08 Math for 3D games

Tvorba a dizajn počítačových hier

Návrh a vývoj počítačových hier

Michal Ferko

9. 11. 2023

What we need the math for

- Placing/moving/rotating/scaling objects
  - Creating hierarchies of objects
- Animation
- Rendering
- Physics & simulation

- Not just 3D, also 2D (but it’s simpler)

Floating point numbers

- IEEE 754 standard
- Single (32-bit) and double precision (64-bit)
  - GPUs almost exclusively single precision
  - Most engines perform all operations as 32-bit floats
- GPU FLOPS
- Half precision sometimes used on GPUs to speed up execution

Vectors and points

- All math we need for 3D games revolves around vectors and points
- We use them to represent 3D locations and directions
- Transforming, animating, rendering, physics…
- 2D variants for 2D games, but the 3rd dimension is still often used
  - Determine which objects are in front of which objects
  - Simulate 3D-like behavior

Vectors and Points - Unity

- Vectors and points share classes
  - `Vector2, Vector3, Vector4`
- Used for positions, directions, other **spatial** functionality
- Basic operations included
  - Addition, subtraction, multiplication, magnitude, normalization
  - Angle, Dot, Cross, Reflect…
  - `Mathf` class for basic operations
  - `Random` class for RNG
- `Transform.position,Transform.lossyScale (+local variants)`
- `Transform.rotation` is a **Quaternion**
- `Transform.forward, Transform.up, Transform.right`

Transformations

- **Affine Transformations** – talk by Jim Van Verth
- **Orientation Representation** – talk by Jim Van Verth
- Transform Translate/Rotate/Scale
- Look At

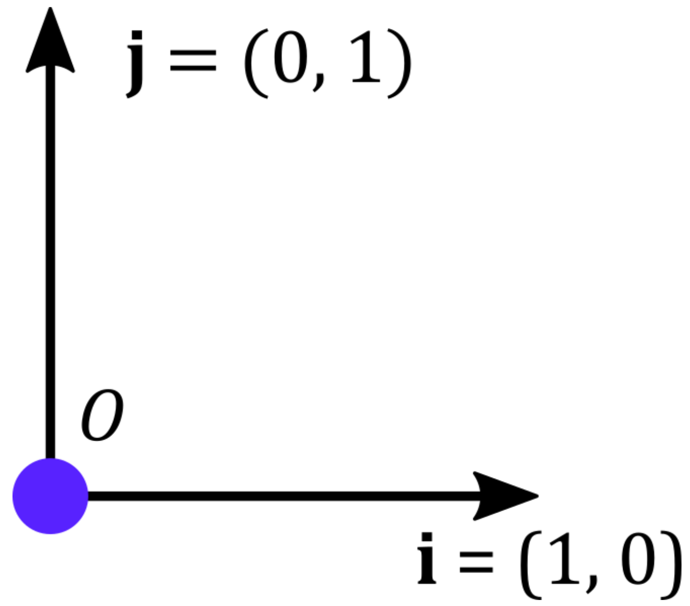
Affine Transformations

- A mapping between affine spaces
- Preserves lines (& planes)
- Preserves parallel lines, but not angles or distances
- Can represent as

$$T(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{y}$$

Affine Space

- Collection of points and vectors
- Represented using a **frame**: $< O, \mathbf{i}, \mathbf{j} >$
  - The frame defines a coordinate space
- Vector: $\mathbf{v} = x\mathbf{i} + y\mathbf{j} \qquad x, y \in \mathbb{R}$
- Point: $P = x\mathbf{i} + y\mathbf{j} + O \quad x, y \in \mathbb{R}$

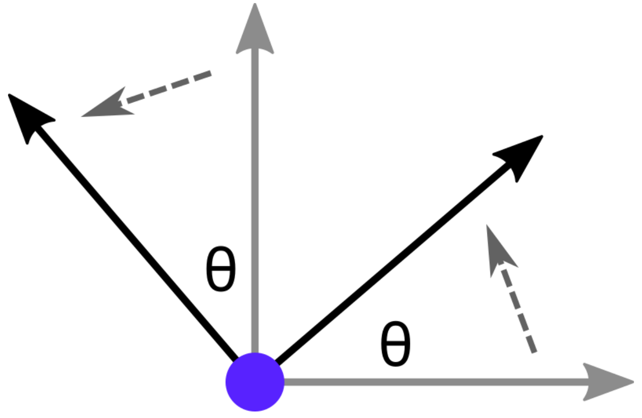


Affine Transformations

- Maps from space to space by using frames
- Determines how axes change - $\mathbf{A}$
- Determines how origin changes - $\mathbf{y}$

$$T(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{y}$$

Examples



- Translation: $T(\mathbf{x}) = \mathbf{x} + \mathbf{t}$ (axes don't change)

$$\mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- Rotation: $T(\mathbf{x}) = \mathbf{R}\mathbf{x}$ (origin doesn't change)

