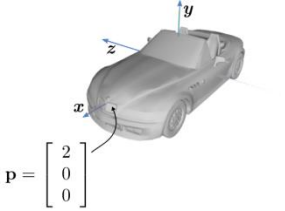
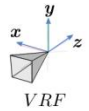
Transformations in the pipeline

- Vertices enter the pipeline with their original position, vertex shader processes one vertex at the time and transforms its position in clip-space coordinates.
- This transformation is obtained as combination of 3 transformations called *model, view* and *projection:*
  - *Model:* used to position the object in the scene
  - *View*: used to transform in view reference coordinates
  - *Projection*: used to transform into clip-space
- Old versions of OpenGL only had two matrices MODELVIEW and PROJECTION

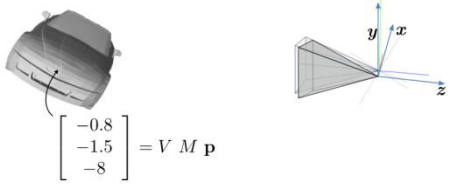
Model space
The frame is *local* to the model. In this example the origin is in the middle of the car



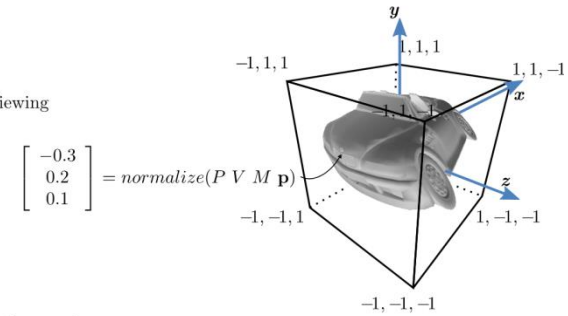
World space
The frame in which all the elements of the scene
are expressed, including the view reference frame.

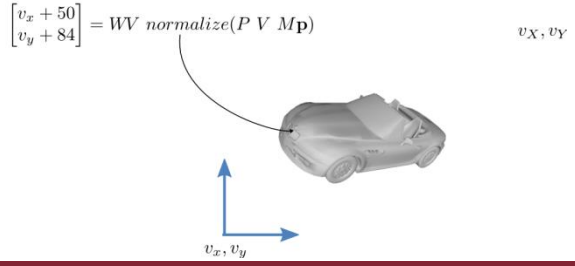


View space
The frame is the view reference frame VRF



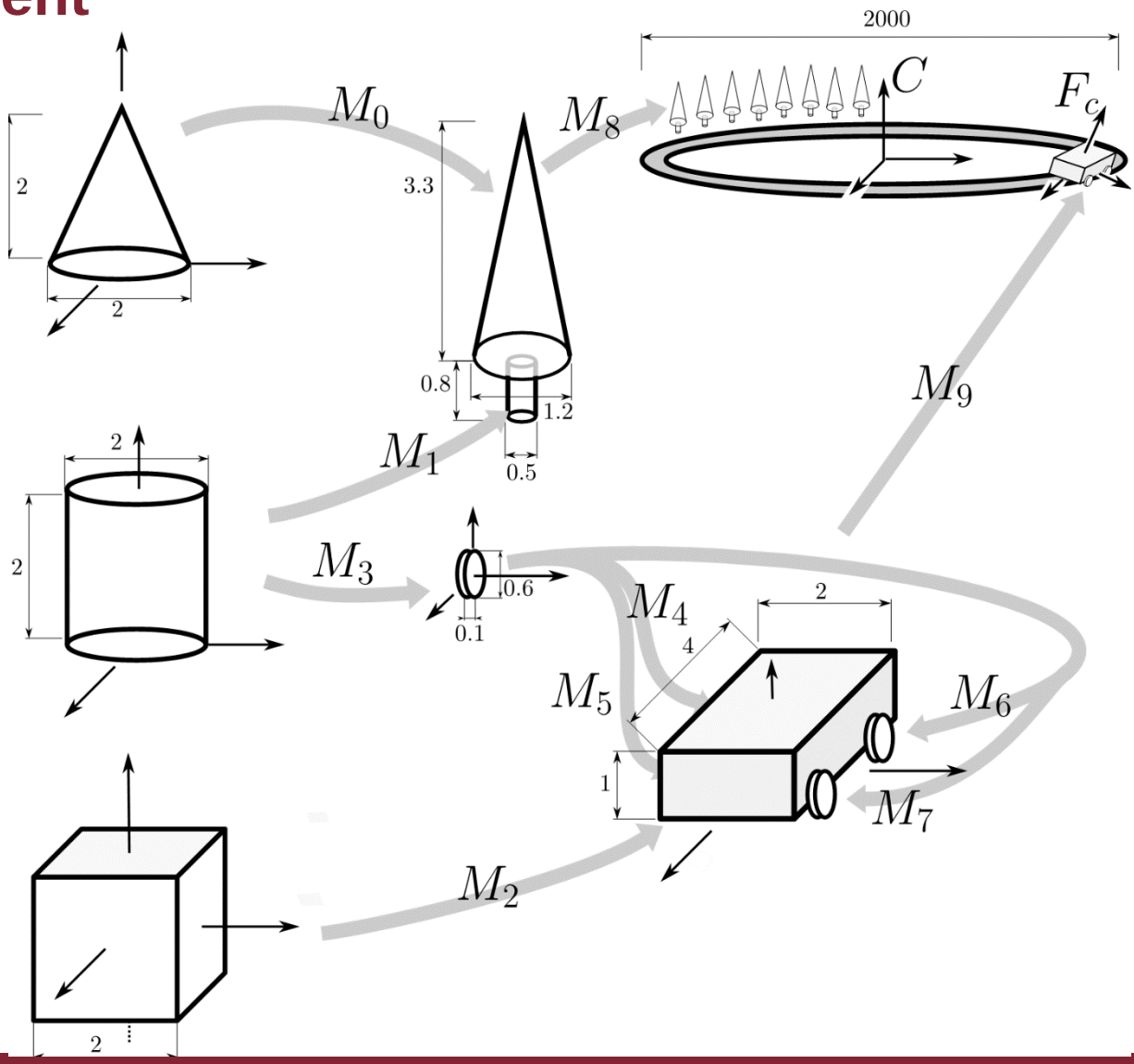
NDC space (Canonical viewing volume)





