KAIST

CS 380

Introduction to Computer Graphics

LAB (4)

2018.04.02

VISUAL COMPUTING Lab

Review Hello 3D

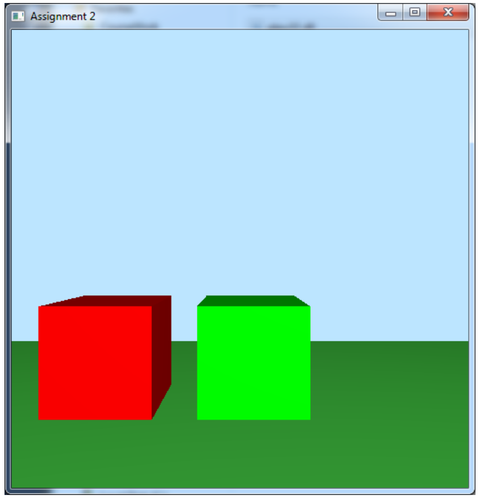
KAIST

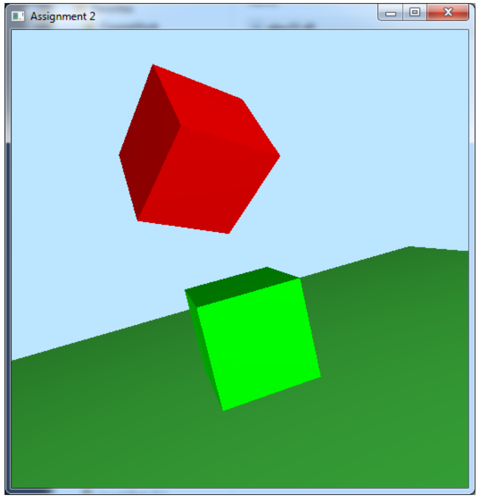
Full affine transformation — Translation — Rotation

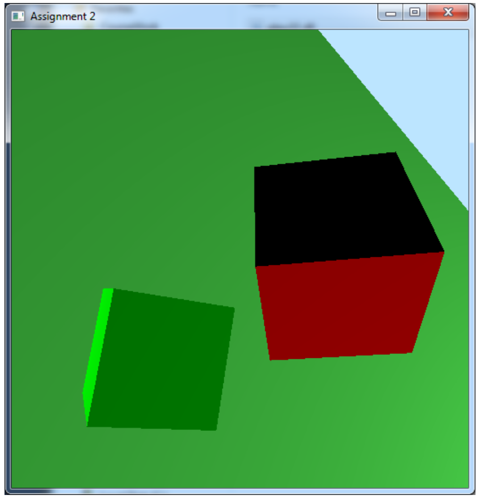
$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & h \\ 0 & 0 & 1 & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & b & c & 0 \\ e & f & g & 0 \\ i & j & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$\begin{bmatrix} l & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} i & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} l & 0 \\ 0 & 1 \end{bmatrix}$$

$$A = TL$$







Quaternion

- Quaternion is implemented in quat.h
- All operations are already implemented in provided code.

- Why not using an Euler(x, y, z) rotation or a rotation matrix?
  - Euler rotation: Gimbal lock problem
  - Rotation matrix: 9 elements (too much computation)

Quaternion Input

KAIST

- Constructors

```
Quat() : q_(1,0,0,0) {}
Quat(const double w, const Cvec3& v) : q_(w, v[0], v[1], v[2]) {}
Quat(const double w, const double x, const double y, const double z) : q_(w, x,y,z) {}
```

  – For given axis and angle θ

$$x = \sin\left(\frac{\theta}{2}\right)\cdot axis.x,\ y = \sin\left(\frac{\theta}{2}\right)\cdot axis.y,\ z = \sin\left(\frac{\theta}{2}\right)\cdot axis.z,\ w = \cos\left(\frac{\theta}{2}\right)$$

- Static Constructors

```
static Quat makeXRotation(const double ang)
static Quat makeYRotation(const double ang)
static Quat makeZRotation(const double ang)
```

