Computer Graphics

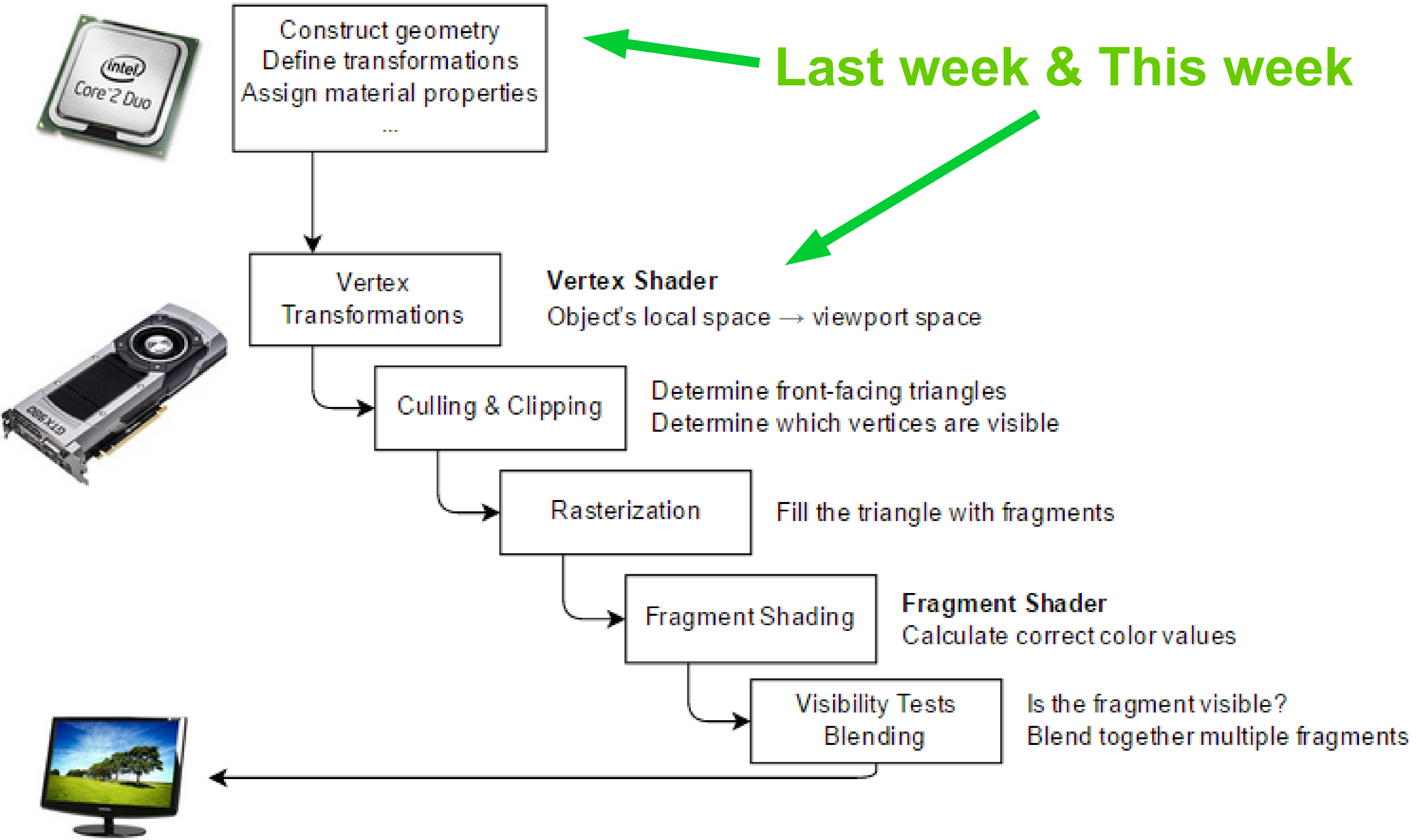
MTAT.03.015

Raimond Tunnel

TARTU ÜLIKOOL · UNIVERSITAS TARTUENSIS · 1632

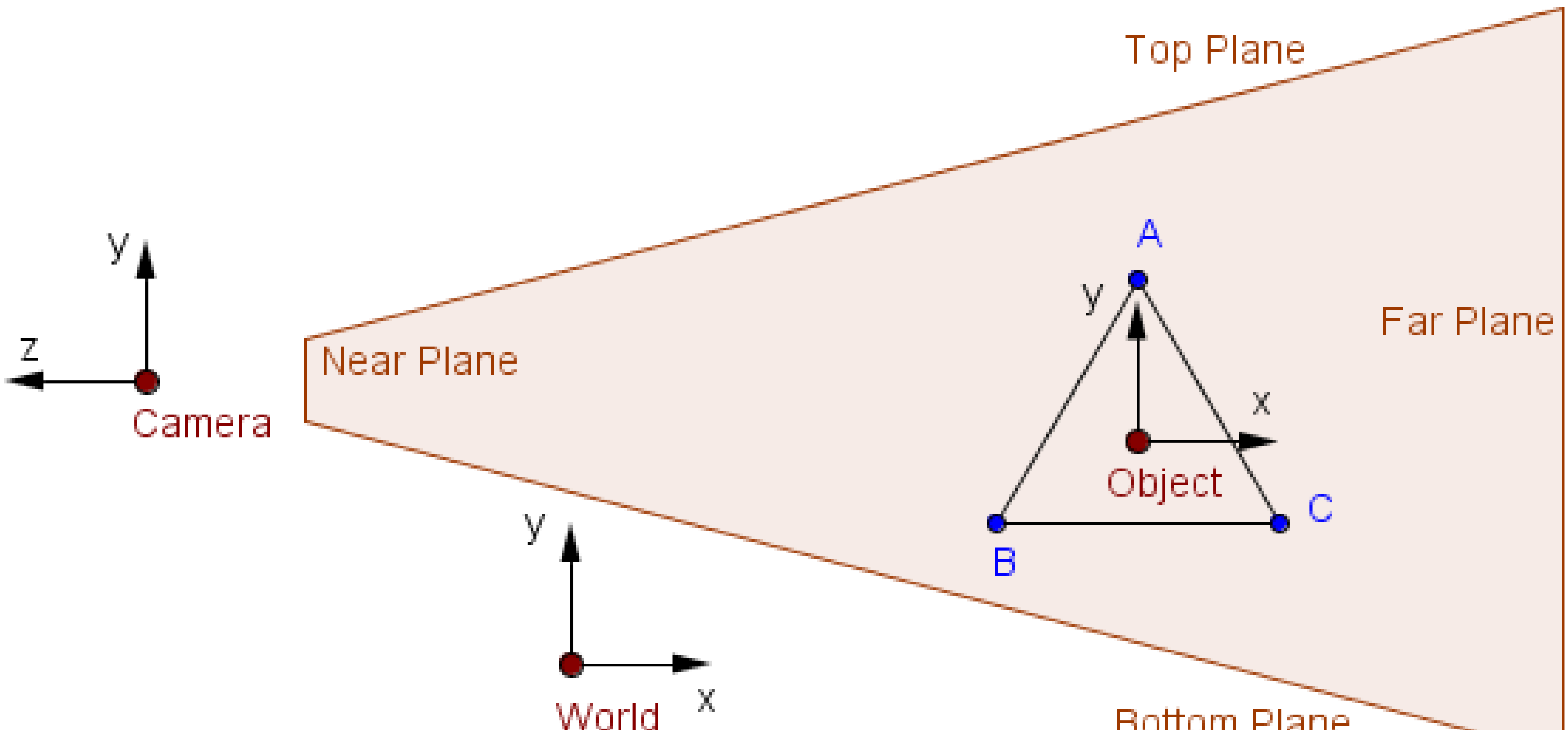
Study IT in .ee

The Road So Far...



