CS380: Introduction to Computer Graphics

Lab Session 3

HYUNJIN KIM

Spring 2023
KAIST

Goal

- **Implement Rigid Body transformation Functions.**
- **Implement arcball Interface.**
- **Fix up the translation.**

Assignment 3

The base of the assignment 3 is the assignment 2.

- **You have to copy files in the assignment 3 into the assignment 2 directory.**

Assignment 3

There are 3 new **header** files in the assignment 3 zip file.

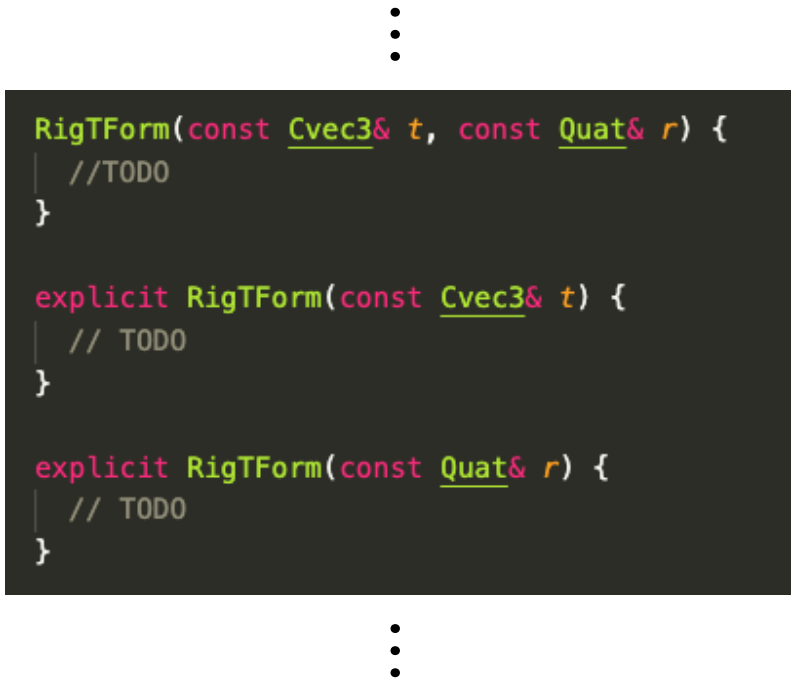
- **rigtform.h** includes new class *'RigTForm'*, which is used to represent rigid body transformation and replace *Matrix4* class.

- **quat.h** includes new class *'Quat'* (quaternion) and its operations.

- **arcball.h** includes some useful functions of arcball.

Task1 – RBT Functions

In Task1, you have to fill '*TODO*' parts in the ***rigtform.h*** file.



```
RigTForm(const Cvec3& t, const Quat& r) {
  //TODO
}

explicit RigTForm(const Cvec3& t) {
  // TODO
}

explicit RigTForm(const Quat& r) {
  // TODO
}
```

Task1 – RBT Functions

You have to implement Rigid Body Transformation (RBT) functions in ***rigtform.h.***

RigTForm class definition:

```
class RigTForm {
  Cvec3 t_; // translation component
  Quat r_;  // rotation component represented as a quaternion
```

Full affine matrix A = Translation T × Linear L

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & h \\ 0 & 0 & 1 & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b & c & 0 \\ e & f & g & 0 \\ i & j & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A = TL$$

RigTForm represents affine matrix A = TL

Task1 – RBT Functions

We can simply rewrite affine matrix A as

$$\underbrace{\begin{bmatrix} r & t \\ 0 & 1 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} i & t \\ 0 & 1 \end{bmatrix}}_{T} \underbrace{\begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix}}_{L}$$

```
class RigTForm {
  Cvec3 t_; // translation component
  Quat r_;  // rotation component represented as a quaternion
```

where t directly becomes t_, and r becomes the matrix conversion of the quaternion r_.

Task1 – RBT Functions

The definition of ***Quat*** class is defined in ***quat.h***.

```
class RigTForm {
  Cvec3 t_; // translation component
  Quat r_;  // rotation component represented as a quaternion
```

$$q = xi + yj + zk + w$$

```
class Quat {
  Cvec4 q_;  // layout is: q_[0]==w, q_[1]==x, q_[2]==y, q_[3]==z

public:
  double operator [] (const int i) const {
    return q_[i];
  }

  double& operator [] (const int i) {
    return q_[i];
  }

  double operator () (const int i) const {
    return q_[i];
  }

  double& operator () (const int i) {
    return q_[i];
  }
```

⋮

Recap: Quaternion

For a given unit quaternion $q = xi + yj + zk + w$ $(x^2 + y^2 + z^2 + w^2 = 1)$, the rotation axis $\vec{u}$ and angle θ can be computed as:

$$\vec{u} = (x, y, z),$$

$$\cos\frac{\theta}{2} = w, \sin\frac{\theta}{2} = \sqrt{x^2 + y^2 + z^2},$$

$$\theta = 2\cos^{-1} w = 2\sin^{-1}\sqrt{x^2 + y^2 + z^2}.$$

Recap: Quaternion

Rotation matrix $\mathrm{R}(q)$ associated with the given unit quaternion

$q = xi + yj + zk + w$ $(x^2 + y^2 + z^2 + w^2 = 1)$ is computed as:

$$R(q) = \begin{bmatrix} 2(w^2 + x^2) - 1 & 2(xy - wz) & 2(xz + wy) \\ 2(xy + wz) & 2(w^2 + y^2) - 1 & 2(yz - wx) \\ 2(xz - wy) & 2(yz + wx) & 2(w^2 + z^2) - 1 \end{bmatrix}.$$

Recap: Quaternion

For given quaternions p and q,

$$R(pq) = R(p)R(q),$$

$$R(q)^{-1} = R(q^{-1}).$$

Recap: Quaternion

In LAB3, we use the below quaternion representations.

$$q = \begin{bmatrix} a \\ \vec{b} \end{bmatrix} = a + b_1 i + b_2 j + b_3 k,$$

$$q_1 q_2 = \begin{bmatrix} a_1 \\ \overrightarrow{b_2} \end{bmatrix} \begin{bmatrix} a_2 \\ \overrightarrow{b_2} \end{bmatrix}.$$

Task1 – RBT Functions

Also, the functions of the Quat class are defined in ***quat.h.*** Use them to implement '*TODO*' parts in ***rigtform.h***.

⋮

```
inline double dot(const Quat& q, const Quat& p) {
  double s = 0.0;
  for (int i = 0; i < 4; ++i) {
    s += q(i) * p(i);
  }
  return s;
}

inline double norm2(const Quat& q) {
  return dot(q, q);
}

inline Quat inv(const Quat& q) {
  const double n = norm2(q);
  assert(n > CS380_EPS2);
  return Quat(q(0), -q(1), -q(2), -q(3)) * (1.0/n);
}

inline Quat normalize(const Quat& q) {
  return q / std::sqrt(norm2(q));
}
```

⋮

Task1 – RBT Functions (Hint)

The multiplication between two affine matrices can be computed as below.

$$\begin{bmatrix} i & t_1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i & t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} i & t_1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1 & r_1 t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} i & t_1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i & r_1 t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_2 & 0 \\ 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} i & t_1 + r_1 t_2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r_1 r_2 & 0 \\ 0 & 1 \end{bmatrix}$$

Task1 – RBT Functions (Hint)

The inversion of the affine matrix can be computed as below.

$$\left(\begin{bmatrix} i & t \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix}\right)^{-1} =$$

$$\begin{bmatrix} r & 0 \\ 0 & 1 \end{bmatrix}^{-1}\begin{bmatrix} i & t \\ 0 & 1 \end{bmatrix}^{-1} =$$

$$\begin{bmatrix} r^{-1} & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i & -t \\ 0 & 1 \end{bmatrix} =$$

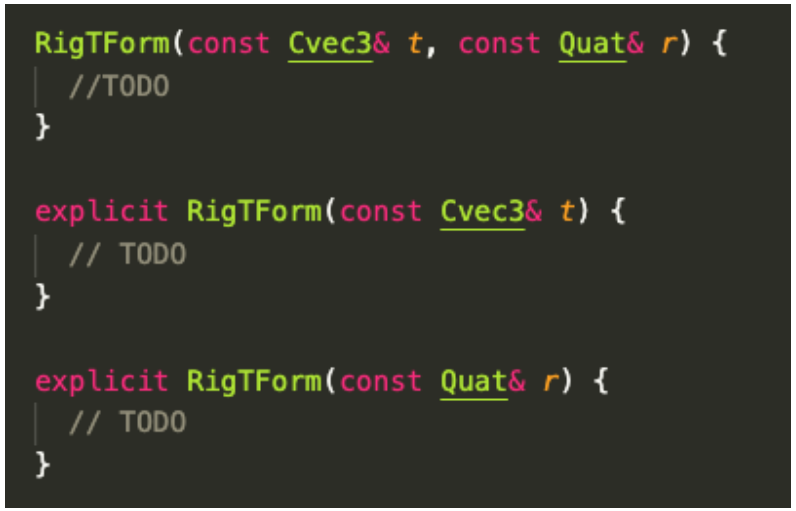
$$\begin{bmatrix} r^{-1} & -r^{-1}t \\ 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} i & -r^{-1}t \\ 0 & 1 \end{bmatrix}\begin{bmatrix} r^{-1} & 0 \\ 0 & 1 \end{bmatrix}$$

Task1 – RBT Functions

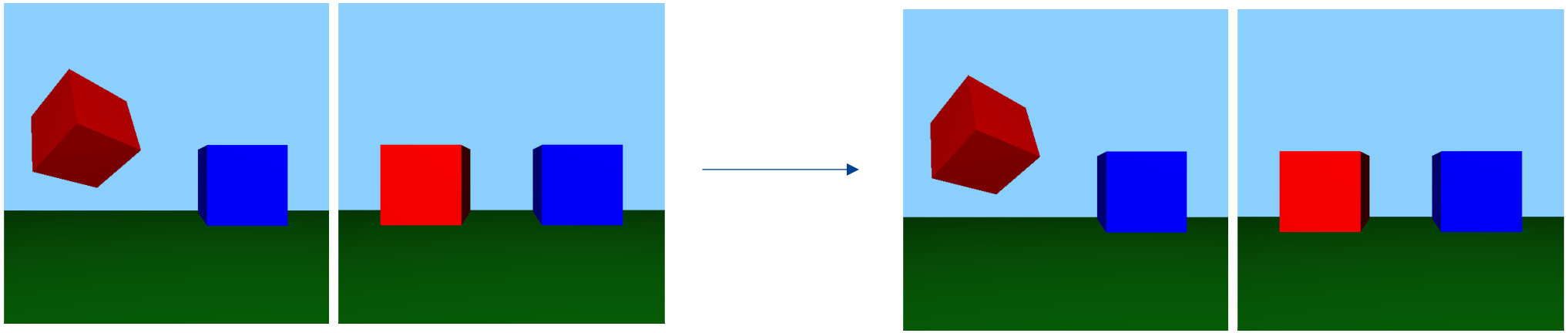
After you fill all '*TODO*' parts in ***rigtform.h***, replace *Matrix4* class with RigTForm class.

