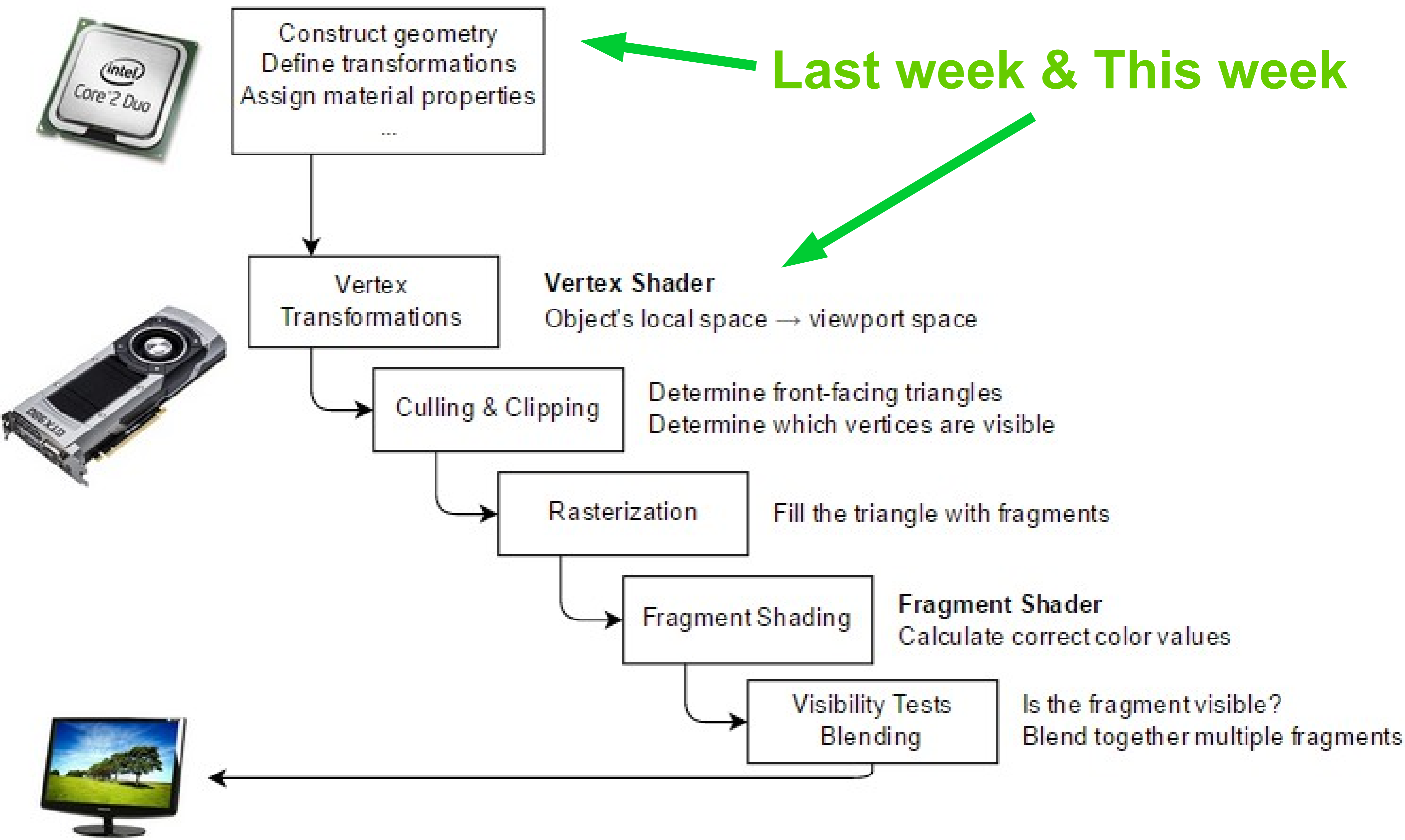
Computer Graphics

MTAT.03.015

Raimond Tunnel

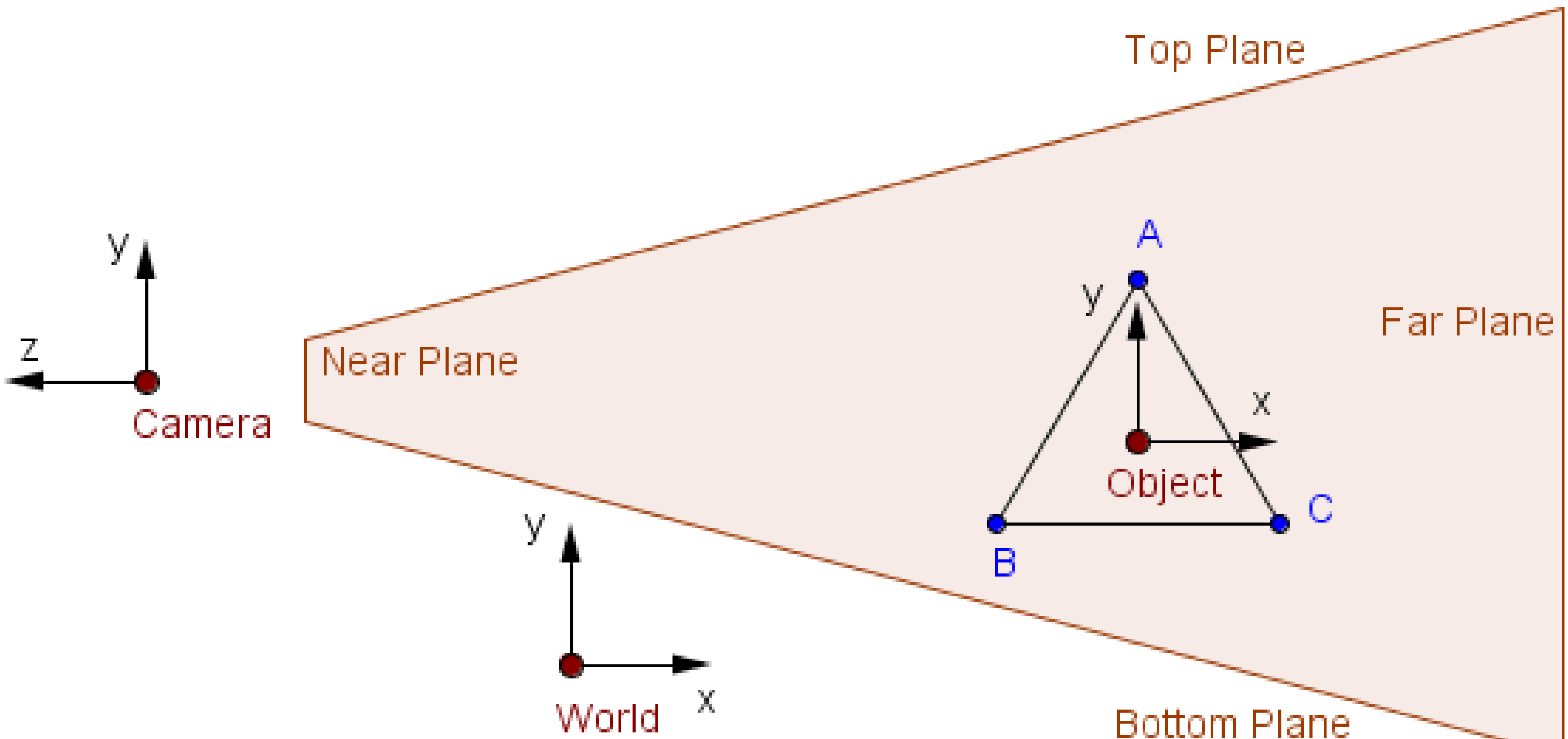
Study IT in .ee

The Road So Far...



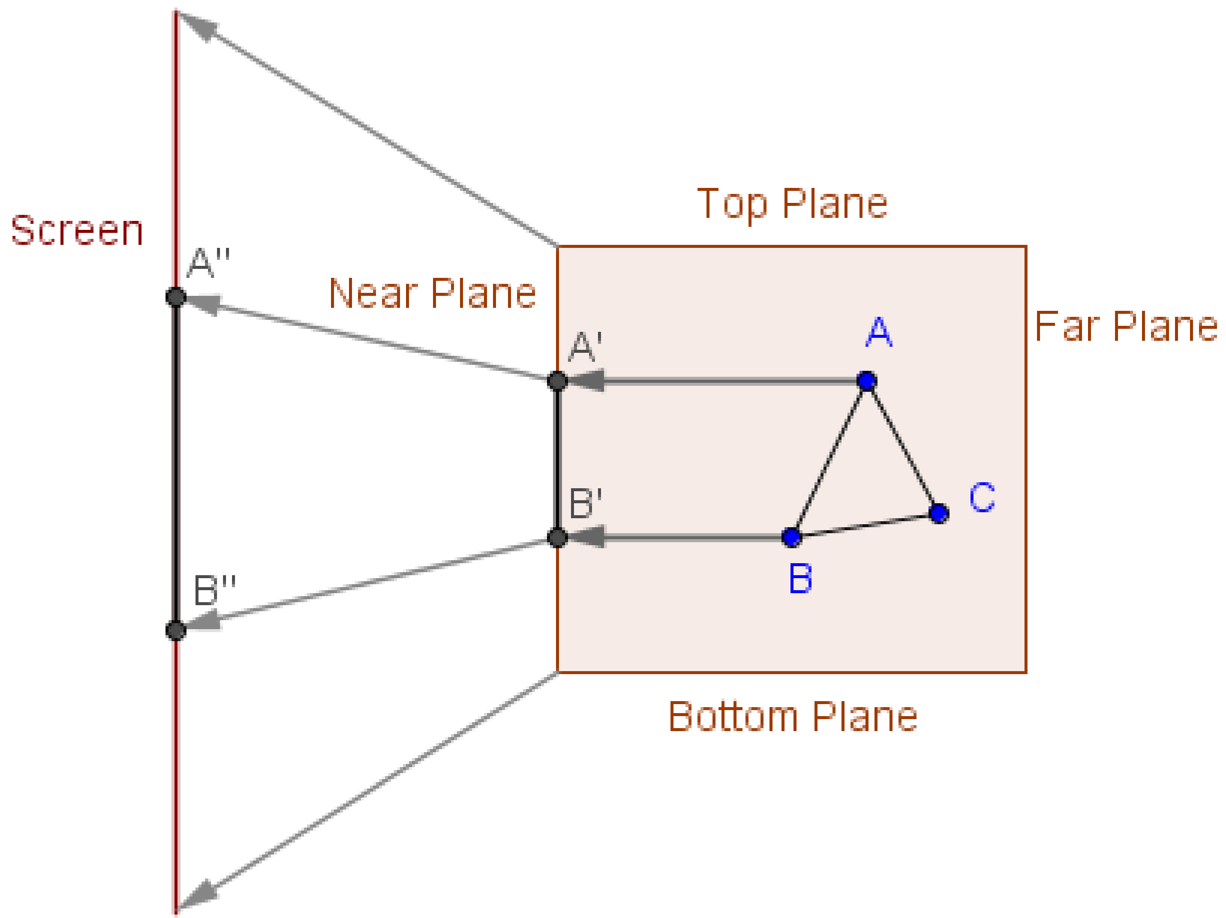
Frames of Reference

- Can you name different spaces (frames of reference) we use?



Frames of Reference

- Can you name different spaces (frames of reference) we use?