Apply Quaternion on Vector

- Perform the following triple quaternion multiplication:
$$\begin{bmatrix} \cos\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right)\hat{\mathbf{k}} \end{bmatrix} \begin{bmatrix} 0 \\ \hat{\mathbf{c}} \end{bmatrix} \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right)\hat{\mathbf{k}} \end{bmatrix}^{-1}$$
- Result is of form: $\begin{bmatrix} 0 \\ \hat{\mathbf{c}}' \end{bmatrix}$
- The vector multiplication is already implemented in skeleton code

```
Cvec4 operator * (const Cvec4& a) const {
  const Quat r = *this * (Quat(0, a[0], a[1], a[2]) * inv(*this));
  return Cvec4(r[1], r[2], r[3], a[3]);
}
```

Apply Quaternion on Vector

KAIST

- Perform the following triple quaternion multiplication:

$$\begin{bmatrix} \cos\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right)\hat{\mathbf{k}} \end{bmatrix} \begin{bmatrix} 0 \\ \hat{\mathbf{c}} \end{bmatrix} \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right)\hat{\mathbf{k}} \end{bmatrix}^{-1}$$

- Result is of form: $\begin{bmatrix} 0 \\ \hat{\mathbf{c}}' \end{bmatrix}$
- The vector multiplication is already implemented in skeleton code

```
Cvec4 operator * (const Cvec4& a) const {
  const Quat r = *this * (Quat(0, a[0], a[1], a[2]) * inv(*this));
  return Cvec4(r[1], r[2], r[3], a[3]);
}
```

Task 1: Rigid Body Transformation

- Define RigTForm class
  - Translation t (3D point vector) and rotation r (4D **Quaternion** vector)
  - Rigid body transformation class
- Computationally efficient than matrix multiplication.
- **RigTForm**'s role is much similar to Matrix4 class, but it is a very helpful **utility class** for further implementation.

Task 1: Rigid Body Transformation

- Implement manipulations
  - Inversion `inline RigTForm inv(const RigTForm& tform)`
  - Multiplication `RigTForm operator * (const RigTForm& a) const {`
  - Conversion to matrix `inline Matrix4 rigTFormToMatrix(const RigTForm& tform)`
  - Conversion from a translation vector `explicit RigTForm(const Cvec3& t)`
  - Conversion form a quaternion `explicit RigTForm(const Quat& r)`
  - Multiplication with a vector `Cvec4 operator * (const Cvec4& a) const`
- Alternate Matrix4 class with RigTForm class in asst2.cpp.
- After you replace Matrix4 by RigTForm, everything should behave same as before.

Task 1: (Hint)

- RigTForm inversion
  - Rotation is just inverse of the quaternion
  - Translation is affected by rotation of itself.
  - $v = RT^{-1}(RT(v))$

- RigTForm multiplication
  - Rotation is just multiplication of two quaternions
  - Translation of first RigTForm is affected by rotation of second RigTForm.
  - (Translation is always affected by previous rotation)

Task 2: Arcball

- Implement the arcball interface
  - Draw a sphere to represent the arcball
  - Implement an arcball function in openGL functions
- Compute rotation
  - Compute two 3D vectors on the screen space.
- Two helper functions in skeleton code
  - getScreenSpaceCoord
  - getScreenToEyeScale
- The radius of the sphere should be 0.25 * min(g_windowWidth, g_windowHeight).

