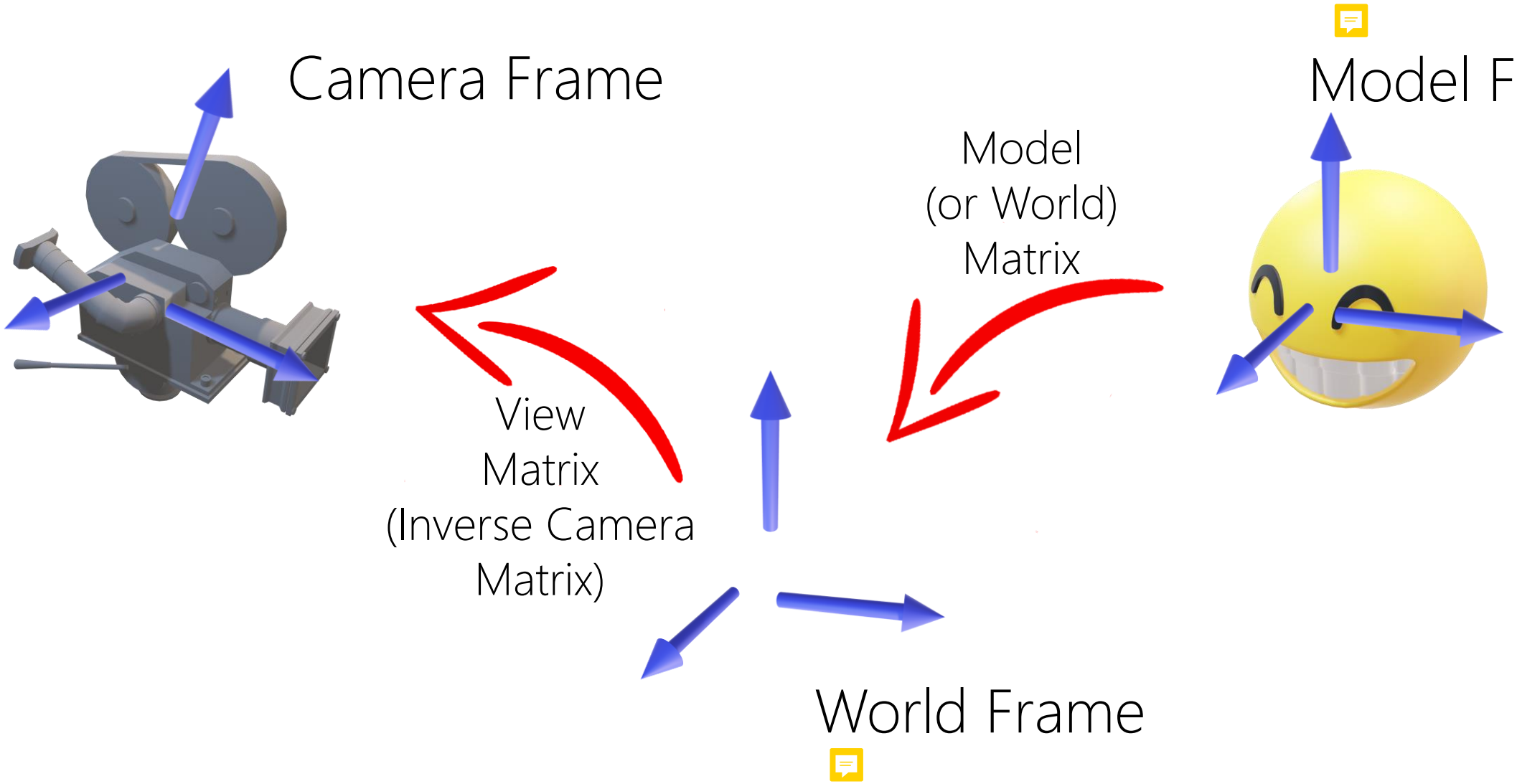
POLITECNICO
MILANO 1863

Normals and Shading Spaces

Computer Graphics 2021

Erica Stella (erica.stella@polimi.it)