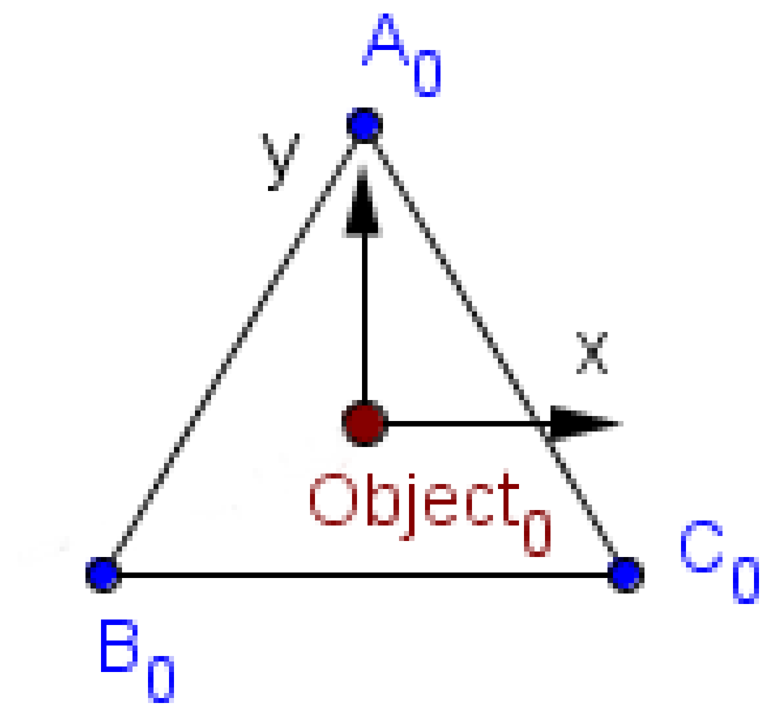


Object Space → World Space

- We model our objects in object space

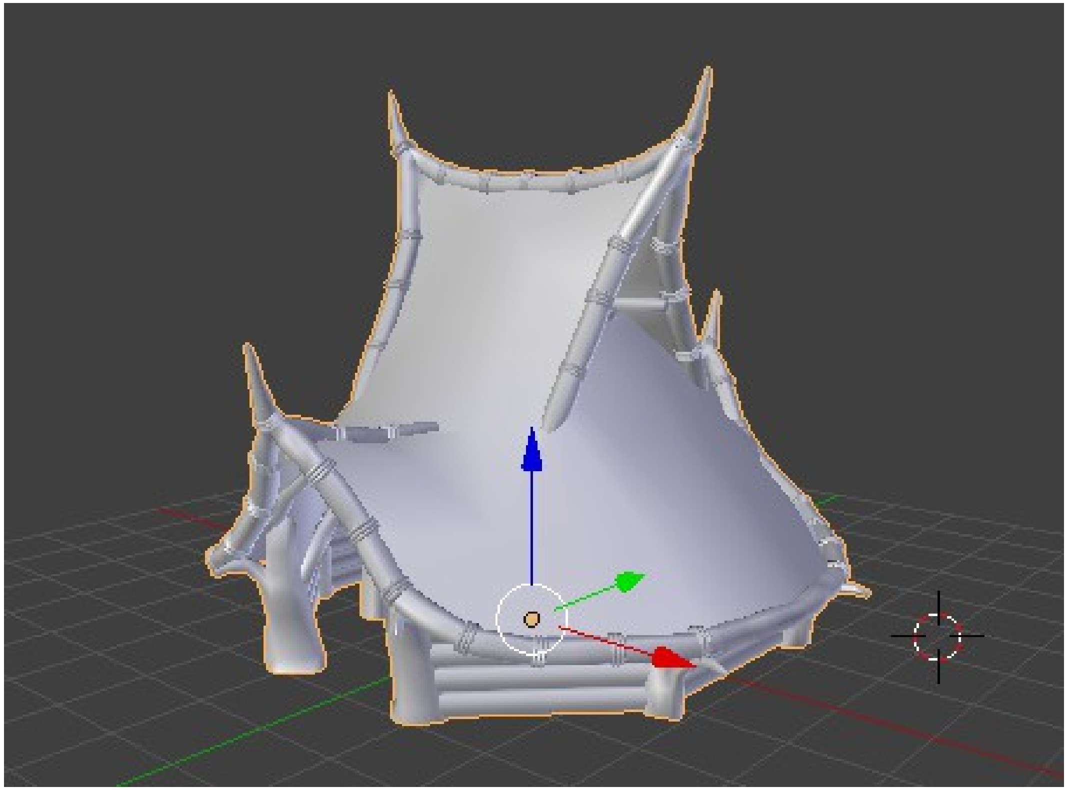
Object Space → World Space

- We model our objects in object space
  - *Symmetrically* from the origin



Object Space → World Space

- We model our objects in object space
  - *Symmetrically* from the origin
  - Up from the origin



Object Space → World Space

- We model our objects in object space
  - *Symmetrically* from the origin
  - Up from the origin
- We position, orient and scale our object with the **model matrix**, thus creating the world space!

Object Space → World Space

- We model our objects in object space
  - *Symmetrically* from the origin
  - Up from the origin
- We position, orient and scale our object with the **model matrix**, thus creating the world space!
- World space is like the root node in the scene graph

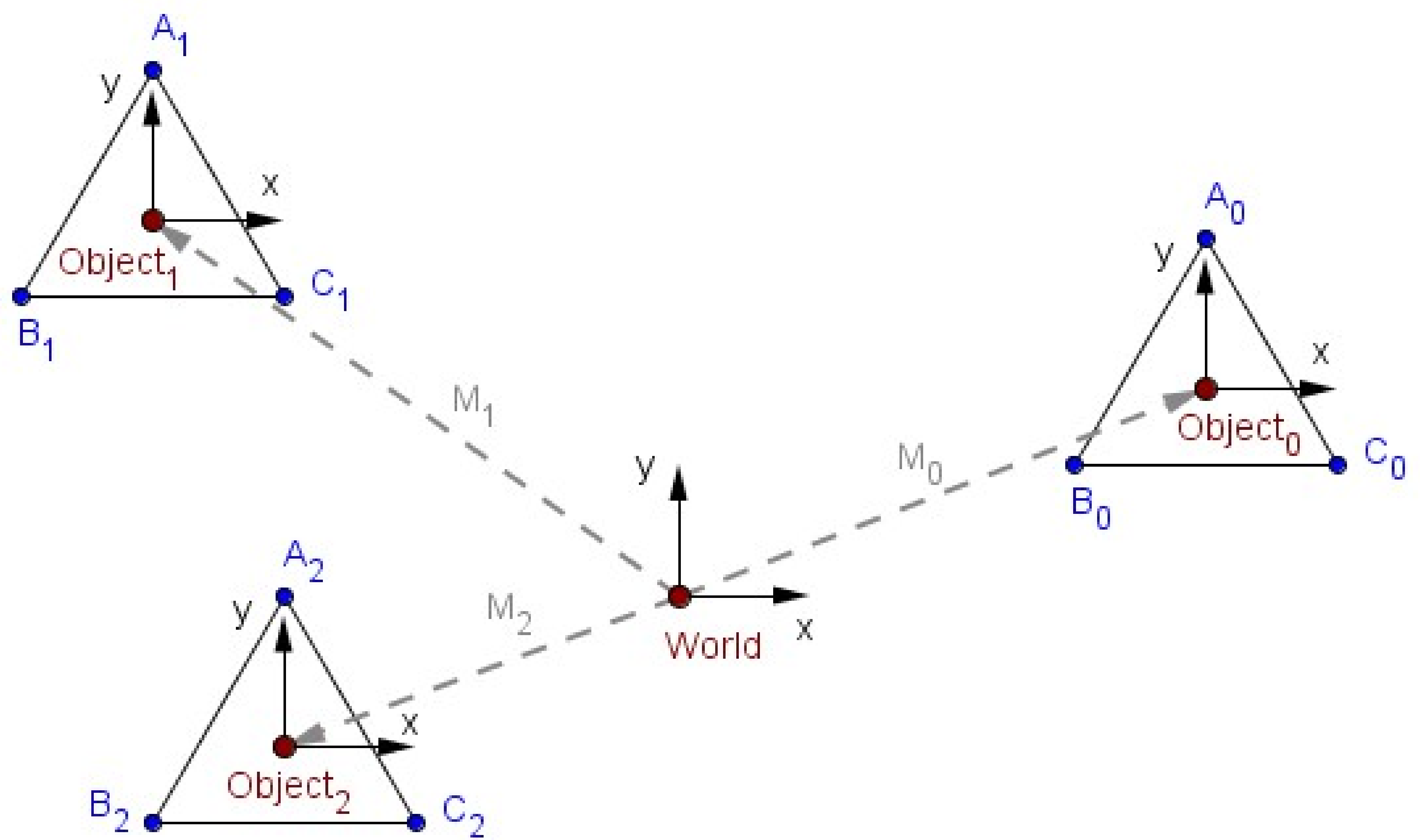
Object Space → World Space

- We model our objects in object space
  - *Symmetrically* from the origin
  - Up from the origin
- We position, orient and scale our object with the **model matrix**, thus creating the world space!
- World space is like the root node in the scene graph:
  - Origin defined by the identity transformation

Object Space → World Space

- We model our objects in object space
  - *Symmetrically* from the origin
  - Up from the origin
- We position, orient and scale our object with the **model matrix**, thus creating the world space!
- World space is like the root node in the scene graph:
  - Origin defined by the identity transformation
  - Every child transformed relative to it

Object Space → World Space



This is what you did last week. :)

Object Space → World Space

$$\text{projectionMatrix} \cdot \text{viewMatrix} \cdot \text{modelMatrix} \cdot v$$

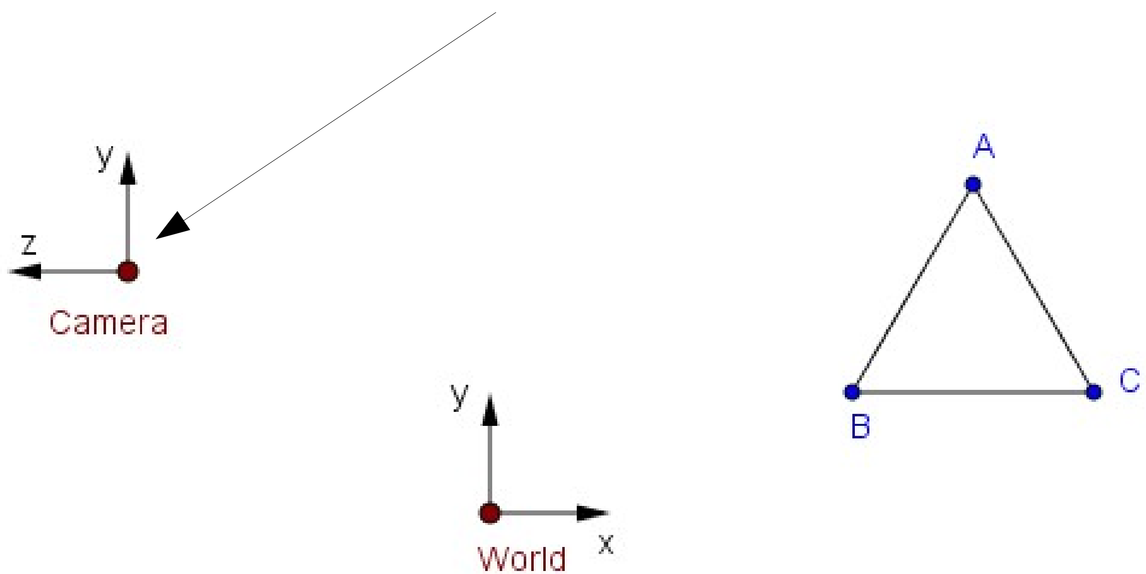
$$P \cdot V \cdot M \cdot v$$

This is what you did last week. :)