Draw Wireframe Mode

KAIST

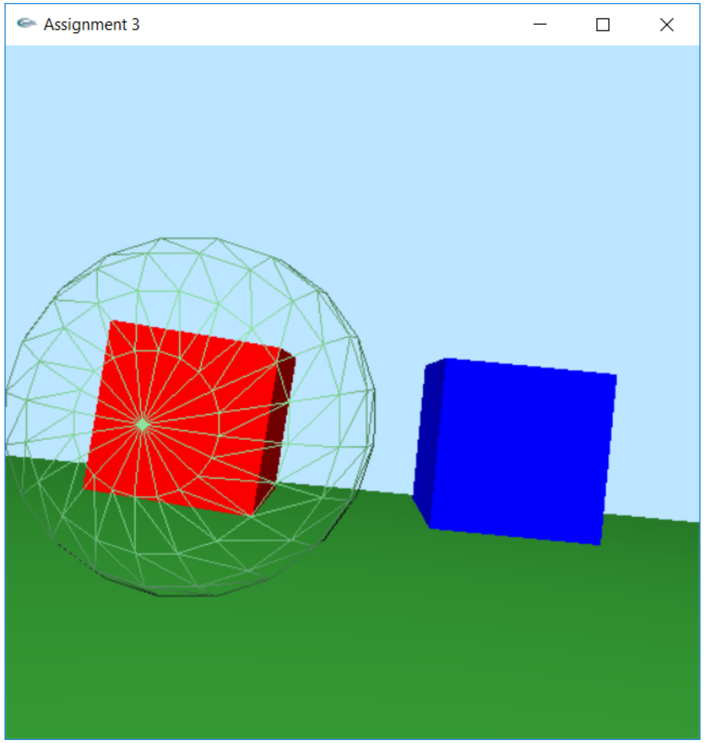
- Draw a wire framed sphere for arcball visualization

```
// switch to wire frame mode
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

// draw something
g_sphere->draw(curSS);

// switch back to solid mode
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

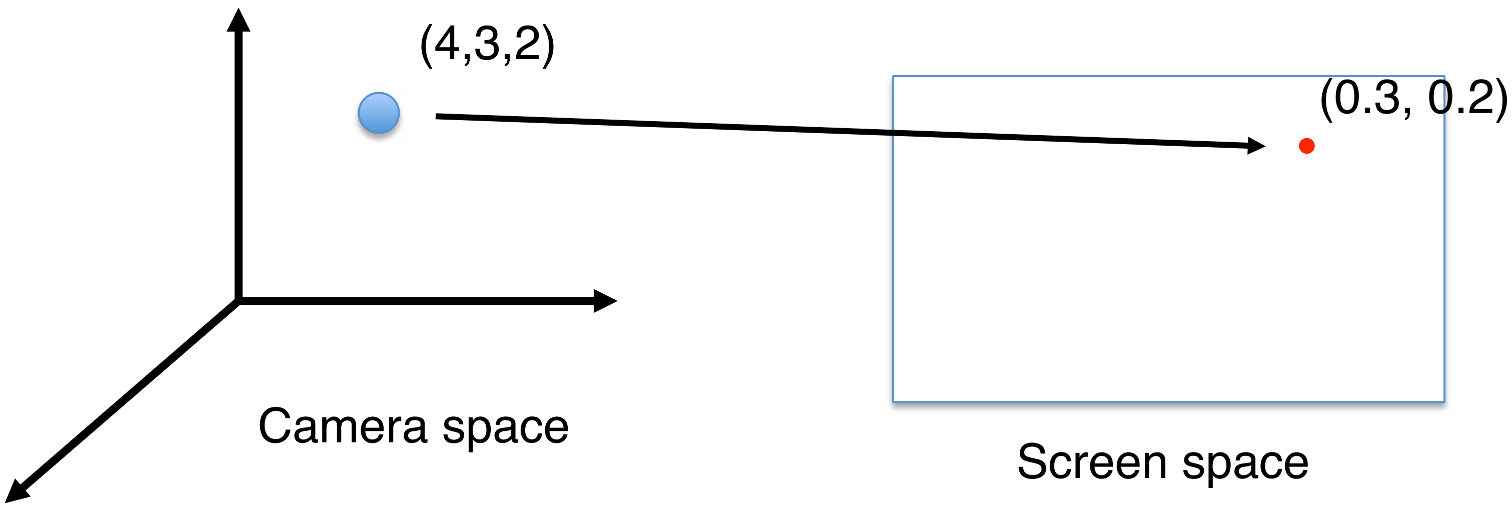
(If you do not switch back to “GL_FILL”, then all objects may be drawn in wireframe)



getScreenSpaceCoord

KAIST

- Convert a 3D vector of the point to a 2D point on a screen space (in pixel units).




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