Our first 3D client

- Using the primitives seen we create the car and the tree
- We must identify the transformations used



Assembling the Tree and the Car

- $M_1 = S_{(0.25,0.4,0.25)}$
- $M_0 = T_{(0,0.8,0)}S_{(0.6,1.65,0.6)}$
- $M_2 = T_{(0,0.3,0)}S_{(1,0.5,2)}T_{(0,1,0)}$
- $M_3 = S_{(0.05,0.3,0.3)}R_{90Z}T_{(0,-1,0)}$
- Transformations M_4 to M_7 are simple translations that position the 4 wheels

Positioning the Trees and the Cars

- We now need to specify the transformations M_8 and M_9 that position the trees and the car.
- M_8 is a simple translation (or a series of translations if we have more trees).
- M_9 will consist of both rotation and translation and is specified in NVMC by defining the frame F_c

Viewing the scene

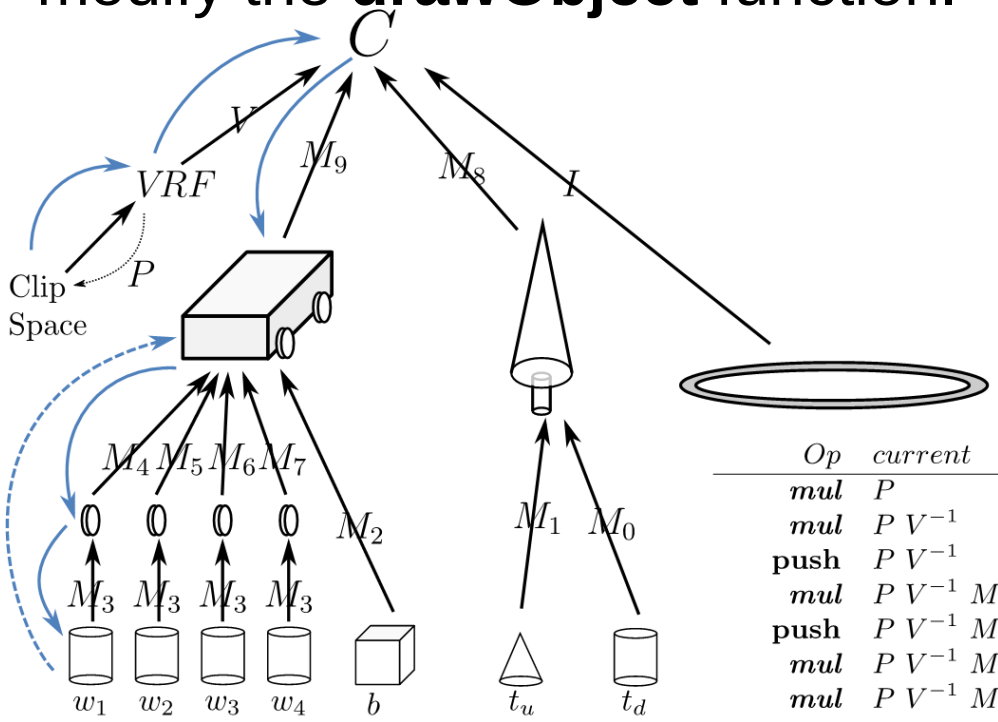
- We define an orthogonal projection from point [0,10,0] in the direction of *y*, the *view* matrix is:

$$V = \begin{bmatrix} x_x & y_x & z_x & O_x \\ x_y & y_y & z_y & O_y \\ x_z & y_z & z_z & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 10 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The projection matrix is obtaind by choosing:
  - *l,r,t,b* = 100 so to include the circuit that is 200mX200m
  - The near plane is set to *n*=0
  - The far plane is set to *f*=11, slightly larger than 10

The code

- We use the primitives we already defined and modify the **drawObject** function.



Op	*current*	*Matrix Stack*
mul	P	$\varnothing$
mul	$P\ V^{-1}$	$\varnothing$
push	$P\ V^{-1}$	$P\ V^{-1}$
mul	$P\ V^{-1}\ M_9$	$P\ V^{-1}$
push	$P\ V^{-1}\ M_9$	$P\ V^{-1}\ M_9; P\ V^{-1}$
mul	$P\ V^{-1}\ M_9\ M_4$	$P\ V^{-1}\ M_9; P\ V^{-1}$
mul	$P\ V^{-1}\ M_9\ M_4\ M_3$	$P\ V^{-1}\ M_9; P\ V^{-1}$
draw w_1	.	.
pop	$P\ V^{-1}\ M_9$	$P\ V^{-1}$

Matrix Stack

- The structure of the code could simply be, for any primitive:

1. Compute the proper transformation
2. Set the vertex shader to apply it
3. Render the primitive

- This method works but requires many unnecessary steps. We can simplify it using a *matrix stack* where the top one is the *current matrix*. We traverse the graph and push a matrix anytime we go to a lower level and a pop when we go up.

Code to set the matrices

```
var ratio = width / height; //line 229, Listing 4.1{
var bbox = this.game.race.bbox;
var winW = (bbox[3] - bbox[0]);
var winH = (bbox[5] - bbox[2]);
winW = winW * ratio * (winH / winW);
var P = SglMat4.ortho([-winW / 2, -winH / 2, 0.0], [winW / 2, winH / 2, 21.0]);
gl.uniformMatrix4fv(this.uniformShader.uProjectionMatrixLocation, false, P);
var stack = this.stack;
stack.loadIdentity(); //line 238}
// create the inverse of V //line 239, Listing 4.2{
var invV = SglMat4.lookAt([0, 20, 0], [0, 0, 0], [1, 0, 0]);
stack.multiply(invV);
stack.push();//line 242
```

Code to initialize

```
NVMCClient.onInitialize = function () {// line 290, Listing 4.2{
        var gl = this.ui.gl;
        NVMC.log("SpiderGL Version : " + SGL_VERSION_STRING + "\n");
        this.game.player.color = [1.0, 0.0, 0.0, 1.0];
        //NVMC.GamePlayers.addOpponent();
        this.initMotionKeyHandlers();
        this.stack = new SglMatrixStack();
        this.initializeObjects(gl); //LINE 297}
        this.uniformShader = new uniformShader(gl);
};
```

Code for shaders

```
var vertexShaderSource = "\
        uniform   mat4 uModelViewMatrix;   \n\
        uniform   mat4 uProjectionMatrix;   \n\
        attribute vec3 aPosition;           \n\
        void main(void)                     \n\
        {                                   \n\
        gl_Position = uProjectionMatrix * uModelViewMatrix    \n\
        * vec4(aPosition, 1.0);                               \n\
        }";
var fragmentShaderSource = "\
        precision highp float;      \n\
        uniform vec4 uColor;        \n\
        void main(void)             \n\
        {                           \n\
        gl_FragColor = vec4(uColor);        \n\
        } ";
```

Code for object rendering

```
gl.uniformMatrix4fv(this.uniformShader.
        uModelViewMatrixLocation, false, stack.matrix);
this.drawObject(gl, this.cylinder, [0.8, 0.2, 0.2, 1.0], [0, 0, 0, 1.0]);
```

- This associates the current matrix of the stack to the uModelViewMatrix and then renders the object
- Snapshot of image created