World Space → Camera Space

- We want to represent everything related to the camera (to make projection easier)

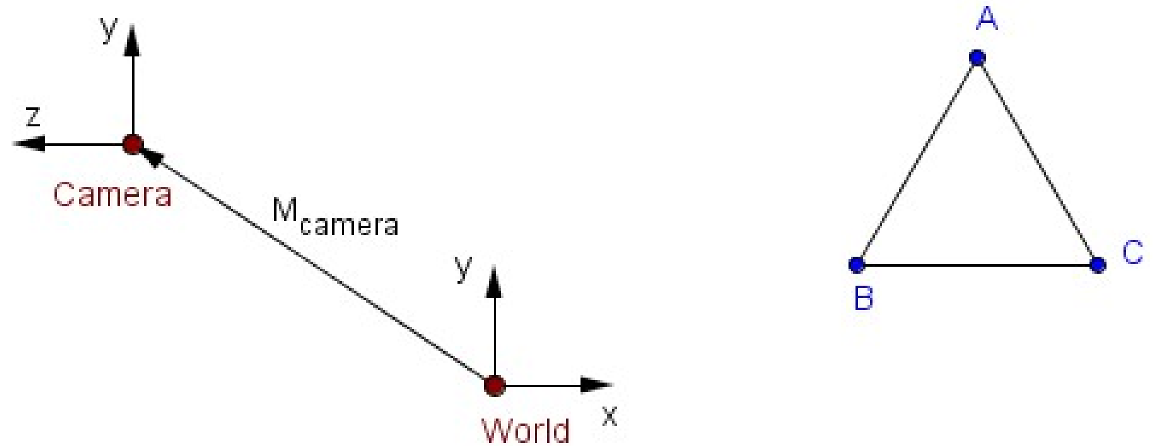
Transform so that this is the origin + basis



Origin & basis – *reeper* (est)

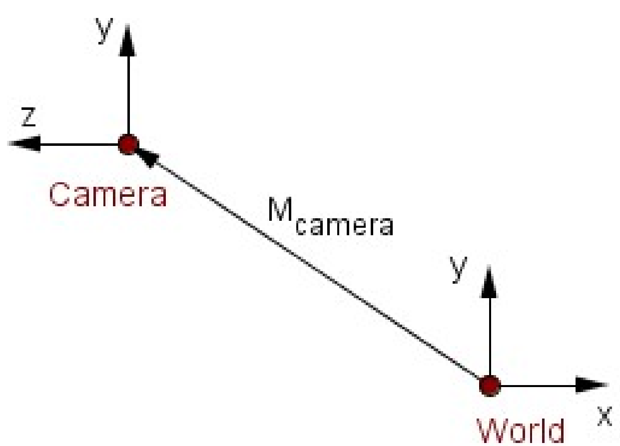
World Space → Camera Space

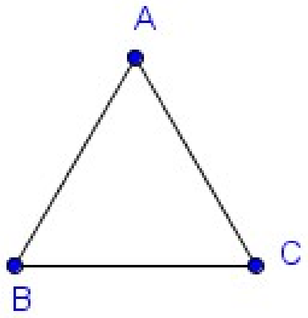
- We want to represent everything related to the camera (to make projection easier)
- We can think of the camera as another object in the scene.



World Space → Camera Space

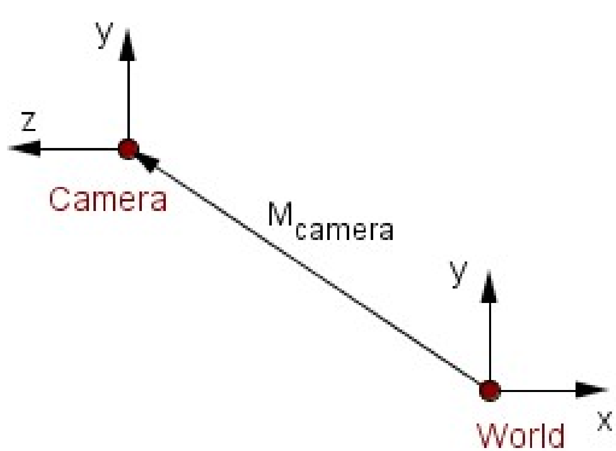
- We want to represent everything related to the camera (to make projection easier)
- We can think of the camera as another object in the scene.
  - It has its own **rotation** and **position**.

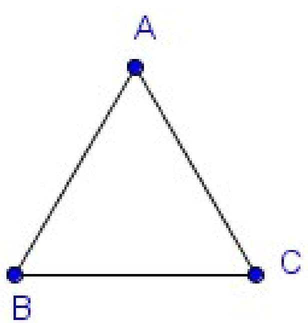




World Space → Camera Space

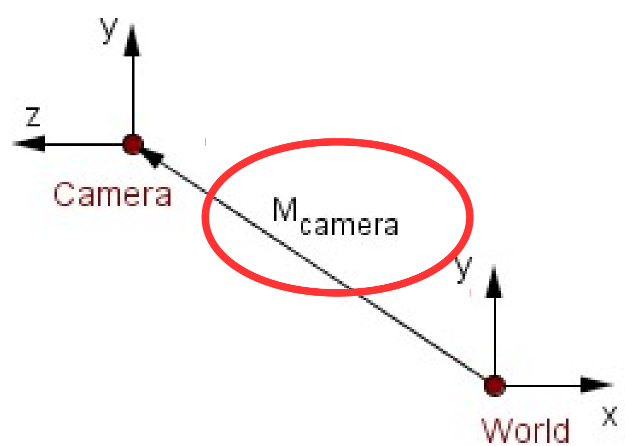
- We want to represent everything related to the camera (to make projection easier)
- We can think of the camera as another object in the scene.
    - It has its own **rotation** and **position**.
    - Scale is not really relevant for the camera.

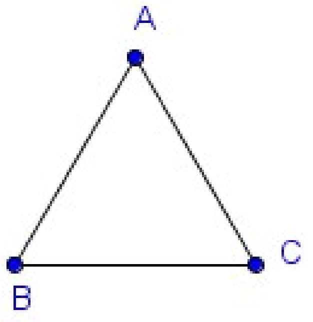




World Space → Camera Space

- Assume that we have a camera's model transformation matrix:

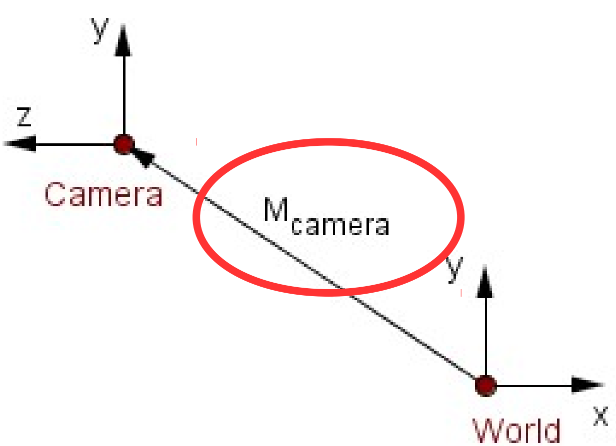


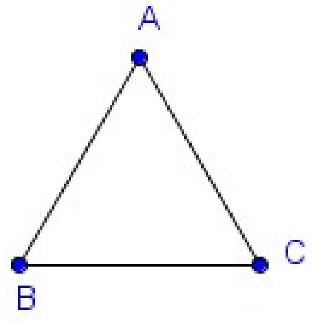


World Space → Camera Space

- Assume that we have a camera's model transformation matrix:

$$M_{camera} = \begin{pmatrix} right_x & up_x & back_x & pos_x \\ right_y & up_y & back_y & pos_y \\ right_z & up_z & back_z & pos_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$





World Space → Camera Space

- Assume that we have a camera's model transformation matrix:

$$M_{camera} = \begin{pmatrix} right_x & up_x & back_x & pos_x \\ right_y & up_y & back_y & pos_y \\ right_z & up_z & back_z & pos_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Remember that the columns are the transformed standard basis...

World Space → Camera Space

- Assume that we have a camera's model transformation matrix:

$$M_{camera} = \begin{pmatrix} right_x & up_x & back_x & pos_x \\ right_y & up_y & back_y & pos_y \\ right_z & up_z & back_z & pos_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Remember that the columns are the transformed standard basis...
- Can you come up with a matrix that describes our world relative to the camera?

World Space → Camera Space

- **View matrix** can be found like this:

World Space → Camera Space

- **View matrix** can be found like this:
  1) Camera's linear transform. is an orthonormal matrix

$$M_{camera} = \begin{pmatrix} right_x & up_x & back_x & pos_x \\ right_y & up_y & back_y & pos_y \\ right_z & up_z & back_z & pos_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

World Space → Camera Space

- **View matrix** can be found like this:
  1) Camera's linear transform. is an orthonormal matrix
  2) Transpose it to find the inverse

$$\begin{pmatrix} right_x & up_x & back_x \\ right_y & up_y & back_y \\ right_z & up_z & back_z \end{pmatrix}^T = \begin{pmatrix} right_x & right_y & right_z \\ up_x & up_y & up_z \\ back_x & back_y & back_z \end{pmatrix}$$

World Space → Camera Space

- **View matrix** can be found like this:
  1) Camera's linear transform. is an orthonormal matrix
  2) Transpose it to find the inverse
  3) Camera's translation can be inverted by negation

$$-\begin{pmatrix} pos_x \\ pos_y \\ pos_z \end{pmatrix} = \begin{pmatrix} -pos_x \\ -pos_y \\ -pos_z \end{pmatrix}$$

$$\begin{pmatrix} right_x & right_y & right_z \\ up_x & up_y & up_z \\ back_x & back_y & back_z \end{pmatrix}$$

World Space → Camera Space

- **View matrix** can be found like this:
  4) Put the two inverse transformations together in the opposite order

$$V = \begin{pmatrix} right_x & right_y & right_z & 0 \\ up_x & up_y & up_z & 0 \\ back_z & back_y & back_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -pos_x \\ 0 & 1 & 0 & -pos_y \\ 0 & 0 & 1 & -pos_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