You have to replace all rigid body transformation matrices with RigTForm (except projection matrix, MVM, NMVM).



Task1 – RBT Functions

Nothing should be changed in the output when you replace *Matrix4* class with *RigTForm* class.

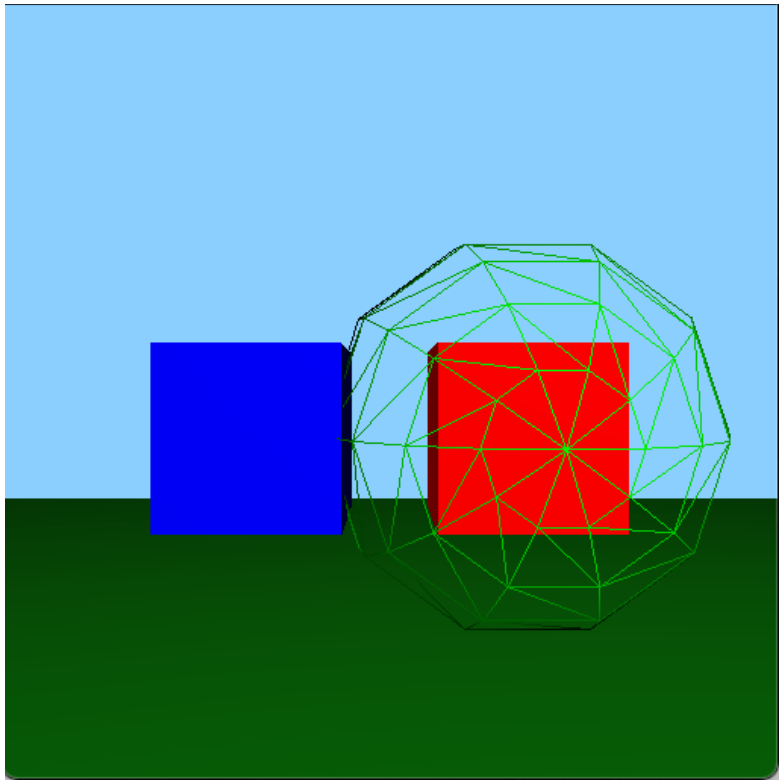


Before replacement

After replacement

Task2 - Arcball

In Task2, you should draw a wireframe sphere called an arcball in a scene, and you will rotate or translate the object with respect to it.



Task2 – Arcball

Task2 is divided into 3 subtasks.

- Drawing a wireframe sphere.
- Controlling the appearance and the position of the arcball.
- Performing transformation with respect to the arcball interface.

Task2 – Arcball

There are 3 helper functions in ***arcball.h*** that will help you implement the arcball interface.

```
inline Cvec2 getScreenSpaceCoord(const Cvec3& p,
                                 const Matrix4& projection,
                                 double frustNear, double frustFovY,
                                 int screenWidth, int screenHeight) {
```

getScreenSpaceCoord() : This takes the 3D position and projection matrix as input and returns the screen coordinates. (Since we change the code, this function will not be used.)

```
inline double getScreenToEyeScale(double z, double frustFovY, int
screenHeight) {
```

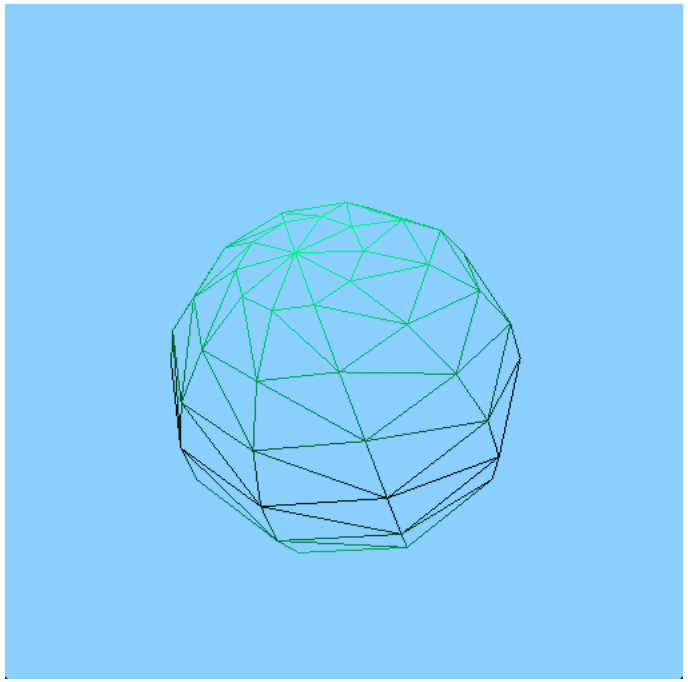
getScreenToEyeScale() : This takes the z value (let z_) of the eyeframe and returns the ratio between screen coordinates (z = 1 plane) and eyeframe (z = z_ plane).

```
inline Cvec3 getModelViewRay(const Cvec2& p, double frustFovY,
                             int screenWidth, int screenHeight) {
```

getModelViewRay() : This takes a point on the screen and returns the ray direction from the camera to the input point.

Task2 – Arcball (Drawing a Sphere)

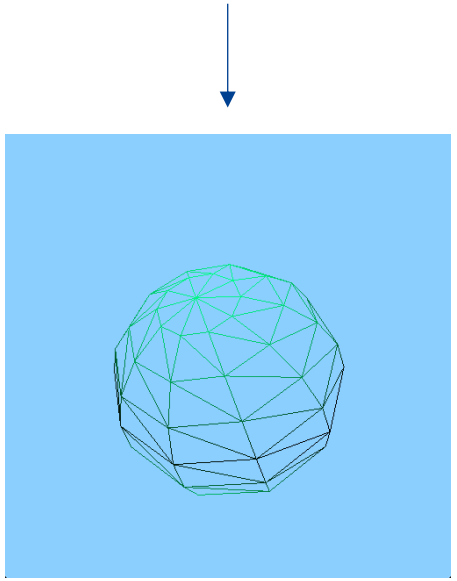
An arcball is a sphere used as an interface of rotation and translation.



Task2 – Arcball (Drawing a Sphere)

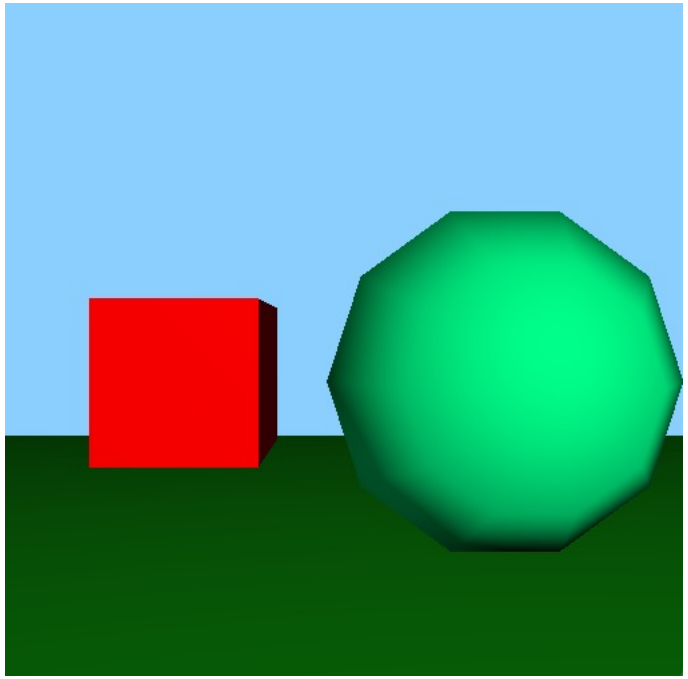
You can use *makeSphere()* function in ***geometrymaker.h*** to draw a sphere.

```
void makeSphere(float radius, int slices, int stacks, VtxOutIter vtxIter, IdxOutIter idxIter) {
```

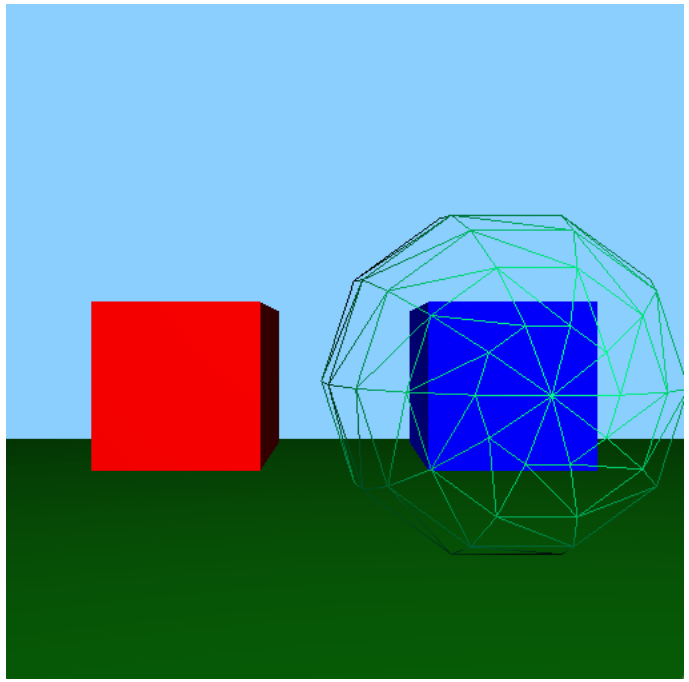


Task2 – Arcball (Drawing a Sphere)

You have to draw not a solid but a wireframe arcball.



