from 3D to 2D, so that we can render them on to the screen. This is where GLM does most of the heavy lifting, in that, we don't have to manually construct the equations to this, but can leave it up to GLM to do this for us.

### The Rendering Pipeline

Throughout this procedure, we will be using 4D vectors  $(x, y, z, w)$ , and  $4 \times 4$  matrices. The reason for this is that we can use the fourth dimension to store information about the vector. I would recommend using [2] as a guide to further understanding the intricate process defined here. This being

<sup>1</sup>Though, obviously we take a memory and computation penalty for this, we can achieve a *smooth limit surface*

<sup>2</sup>This is a exercise left to the reader

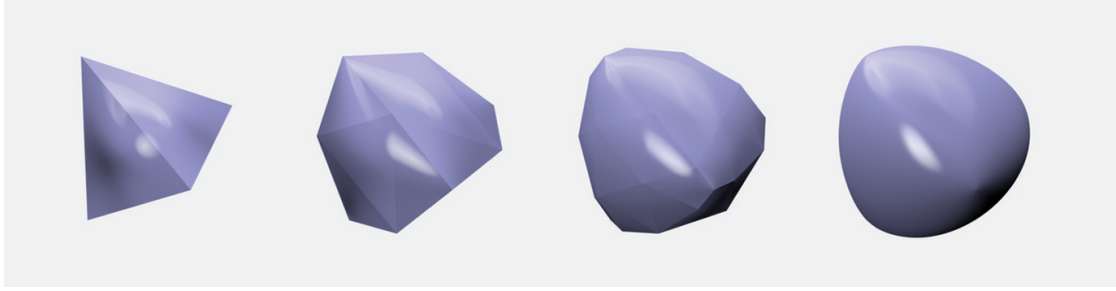


Figure 1: Image shows the tetrahedron being subdivided 0, 1, 2, 6 times

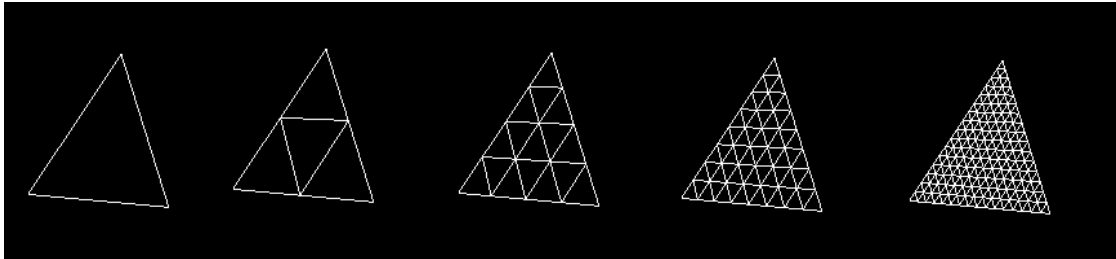


Figure 2: shows the subdivison process on a equilateral triangle

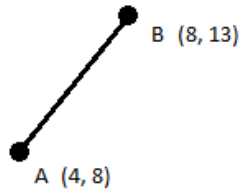


Figure 3: Image shows the tetrahedron being subdivided 0, 1, 2, 6 times

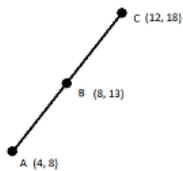


Figure 4: shows the point C, which is 12 units away from A

- If  $w = 0$ , then the vector is a direction vector
- If  $w = 1$ , then the vector is a point

To begin, all of the points that describe a sphere are relative to the origin (obviously), however this origin is not necessarily the origin of the world (but rather relative to the origin of the model). To make it relative to the world, we can apply a *model* matrix transformation. The model matrix is consistent of:

- A translation matrix – which describes the position of the object in the world relative to the origin of the world.
- A rotation matrix – which describes the orientation of the object in the world relative to the basis vectors of the world.
- A scaling matrix – which describes the size of the object in the world relative to the basis vectors of the world.