World Space → Camera Space

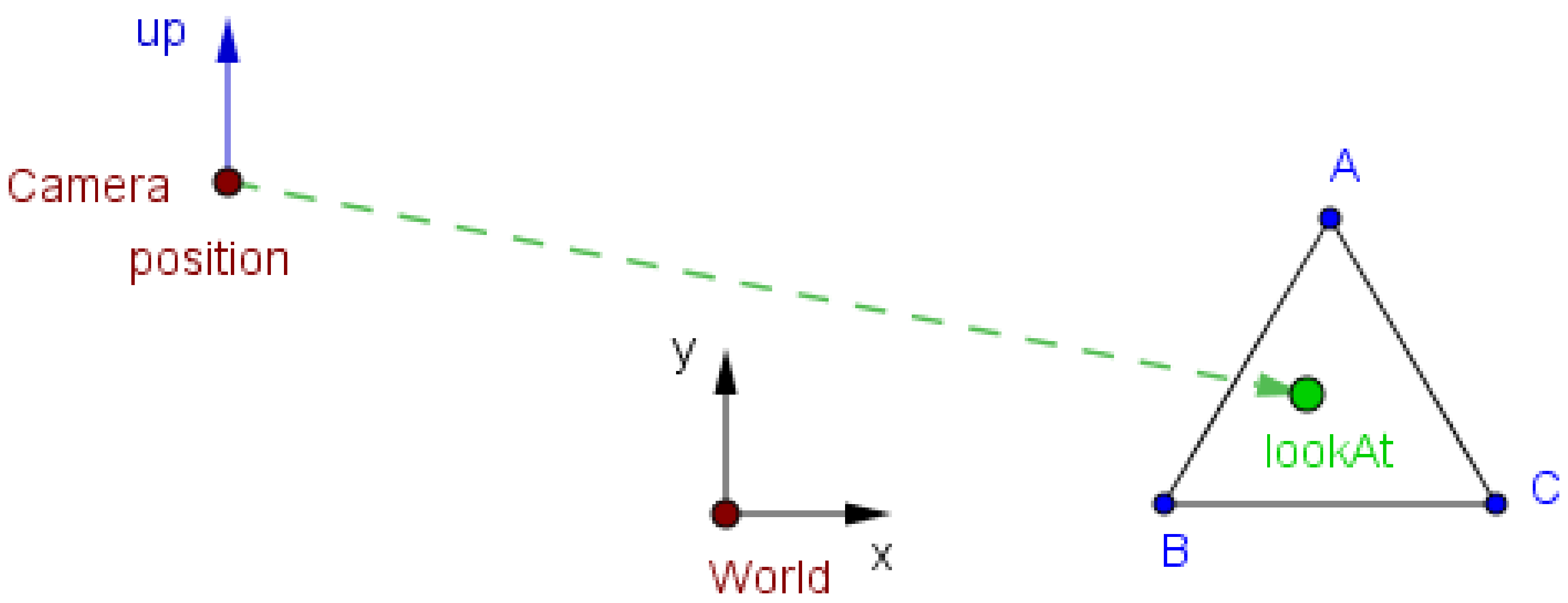
- **View matrix** can be found like this:

$$V = \begin{pmatrix} right_x & right_y & right_z & 0 \\ up_x & up_y & up_z & 0 \\ back_z & back_y & back_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -pos_x \\ 0 & 1 & 0 & -pos_y \\ 0 & 0 & 1 & -pos_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1. Transpose the rotation to inverse it
2. Negate the translation to inverse it
3. Multiply together in the reverse order

World Space → Camera Space

- Usually it is more intuitive to specify the camera by its **position**; **point** it is **looking at**; and the **up-vector**



World Space → Camera Space

- Usually it is more intuitive to specify the camera by its **position**; **point** it is looking at; and the **up-vector**

Three.js:

```
camera.position.set(x, y, z);
camera.up.set(upX, upY, upZ);
camera.lookAt(point);
```

World Space → Camera Space

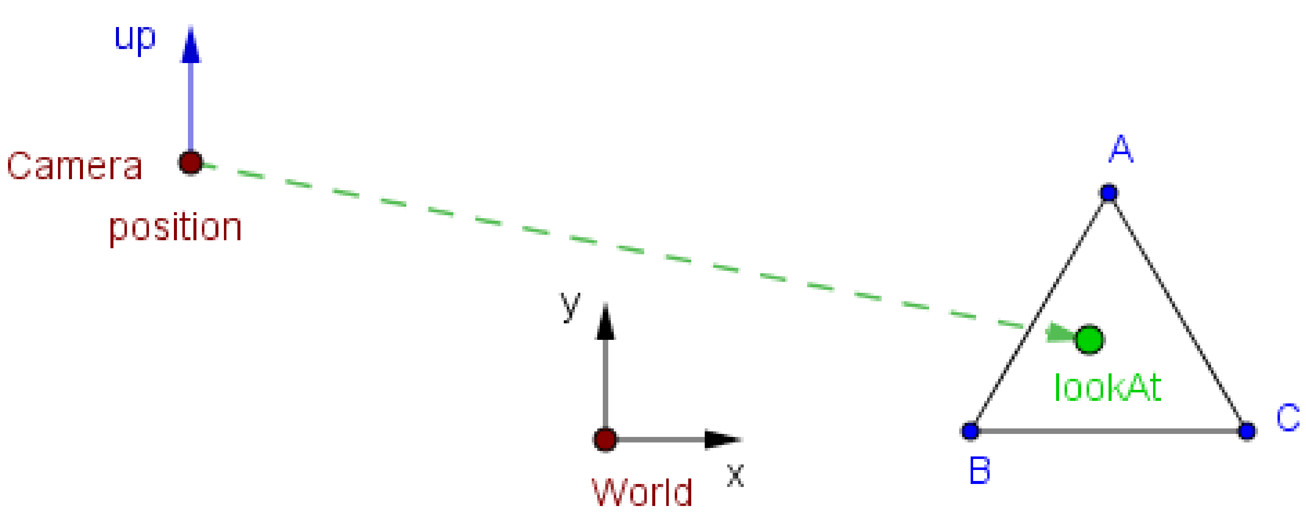
- Usually it is more intuitive to specify the camera by its **position**; **point** it is looking at; and the **up-vector**

OpenGL:

```
glm::mat4 view = glm::lookAt(
   glm::vec3(x, y, z),
   glm::vec3(pX, pY, pZ),
   glm::vec3(upX, upY, upZ)
);
```

World Space → Camera Space

- Usually it is more intuitive to specify the camera by its **position**; **point** it is **looking at**; and the **up-vector**
- The up-vector may not be the same as the y-direction of camera's space.
  It just gives a rough orientation.



World Space → Camera Space

- Using the lookAt() command parameters, how to find the correct matrix?
- What do we have and what do we need?

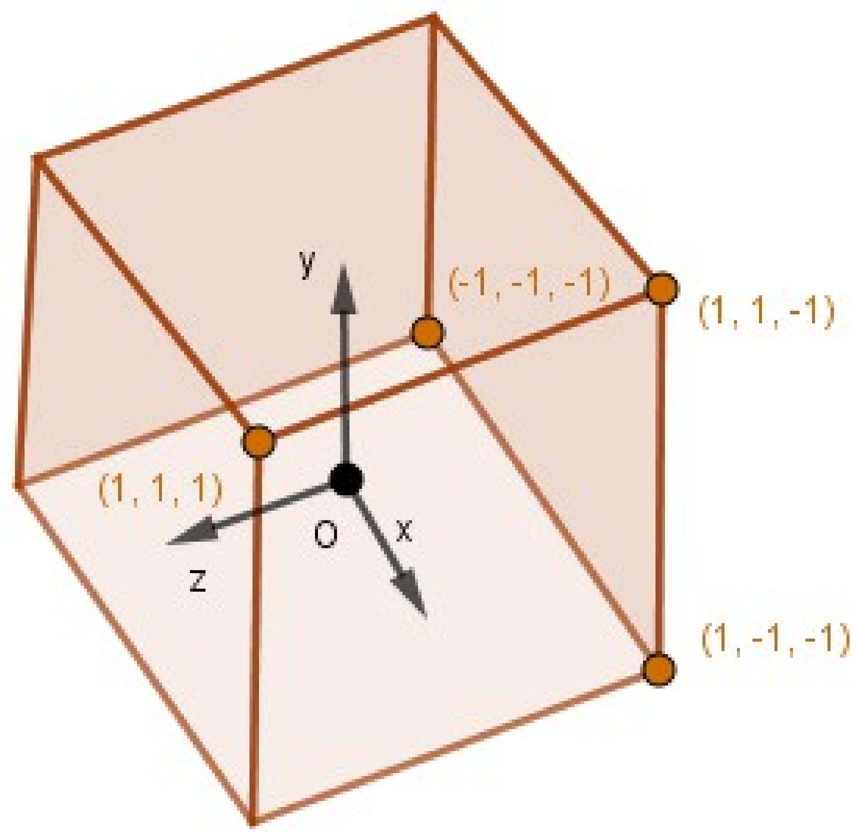
World Space → Camera Space

$$\text{projectionMatrix} \cdot \text{viewMatrix} \cdot \text{modelMatrix} \cdot v$$

$$P \cdot V \cdot M \cdot v$$

Camera Space → ND Space

- For the **normalized device space**, we transform the view frustum into a cube $[-1, 1]^3$.



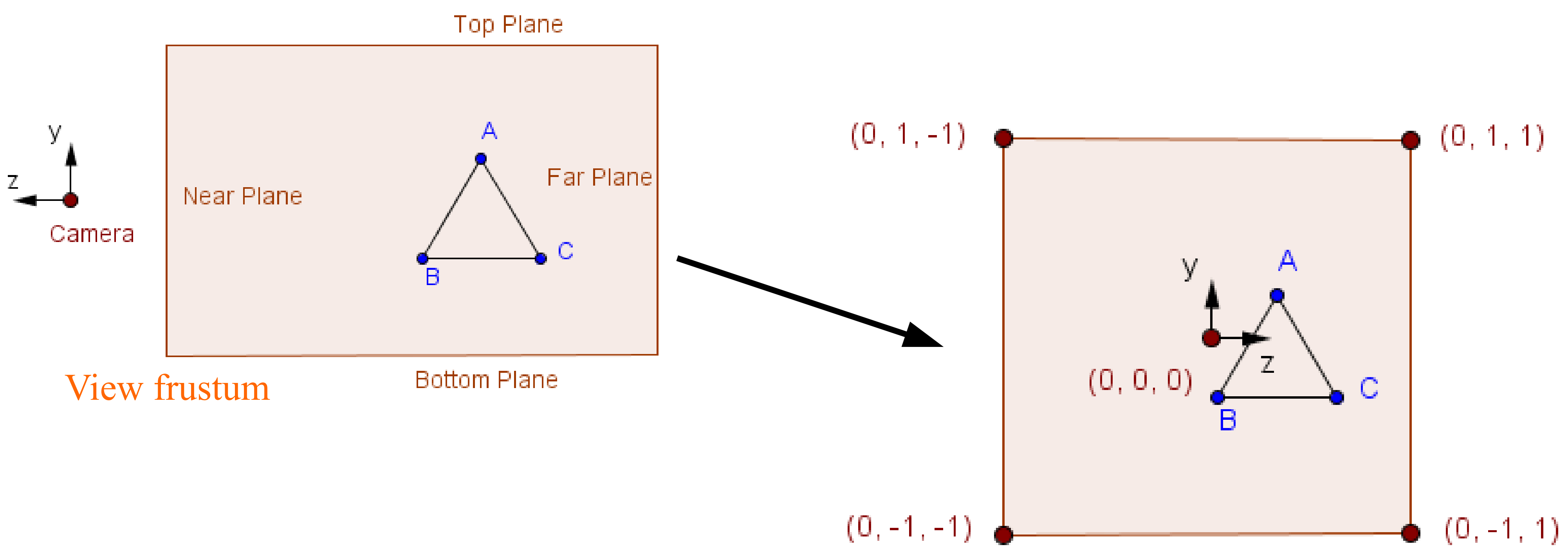
Canonical view volume

Frustum – *tüvipüramiid* (est)

Camera Space → ND Space

- For the **normalized device space**, we transform the view frustum into a cube $[-1, 1]^3$.