Transformations



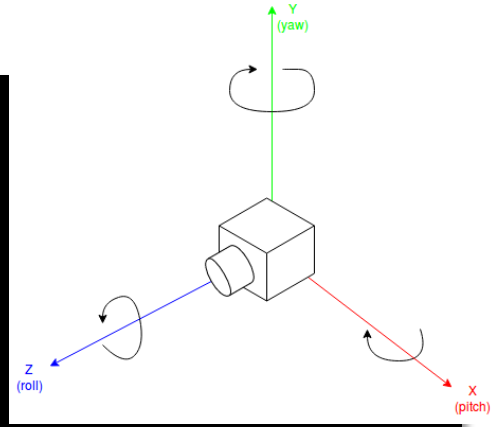
Transformations utils

M =World Matrix / Model Matrix

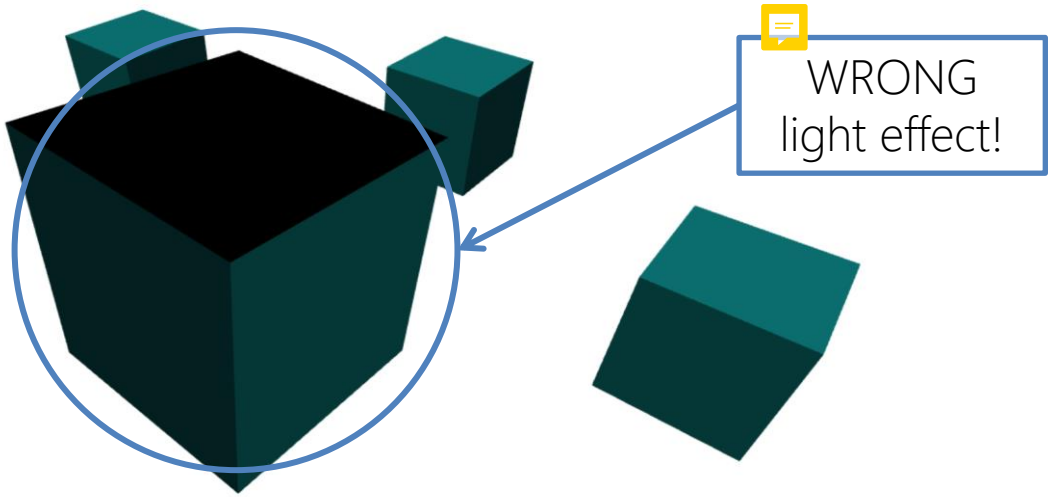
V = View Matrix / Eye Matrix

Utils library, provided with the examples, implements useful functions to deal with matrices and to generate model, view, perspective matrices

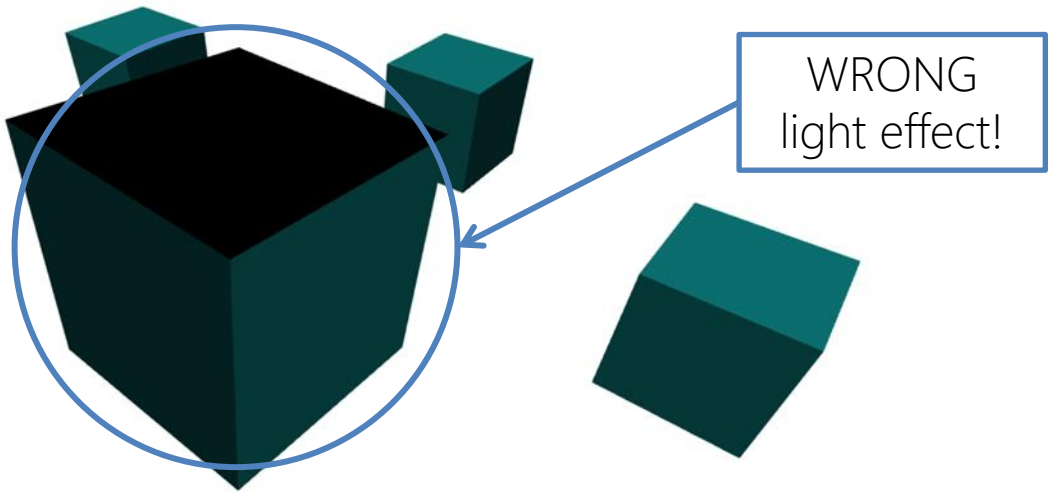
```
fieldOfView = 90.0; aspectRatio = w/h; nearPlane = 0.1; farPlane = 100.0;
xCam = 3.0; yCam = 3.0; zCam = 3.0; elevCam = -45.0;//Xaxis
angleYCam = -45.0;
xObj = 0.0; yObj = 0.0; zObj = 2.5;
rotXObj = 3.0; rotYObj = 3.0; royZObj = 2.5; scaleObj = 1.0;


Perspective = utils.MakePerspective(fieldOfView, aspectRatio, nearPlane, farPlane);

V = utils.MakeView(xCam, yCam, zCam, elevCam, angleYCam);

M = utils.MakeWorld( xObj, yObj, zObj, rotXObj, rotYObj, royZObj, scaleObj);

MV = utils.multiplyMatrices(V, M);

Projection = utils.multiplyMatrices(Perspective, MV);
```