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

- Scale: $T(\mathbf{x}) = \mathbf{S}\mathbf{x}$ (origin doesn't change)

$$\mathbf{S} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

Combining Transforms

$$T(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{y}$$
$$S(\mathbf{w}) = \mathbf{B}\mathbf{w} + \mathbf{z}$$
$$S\big(T(\mathbf{x})\big) = \mathbf{B}(\mathbf{A}\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{B}\mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} + \mathbf{z}$$

- Order dependent!

$$S\big(T(\mathbf{x})\big) \neq T\big(S(\mathbf{x})\big)$$

- Can also do inverse

$$T^{-1}(\mathbf{z}) = \mathbf{A}^{-1}\mathbf{z} - \mathbf{A}^{-1}\mathbf{y}$$

Graphics APIs use matrix form

$$\mathbf{T} = \begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

- Can then use simple matrix multiplication (column vectors)

$$T\big(S(x)\big) = \mathbf{T}(\mathbf{Sx})$$

- Is a 4x4 matrix for 3D
- But most engines combine TRS into a single transform
  - Translation vector (position in Unity)
  - Scale vector (scale in Unity)
  - Rotation (rotation in Unity) - Euler angles in Unity, but are **quaternions** under the hood
    - `Transform.rotation` is of type `Quaternion`

Unity transform details

- Unity combines all into a 4x4 matrix that's sent to the GPU for rendering
- GPU takes the mesh vertex position and multiplies with the matrix
  - Result is final position in **world space**

```
var t = transform;
var matrix = Matrix4x4.TRS(t.localPosition, t.localRotation, t.localScale);
var matrix2 = transform.localToWorldMatrix;   // Same result if has no parent
```

- If the object has a parent, it's more difficult
  - Have to combine all parent transforms (multiply all matrices)
  - Result is a single 4x4 matrix to get world space coordinates
    - `transform.localToWorldMatrix`

Orientation vs Rotation

- Orientation – described as relative to a reference frame
- Rotation – changes object from one orientation to another
- Orientation can be represented as a rotation
  - From the reference frame $< O, \mathbf{i}, \mathbf{j} >$

- Representing rotation is tricky in 3D – we need to do
  - Concatenation – add two rotations together to get the resulting rotation
  - Interpolation – animate between two orientations
  - Rotation itself – applying rotation transform to vertex positions

Orientation Representation

Name	Concatenation	Interpolation	Applying rotation	Intuitive	Example
3x3 matrix	**Easy**, multiply matrices	Hard	**Easy**, multiply vector by matrix	No	$\begin{pmatrix} 0.41 & -0.67 & 0.61 \\ 0.86 & 0.08 & -0.5 \\ 0.29 & 0.74 & 0.61 \end{pmatrix}$
Euler angles	Hard, can lead to gimbal lock	Hard, cannot be direct	**Easy**, convert to matrix	**Yes**	$(45°, 30°, 85°)$
Axis+Angle	Hard	**Easy**	**Easy**, convert to matrix	**Yes**	$\mathbf{v} = (1,0,0)$ $\alpha = 45°$
Quaternions	**Easy**	**Easy**, spherical linear interpolation (slerp)	**Easy**, convert to matrix	No	$\mathbf{q} = \left(\frac{\sqrt{2}}{2}, 0, 0, \frac{\sqrt{2}}{2}\right)$ For $\mathbf{v} = (0,0,1), \alpha = 45°$

Quaternions

- Generalized complex numbers
- In 2D ($\mathbf{i}^2 = -1$)
  - A complex number $a + b\mathbf{i}$ that's normalized: $\sqrt{a^2 + b^2} = 1$
  - Represents rotation of angle α, where $\mathrm{a} = \cos\alpha\,, b = \sin\alpha$
- A quaternion is basically that, but in 3D
  - Written as $w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$ or (w, x, y, z)
  - That's normalized: $\sqrt{w^2 + x^2 + y^2 + z^2} = 1$