Orthographic

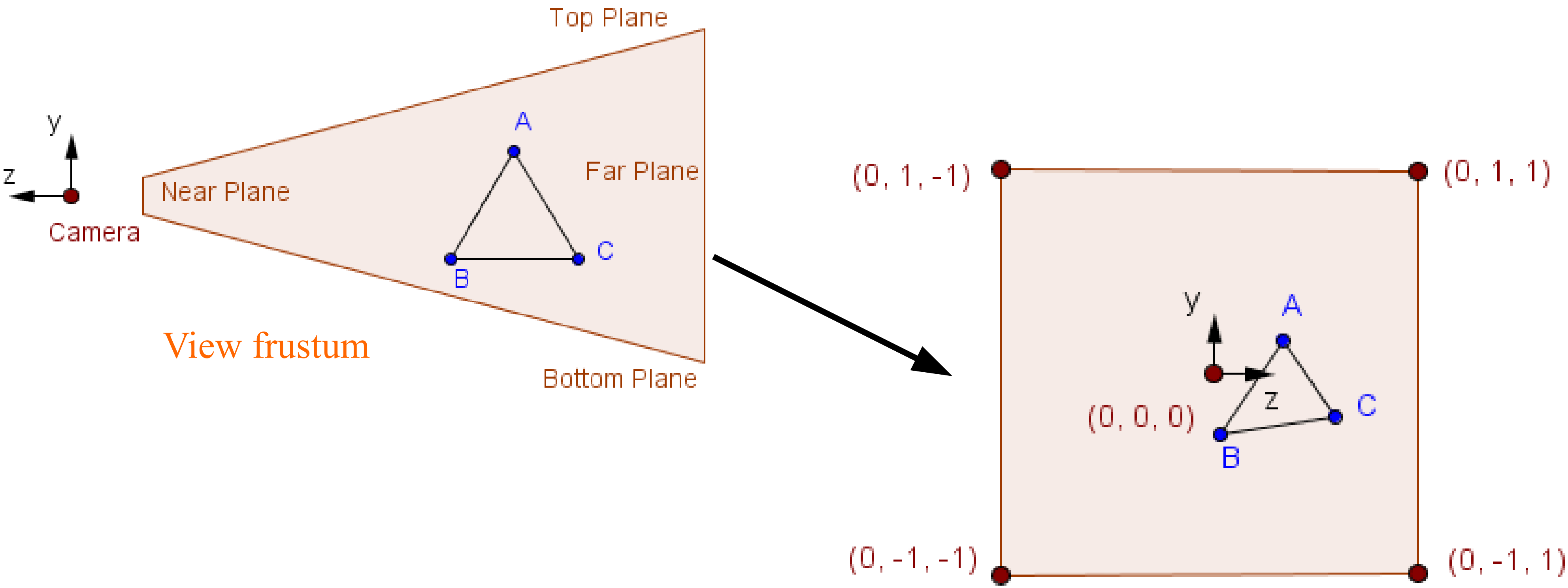


Slices from x=0 plane

Camera Space → ND Space

- For the **normalized device space**, we transform the view frustum into a cube $[-1, 1]^3$.

Perspective

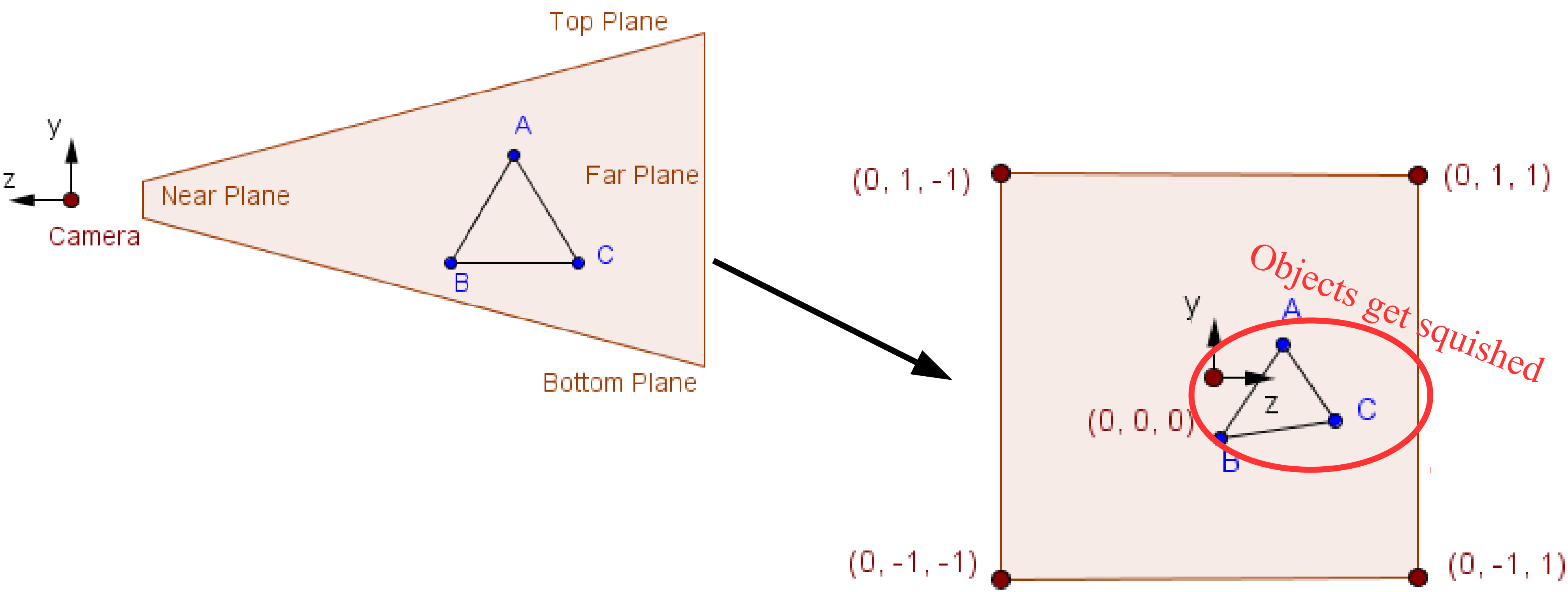


Slices from x=0 plane

Camera Space → ND Space

- For the **normalized device space**, we transform the view frustum into a cube $[-1, 1]^3$.

Perspective



See: http://tume-maailm.pri.ee/ylikool/CG/examples/perspectiveProjection/

Camera Space → ND Space

- For the **normalized device space**, we transform the view frustum into a cube $[-1, 1]^3$.
- We want to flip the *z*-axis, because our near and far planes are positive values.

Camera Space → ND Space

- For the **normalized device space**, we transform the view frustum into a cube $[-1, 1]^3$.
- We want to flip the z axis, because our near and far planes are positive values.
- This is the job for the **projection matrix** together with the **point normalization**.

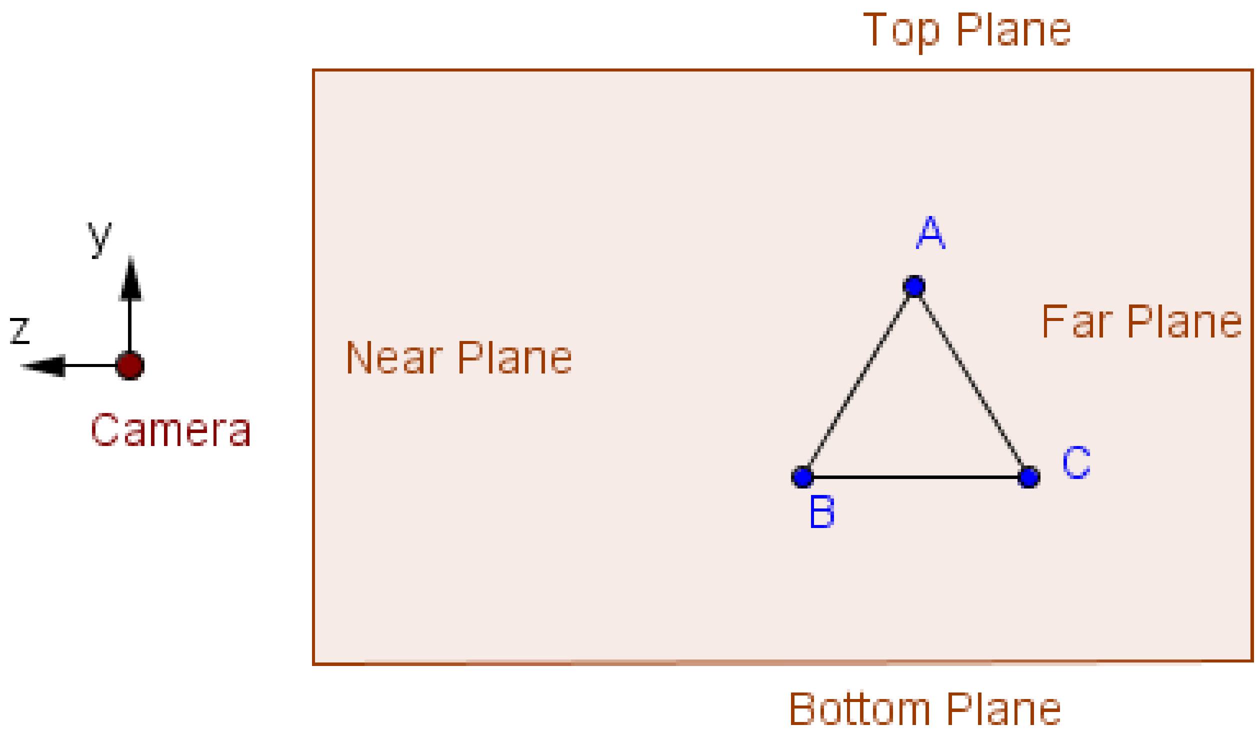
Camera Space → ND Space

$$\text{projectionMatrix} \cdot \text{viewMatrix} \cdot \text{modelMatrix} \cdot v$$

$$P \cdot V \cdot M \cdot v$$

Orthographic Projection

- We define our view volume with the values for **left**, **right**, **top**, **bottom**, **near** and **far** planes.