Implementing lighting: Issue



Implementing lighting: Issue



Vertices are transformed using world/view/projection matrix, but normals are not…

```
[...]
void main() {
  fsNormal = inNormal;
  gl_Position = matrix * vec4(inPosition, 1.0);
}
```

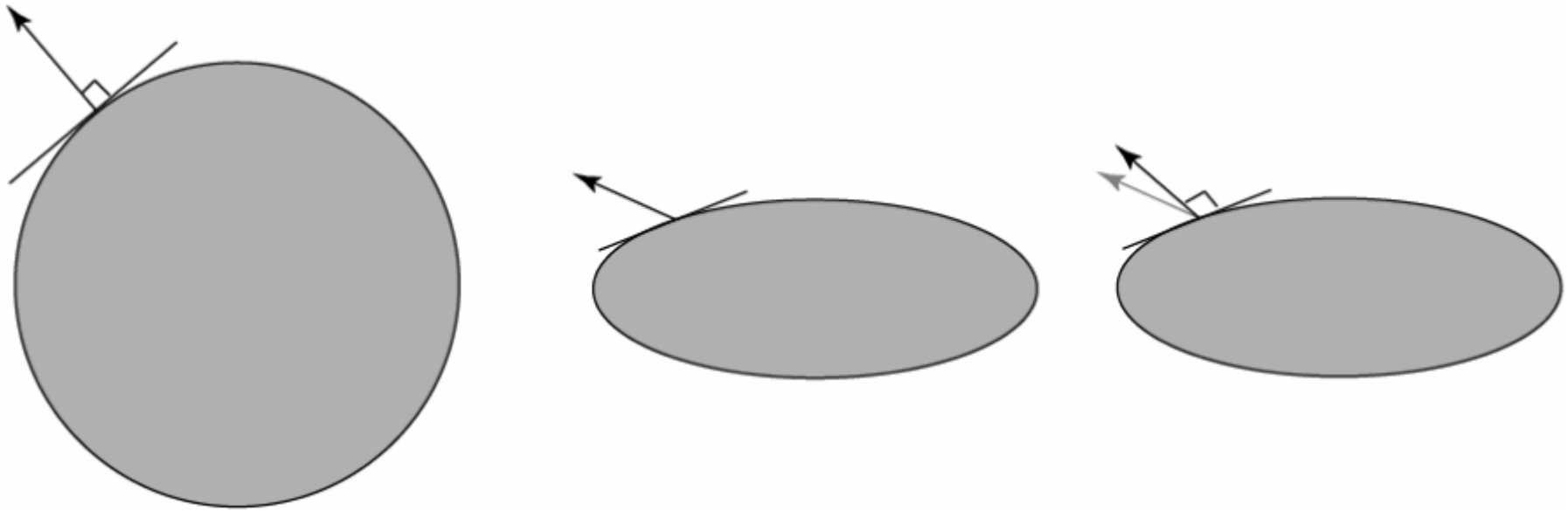
Vertex Shader

Normal and vertices position are (almost) always expressed in *local (or model) coordinates*

But lights and observer are expressed in *world coordinates*.

Transforming normals

- We cannot apply directly model/view matrix trasformations (when non-uniform scaling is enabled)



Transforming normals

- By definition: $\boldsymbol{t} \cdot \boldsymbol{n} = \boldsymbol{t}^T \boldsymbol{n} = 0$
- After transform $\boldsymbol{M}$: $(M\boldsymbol{t})^T (X)\boldsymbol{n} = 0$
- Which implies: $\boldsymbol{t}^T M^T X \boldsymbol{n} = 0$
- To make it valid we have: $\boldsymbol{t}^T M^T (M^T)^{-1} \boldsymbol{n} = 0$

- Normals are transformed by means of the inverse transpose transformation matrix