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

  - Represents rotation of angle β around axis (a_x, a_y, a_z)
  - Where $\mathrm{w} = \cos\beta; x = a_x \sin\beta\,;\ y = a_y \sin\beta\,;\ z = a_z \sin\beta$

Quaternions

$$\mathbf{M_q} = \begin{pmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{pmatrix}$$

- Can easily transform into a matrix
- Easy math for rotating vectors without using matrix form: $\mathbf{p}' = \mathbf{q}\,\mathbf{p}\,\mathbf{q}^{-1}$
- Easy uniform interpolation with slerp
- Unity uses it for all rotations
  - For transforms, animation, interpolation…
  - Provides easy conversions into it

$$\text{slerp}(\mathbf{p},\mathbf{q}:t) = \frac{\sin((1-t)\alpha)}{\sin\alpha}\mathbf{p} + \frac{\sin(t\alpha)}{\sin\alpha}\mathbf{q}$$

$$\cos\alpha = \mathbf{p} \bullet \mathbf{q}$$

```
var q1 = Quaternion.AngleAxis(45, new Vector3(0, 1, 0));
var q2 = Quaternion.Euler(45, 30, 85);
var q3 = Quaternion.Slerp(q1, q2, 0.3f);
var q4 = Quaternion.FromToRotation(Vector3.forward, Vector3.right);
Vector3 newPosition = q4 * transform.position;
```

Quaternions

- Good to know how they work
- You will never have to do Quaternion math directly
  - Can use other formats, engines have support for it
- Always converted to a 4x4 matrix before being used on the GPU

Why 4x4 matrix?

- We want to transform points as well as vectors
- Using a single transform
- We can differentiate between points and vectors:
  - Point: $P = \left(P_x, P_y, P_z, 1\right)^T$
  - Vector: $\mathbf{v} = \left(v_x, v_y, v_z, 0\right)^T$
- And if we combine 3x3 rotation matrix with 3x3 scale matrix and translation vector:

$$\mathbf{M} = \begin{bmatrix} \boldsymbol{S} \cdot \boldsymbol{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

- Then all the math works flawlessly!

4x4 matrix allows other transforms

- You can even use non-affine transformations with 4x4 matrices
- And using these, you even can transform points to vectors
  - $\mathbf{v} = (a, b, c, 0), \mathbf{Tv} = P, P = (d, e, f, 1)$ ($\mathbf{T}$ – non-affine transform)
- This is then called **homogenous coordinates**
- For these transformations, you can have a result of (d, e, f, g), where $g \neq \{0,1\}$. In those cases, the resulting point is:

$$\left(\frac{d}{g}, \frac{e}{g}, \frac{f}{g}, 1\right)$$

- This is used for e.g. perspective camera

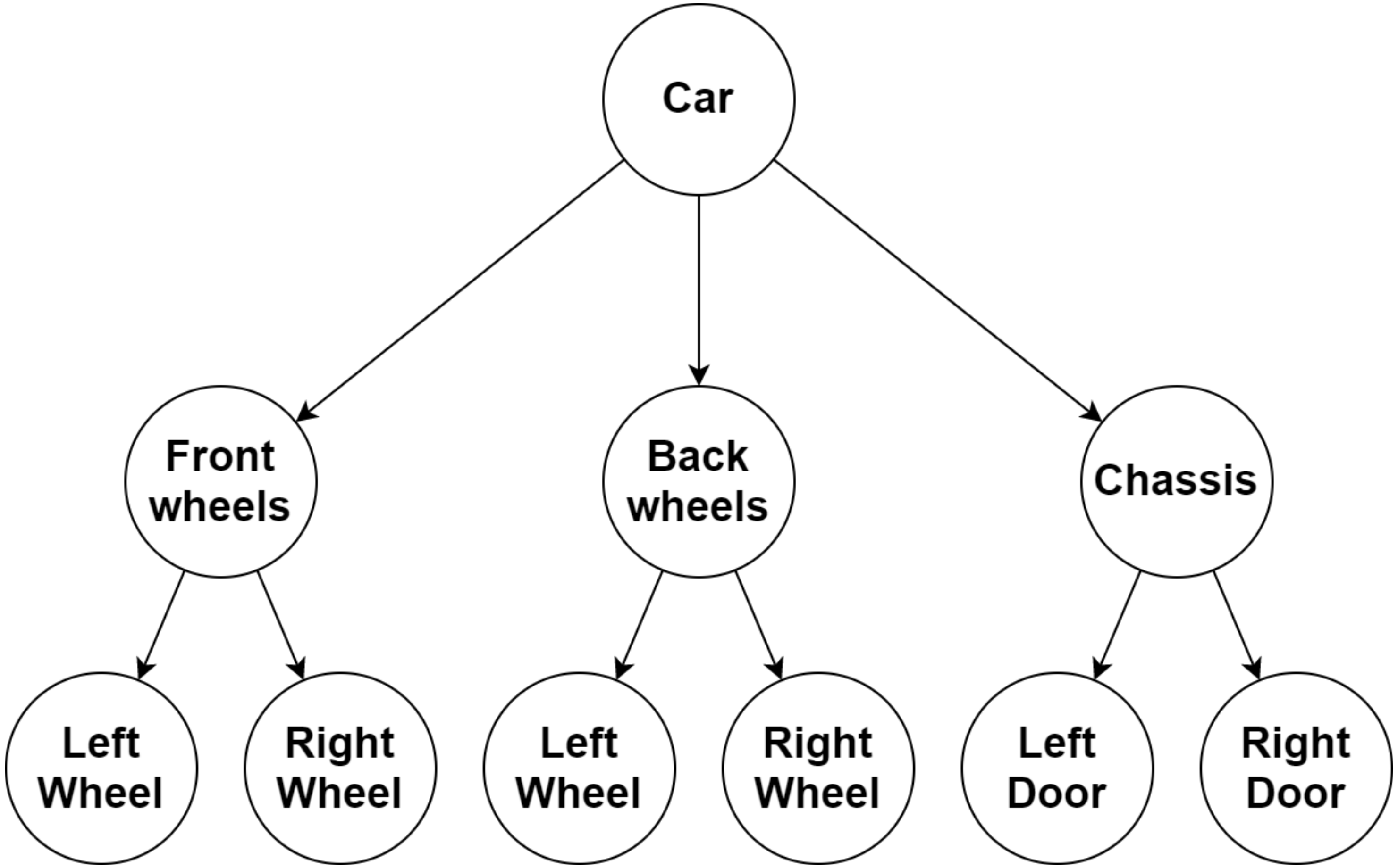
3D Transformation Pipeline

- We want to solve 3 problems
  - Construct hierarchies of objects
    - Transformation of object combined with its parents
  - Manipulate camera
    - Viewing transformation
  - Render object to screen
    - Projection/screen transformation
- We have objects as 3D meshes (list of vertex positions in some space)
  - How do they become pixels?

Scene Graph

- Basic structure for hierarchical scenes
  - Used almost anywhere, even for e.g. video editing
- It is a tree structure – directed acyclic graph
  - Nodes can have any number of children

Scene Graph


Car
Front wheels
Back wheels
Chassis
Left Wheel
Right Wheel
Left Wheel
Right Wheel
Left Door
Right Door

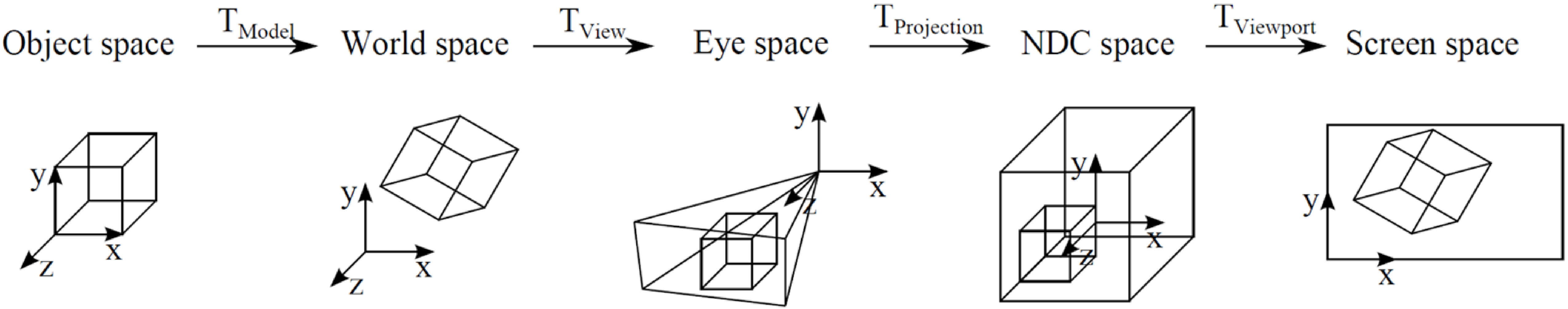
Object transformation

- Each scene graph node has an affine transformation
  - Affine transform is a combination of Translation, Scale & Rotation
  - Affine transform always outputs a 4x4 matrix
  - Transform component in Unity
- Final object transformation
  - Its own transform
    - Multiplied by the parent's transformation
      - Multiplied by the grandparent's transformation
        - …
  - The result is a multiplication of several 4x4 matrices