Orthographic Projection

- We define our view volume with the values for **left**, **right**, **top**, **bottom**, **near** and **far** planes.

```
OrthographicCamera( left, right, top, bottom, near, far )

left — Camera frustum left plane.
right — Camera frustum right plane.
top — Camera frustum top plane.
bottom — Camera frustum bottom plane.
near — Camera frustum near plane.
far — Camera frustum far plane.

Together these define the camera's viewing frustum.
```

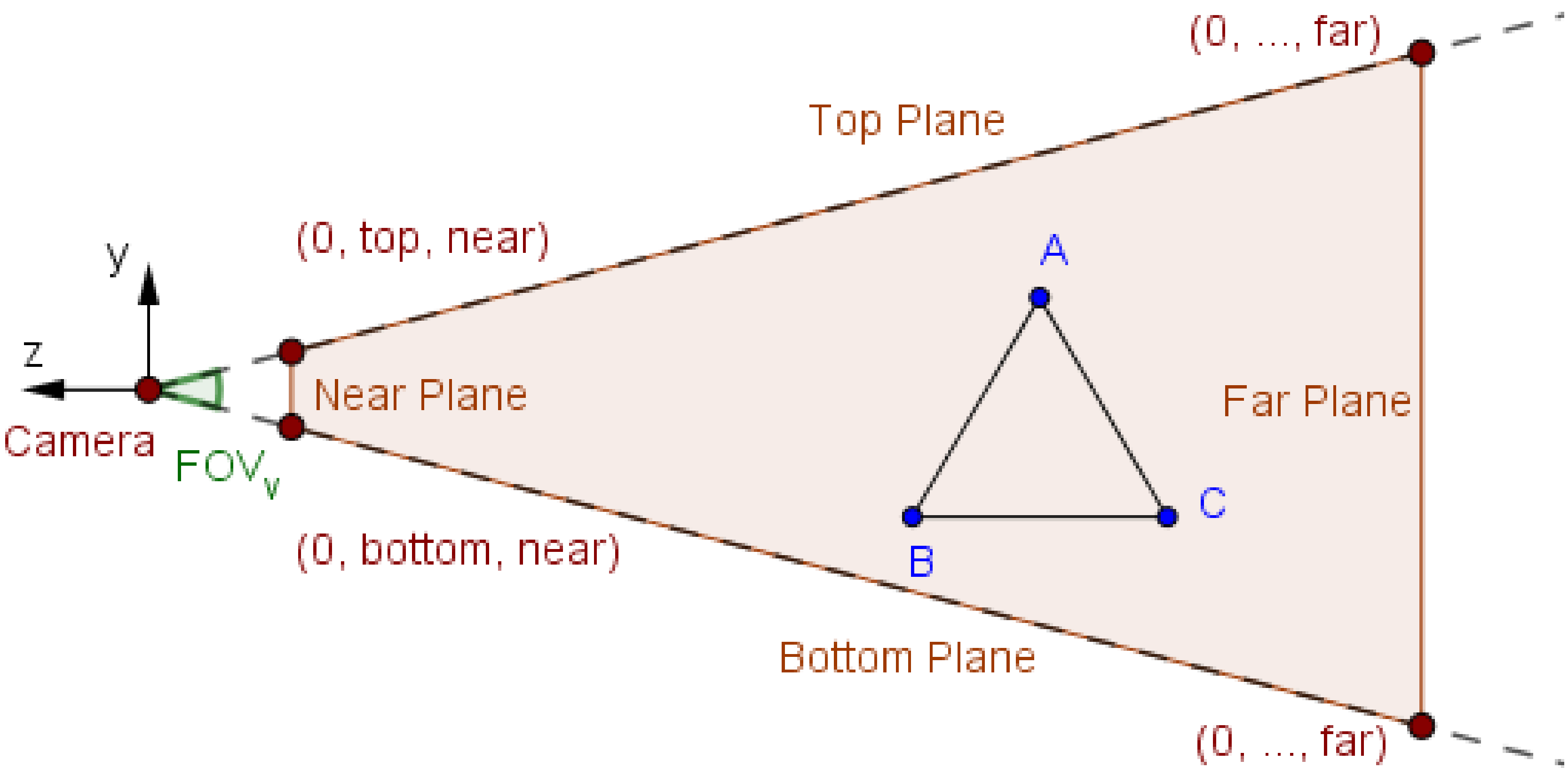
From Three.js docs.

Orthographic Projection

- We define our view volume with the values for **left**, **right**, **top**, **bottom**, **near** and **far** planes**.**
- What would be the matrix that transforms the *orthographic view volume* into the *canonical view volume* ($[-1, 1]^3$)?

Perspective Projection

- Usually defined by the **vertical angle** for the field-of-view (**FOV**), the **aspect ratio** and the **near** and **far** planes.



Perspective Projection

- Usually defined by the **vertical angle** for the field-of-view (**FOV**), the **aspect ratio** and the **near** and **far** planes.

PerspectiveCamera(fov, aspect, near, far)

fov — Camera frustum vertical field of view.

aspect — Camera frustum aspect ratio.

near — Camera frustum near plane.

far — Camera frustum far plane.

Together these define the camera's viewing frustum.

From Three.js docs.

Perspective Projection

- Usually defined by the vertical angle for the field-of-view (**FOV**), the **aspect ratio** and the **near** and **far** planes.
- Find the ***left***, ***right***, ***top*** and ***bottom*** on the near plane, when the projection is *symmetric*?

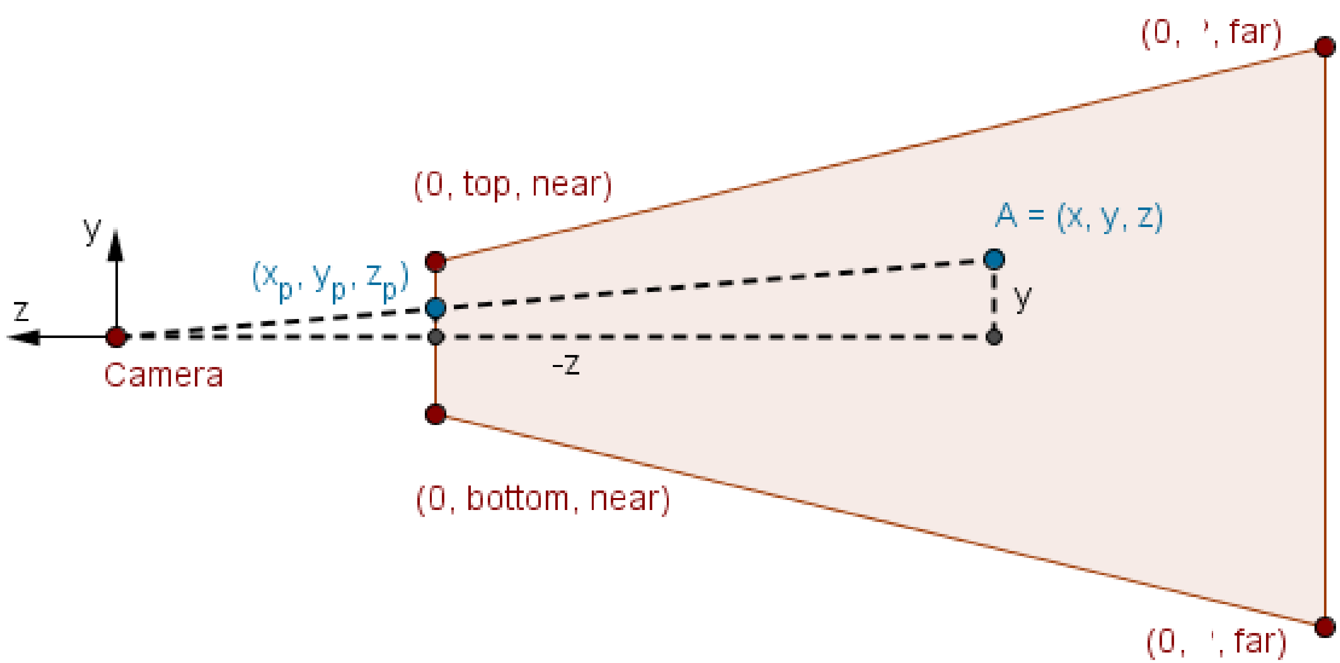
top = -bottom

left = -right

Use the image from 2 slides ago.

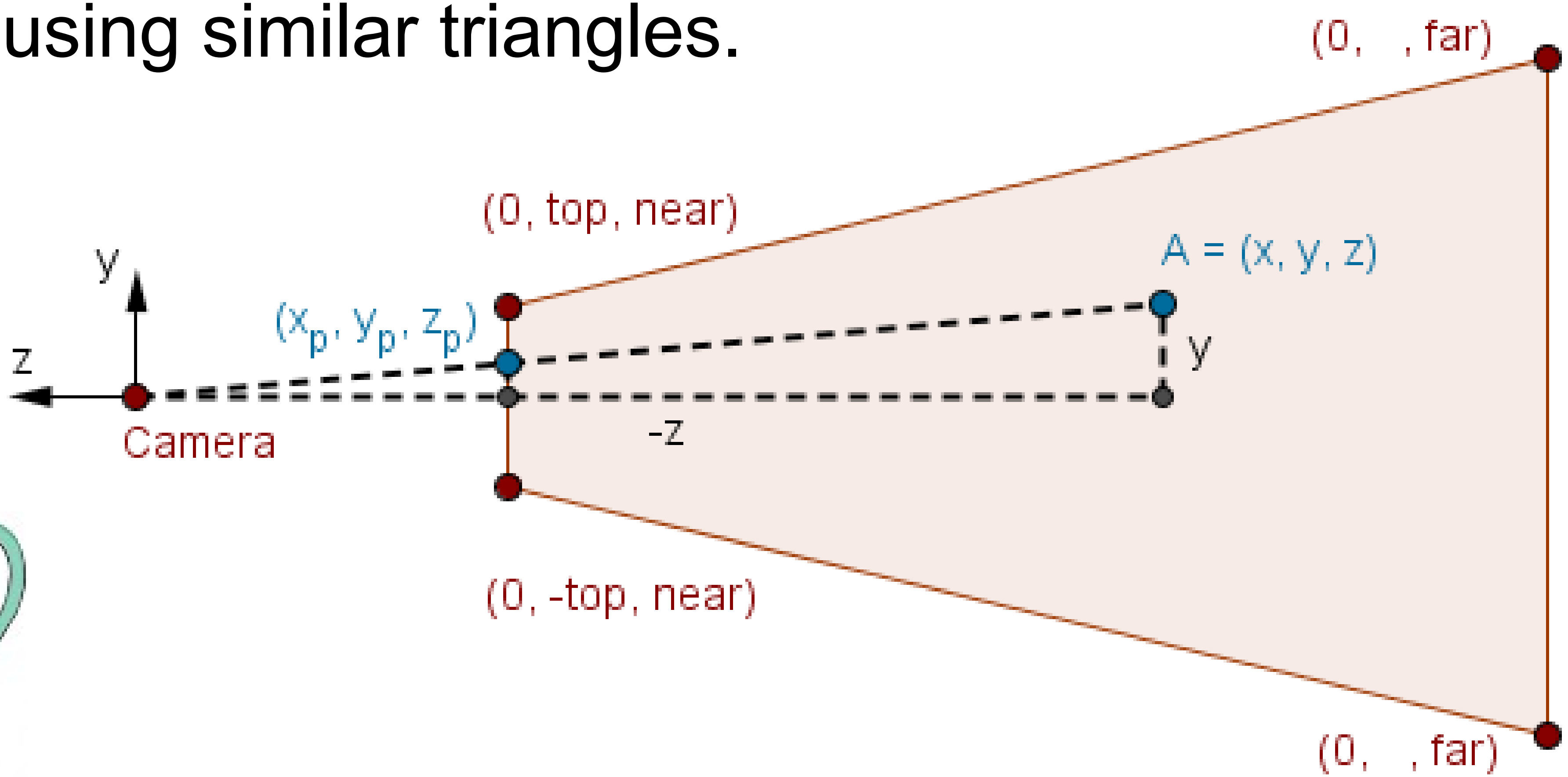
Perspective Projection

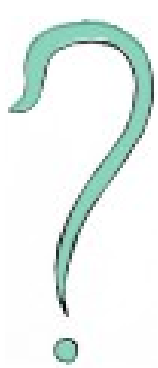
- Differently from the orthographic projection, here we have a viewer located in a single point.
- Similarly we want to find the normalized device coordinates for all points inside the view volume.