After applying the model matrix, our coordinates are now in *world space* (points are defined relative to the origin of the world). Quote from Futurama:

‘The engines don’t move the ship at all. The ship stays where it is and the engines move the universe around it’

For example, if you want to view a mountain from a different angle, you can either move the camera or move the mountain. Whilst not practical in real life, the latter is easier and simpler in CG than the former

Initially, your camera is at the origin of the world space and you want to move your camera 3 units to the right, this would

be equivalent of moving the entire world 3 units to the left instead. Mathematically, this is equivalent of describing everything in terms of the basis vectors defined relative to the camera, rather than in world space. This is the idea behind the view matrix.

Now that we are in Camera Space, we can start to project our points onto the screen. This is done by the projection matrix. We obviously have to use the  $x$  and  $y$  coordinates of the points to determine where to place our points on the screen, however we must also use the  $z$  coordinate to determine which point should be more on the screen than the other. The projection matrix converts the frustum of the camera to a cube, and then scales the cube to the screen. 5 shows the steps taken described here. Once our coordinates have been projected onto the screen, we can then render them using SFML, this is done by creating a vertex array, and then filling it with the points we have projected onto the screen.

---

```
<<Get UV coordinate for a point
    xyz>>
<<Rendering a Sphere>>=
<<Get subdivided octahedron>>
<<Map the octahedron onto a sphere>>

sf::Texture texture = sf::Texture();
for (int i = 0; i <
    triangles.size(); i++) {
    glm::vec3 v1 = triangles[i].v1;
    glm::vec3 v2 = triangles[i].v2;
    glm::vec3 v3 = triangles[i].v3;

    glm::vec4 p1 = MVP * glm::vec4(v1,
        1.0f);
    glm::vec4 p2 = MVP * glm::vec4(v2,
        1.0f);
    glm::vec4 p3 = MVP * glm::vec4(v3,
        1.0f);
```