    ⇒ one 4x4 transformation matrix

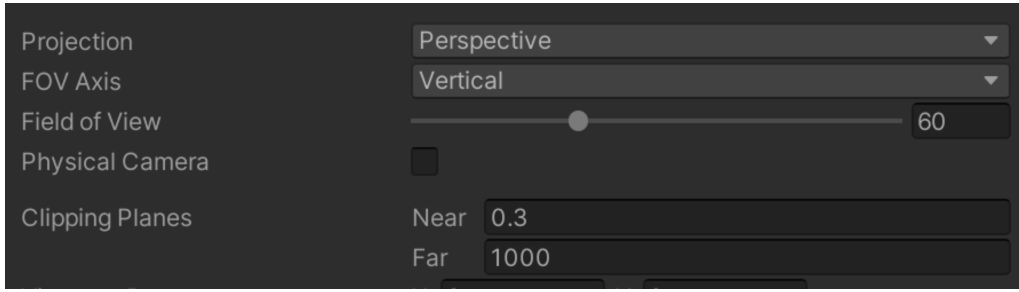
Different spaces and transforms

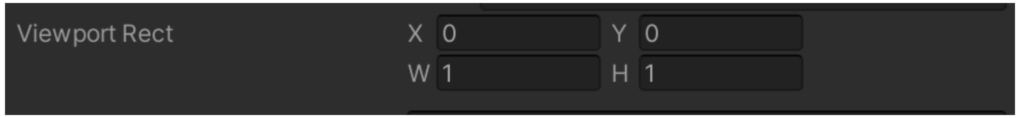
- We determine object positions in the "world" ($\mathbf{T}_{Model}$)
  - Created from a hierarchy of transformations – from object through its parents
- By placing the camera in the world, we determine from where (position, direction) we are looking at the world ($\mathbf{T}_{View}$)
- The type of camera (orthographic/perspective) determines our projection 3D ⇒ 2D ($\mathbf{T}_{Projection}$)
- We select the part of the window we render to ($\mathbf{T}_{Viewport}$)

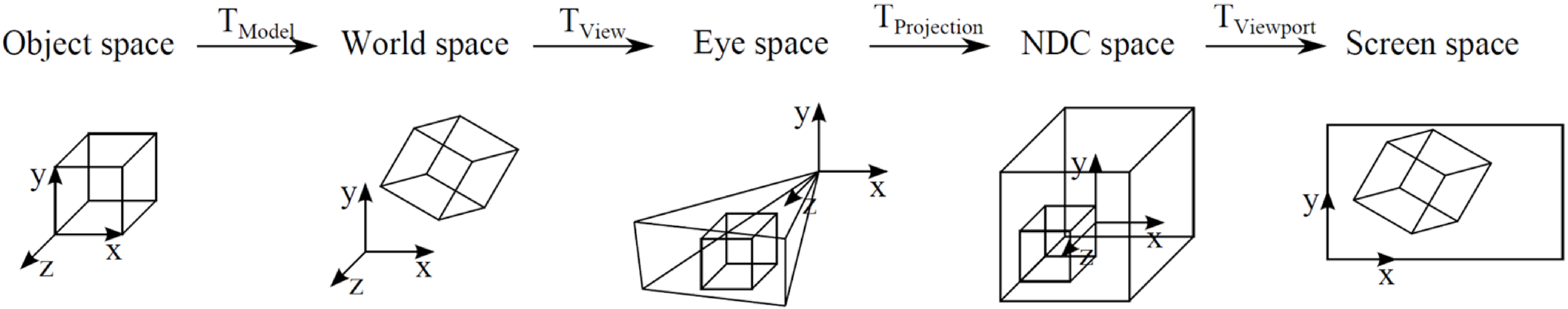


Spaces in Unity

- Object ⇒ World – taken from object transform (transform.localToWorldMatrix)
- World ⇒ Eye – taken from camera transform (camera.transform.localToWorldMatrix)
- Eye ⇒ NDC – taken from camera parameters
- NDC ⇒ Screen – taken from camera viewport