Shading Spaces

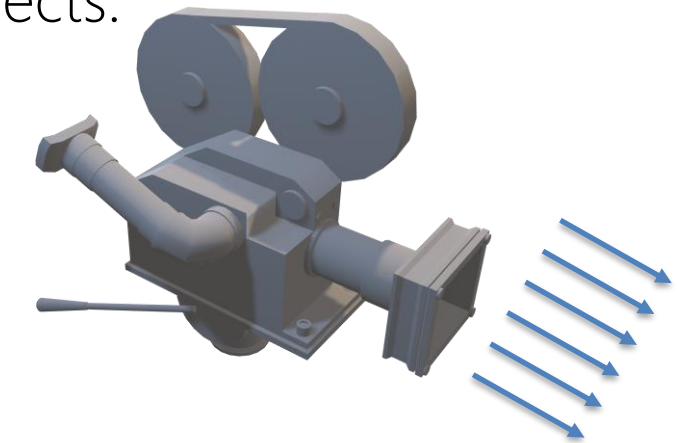
To compute shading equations coherently,
data about normals, lights and eye/camera position
must be expressed in the same coordinates system, or *space*.

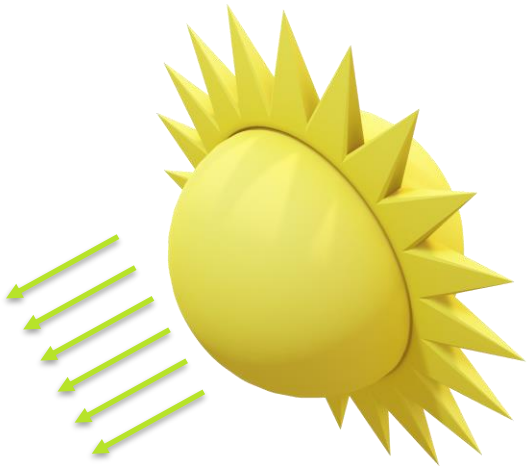
The most common spaces are:

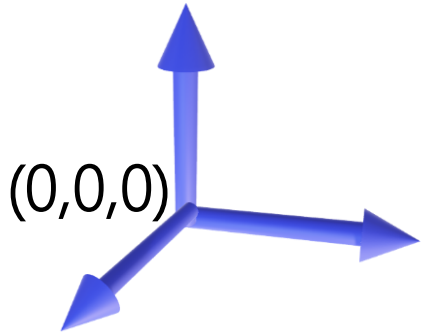
- World Space
- Camera Space
- Object Space
- Static Space

World Space

When the illumination model is computed in **World Space**, vertices positions and normals must be transformed according to the World transforms of the considered objects.







World Space

World Space *usually* requires to pass 3 matrices to the vertex shader:

- WVP projection (as usual)
- Normal Matrix to transform the normals in world space
- World Matrix to transform the positions in world space

Data	Are usually in	Must be transformed in World Space?
Vertex Position	Local Space	Yes
Vertex Normals	Local Space	Yes*
Light Direction	World Space	No
Light Position	World Space	No
Eye Position	World Space	No

- **Note***: to transform normals it is possible to use the world matrix **IF** there are no non-uniform scale modifications on the object. Otherwise, the 3x3 submatrix of the inverse transpose world matrix is required.

World Space

Data	Are usually in	Must be transformed in World Space?
Vertex Position	Local Space	Yes
Vertex Normals	Local Space	Yes*
Light Direction	World Space	No
Light Position	World Space	No
Eye Position	World Space	No

- PRO: easy to implement
- CON: Light computation using World space is computationally expensive (a lot of per vertex matrix multiplication) but it is easy to understand.

Lambertian Reflection in World Space

```
#version 300 es

in vec3 inPosition;
in vec3 inNormal;
out vec3 fsNormal;

uniform mat4 matrix;
uniform mat4 nMatrix;  //matrix to transform normals

void main() {
  fsNormal = mat3(nMatrix)*inNormal;
  //This is okay too, but w MUST be 0
  //Otherwise, translations will distort the direction of the normal
  //fsNormal = (nMatrix * vec4(inNormal,0.0)).xyz;
  gl_Position = matrix * vec4(inPosition, 1.0);
}
```

Vertex Shader

Lambertian Reflection in World Space

Remember to pass the right matrices from the javascript source

```
//Retrieve location of normal matrix
normalMatrixPositionHandle = gl.getUniformLocation(program, 'nMatrix');
matrixLocation = gl.getUniformLocation(program, 'matrix');

perspectiveMatrix = utils.MakePerspective(90, w/h, 0.1, 100.0);
viewMatrix = utils.MakeView(3.0, 3.0, 2.5, -45.0, -40.0);
cubeWorldMatrix = utils.MakeWorld( -3.0, 0.0, -1.5, 0.0, 0.0, 0.0, 0.5);
//To invert the matrix, use the utils.invertMatrix function
cubeNormalMatrix = utils.invertMatrix(utils.transposeMatrix(cubeWorldMatrix));

projectionMatrix = utils.multiplyMatrices(viewMatrix, cubeWorldMatrix);
projectionMatrix = utils.multiplyMatrices(perspectiveMatrix, projectionMatrix);

gl.uniformMatrix4fv(matrixLocation, gl.FALSE, utils.transposeMatrix(projectionMatrix));
//Pass normal matrix
gl.uniformMatrix4fv(normalMatrixPositionHandle, gl.FALSE,
                                        utils.transposeMatrix(cubeNormalMatrix));
```