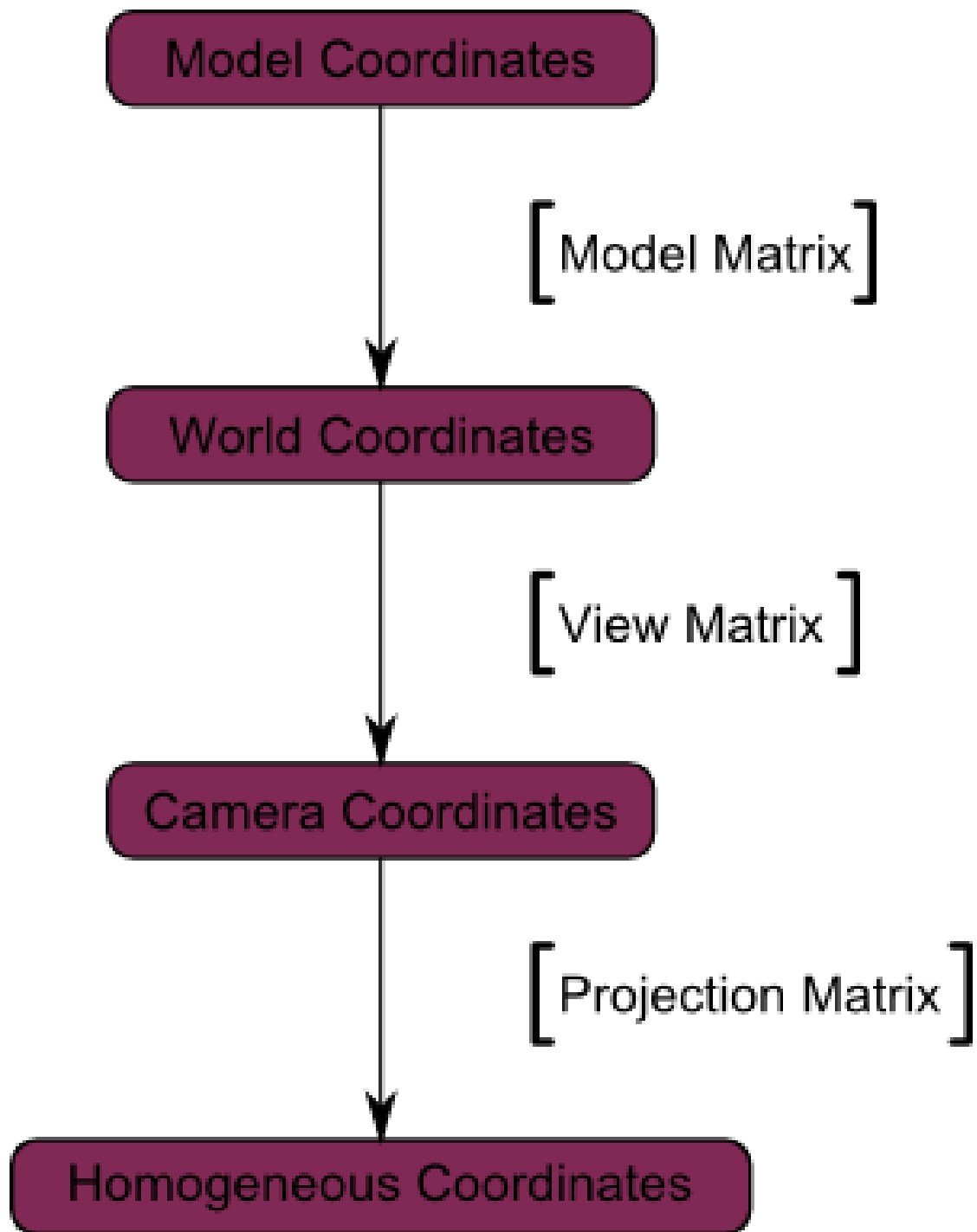


Figure 5: shows the steps taken to get screen coordinates

```

sf::VertexArray
    triangle(sf::Triangles, 3);
triangle[0].position =
    sf::Vector2f(p1.x, p1.y);
triangle[1].position =
    sf::Vector2f(p2.x, p2.y);
triangle[2].position =
    sf::Vector2f(p3.x, p3.y);

<<Set UV coordinates>>
    window.draw(triangle,
        &texture);
}

```

---

### 1.2.3 Mapping a texture onto the sphere

After the arduous task of getting the triangles we want on to the screen, we can now move on to the task of mapping a texture onto the sphere. To do so, we must introduce the idea of *uv* coordinates. These coordinates specify the location of a 2D source image (or in some 2D parameterized space). We need to find a mapping of a point from a 3D surface (in this case of a sphere) onto *uv* coordinates.

*uv* coordinates are defined in the range  $[0, 1]$ , where  $u$  is the horizontal coordinate and  $v$  is the vertical coordinate. Their range allows them to be used in any texture, regardless of size, since they are relative to the size of the texture.

For spheres, surface coordinates are defined in terms of two angles  $\theta$  and  $\phi$ , where  $\theta$  measures the angle made between the  $y$  axis and the point and  $\phi$  is the angle about

the  $y$  axis<sup>3</sup>. To begin with then<sup>4</sup>:

$$\begin{aligned}
 y &= -\cos(\theta) \\
 x &= -\cos(\phi) \sin(\phi) \\
 z &= \sin(\phi) \sin(\theta)
 \end{aligned}$$

From this we can infer that:

$$\begin{aligned}
 \theta &= \arccos(-y) \\
 \phi &= \text{atan2}(z, -x)
 \end{aligned}$$

Where  $\text{atan2}$  is the four-quadrant inverse tangent function. This returns values in the range  $[-\pi, \pi]$ , however these values go from 0 to  $\pi$ , then flip to  $-\pi$ , proceeding back to 0. While mathematically correct, this cannot be used to map *uv* coordinate, since we want a smooth transition from 0 to 1.