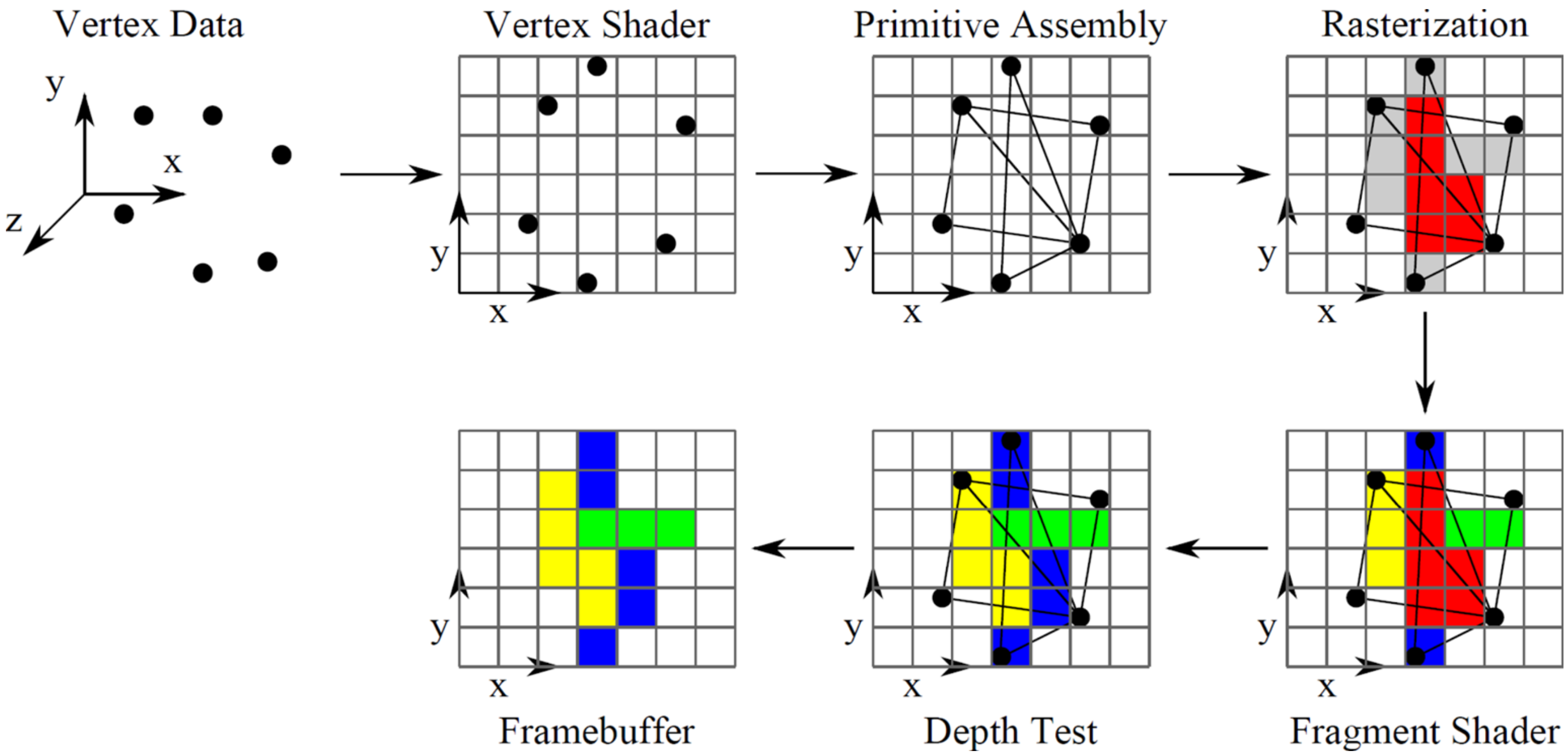




Real-time Rendering Pipeline

- The GPU renders 3D scenes from triangles

Offline rendering is very different (ray tracing instead of rasterization)



References

- Mathematics for 3D Game Programming and Computer Graphics
- Essential Mathematics for Games and Interactive Applications: A Programmer's Guide
- http://www.essentialmath.com/tutorial.htm