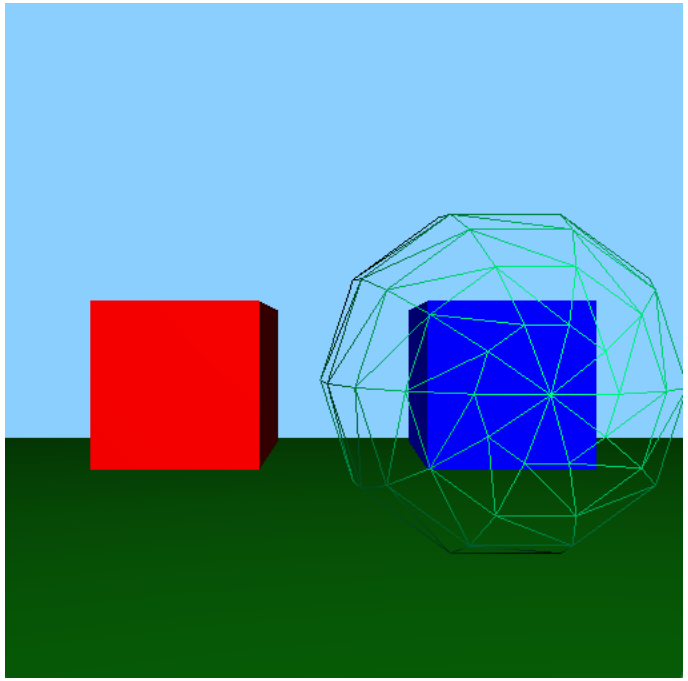
Wrong



Correct

Task2 – Arcball (draw sphere)

If you change *glPolygonMode* to *GL_LINE*, then you can draw geometry with a wireframe.



```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

Switch to wireframe mode.

```
ArcballGeometry->draw(curSS);
```

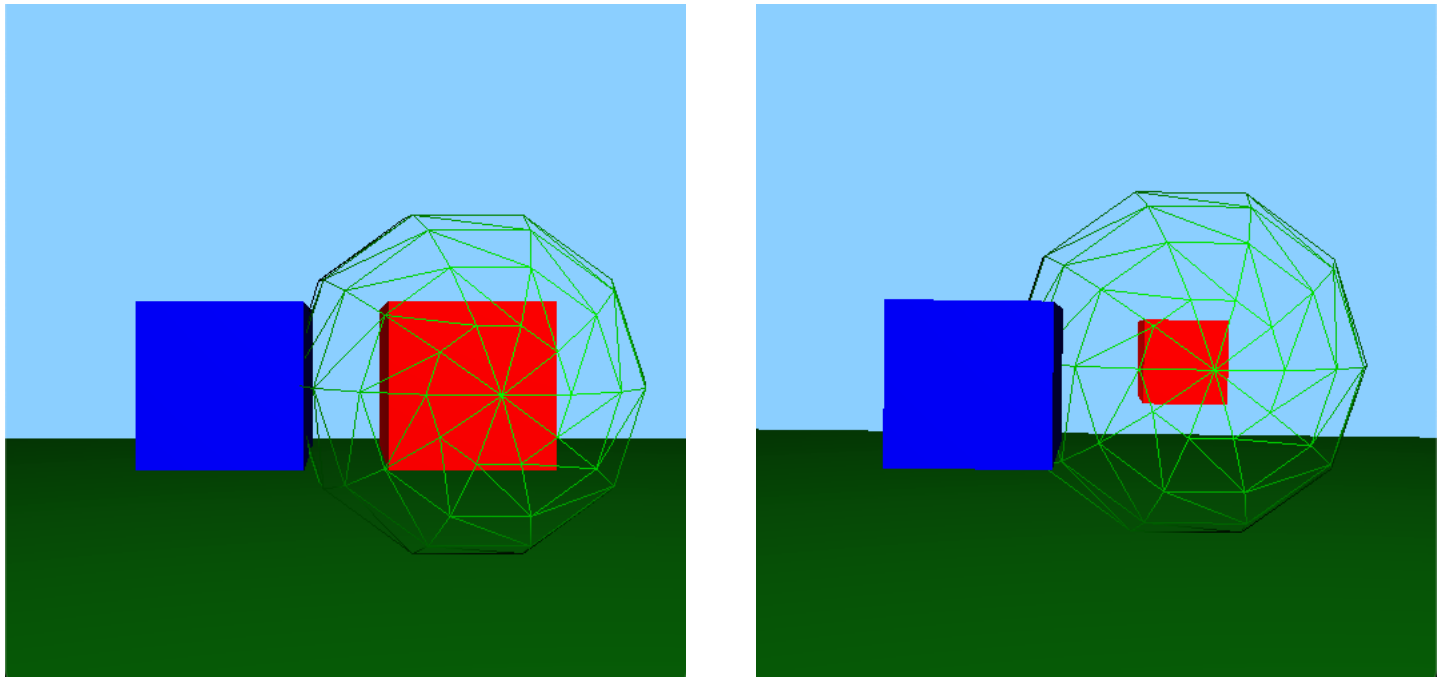
Draw in the scene.

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

Switch back to solid mode.

Task2 – Arcball (Appearance / Position)

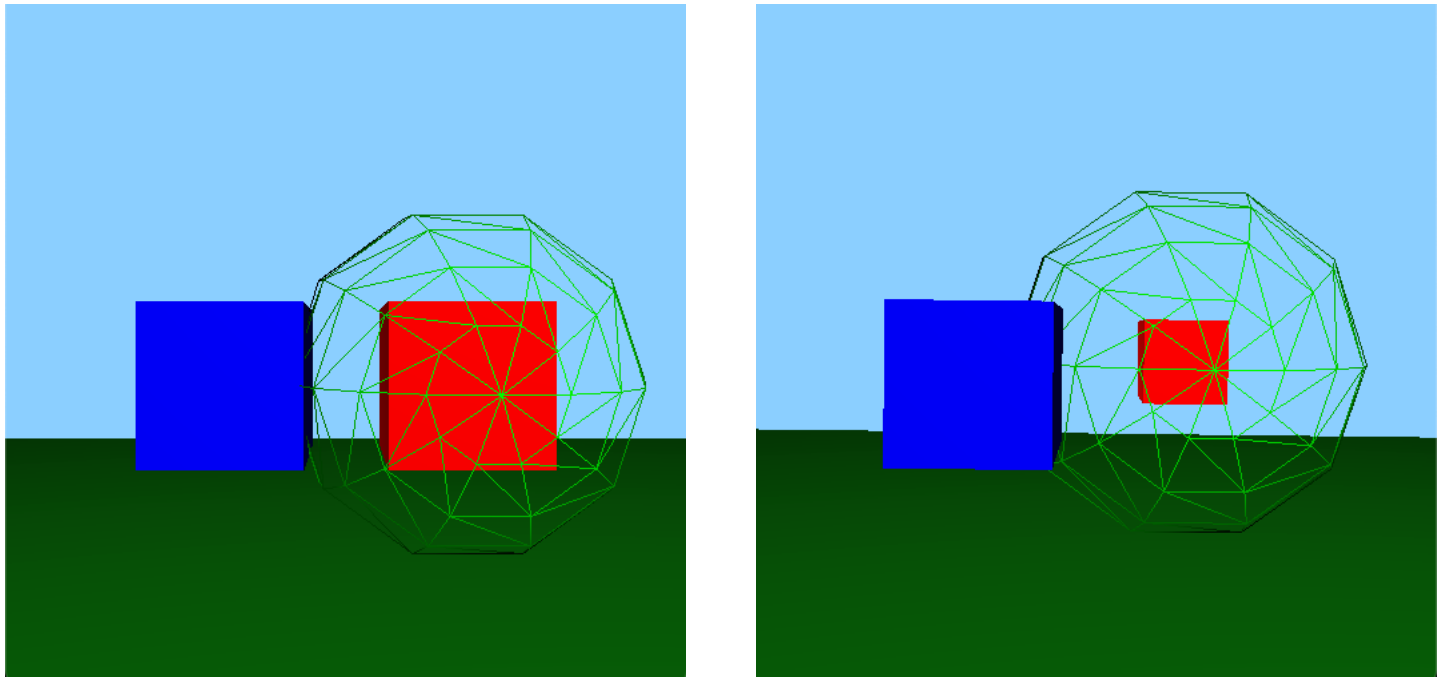
The radius of the arcball should be dependent to the window size.



The arcball radius is not dependent to the arcball position in the 3D world, but the window size.

Task2 – Arcball (Appearance / Position)

The screen radius of arcball should be 0.25*min(g_windowWidth, g_windowHeight).



The arcball radius is not dependent to the arcball position in the 3D world, but the window size.

Task2 – Arcball (Appearance / Position)

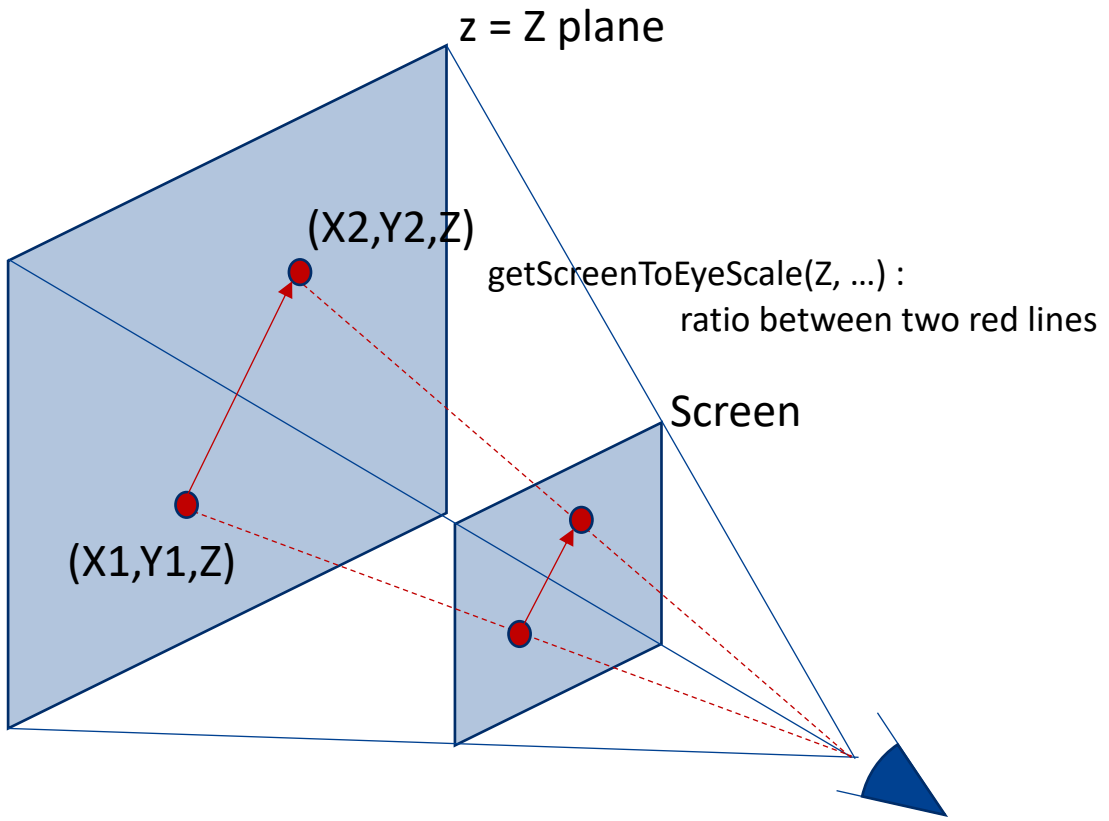
In *arcball.h*, there is a function *getScreenToEyeScale* that returns the ratio between screen coordinates and the eyeframe.