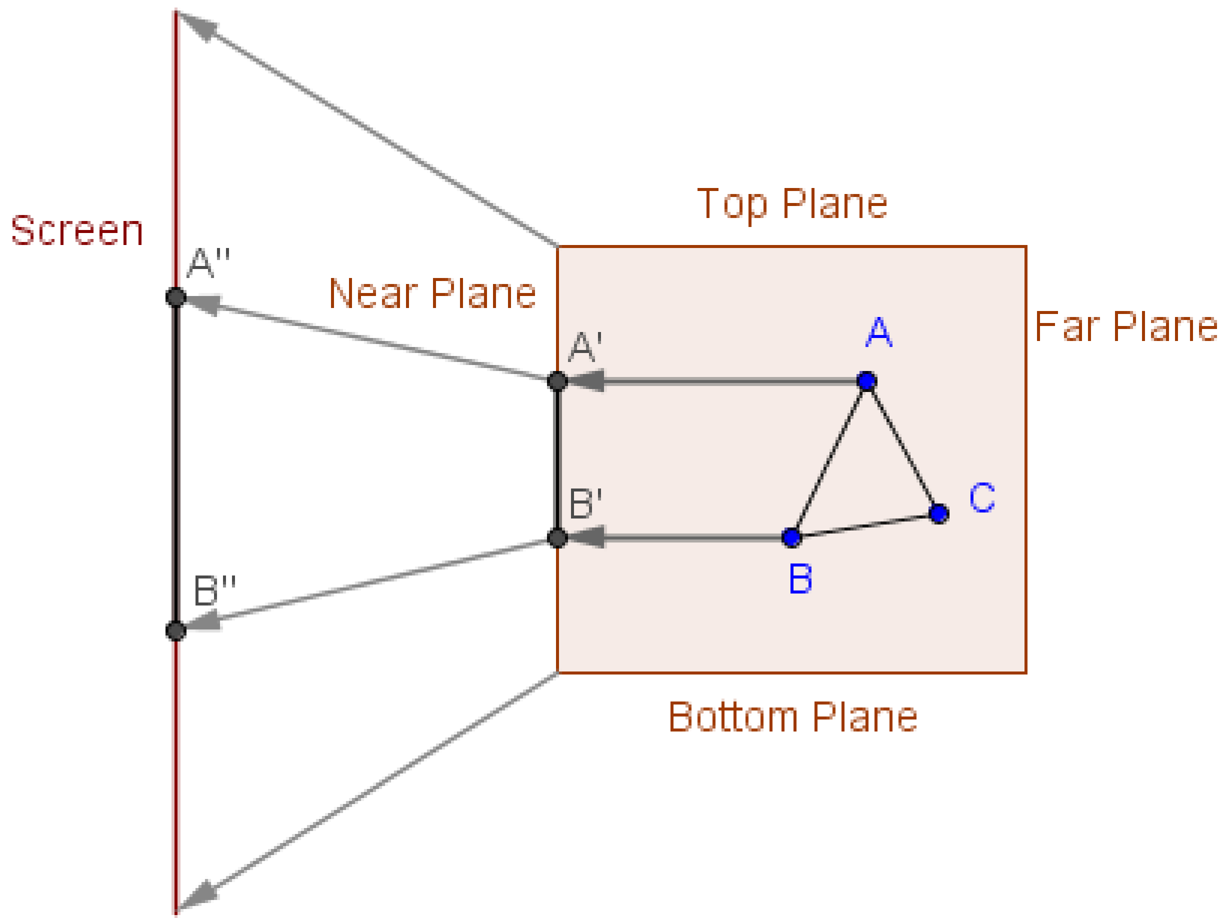
Frames of References

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



Frames of References

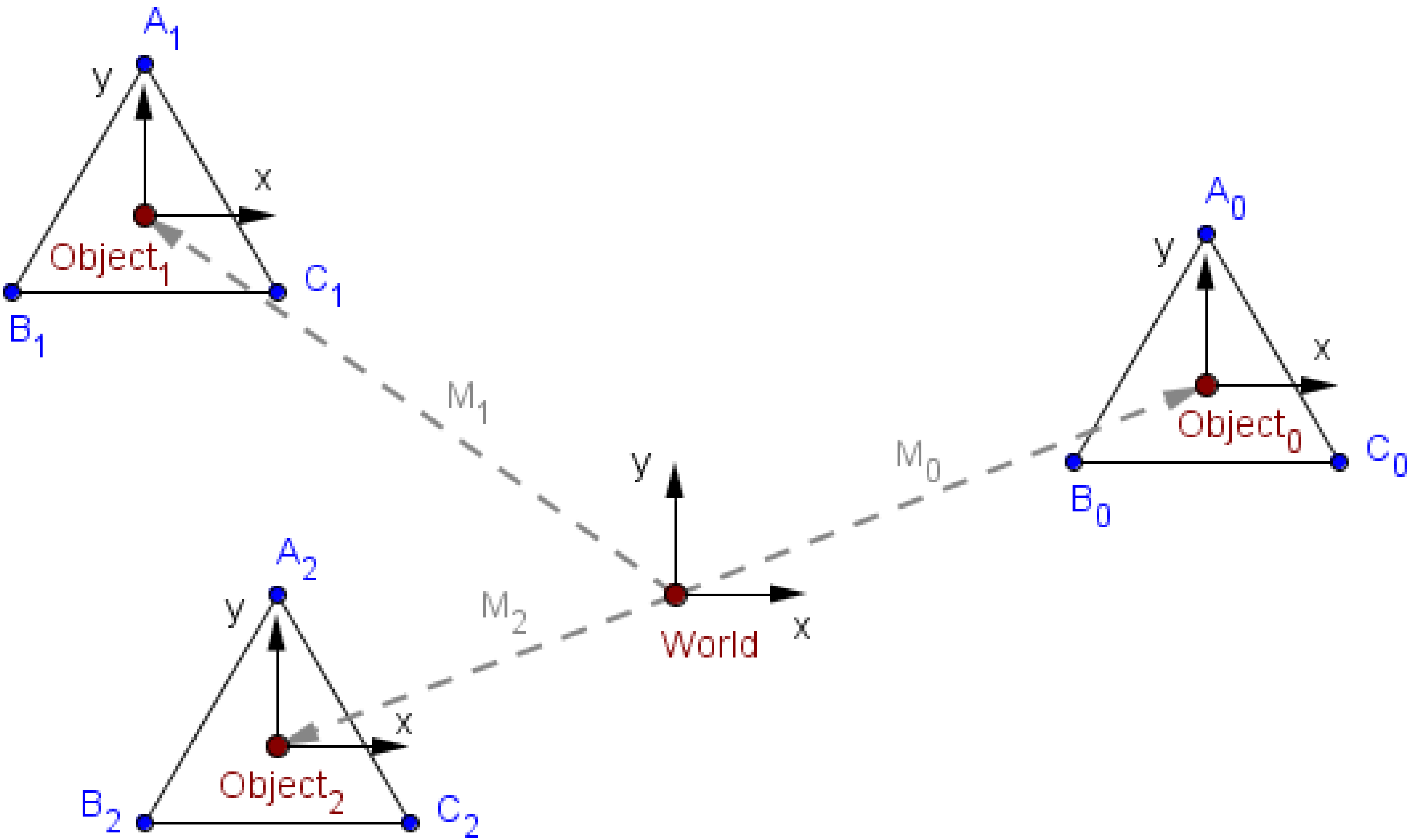
- Can you name different spaces (frames of references) we use?