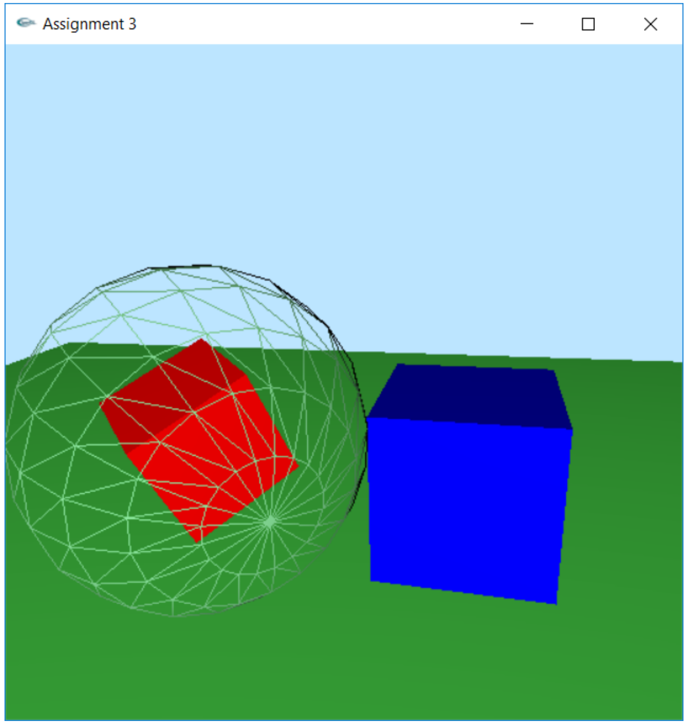
getScreenSpaceCoord

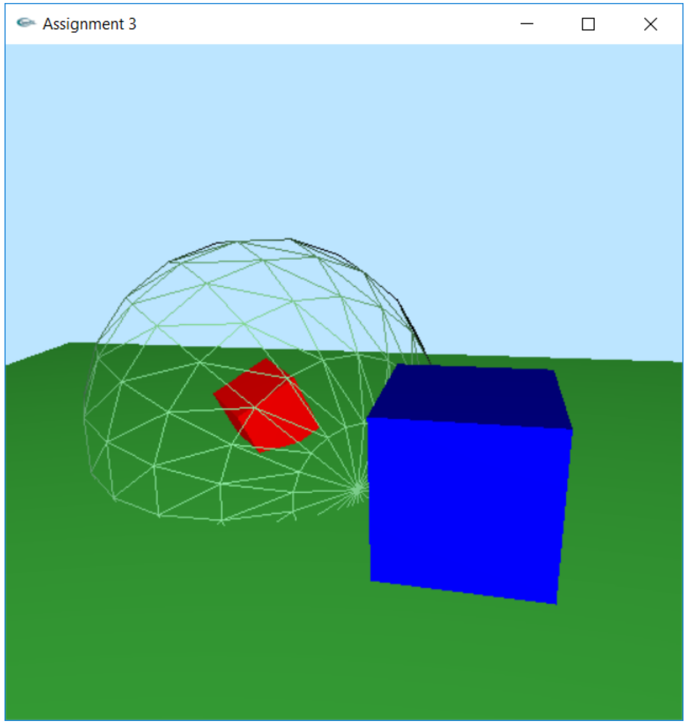
- Why use it?
  - Mouse position is given as 2D point on a screen space
  - Arcball is a 3D object.

  - We should convert one of two points (mouse position, arcball position) to the other coordinate to calculate arcball manipulation.

getScreenToEyeScale

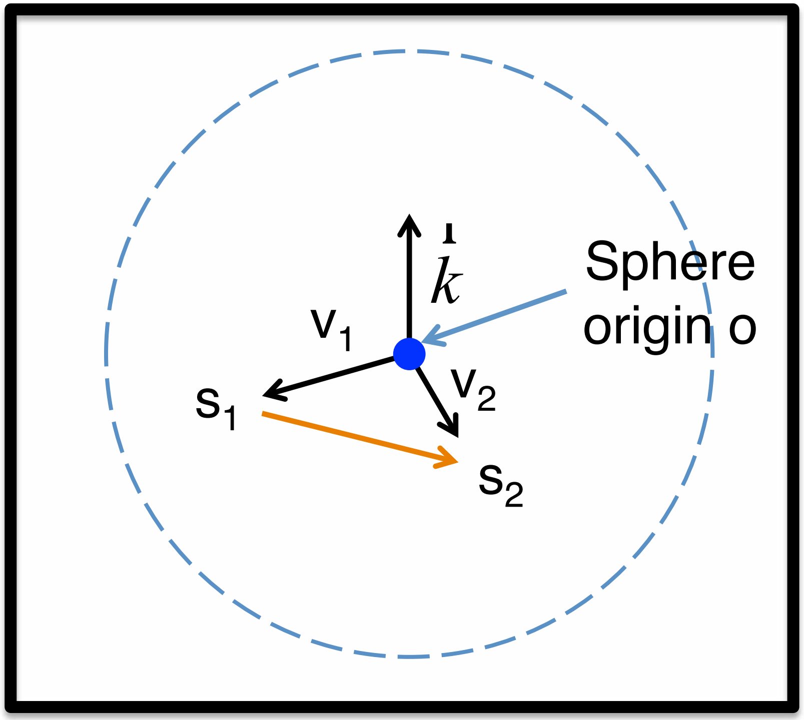
- Arcball size should not be changed in Screen Space even if size of the cube is changed in screen space due to translation

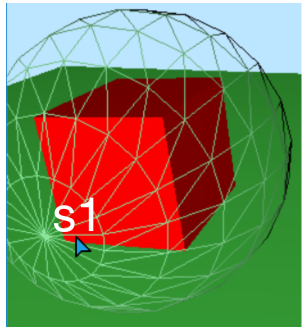


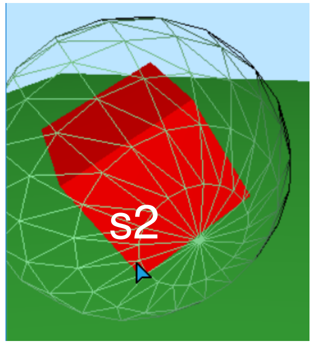