World Space

M =World Matrix / Model Matrix

V = View Matrix / Eye Matrix

What to transform?

- Vertex Positions: $M * \boldsymbol{v}$ (for each vertex → Vertex Shader)
- Vertex Normals : $\text{sub3x3}(M^{-T}) * \boldsymbol{n}$ (for each vertex → Vertex Shader)

```
[...]
uniform mat4 matrix;
uniform mat4 pMatrix;
uniform mat4 nMatrix;

void main() {
  fsNormal = mat3(nMatrix)*inNormal;
  fsPosition = (pMatrix * vec4(inPosition,
               1.0)).xyz;
  gl_Position = matrix * vec4(inPosition,
               1.0);
}
```

Vertex Shader

```
[...]

vec3 nEyeDirection = normalize(eyePosition
                     - fsPosition);
vec3 nLightDirection =
                normalize(-lightDirection);
vec3 nNormal = normalize(fsNormal);

[...phong/blinn equations...]
```

Fragment Shader

World Space

M =World Matrix / Model Matrix

V = View Matrix / Eye Matrix

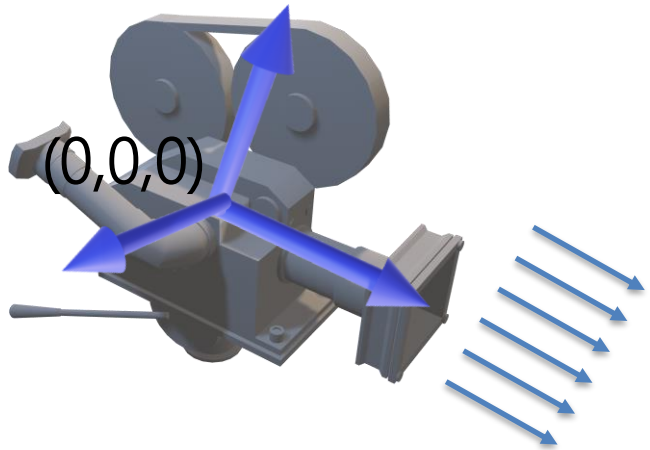
```
[...]
uniform mat4 matrix;
uniform mat4 nMatrix;
uniform mat4 pMatrix;

void main() {
  [...]
}
```

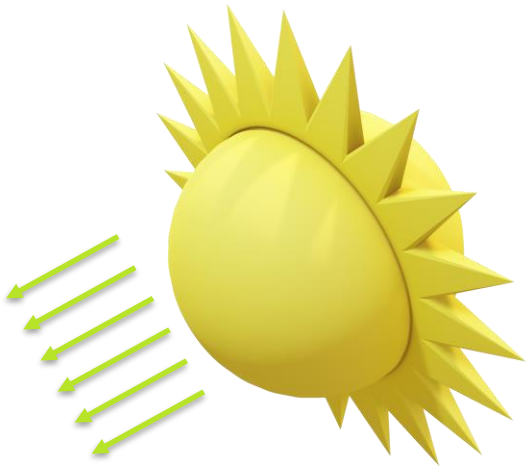
Vertex Shader

```
Perspective = utils.MakePerspective(90, w/h, 0.1, 100);
V = utils.MakeView(3.0, 3, 2.5, -45, -40);
M = utils.MakeWorld( -3, 0, -1, 0, 0, 0, 0.5);
normalMatrix = utils.invertMatrix(utils.transposeMatrix(M));

MV = utils.multiplyMatrices(V, M);
Projection = utils.multiplyMatrices(Perspective, MV);

gl.uniformMatrix4fv(matrixLocation, gl.FALSE,
                    utils.transposeMatrix(Projection));
gl.uniformMatrix4fv(normalMatrixLocation, gl.FALSE,
                    utils.transposeMatrix(normalMatrix));
gl.uniformMatrix4fv(vertexMatrixPositionHandle, gl.FALSE,
                    utils.transposeMatrix(M));
```

Camera Space

In **Camera Space**, positions and normals must be transformed using both the View and the World transforms. Lights (i.e. direction and position) must be transformed together with objects and normal, but only with the View matrix.