```
// Return the scale between 1 unit in screen pixels and 1 unit in the eye-frame
// (or world-frame, since we always use rigid transformations to represent one
// frame with resepec to another frame)
//
// Ideally you should never call this using a z behind the Z=0 plane,
// sinch such a point wouldn't be visible.
//
// But if you do pass in a point behind Z=0 plane, we'll just
// print a warning, and return 1
inline double getScreenToEyeScale(double z, double frustFovY, int screenHeight) {
  if (z > -CS380_EPS) {
    std::cerr << "WARNING: getScreenToEyeScale on z near or behind Z=0 plane. Returning 1 instead." << std::endl;
    return 1;
  }
  return -(z * tan(frustFovY * CS380_PI/360.0)) * 2 / screenHeight;
}

#endif
```

Task2 – Arcball (Appearance / Position)

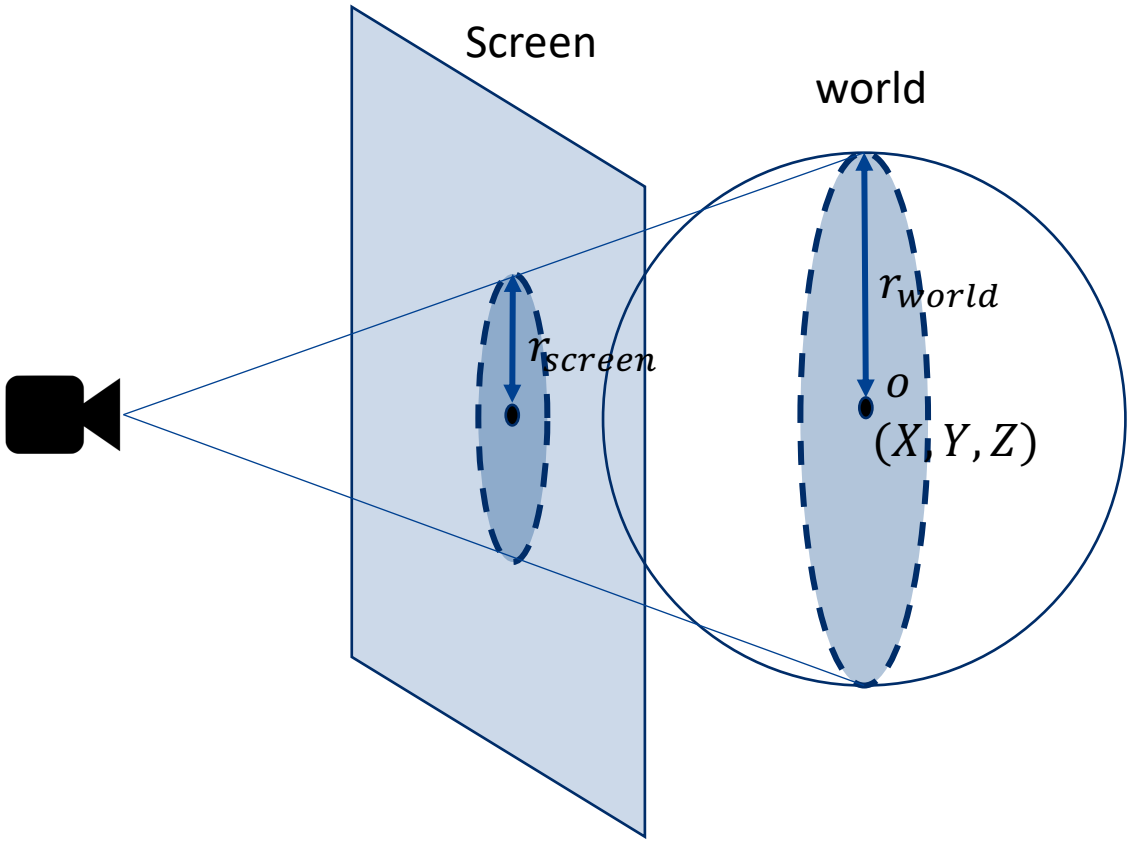
If you input Z to the *getScreenToEyeScale* function, you will get the ratio between the screen and the z = Z plane.

```
// Return the scale between 1 unit in screen pixels and 1 unit in the eye-frame
// (or world-frame, since we always use rigid transformations to represent one
// frame with resepec to another frame)
//
// Ideally you should never call this using a z behind the Z=0 plane,
// sinch such a point wouldn't be visible.
//
// But if you do pass in a point behind Z=0 plane, we'll just
// print a warning, and return 1
inline double getScreenToEyeScale(double z, double frustFovY, int screenHeight) {
  if (z > -CS380_EPS) {
    std::cerr << "WARNING: getScreenToEyeScale on z near or behind Z=0 plane. Returning 1 instead." << std::endl;
    return 1;
  }
  return -(z * tan(frustFovY * CS380_PI/360.0)) * 2 / screenHeight;
}