Fortunately,

$$\text{atan2}(a, b) = \text{atan2}(-a, -b) + \pi$$

This formulation gives values in the desired smooth range of  $[0, 2\pi]$ , therefore

$$\phi = \text{atan2}(z, -x) + \pi$$

Since we have our  $\theta$  and  $\phi$  values, we can now convert them to *uv* coordinates. This is done by:

$$\begin{aligned}
 u &= \frac{\phi}{2\pi} \\
 v &= \frac{\theta}{\pi}
 \end{aligned}$$

Now that we have our *uv* coordinates, SFML provides a method of interpolation between these coordinates defined by the vertices of the triangles, so we need not worry about the interpolation of the *uv* coordinates:

<sup>3</sup> Annoyingly, many textbook definitions of  $\phi$  and  $\theta$  are not only swapped, but also the axes of measurement are also changed, we consider the "poles" of our sphere to be the  $y$  axis, however many textbooks consider the "poles" to be the  $z$  axis, which ends up changing the equations in a subtle, yet frustrating to debug manner.

<sup>4</sup> Assuming unit sphere

---

```

<<Get UV coordinate for a point
xyz>>=
glm::vec2 getUV(glm::vec3 xyz) {
    float theta = acos(-xyz.y);
    float phi = atan2(xyz.z,
        -xyz.x) + M_PI;
    return glm::vec2(phi / (2 *
        M_PI), theta / M_PI);
}
<<Set UV coordinates>>=
glm::vec2 uv1 = getUV(v1);
glm::vec2 uv2 = getUV(v2);
glm::vec2 uv3 = getUV(v3);

triangle[0].texCoords =
    sf::Vector2f(uv1.x, 1 - uv1.y);
triangle[1].texCoords =
    sf::Vector2f(uv2.x, 1 - uv2.y);
triangle[2].texCoords =
    sf::Vector2f(uv3.x, 1 - uv3.y);

```

---

Interestingly, we have had to reverse our  $v$  coordinate, this is because SFML's reading of texture coordinates is from the top left corner, rather than the bottom left corner. This is a common convention in computer graphics, and is something to be aware of.

#### 1.2.4 Drawing a point on the sphere

We decided that the user would be allowed to select a launch point (as this point would act as the starting point for our projectile). And the easiest way for the user was to select latitude and longitude points, as these are the most intuitive to the user.

The process from here is simply the inverse of the process described above.

Also note that in our model, latitude/longitude (0,0) is the point (0,0,-1)

We can derive these equations, by realising that since our sphere revolves around the  $y$  axis, only the latitude component will affect our final  $y$  coordinate. Since our sphere is also centered at the origin, we can

conclude that our  $y$  coordinate will be the sine of the latitude.

By the same logic, we can infer that the  $x$  and  $z$  coordinates will be the cosine of the latitude since the  $x$  and  $z$  coordinates are the projection of the latitude onto the  $xz$  plane.

The longitude will affect the  $x$  and  $z$  coordinates, since the longitude is the angle about the  $y$  axis. The  $x$  coordinate will be the cosine of the longitude, and the  $z$  coordinate will be the sine of the longitude.

Therefore the equations are:

$$\begin{aligned}
 y &= \sin(\text{latitude}) \\
 x &= \cos(\text{latitude}) \cos(\text{longitude}) \\
 z &= \cos(\text{latitude}) \sin(\text{longitude})
 \end{aligned}$$

#### 1.2.5 Computing and drawing the trajectory of the projectile

We gave the user the option to select these configuration items for the projectile:

- Latitude
- Longitude
- Launch velocity
- Launch angle (cardinal)
- Elevation angle

The latitude and longitude are easy to understand, and the launch velocity is the speed at which the projectile is launched. The launch angle is the angle at which the projectile is launched, with reference to the Westerly direction. The elevation angle is the angle at which the projectile is launched, with reference to the horizon.

To visualise these the last 3 parameters properly, suppose a local coordinate system, where the normal to the sphere is (by definition) orthogonal to the to the two

other basis vectors of the coordinate system. We can define the  $z'$  axis to be the normal to the sphere, and the  $x'$  axis to be the basis vector ‘facing’ the Westerly direction. The  $y'$  axis is then the basis vector facing the Northerly direction<sup>5</sup>.

We can create a local coordinate system by applying the cross product two times to the normal of the sphere. The implementation we used is a derivation of the one defined in [4].