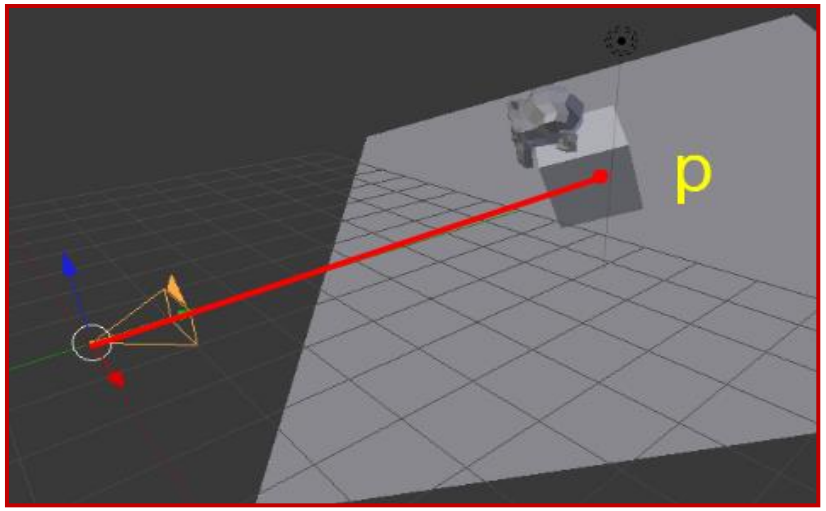
Camera Space

- Requires to pass **3** matrices to the **vertex shader,** as in World Space, to project vertices(1) and to transform positions(2) and normals(3).
- Normal directions are transformed with the inverse transpose of the 3x3 submatrix of the ModelView (or WorldView) matrix.
- Light positions are transformed by applying the View transform to them.
- Light directions are transformed by considering the inverse transpose of the 3x3 submatrix of the View transform.
  - However, since scale transforms are usually not used for the View matrix, the 3x3 submatrix is usually enough

Camera Space

Data	Are usually in	Must be transformed in Camera Space?
Vertex Position	Local Space	Yes
Vertex Normals	Local Space	Yes
Light Direction	World Space	Yes
Light Position	World Space	Yes
Eye Position	World Space	No

- PRO: the **observer direction** (computed in the shaders for specular reflection), can be obtained by **normalizing** the coordinates of the considered point of the object in the 3D space (because, after the View transform, the center of projection is in the origin of the axis).
- CON: less intuitive than world space



Lambertian Reflection in Camera Space

- Shaders don't change but we must change the matrices passed to the shaders
- We must also transform the directions of the ligths before passing them to the shaders

```
//define directional light
var dirLightAlpha = -utils.degToRad(60);
var dirLightBeta  = -utils.degToRad(120);

//Spherical coordinates math to get a direction from two angles
var directionalLight = [Math.cos(dirLightAlpha) * Math.cos(dirLightBeta),
            Math.sin(dirLightAlpha),
            Math.cos(dirLightAlpha) * Math.sin(dirLightBeta)
            ];
var directionalLightColor = [0.1, 1.0, 1.0];

var lightDirectionHandle = gl.getUniformLocation(program, 'lightDirection');
var lightColorHandle = gl.getUniformLocation(program, 'lightColor');
```

Lambertian Reflection in Camera Space

```
var normalMatrixPositionHandle = gl.getUniformLocation(program, 'nMatrix');
var matrixLocation = gl.getUniformLocation(program, "matrix");

var perspectiveMatrix = utils.MakePerspective(90, gl.canvas.width / gl.canvas.height, 0.1, 100.0);
var viewMatrix = utils.MakeView(3.0, 3.0, 2.5, -45.0, -40.0);
var worldViewMatrix = utils.multiplyMatrices(viewMatrix, cubeWorldMatrix[i]);
var projectionMatrix = utils.multiplyMatrices(perspectiveMatrix, worldViewMatrix);

//Matrix to transform the light direction from world space to camera space
var lightDirMatrix = utils.invertMatrix(utils.transposeMatrix(viewMatrix));
//Inverse transpose of the world view matrix for the normals
var cubeNormalMatrix = utils.invertMatrix(utils.transposeMatrix(worldViewMatrix));
//Directional light transformed by the 3x3 submatrix
var directionalLightTransformed = utils.multiplyMatrix3Vector3(
    utils.sub3x3from4x4((lightDirMatrix)), directionalLight);

gl.uniform3fv(lightColorHandle, directionalLightColor);
//Pass the already-transformed direction of the light
gl.uniform3fv(lightDirectionHandle, directionalLightTransformed);

gl.uniformMatrix4fv(matrixLocation, gl.FALSE, utils.transposeMatrix(projectionMatrix));
gl.uniformMatrix4fv(normalMatrixPositionHandle, gl.FALSE, utils.transposeMatrix(cubeNormalMatrix));
```