Perspective Projection

- First **find** and then **map** the x and y coordinates of the ***projected point*** to the correct range using similar triangles.





Find x_p and y_p

Perspective Projection

$$P = \begin{pmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- If the third row would be (0, 0, 1, 0), then all *z* coordinates become -1 (beacuse we found the projected coordinates on the near plane)

Perspective Projection

- We want to map the *z* value from the range [near, far] to the range [-1, 1].
- We can use scale and translation.

$$P = \begin{pmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & s & t \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective Projection

- We want to map the *z* value from the range [near, far] to the range [-1, 1], so...

$$\begin{cases} \dfrac{s \cdot near + t}{near} = -1 \\[2ex] \dfrac{s \cdot far + t}{far} = 1 \end{cases}$$

$$P = \begin{pmatrix} \dfrac{near}{right} & 0 & 0 & 0 \\ 0 & \dfrac{near}{top} & 0 & 0 \\ 0 & 0 & s & t \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Can this be solved for the *s* and *t* unknowns?

Division by *near* and *far* is because of the homogeneous division.

Hint

Perspective Projection

- After applying this matrix and doing the point normalization (dividing with *w*), you have the perspective projection.

$$P = \begin{pmatrix} \frac{near}{right} & 0 & 0 & 0 \\ 0 & \frac{near}{top} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective Projection

- When using the FOV (α) and aspect ratio (ar).

$$P = \begin{pmatrix} \dfrac{1}{ar \cdot \tan\left(\dfrac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \dfrac{1}{\tan\left(\dfrac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & -\dfrac{far + near}{far - near} & -\dfrac{2 \cdot far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Clip Space

- After the projection matrix multiplication and before the *w*-division, vertices are in a *clip space*.

Read more here: https://stackoverflow.com/a/21841924/3067608

Clip Space

- After the projection matrix multiplication and before the *w*-division, vertices are in a *clip space*.
- That is the space, where it is the most easiest to determine, which triangles need to be clipped or culled.

Read more here: https://stackoverflow.com/a/21841924/3067608

Clip Space

- After the projection matrix multiplication and before the *w*-division, vertices are in a *clip space*.
- That is the space, where it is the most easiest to determine, which triangles need to be clipped or culled.
- **Clipping** – performed when some part of the triangle is inside the view volume.

Read more here: https://stackoverflow.com/a/21841924/3067608

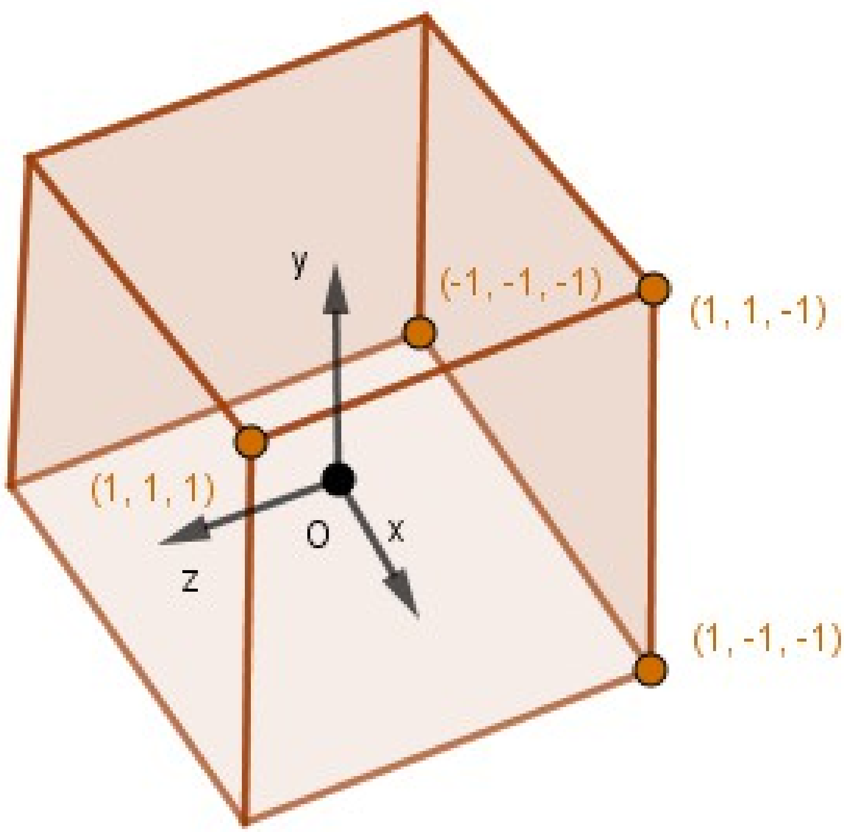
Clip Space

- After the projection matrix multiplication and before the *w*-division, vertices are in a *clip space*.
- That is the space, where it is the most easiest to determine, which triangles need to be clipped or culled.
- **Clipping** – performed when some part of the triangle is inside the view volume.
- **Culling** – performed when the triangle is not inside the view volume. Or is back-facing.

Read more here: https://stackoverflow.com/a/21841924/3067608

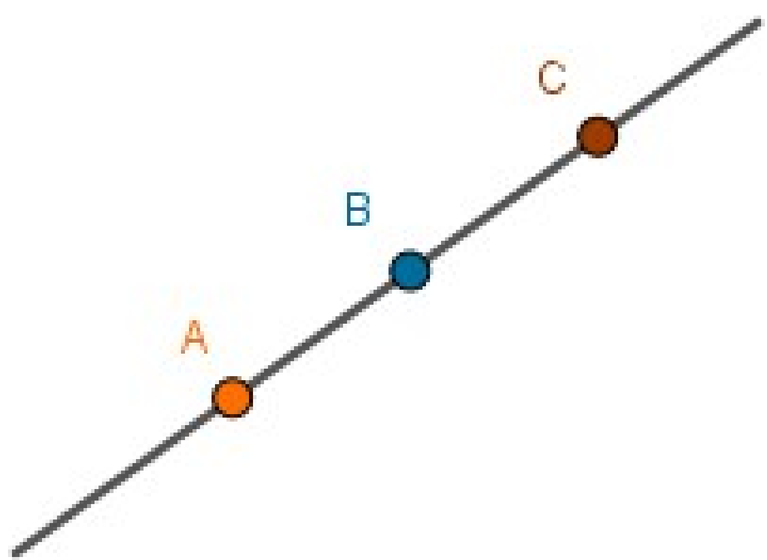
ND Space → Screen Space

- We have everthing we want to show now in the $[-1, 1]^3$ cube (normalized device space).



ND Space → Screen Space

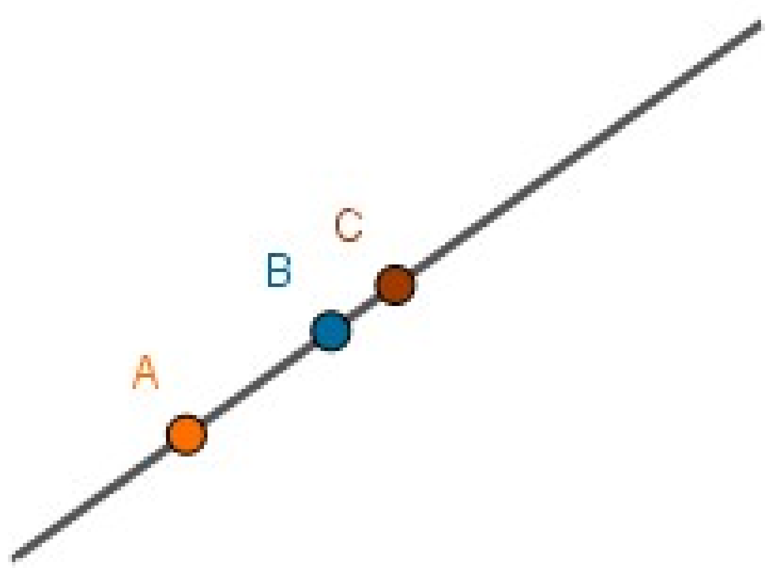
- We have everthing we want to show now in the $[-1, 1]^3$ cube (normalized device space).
- We also know the correct relative depth of the vertices.



Before the perspective projection

ND Space → Screen Space

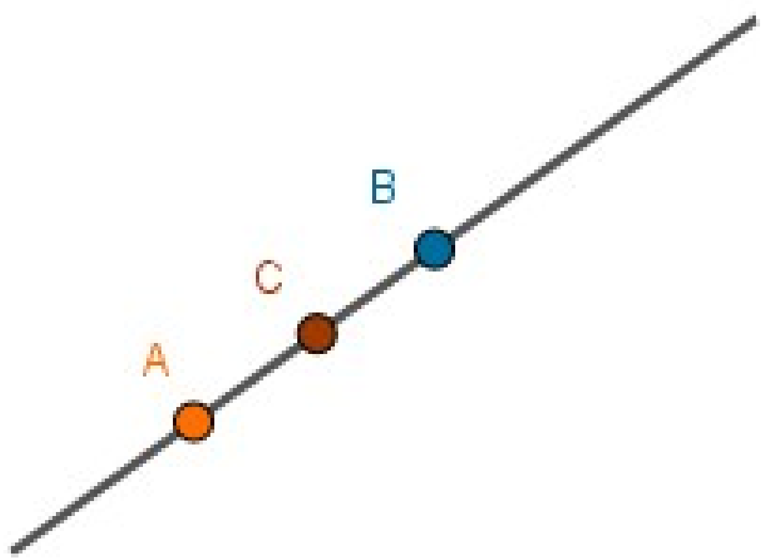
- We have everthing we want to show now in the $[-1, 1]^3$ cube (normalized device space).
- We also know the correct relative depth of the vertices.



After the perspective projection

ND Space → Screen Space

- We have everthing we want to show now in the $[-1, 1]^3$ cube (normalized device space).
- We also know the correct relative depth of the vertices.



This will not happen!

ND Space → Screen Space

- We have everthing we want to show now in the $[-1, 1]^3$ cube (normalized device space).
- We also know the correct relative depth of the vertices.
- How to know where to draw on the screen?

Come up with that matrix...

ND Space → Screen Space

- This is done for you, the screen space matrix is constructed when you specify the viewport size.