Object Space → World Space

- We model our objects in object space
  - *Symmetrically* from the origin
- We position, orient and scale our object in the world space with the **model matrix**
- World space is like the root node in the scene graph
  - Located in an origin
  - Every child transformed relative to it

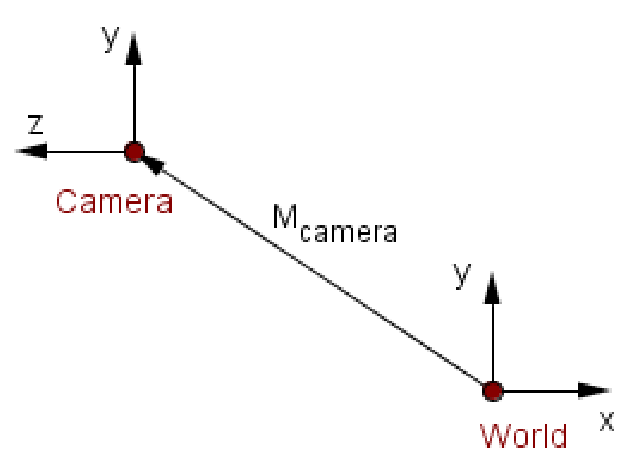
Object Space → World Space

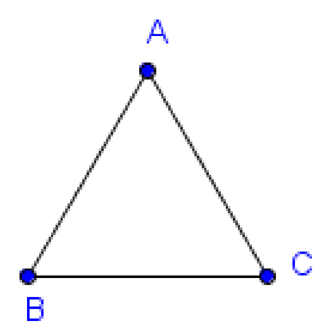


This is what you did last week. :)

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:

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

- 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:

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

- Transpose the rotation to inverse it
- Negate the translation to inverse it
- 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**
  - The up-vector may not be the same as the y-direction of camera's space. It just give a rough orientation.

Three.js:

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

OpenGL:

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

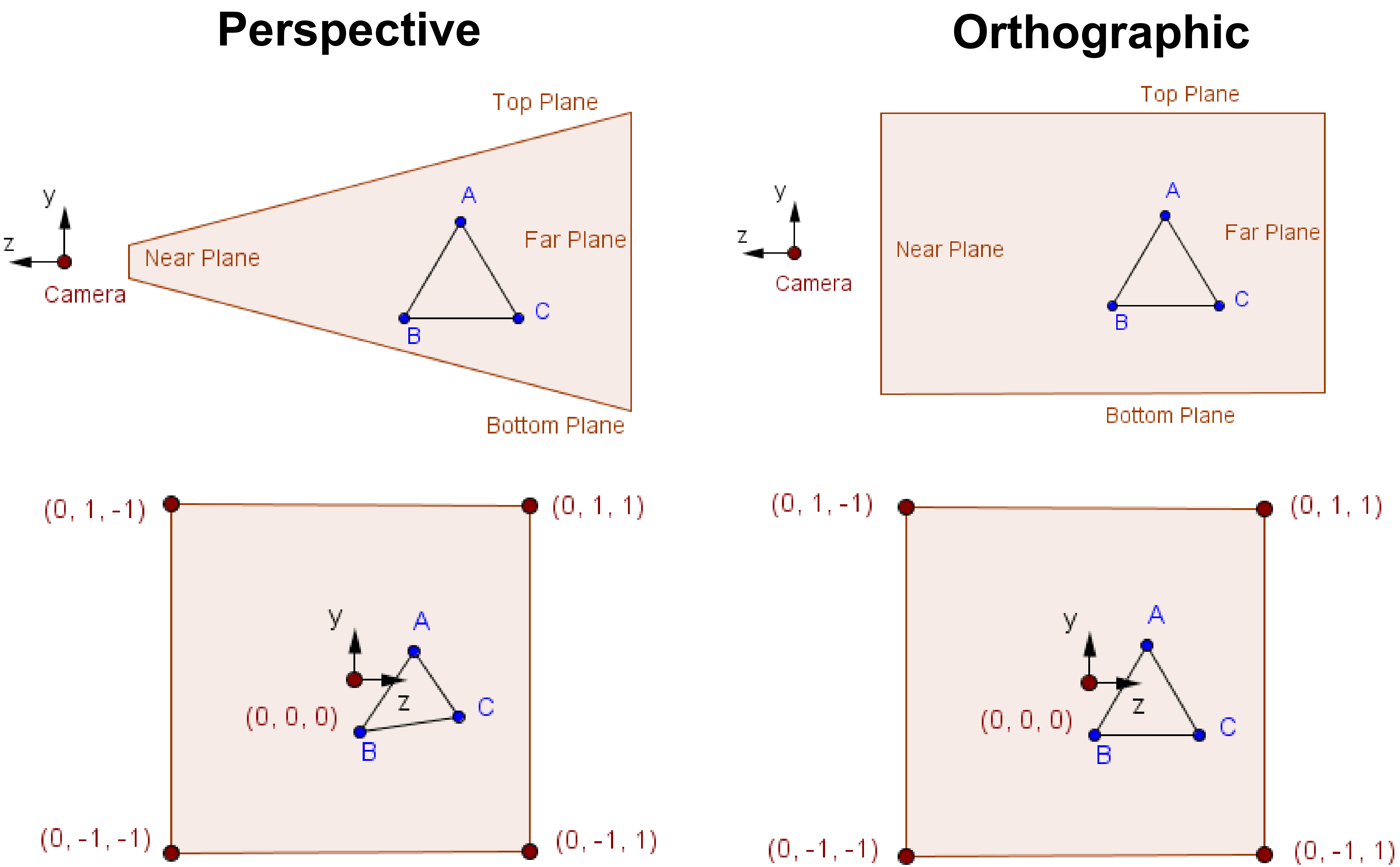
World Space → Camera Space

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

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**.
- But there are different types of projection:
  - Orthographic
  - Oblique
  - Perspective