```
inline double getScreenToEyeScale(double z, double frustFovY, int screenHeight)
```

Arcball Rotation







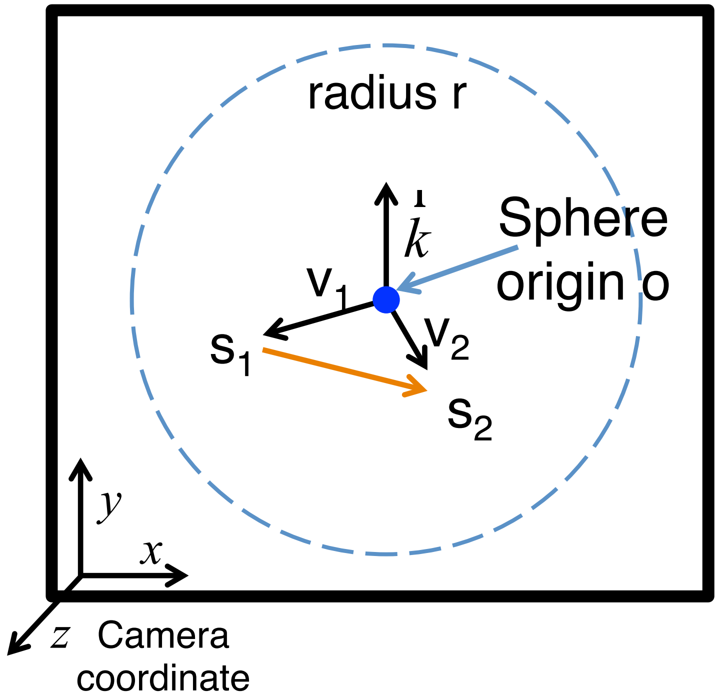
- Sphere origin o – center of sphere, projection of a frame origin
- s_1 – clicked screen coordinate
- s_2 – dragged mouse screen coordinate

VISUAL COMPUTING Lab

Arcball Rotation (hint)

KAIST

- v_1, v_2 – the directional vectors
- $v_{1x} = s_x - o_x$, $v_{1y} = s_y - o_y$, $v_{1x}^2 + v_{1y}^2 + v_{1z}^2 = r^2$

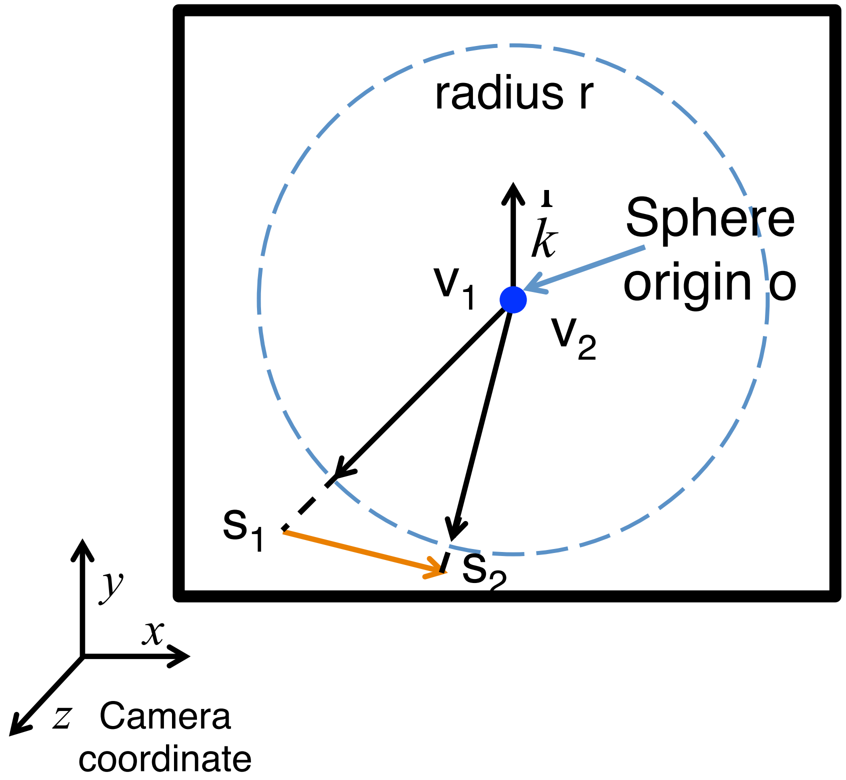


VISUAL COMPUTING Lab

Arcball Rotation (hint)

KAIST