Camera Space

M =World Matrix / Model Matrix

V = View Matrix / Eye Matrix

What to transform?

- Vertex Positions: $\boldsymbol{MV} * \boldsymbol{v}$ (for each vertex → Vertex Shader)
- Vertex Normals: sub3x3$((\boldsymbol{MV})^{-T}) * \boldsymbol{n}$ (for each vertex → Vertex Shader)
- Light Position: $\boldsymbol{V} * \boldsymbol{v}$ (Once for the whole scene → CPU)
- Light Direction: sub3x3$((\boldsymbol{V})^{-T}) * \boldsymbol{n}$ (Once for the whole scene → CPU)

```
[...]
uniform mat4 matrix;
uniform mat4 nMatrix;
uniform mat4 pMatrix;

void main() {
  fsNormal = mat3(nMatrix) * inNormal;
  fsPosition = (pMatrix * vec4(inPosition, 1.0)).xyz;
  gl_Position = matrix * vec4(inPosition, 1.0);
}
```

Vertex Shader

```
[...]

vec3 nEyeDirection = normalize(0 - fsPosition);
vec3 nLightDirection = normalize(-lightDirection);

[...phong/blinn equations...]
```

Fragment Shader

Camera Space

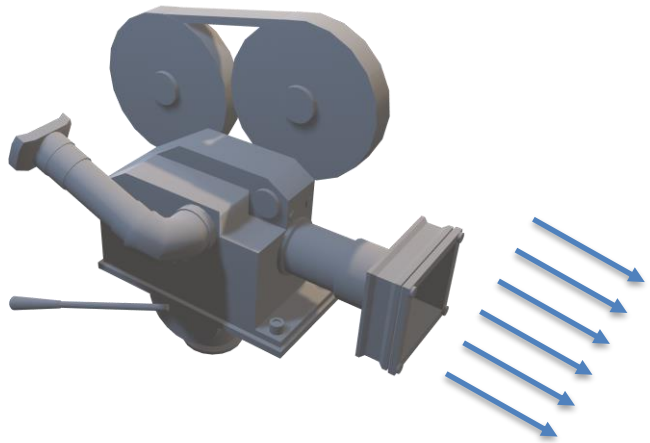
M =World Matrix / Model Matrix

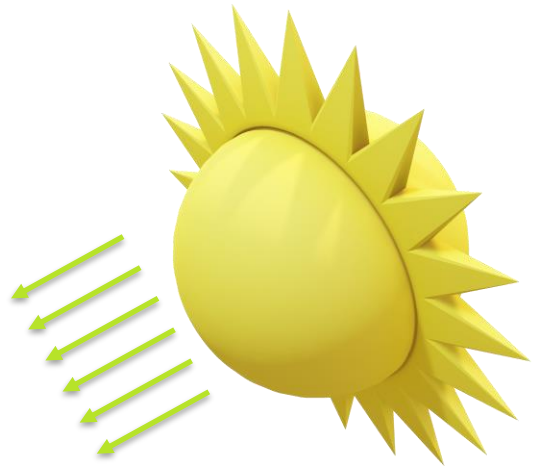
V = View Matrix / Eye Matrix

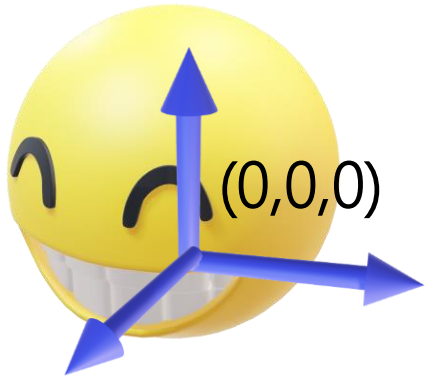
```
var alpha = -utils.degToRad(60);
var beta  = -utils.degToRad(120);

var d_l = [Math.cos(alpha) * Math.cos(beta),
           Math.sin(alpha), Math.cos(alpha) * Math.sin(beta)  ];

var M = utils.MakeWorld( -3, 0, -1, 0, 0, 0, 0.5);
var V = utils.MakeView(3.0, 3.0, 2.5, -45.0, -40.0);
var lightDirMatrix = V; //if I am sure there is no scaling, otherwise inverse transpose
var lightPosMatrix = V;
var d_l_trans = utils.multiplyMatrix3Vector3(utils.sub3x3from4x4(lightDirMatrix), d_l);

var MV = utils.multiplyMatrices(V, M);
var Projection = utils.multiplyMatrices(Perspective, MV);

gl.uniformMatrix4fv(vertexMatrixPositionHandle, gl.FALSE, utils.transposeMatrix(MV));
gl.uniformMatrix4fv(matrixLocation, gl.FALSE, utils.transposeMatrix(projectionMatrix));
gl.uniform3fv(lightDirectionLocation, d_l _trans);
```

Object Space

In the **Object Space**, the lights and the observer directions are transformed in the local coordinates of the considered object, **for each** scene object.







Object Space

Lights' direction→ inverse transpose of the 3x3 submatrix of the inverse World matrix (+normalization) == transpose of the World matrix

Eye position → inverse of the World transform matrix

Data	Are usually in	Must be transformed in Object Space?
Vertex Position	Local Space	No
Vertex Normals	Local Space	No
Light Direction	World Space	Yes
Light Position	World Space	Yes*
Eye Position	World Space	Yes

*Lights that require calculating the direction from the fragment (or vertex) to the light's position require extra matrices to be passed because angles in object space are distorted so we won't look at them.

Object Space