#endif
```



Task2 – Arcball (Appearance / Position)

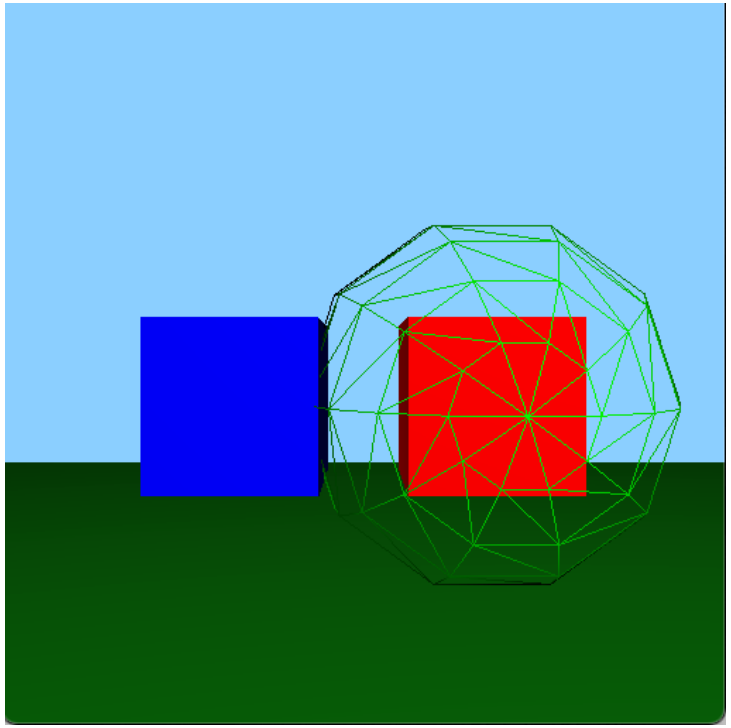
If you know the screen radius of arcball, you can directly get arcball radius in 3D world using the *getScreenToEyeScale* function.



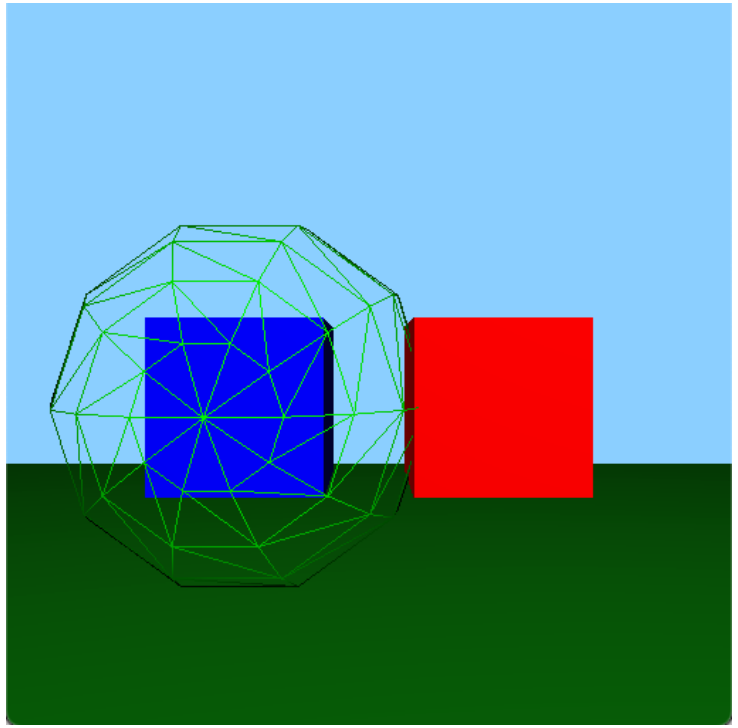
$$Ratio = getScreenToEyeScale(Z, \dots)$$
$$worldRadius = screenRadius * Ratio$$

Task2 – Arcball (Appearance / Position)

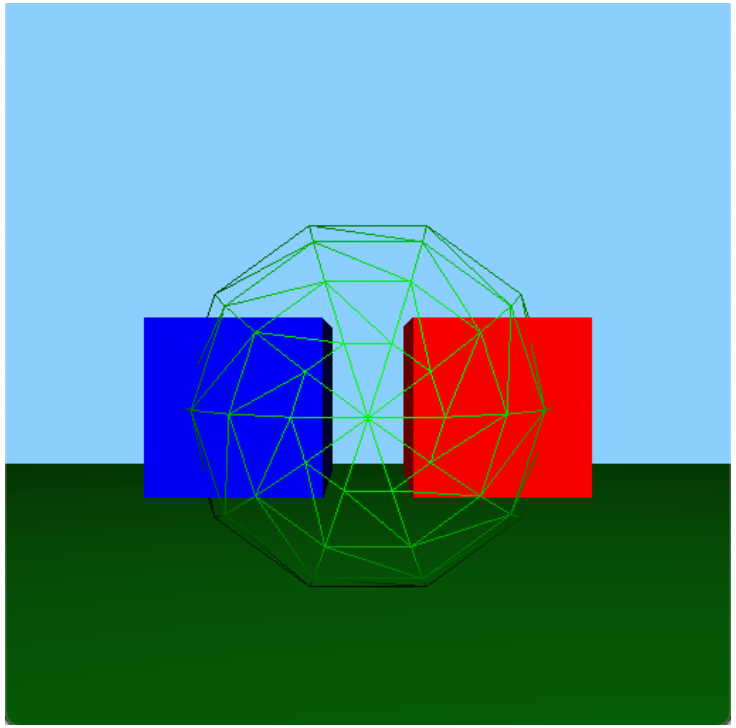
The arcball should be centered on the frame of the object you are manipulating.



Manipulating Red Cube



Manipulating Blue Cube

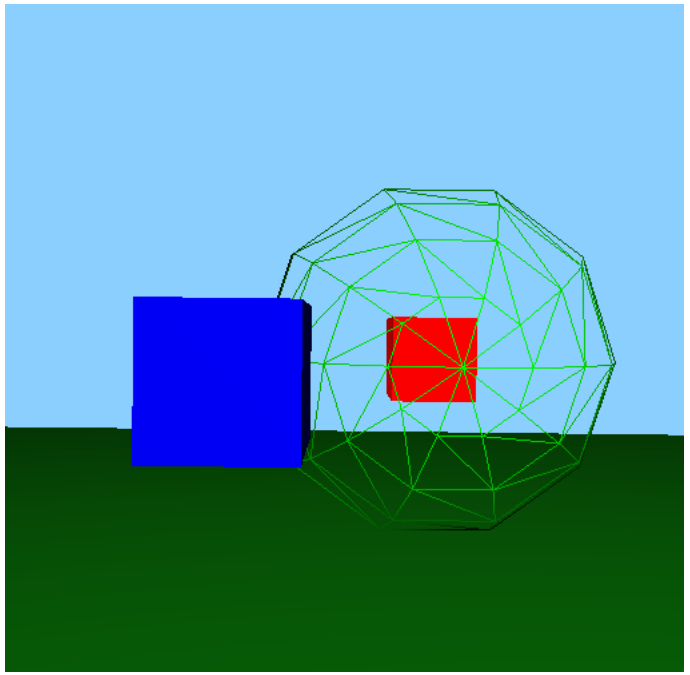


Manipulating Sky camera w.r.t world-sky coordinate

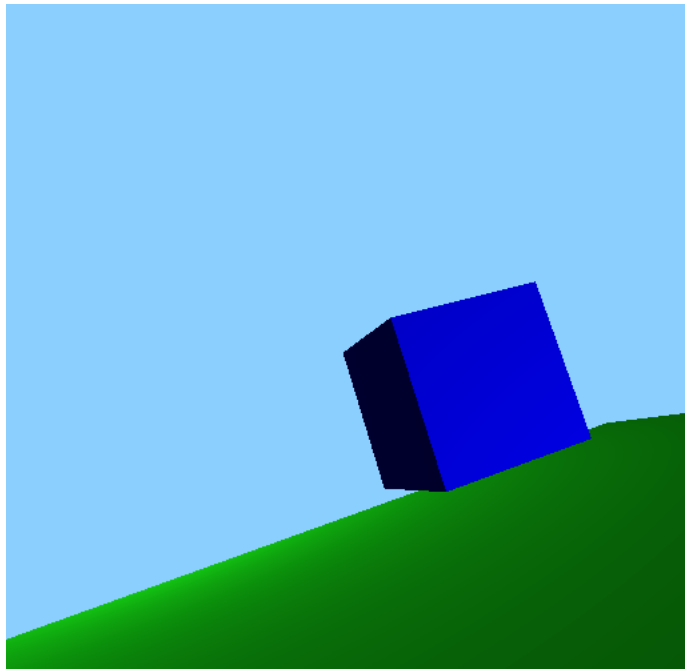
Task2 – Arcball (Appearance / Position)

Arcball should appear if

1) You are manipulating a cube, but manipulated cube should not be the current eyeframe.