- When you drag outside of the arcball, use nearest point of the arcball for manipulation.



VISUAL COMPUTING Lab

Task 3: Translation Fix Up

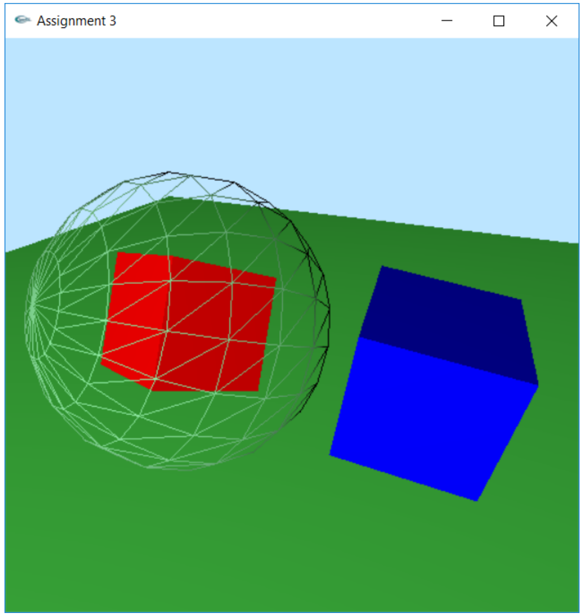
KAIST

- Translate the object as same as the mouse movement.
- Use g_arcballScale.
- Wherever the object is, the object should follow a mouse pointer.

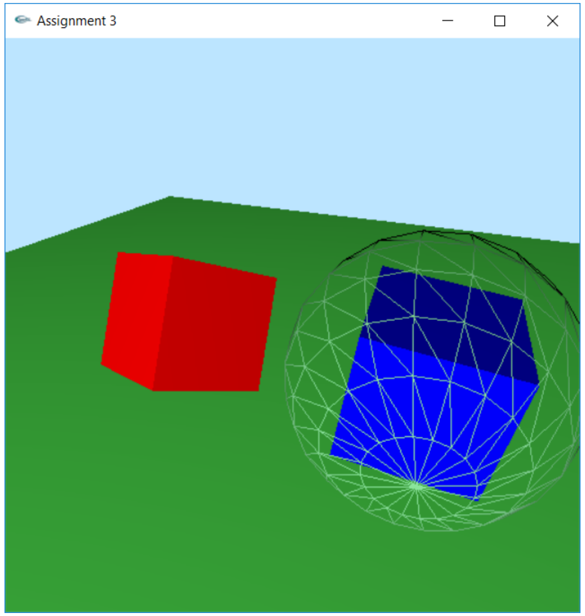
Active Object

- When ‘v’ is pressed

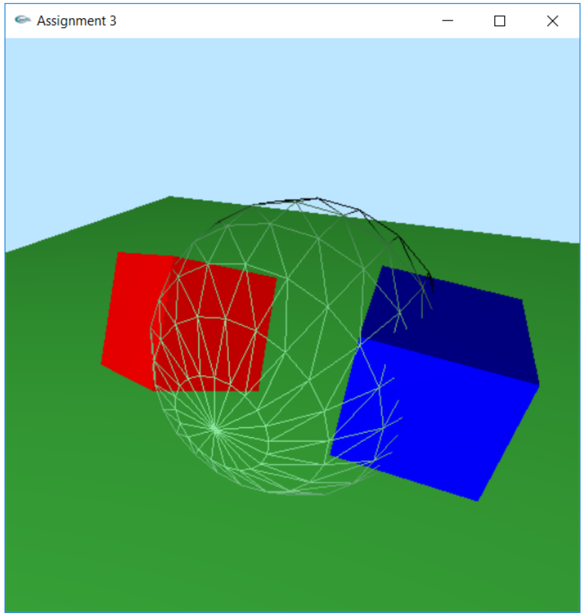
Red-box is an active object



Blue-box is an active object



Sky is an active object

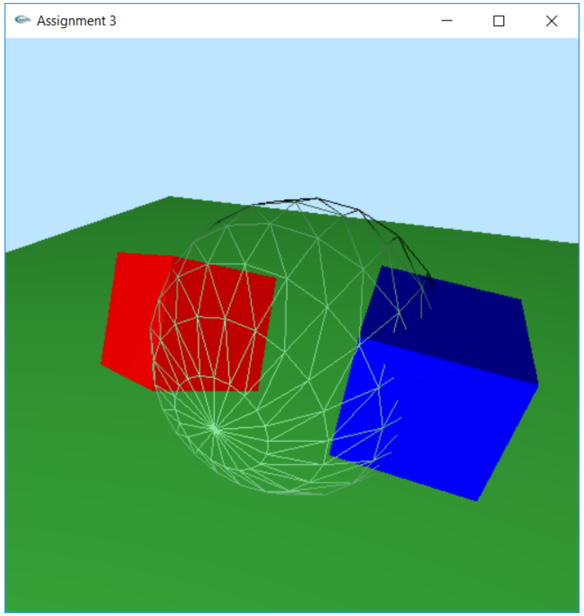


Sky Mode

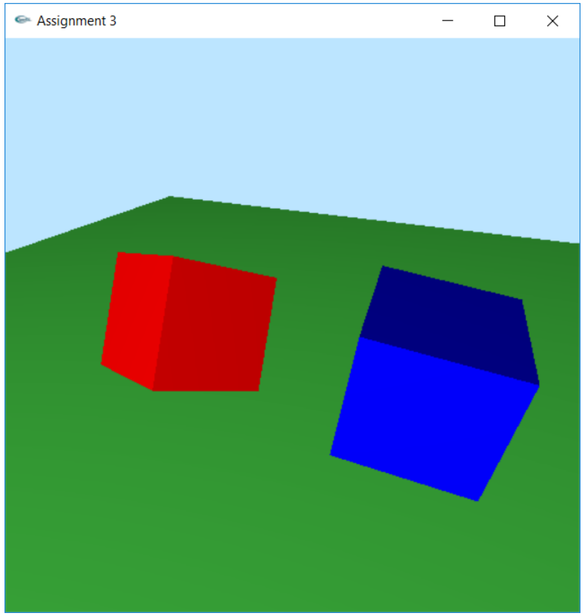