Data	Are usually in	Must be transformed in Object Space?
Vertex Position	Local Space	No
Vertex Normals	Local Space	No
Light Direction	World Space	Yes
Light Position	World Space	Yes*
Eye Position	World Space	Yes

- PRO: Avoids transforming the normals of the objects (more efficient when the object has many vertices)
- CON: Different transformation for each object, angle distortion requires extra matrices for point lights

Object Space

M =World Matrix / Model Matrix

V = View Matrix / Eye Matrix

What to transform?

- Light Direction: sub3x3$(M^T) * \boldsymbol{n}$ (Once for the each model → CPU)
- Camera Position: $(MV)^{-1} * (\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{1})$ (Once for the each model → CPU)

```
[...]
uniform mat4 matrix;

void main() {
  fsNormal = inNormal;
  gl_Position = matrix * vec4(inPosition, 1.0);
}
```

Vertex Shader

Object Space

M =World Matrix / Model Matrix

V = View Matrix / Eye Matrix

```
var alpha = -utils.degToRad(60);
var beta  = -utils.degToRad(120);

var d_l = [Math.cos(alpha) * Math.cos(beta),
           Math.sin(alpha), Math.cos(alpha) * Math.sin(beta)  ];

var M = utils.MakeWorld( -3, 0, -1, 0, 0, 0, 0.5);
var V = utils.MakeView(3.0, 3.0, 2.5, -45.0, -40.0);
var lightDirMatrix = utils.transposeMatrix(M); //From world space to object space
var d_l_trans = utils.multiplyMatrix3Vector3(utils.sub3x3from4x4(lightDirMatrix), d_l);

var MV = utils.multiplyMatrices(V, M);
var Projection = utils.multiplyMatrices(Perspective, MV);

gl.uniformMatrix4fv(matrixLocation, gl.FALSE, utils.transposeMatrix(projectionMatrix));
gl.uniform3fv(lightDirectionLocation, d_l _trans);
```

Static Space

In **Static Space**, local and world coordinates are identical (all the world matrices are identity matrices). This also means that both World and Object space are coincident

In this case, we do not need matrices to transform positions and normals as in object space.

We do not even need to transform either lights or observer position, since they are already correct for the considered space.

Data	Expressed in	Must be transformed in Static Space?
Vertex Position	World Space	No
Vertex Normals	World Space	No
Light Direction	World Space	No
Light Position	World Space	No
Eye Position	World Space	No

Static Space

Data	Expressed in	Must be transformed in Static Space?
Vertex Position	World Space	No
Vertex Normals	World Space	No
Light Direction	World Space	No
Light Position	World Space	No
Eye Position	World Space	No

- PRO: The simplest to use, since it does not require any special transformation.
- CON: it does not allow to move objects during the application.

However, there are many cases in which it can be applied: medical and scientific visualization, fixed backgrounds in virtual reality applications