Manipulating red cube (eyeframe : sky camera)



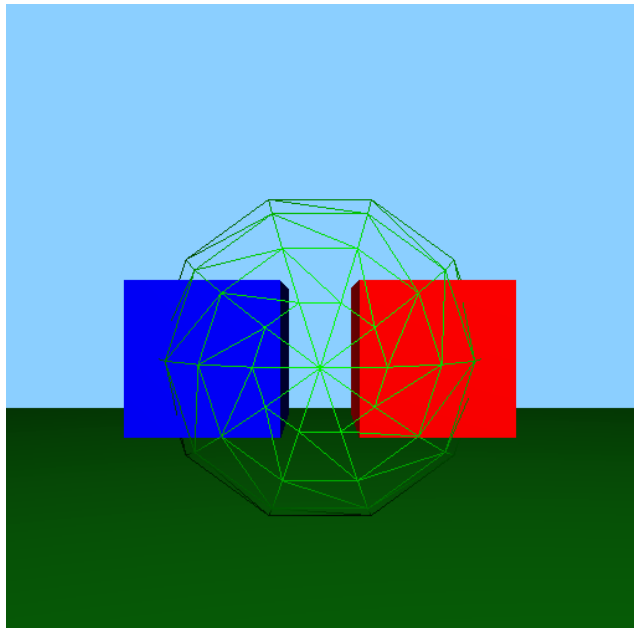
Manipulating red cube (eyeframe : red cube)

Task2 – Arcball (Appearance / Position)

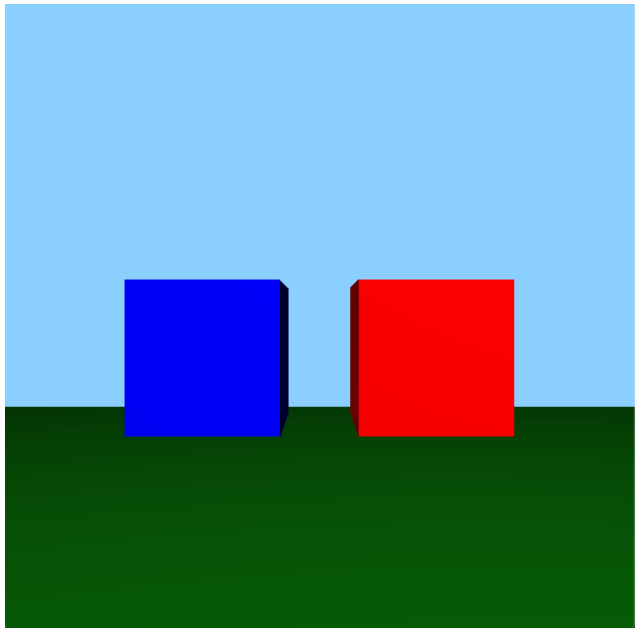
Arcball should appear if

1) You are manipulating a cube, but manipulated cube should not be the current eyeframe.
2) You are manipulating the sky camera w.r.t the world-sky coordinate.

In case (2), arcball center should be the world's origin.



Manipulating syk camera w.r.t world-sky coordinate



Manipulating syk camera w.r.t sky-sky coordinate

Task2 – Arcball (Appearance / Position)

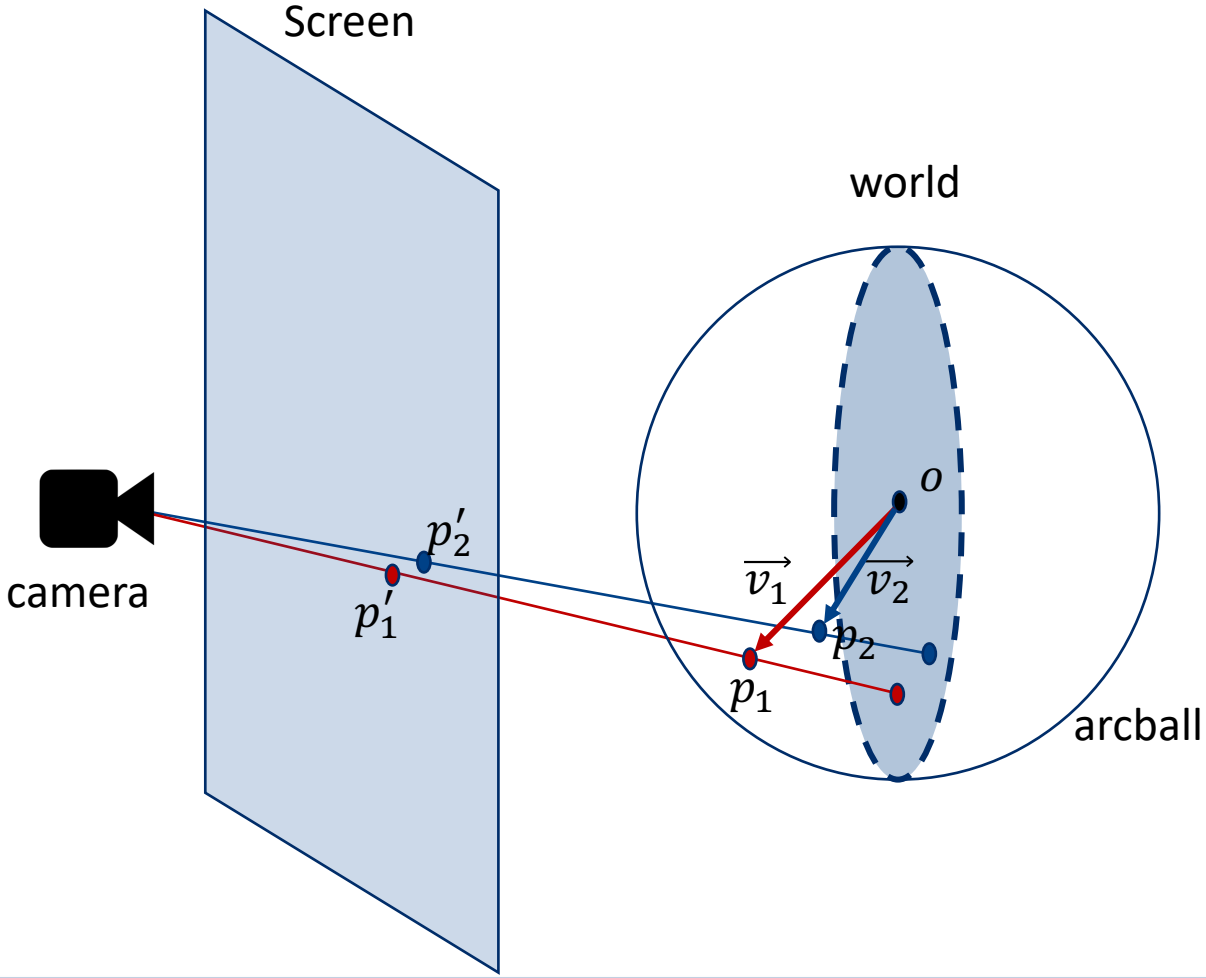
Arcball should appear if

1) You are manipulating a cube, but manipulated cube should not be the current eyeframe.
2) You are manipulating the sky camera w.r.t the world-sky coordinate.
3) Otherwise, arcball should not appear.

In case (2), arcball center should be the world's origin.