KAIST

- This is mode is only effective when sky is an active object
- When ‘m’ is pressed

World-sky frame

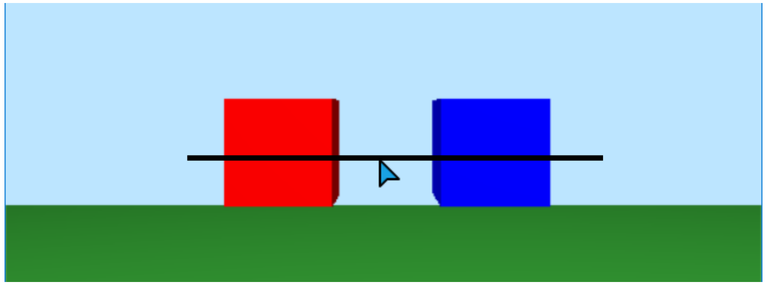


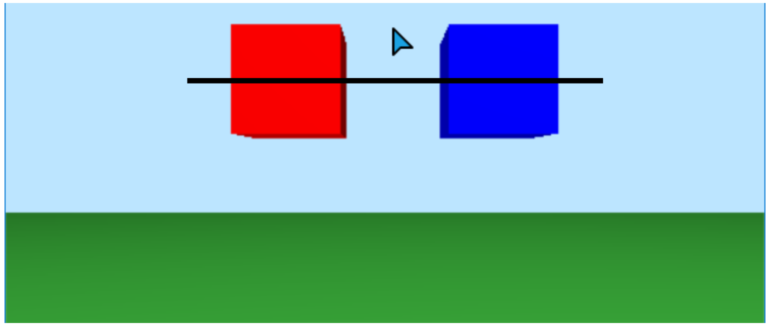
World-sky frame (Ego-motion)



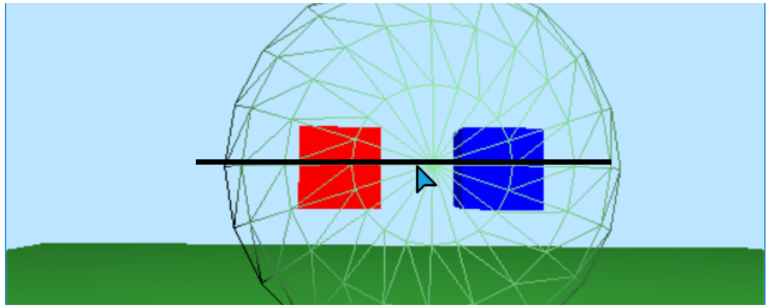
Task 3: Translation Fix Up

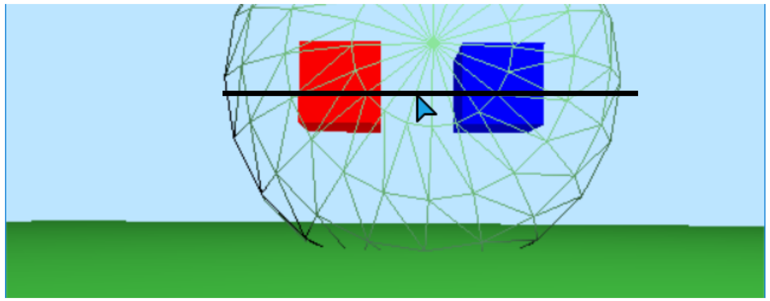
Solution of HW2





Solution of HW3



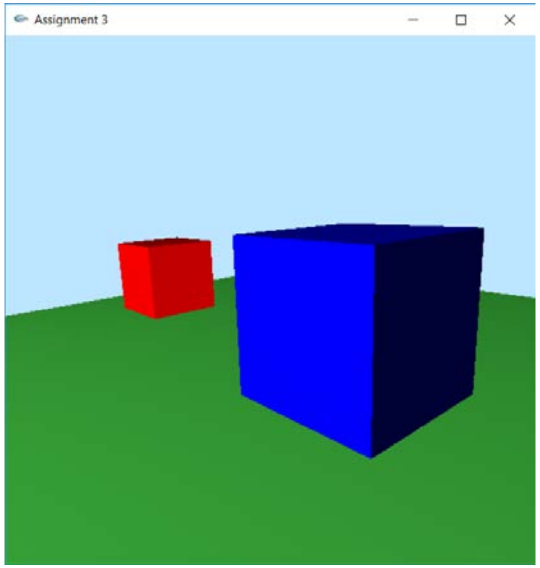


- When scene is minified, the translation should also be magnified to follow mouse cursor

Task 3: Translation Fix Up

KAIST

- Perspective (원근감)
  - For same size of objects, far objects look smaller



  - Also, for same translation, far objects look translate smaller than near objects.
  - However, translation of HW3 should be same as translation of mouse cursor.
  - We will learn perspective projection next week.

VISUAL COMPUTING Lab

Task 3: Translation Fix Up

KAIST

- There is a statement in HW3 document:
  - "When the arcball is not in use (e.g., in ego motion), g_arcballScale may not be correctly defined, so feel free to fall back to the hard-coded number in that case."

- Which means translation should not be scaled in sky-sky mode.