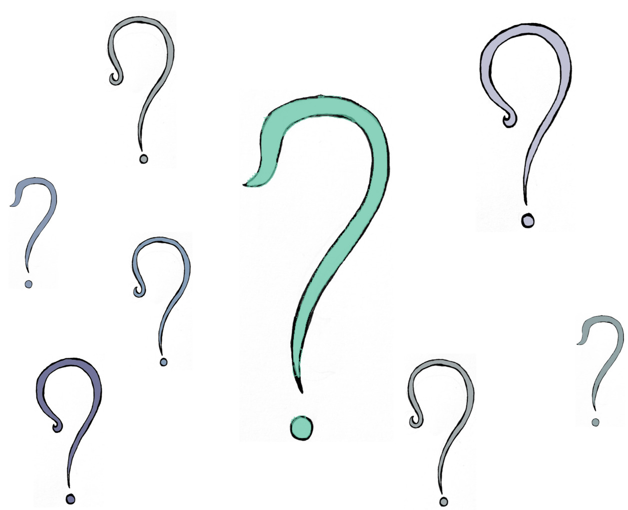
Camera Space → ND Space



Slices from x=0 plane

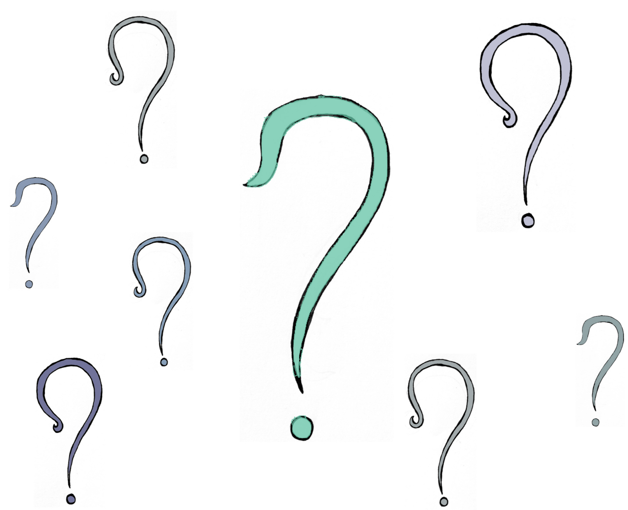
Orthographic Projection

- 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 view volume into a 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.
- How to find the left, right, top and bottom values, assuming that the projection is symmetric?