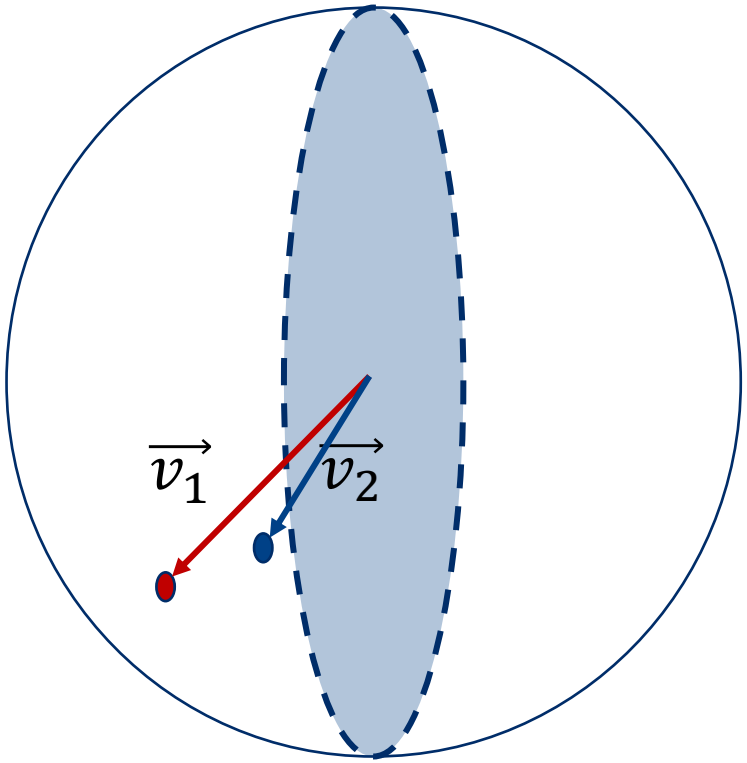
Task2 – Arcball (Transform)

Next, you have to implement transformation w.r.t the arcball interface.



Let's think about rotation aligning $\overrightarrow{v_1}$ to $\overrightarrow{v_2}$.

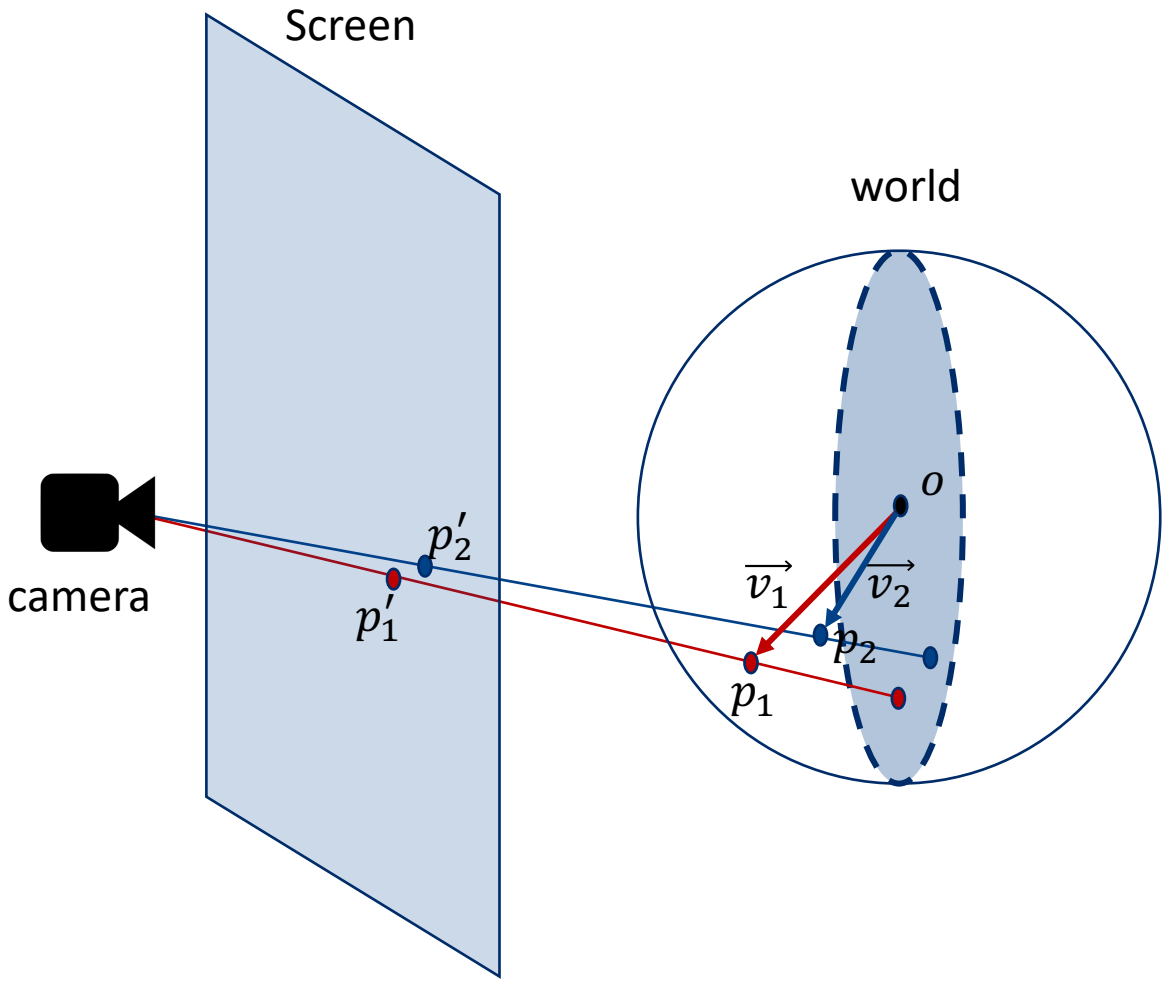
Task2 – Arcball (Transform)



If we know $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$, then we can compute rotation by using them.

Then how can we compute $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$?

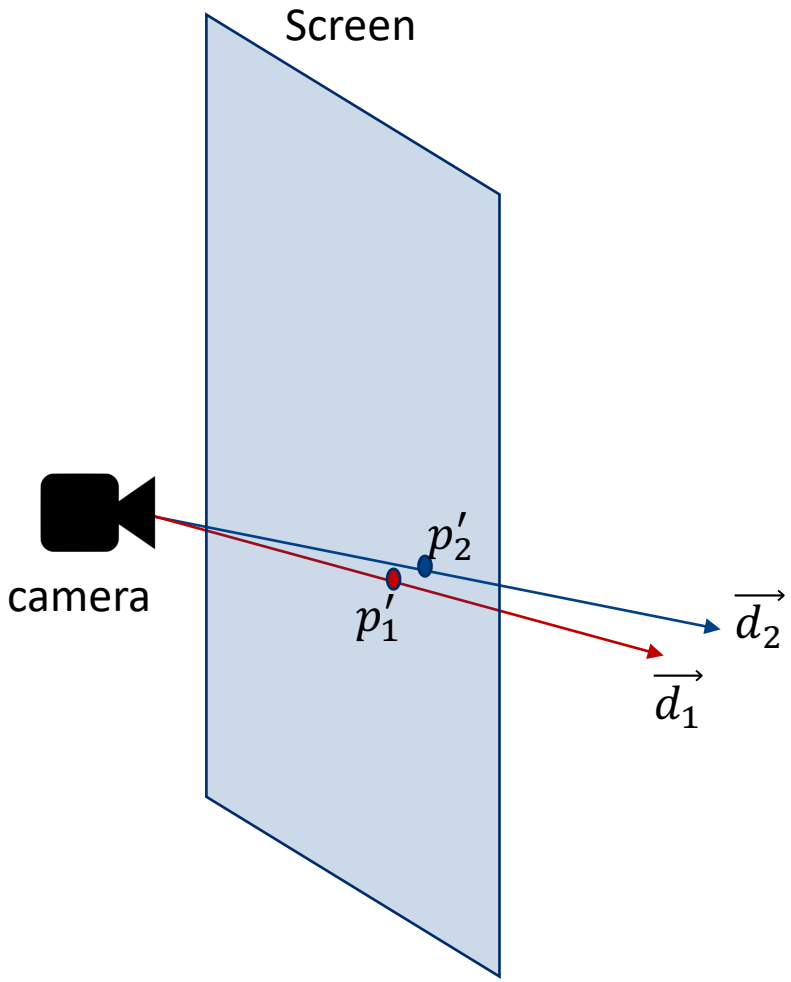
Task2 – Arcball (Transform)



Let p_1' and p_2' be the clicked points.

p_1' and p_2' correspond to points p_1 and p_2 (respectively) which are the intersection between rays from the camera passing p_1' and p_2' (respectively) and the arcball.

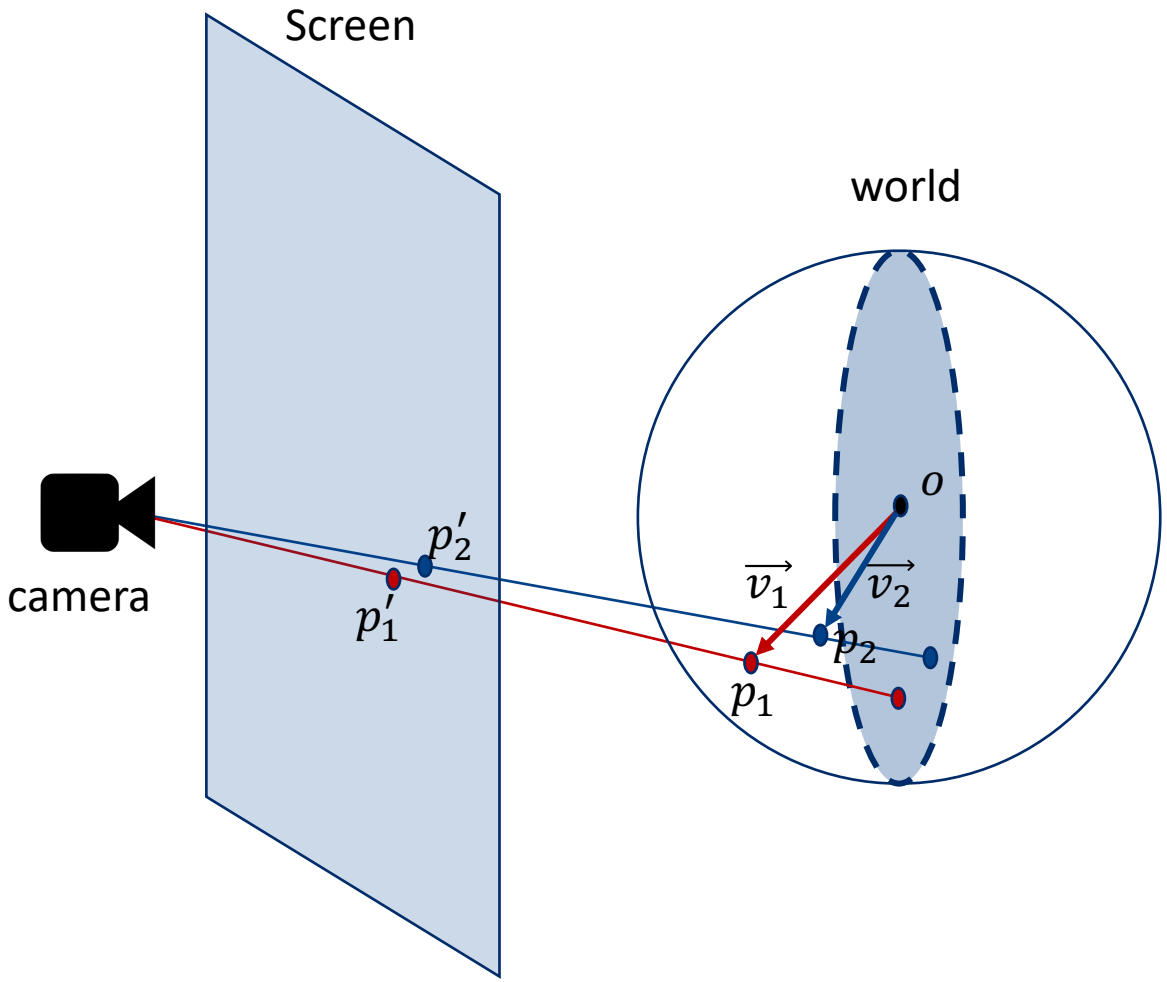
Task2 – Arcball (Transform)