---

```
<<Get local coordinate system>>=
void CoordinateSystem(const
    glm::vec3 &v1, glm::vec3 *v2,
    glm::vec3* v3)
{
    *v2 = glm::vec3(-v1.z, 0,
        v1.x) / std::sqrt(v1.x *
        v1.x + v1.z * v1.z);

    *v3 = glm::cross(v1, *v2);
}
```

---

From this assumption, we can define all possible directions where the projectile can be thrown as a hemisphere, with radius of the launch velocity. Further, we can define the launch angle to be the ‘longitude’ and the elevation angle to be the ‘latitude’ of this hemisphere.

From this we can use the formulation given in [5] to find the components of velocity vector with reference to the local coordinate system<sup>6</sup>:

$$\begin{aligned} v_x &= v \cos(\text{elevation}) \cos(\text{launch}) \\ v_y &= v \cos(\text{elevation}) \sin(\text{launch}) \\ v_z &= v \sin(\text{elevation}) \end{aligned}$$

Now that we know the velocity vector of the projectile, we can use verlet integra-

tion to find the position of the projectile at any given time. We can infer the direction in which gravity will act in, since it will into the center of the sphere. Since our sphere is centered at  $(0, 0, 0)$ , we know that the direction is simply the negative of the position vector of the projectile.

Next we can calculate the acceleration due to gravity, by using the formula:

$$a = \frac{GM}{r^2}$$

One inaccuracy of our model, is that our mass of Earth (or Mars or Moon)  $M$ , is scaled and not accurate to the real mass of the planet. Our scaled mass was calculated as:

$$M = g * r^2 / G$$

where  $g$  is the acceleration due to gravity on the surface of the planet,  $r$  is the radius of the planet (in our scaled version), and  $G$  is the gravitational constant.

We can then calculate the acceleration due to gravity as:

---

```
float distanceFromCenter =
    glm::distance(glm::vec3(0,
        0, 0), xyzPosition);
g = launchControlSettings.bigG *
    planetMass /
    (distanceFromCenter *
        distanceFromCenter);

acceleration = -g * difference;
```

---

This would integrate to the rest of the code as follows:

---

```
float g =
    launchControlSettings.bigG *
    planetMass /
```

---

<sup>5</sup>This is to say that  $x'$  and  $y'$  is a propotional representation of the  $x$  and  $y$  axis of the world space (since our sphere’s poles are through the  $y$  axis)

<sup>6</sup>Again, note that  $v_z$  is the cosine not the sine. This is because we want zero elevation to be the horizon, and not the zenith.



```

        (launchControlSettings.radius *
         launchControlSettings.radius);
        planetMass /
        (distanceFromCenter *
         distanceFromCenter);

glm::vec3 difference =
    glm::normalize(xyzPosition);
glm::vec3 acceleration = -g *
    difference;
    acceleration = -g * difference;

    points.push_back(xyzPosition);
    numPoints++;
}

```

```

int numPoints = 0;
int maxPoints = 1000;

float dt = 0.001f;
while (glm::distance(glm::vec3(0, 0,
    0), xyzPosition) >=
        launchControlSettings.radius
        &&
        numPoints < maxPoints) {
    // update our position
    xyzPosition += xyzVelocity * dt +
        0.5f * acceleration * dt * dt;

    // adjust our position based on
    // rotation
    xyzPosition = adjustforRotation(
        xyzPosition,
        launchControlSettings.angularVelocity,
        dt, numPoints);

    // update our velocity
    xyzVelocity += acceleration * dt;

    // update our acceleration
    difference =
        glm::normalize(xyzPosition);

    // Acceleration is calculated by
    // working out which component of
    // the velocity
    // will be affected the most of
    // immediate effect of gravity
    // by calculating the normaised
    // difference between the
    // position and the
    // center of the earth
    float distanceFromCenter =
        glm::distance(glm::vec3(0, 0,
        0), xyzPosition);
    g = launchControlSettings.bigG *

```

Note how we keep a track of the number of points, as we don't want to calculate the trajectory of the projectile indefinitely, causing memory and computation issues later on (plus SFML's draw calls for points more than 1000 points is not the most efficient.).