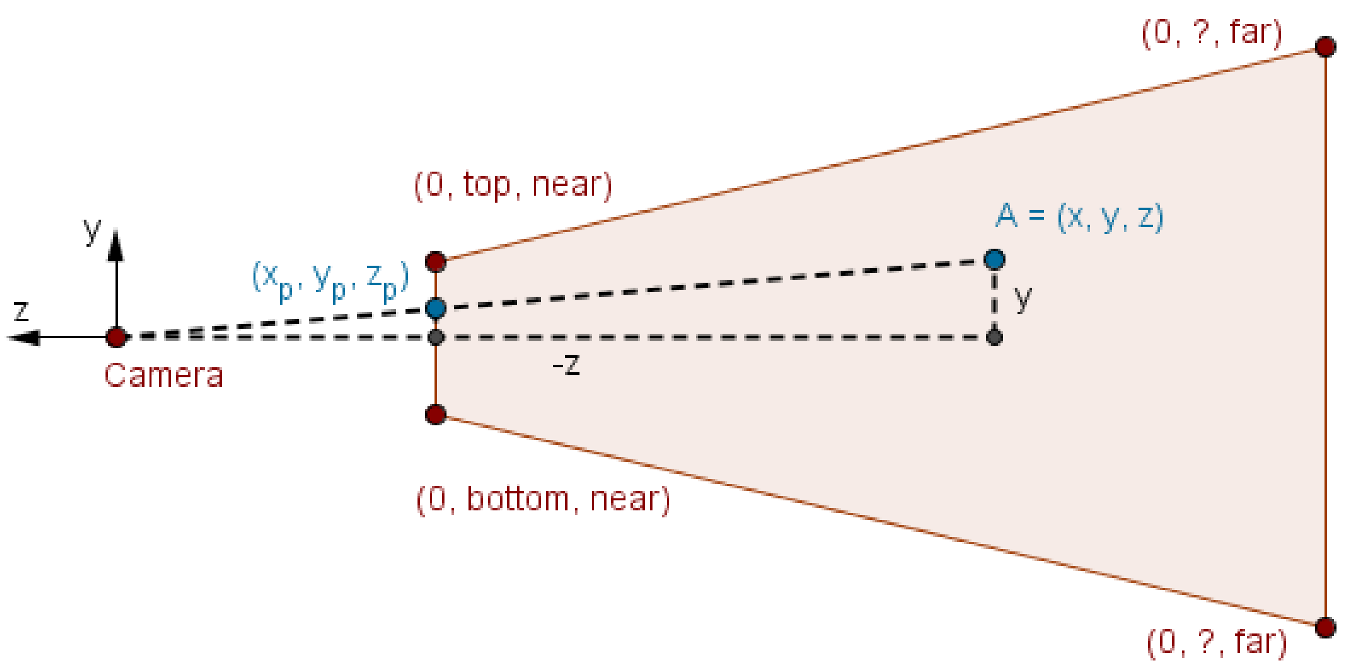


top = -bottom

left = -right

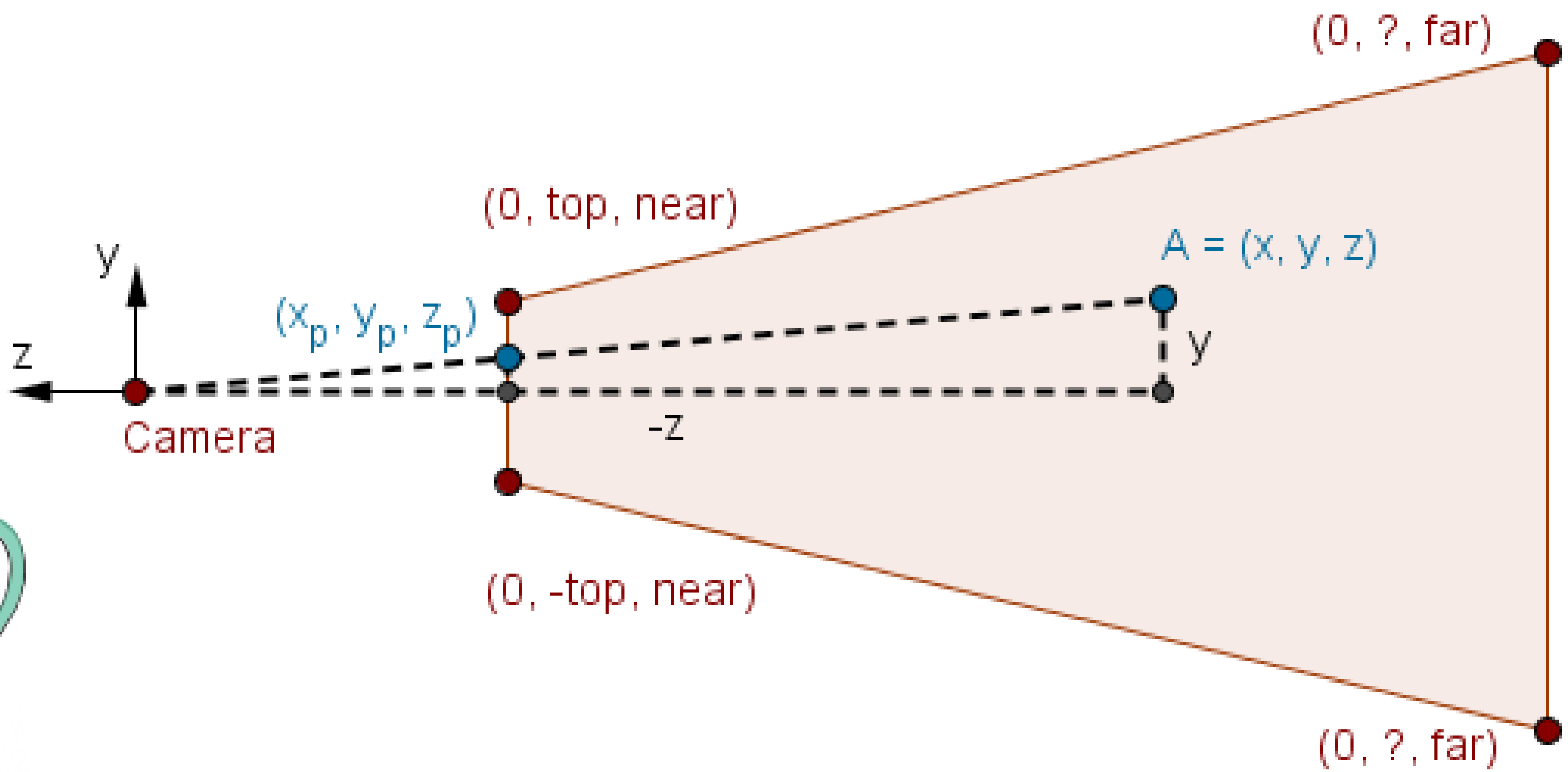
Perspective Projection

- 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 map the x and y coordinates to the correct range using similar triangles.



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.

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

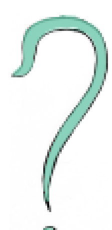
Perspective Projection

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

$$\begin{cases} s \cdot near + t = -1 \\ s \cdot far + t = 1 \end{cases}$$

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

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



Perspective Projection

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

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

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.

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 matrix is constructed when you specify the viewport size.

Three.js

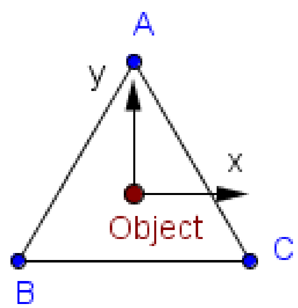
```
renderer = new THREE.WebGLRenderer();
renderer.setSize(width, height);
```