```
inline Cvec3 getModelViewRay(const Cvec2& p, double frustFovY,
                             int screenWidth, int screenHeight) {
  const double aspectRatio = screenWidth / static_cast <double> (screenHeight);
  const double ang = frustFovY * 0.5 * CS380_PI/180;
  const double f = std::abs(std::sin(ang)) < CS380_EPS ? 0 : 1/std::tan(ang);

  const double clipcoord_x = (p[0] - (screenWidth - 1)/2.0) * 2.0 / screenWidth;
  const double clipcoord_y = (p[1] - (screenHeight - 1)/2.0) * 2.0 / screenHeight;

  Cvec3 d(-clipcoord_x * aspectRatio / f, -clipcoord_y / f, 1.0);
  d.normalize();
  return d;
}
```

You can get the normalized ray direction using the *getModelViewRay* function defined in ***arcball.h***.

Task2 – Arcball (Transform)



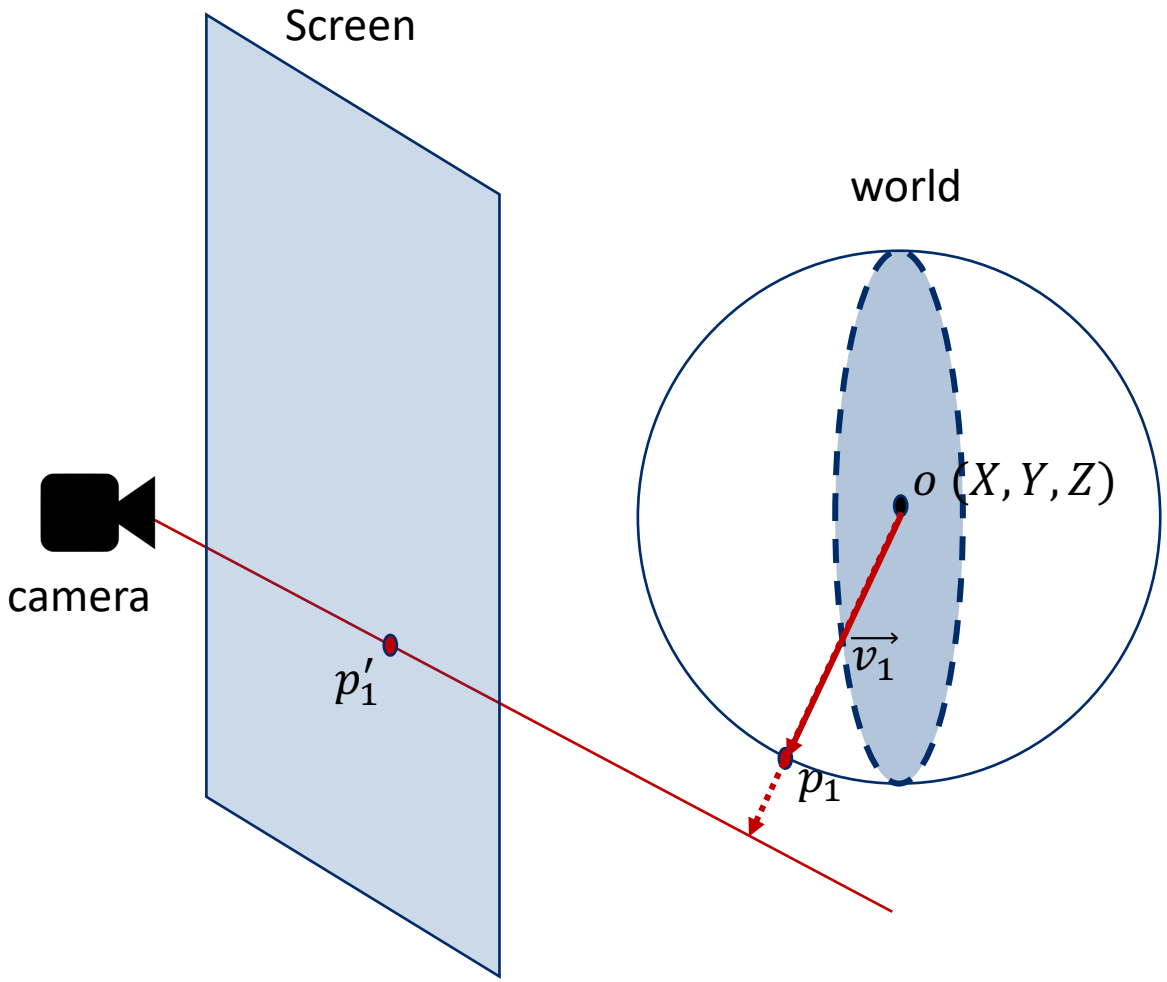
Since you have ray direction and know the sphere (arcball) center and radius, you can compute intersection points p_1 and p_2. Thus, you can get $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$.

Hint:

Ray equation : $t\vec{d} = (\mathrm{td_x}, \mathrm{td_y}, \mathrm{td_z})$

Sphere equation : $(x - c_x)^2 + \left(y - c_y\right)^2 + (z - c_z)^2 = r^2$

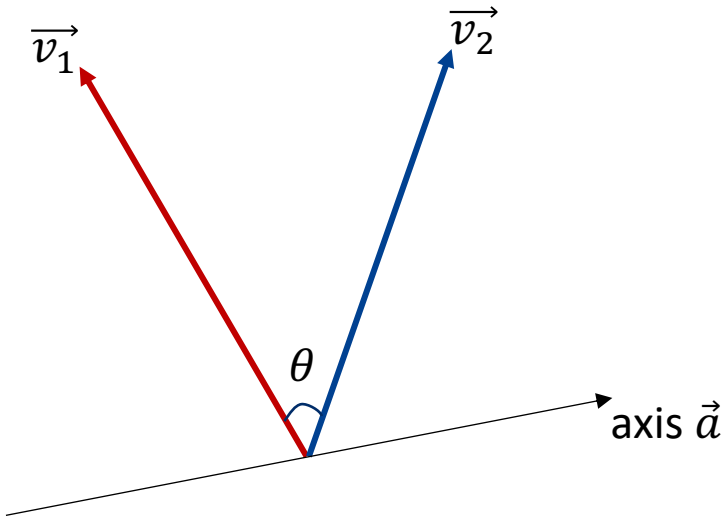
Task2 – Arcball (Transform)



If there is no intersection between the ray and the sphere, choose the closest point on the sphere.

Hint:
The equation $at^2 + bt + c = 0$ has real roots if $b^2 - 4ac \geq 0$

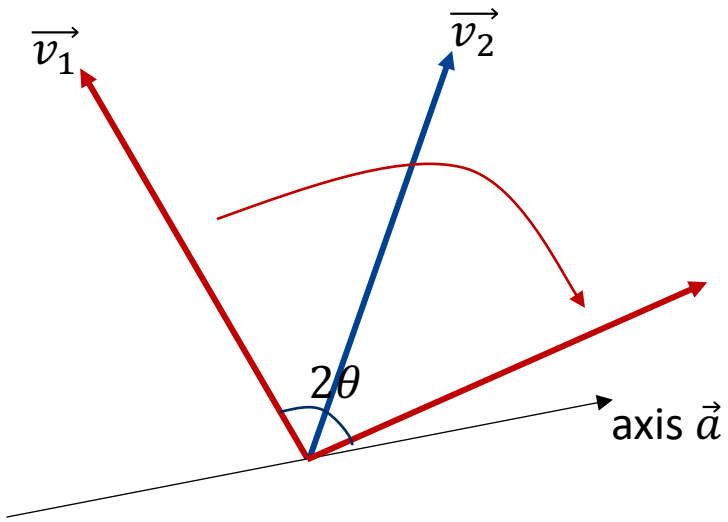
Task2 – Arcball (Transform)



After you get $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$, you can compute the rotation whose axis is $\vec{a}$ (= normalize($\overrightarrow{v_1}\times\overrightarrow{v_2}$)) and angle is θ (= $\arccos\left(\frac{\overrightarrow{v_1}\cdot\overrightarrow{v_2}}{\|\overrightarrow{v_1}\|\|\overrightarrow{v_2}\|}\right)$).

Task2 – Arcball (Transform)

However, in the arcball rotation, you have to rotate $\overrightarrow{v_1}$ not θ but 2θ. (The axis is still $\vec{a}$.)



We can represent rotation as a quaternion. The quaternion of rotation with angle 2θ and axis $\vec{a}$ is written as:

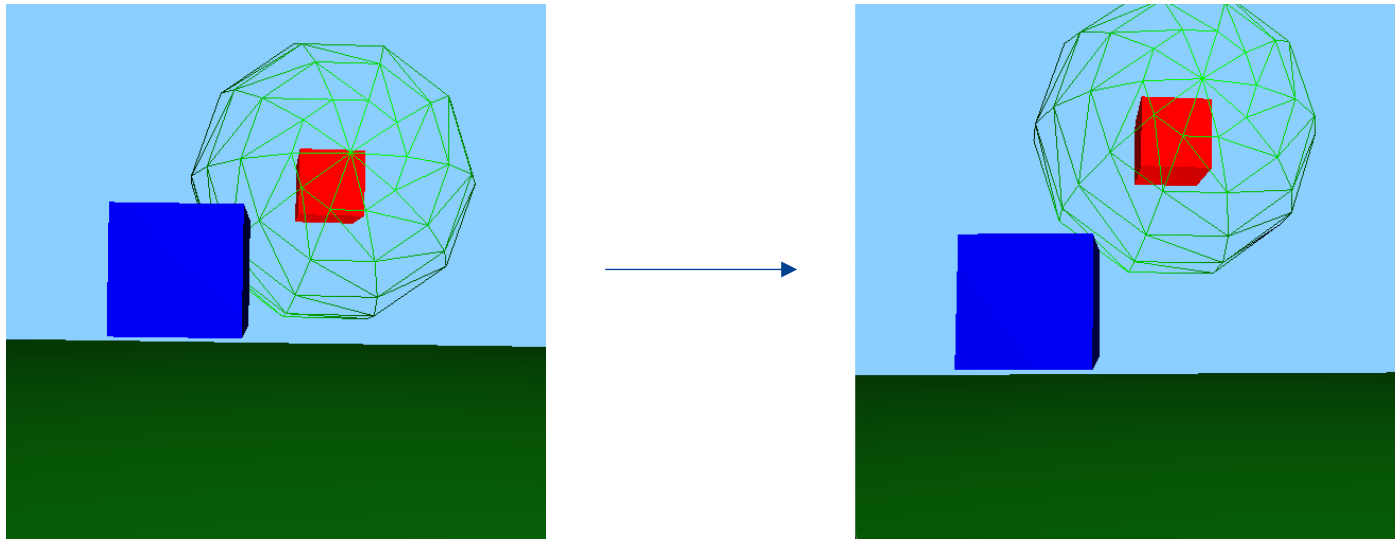
$$q = \begin{bmatrix} \cos\theta \\ \sin\theta\,\vec{a} \end{bmatrix} = \begin{bmatrix} \overrightarrow{v_1} \cdot \overrightarrow{v_2} \\ \overrightarrow{v_1} \times \overrightarrow{v_2} \end{bmatrix} = \begin{bmatrix} 0 \\ \overrightarrow{v_2} \end{bmatrix} \begin{bmatrix} 0 \\ -\overrightarrow{v_1} \end{bmatrix}$$

where $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ are normalized.

Task3 – Translation Fix Up

The last task is to fix the translation proportion to the arcball size.

If you translate the object, it should move w.r.t arcball size.

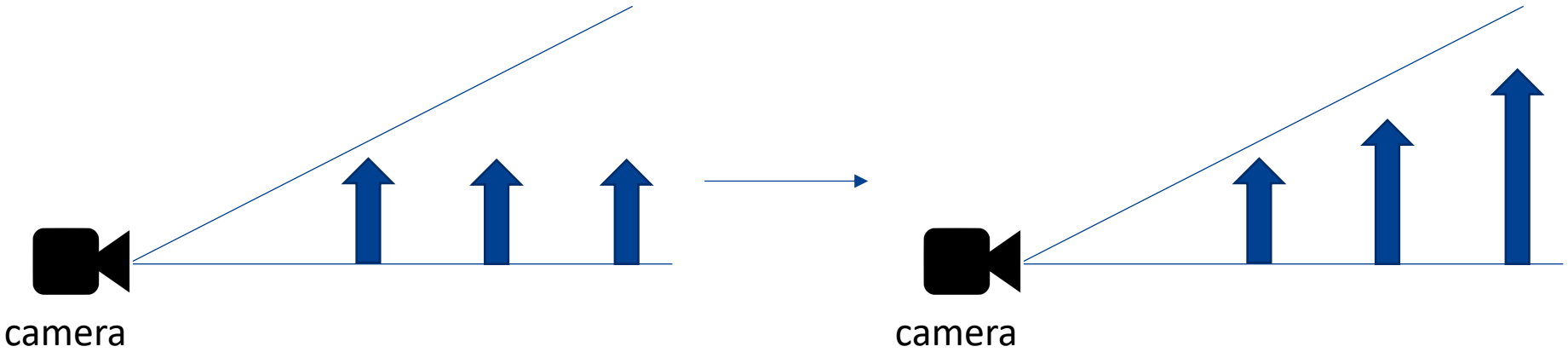


Before fix up

After fix up

Task3 – Translation Fix Up

You can simply fix up by replacing the translation factor 0.01 to arcball scale.



Before fix up

After fix up

Evaluation

- Is the RigTForm class functions implemented correctly?
- Does the arcball appear appropriately?
- Is the size of the arcball defined correctly?
- Does the rotation work well w.r.t the arcball interface?
- Is the translation fixed up?

How to Submit

Please zip all code files and submit the zip file to Gradescope.

The due date is 4/9 (SUN) 11:59 pm KST.