Three.js

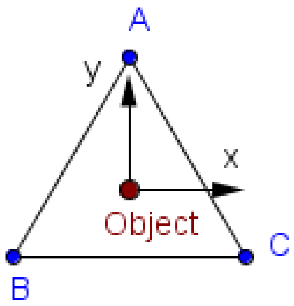
renderer = new THREE.WebGLRenderer();
renderer.setSize(**width**, **height**);

OpenGL + GLFW

win = glfwCreateWindow(**width**, **height**,
"Hello GLFW!", NULL, NULL)

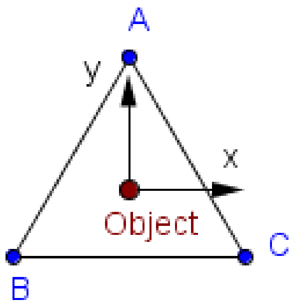
Overall

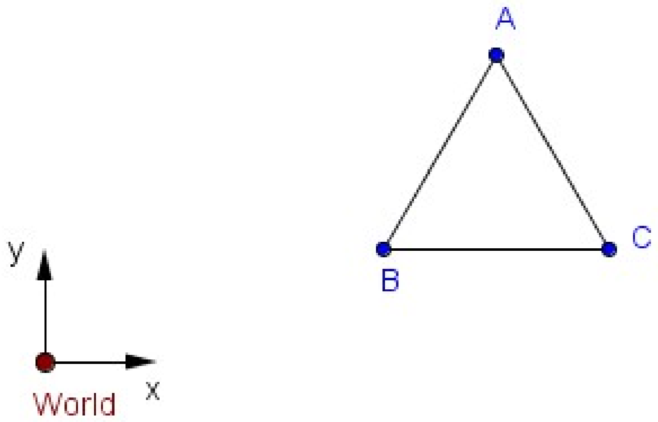
Object Space



Overall

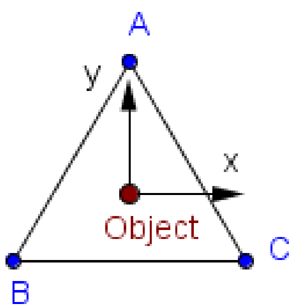
Object Space → **World Space**

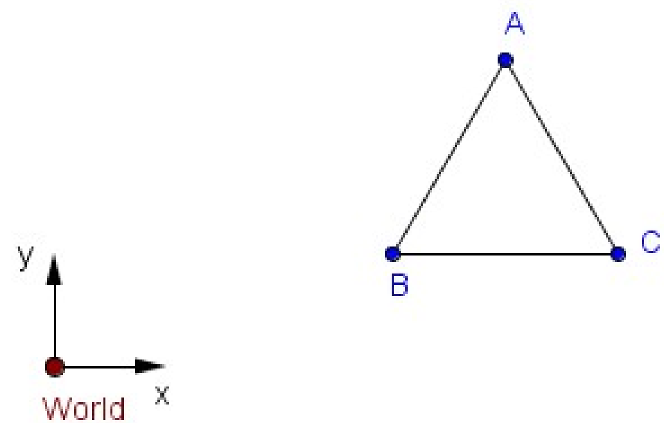




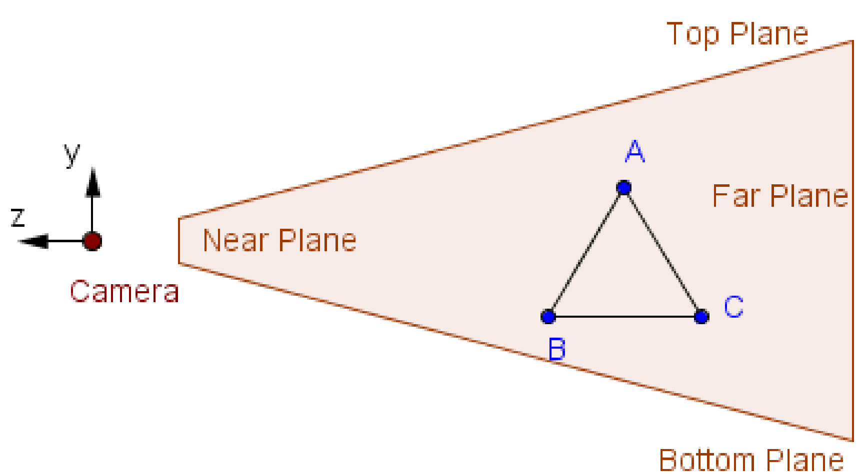
Overall

Object Space → **World Space** →



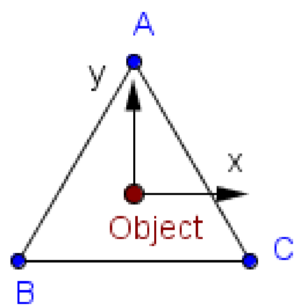


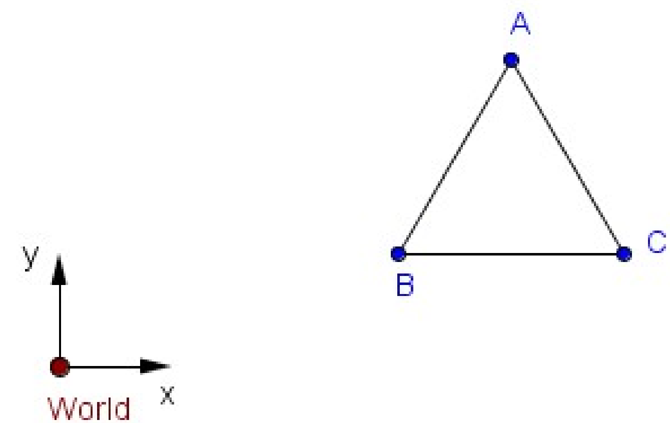
→ **Camera (View) Space**



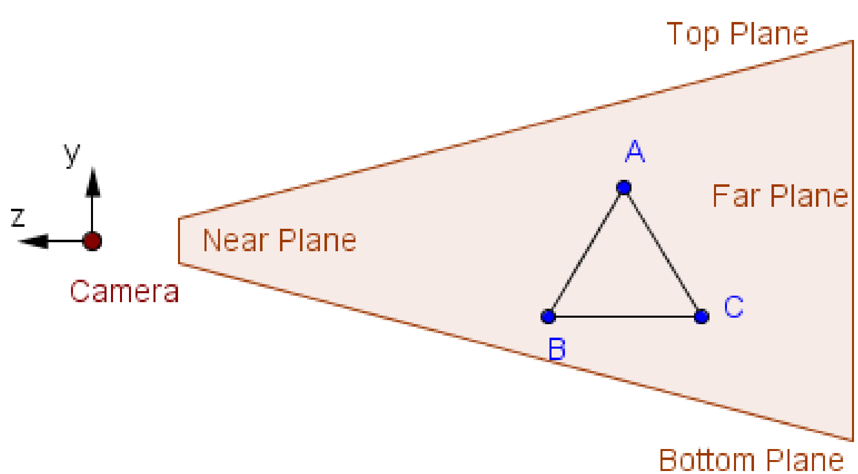
Overall

Object Space → World Space →





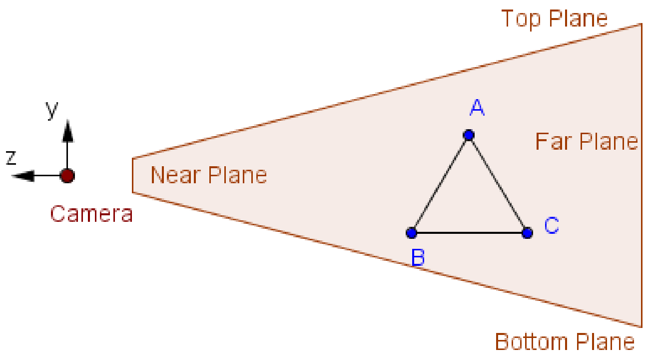
→ Camera (View) Space

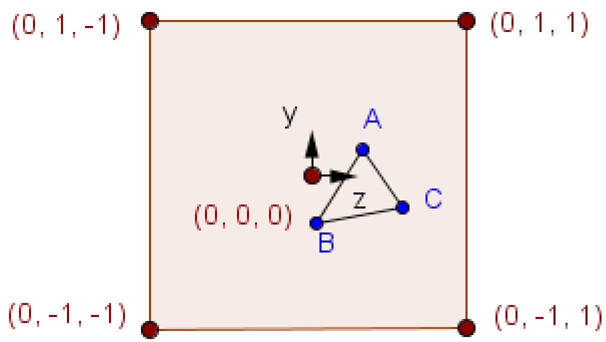


Light calculations are usually in this space!

Overall

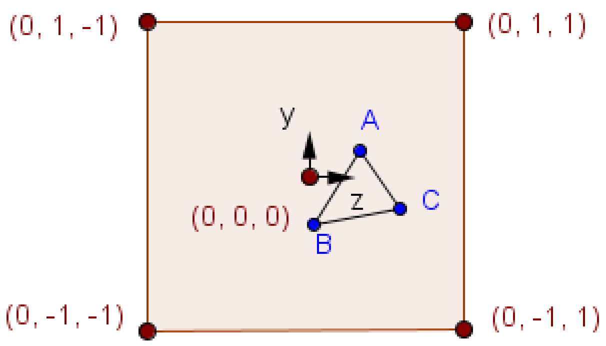
Camera (View) Space → Normalized Device Space



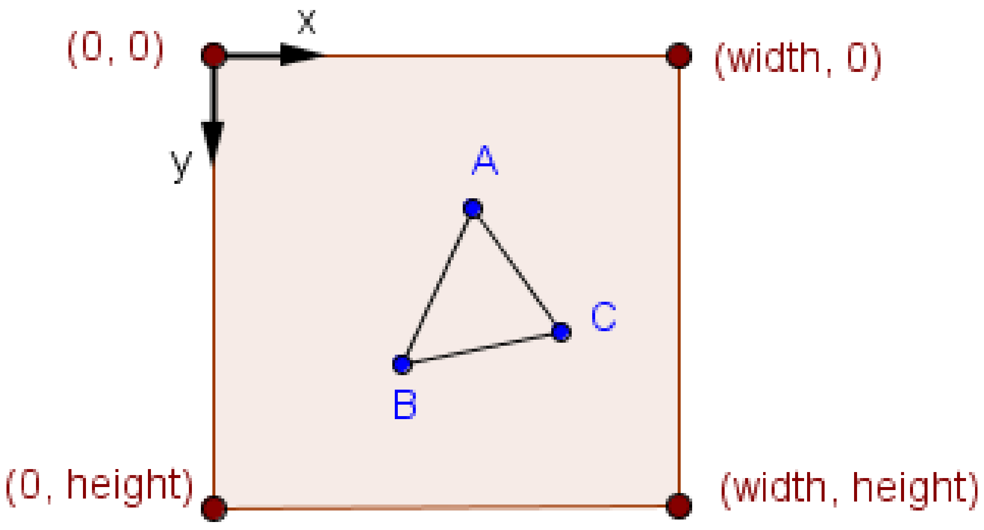


Overall

→ **Normalized Device Space** →



→ **Sceen Space**



Overall

- Vertex shader must return homogeneous coordinates in the clip space – that is in normalized device space without the *w*-division.

```
gl_Position = projection * view * model * vec4(position, 1.0);

gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);

gl_Position = modelViewProjectionMatrix * vec4(position, 1.0);
```

Overall

- Vertex shader must return homogeneous coordinates in the clip space – that is in normalized device space without the *w*-division.

```
gl_Position = projection * view * model * vec4(position, 1.0);

gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);

gl_Position = modelViewProjectionMatrix * vec4(position, 1.0);
```

- Then GPU does:
  - *w*-division
  - Screen space transformation

Additional Links

- General overview: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
- How to derive the view matrix: http://3dgep.com/understanding-the-view-matrix/
- How to derive the projection matrices: http://www.songho.ca/opengl/gl_projectionmatrix.html
- About transforming the surface normals: http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/

What was interesting for you today?

What more would you like to know?

Next time

Shading and Lighting