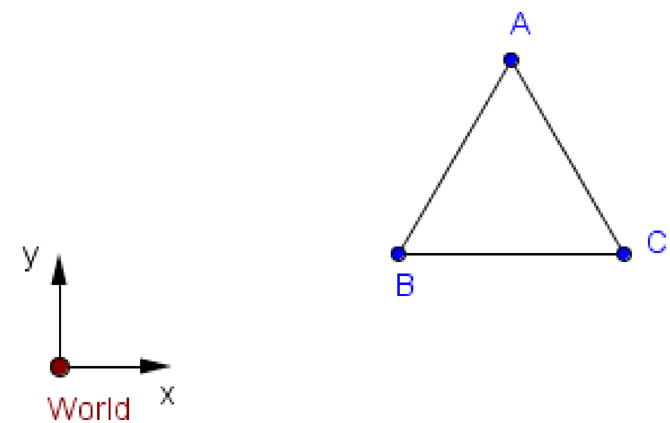
OpenGL + GLFW

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

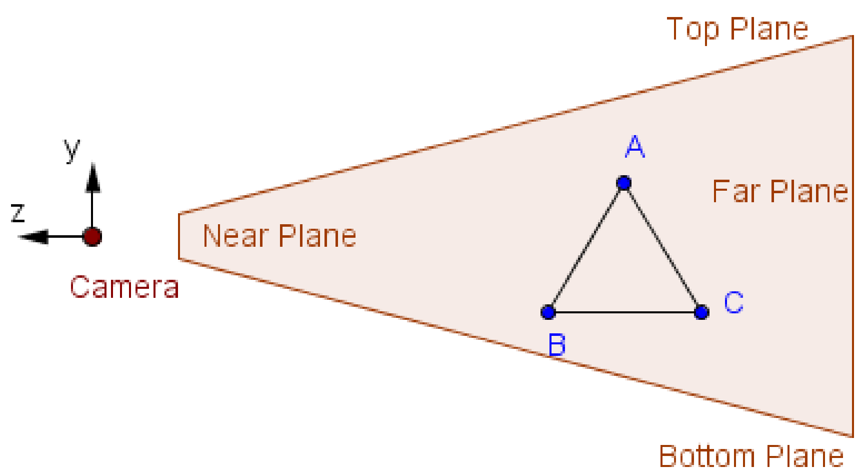
Overall

Object Space → World Space →





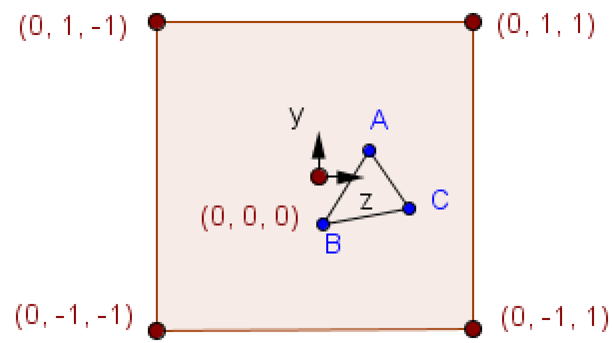
→ Camera Space →



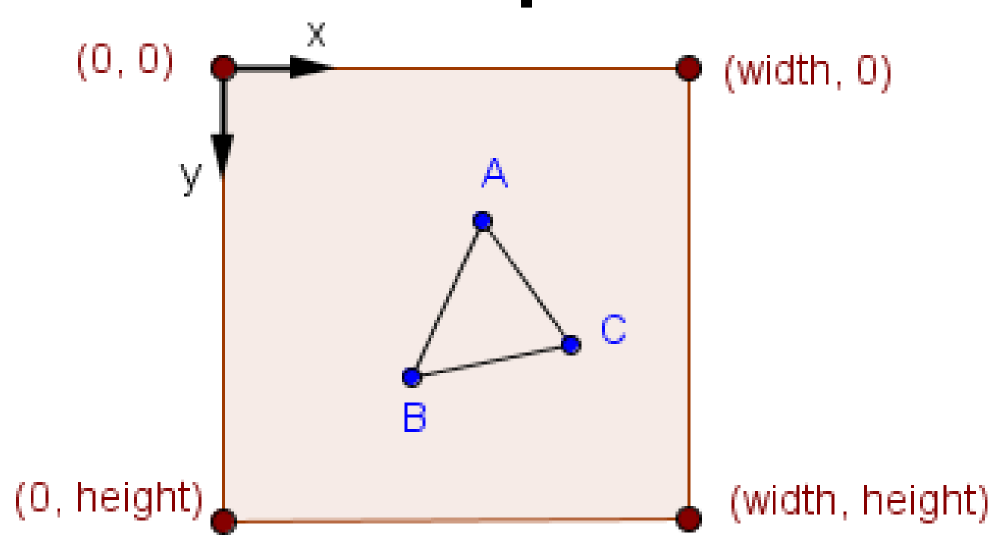
Light calculations are usually in this space!

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 * model * view * vec4(position, 1.0);

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

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

- Next 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 did you learn today?

What more would you like to know?

Next time

Shading and Lighting