## 1.2.6 Accounting for the rotation of the Earth

We account for the rotation of the Earth, by shifting each point on the projectile by the same amount that the Earth has rotated, in the time taken for the projectile to be at that point. This is done by:

```

// This function calculates the
// current spherical coordinates
// of the projectile,
// And takes away some component with
// respect to the angular velocity
glm::vec3 adjustforRotation(glm::vec3
    currentPos, float angularVel,
    float dt, int pointIndex) {
    float length =
        glm::length(currentPos);
    currentPos =
        glm::normalize(currentPos);
    float theta =
        std::acos(-currentPos.y) -
        glm::pi<float>() / 2.f;
    float phi =
        std::atan2(-currentPos.z,
        currentPos.x);
    phi -= angularVel * dt * pointIndex;
    return

```

```

    getCartesian(glm::degrees(theta),
glm::degrees(phi), 1) * length;
}

```

---

It is good to note that we could have also simply moved the landing position of the projectile by the same amount the Earth had rotated (as described in [1]), as an alternative method to account for the rotation of the Earth.

### 1.2.7 Animating the projectile

The most trivial process of the entire algorithm is the animation of the projectile. This is done by calculating how many points in the projectile there are, with respect to a fixed time limit (e.g. 5 seconds) and then waiting for that many frames to pass before moving on to the next point.

---

```

float timePerPoint = 5.f /
    projectilePath.size();
if (animationClock.
getElapsedTime().asSeconds()
>= timePerPoint)
{
    currentAnimatedPoint++;
    if (currentAnimatedPoint
>= projectilePath.size()) {
        launchControlSettings.
isAnimated = false;
        currentAnimatedPoint = 0;
    }
    animationClock.restart();
}
for (int i = 0; i <
    currentAnimatedPoint; i++)
{
    glm::vec3 transformedPoint = mvp
        *
        glm::vec4(projectilePath[i],
            1.0f);

    transformedPoint.x =
        (transformedPoint.x + 1.0f)
        * 0.5f * RENDER_WIDTH;

```

```

transformedPoint.y =
    (transformedPoint.y + 1.0f)
    * 0.5f * RENDER_HEIGHT;

sf::CircleShape circle(2);
if (transformedPoint.z > 4.8) {
    circle.setFillColor(sf::Color::Red);
} else {
    circle.setFillColor(sf::Color::Magenta);
}
circle.setPosition(transformedPoint.x,
    transformedPoint.y);
window.draw(circle);
}

```

---

## 1.3 Results

Our model is a good approximation of the real world, and can be used to simulate the trajectory of a projectile on Earth, Mars, and the Moon. The model is not perfect, and there are some inaccuracies, such as the mass of the planets, and the fact our model does not account for the Earth's true shape, nor its atmosphere. One big problem with our model, is that there seems to be artefacting of the texture on the sphere. We believe this to be an issue with SFML's texture interpolation algorithm, and we are not sure how to fix this. We have tried to increase the resolution of the sphere, but this has not fixed the issue.

## References

- [1] Department of Applied Mathematics and Theoretical Physics. “Deflection of a Projectile due to the Earth’s Rotation”. In: (). URL: <https://www.damtp.cam.ac.uk/user/reh10/lectures/ia-dyn-handout14.pdf>.
- [2] *Essence of Linear Algebra*. <http://www.youtube.com/playlist?list=PLZHqObOWTQDDbWizkM2WtmgF8hE-ab>. (Visited on 08/09/2024).
- [3] Jos Stam. “Evaluation of Loop Subdivision Surfaces”. In: 2010. URL: <https://api.semanticscholar.org/CorpusID:8420692>.
- [4] *Vectors*. [https://pbr-book.org/3ed-2018/Geometry\\_and\\_Transformations/Vectors#Coordina](https://pbr-book.org/3ed-2018/Geometry_and_Transformations/Vectors#Coordina). (Visited on 08/12/2024).
- [5] Eric W. Weisstein. *Spherical Coordinates*. <https://mathworld.wolfram.com/SphericalCoordinates.html>. Text. (Visited on 08/12/2024).