VR-2025-Spring API Documentation

Basics

Scenes

All scenes are defined in `js/scenes/scenes.js`

In that file you will see lines like this:

```
{ name: "example1", path: "./example1.js", public: true },
```

Where `name` is the name that appears in the XR experience, `path` is the relative path to the scene file, and `public` defines whether you can see it directly in the scene menu after you enter AR mode.

To access non-public scenes in the XR experience: point to and hold any button in the scene menu with one controller, and click the same button with the other controller.

Each scene is a separately defined .js file located under `js/scenes`
The basic structure of a scene looks like this:

```
export const init = async model => {
   // initialization code, executed once on load
   model.animate(() => {
      // update code, executed every frame
   });
}
```

In short, `init` and `animate` are the most important functions in a scene.
You can refer to `js/scenes/demoSimplest.js` for a simple example.

cg.js

A vector can be represented as a an array of 3 floating point numbers:

```
[x,y,z]
```

A 4x4 matrix can be represented as a column-major array of 16 floating point numbers:

```
[x0,x1,x2,x3, y0,y1,y2,y3, z0,z1,z2,z3, w0,w1,w2,w3]
```

SCALAR FUNCTIONS

```
cg.def(value, default)  // If value is undefined, return default, else return value.
cg.ease(t)              // Cubic ease curve: If t<0 then 0 else if t>1 then 1 else t*t*(3 - 2*t)
cg.hermite(t,a,b,da,db) // scalar hermite interpolation
cg.ik(a,b,C,D)          // 2-link inverse kinematics:
                        // Shoulder is assumed to be at the origin.
                        //    Input : a,b are limb lengths
                        //            C is a vector specifying position of wrist.
                        //            D is a vector specifying preferred elbow direction.
                        //    Output: computed elbow position is placed in D.
cg.mixAngle(a,b,t)      // Interpolate between angles a and b in radians; handle wrap-around properly.
```

```
cg.mixf(a,b,t)          // Return a+t*(b-a), where a and b are floating point numbers.
cg.mixf(a,b,t,u)        // Return a*t + b*u, where a and b are floating point numbers.
cg.noise(x,y,z)         // Return floating point value of noise at point [x,y,z] (Perlin1985).
cg.plateau(a,b,c,d,t)   // As t varies from a⇒b⇒c⇒d, return linear ramp of values 0⇒1⇒1⇒0.
cg.roundFloat(n,f)      // Floating point value of f rounded to n digits.
cg.uniqueID()           // Return a unique randomly defined integer ID.
```

STRING FUNCTIONS

```
cg.decimal(f,n)         // String representation of f with exactly n decimal digits.
cg.fixedWidth(f,n,d)    // String representation of f with n characters before & d digits after the dot.
cg.hexToRgba(hex)       // Convert hex representation of a color to an [r,g,b,a] vector.
cg.pack(array,lo,hi)    // Pack an array into a string in base 92*92; must specify lo and hi values.
cg.round(f,n)           // String representation of f rounded to n digits.
cg.unpack(string,lo,hi) // Unpack a based 92*92 packed array; lo,hi must match values when packed.
```

VECTOR FUNCTIONS

```
cg.add(a,b)             // return sum a+b of two vectors
cg.cross(a,b)           // return cross product of two vectors
cg.dot(a,b)             // return dot product of two vectors
cg.mix(a,b,t)           // return a+t*(b-a), where a and b are vectors
cg.mix(a,b,t,u)         // return a*t + b*u, where a and b are vectors
cg.norm(v)              // return norm (geometric length) of a vector
cg.normalize(v)         // return v scaled to unit length
cg.roundVec(n,v)        // return approximation of v with each component rounded to n digits
cg.scale(v, s)          // return v scaled by s
cg.subtract(a,b)        // return difference b-a of two vectors
```

CURVE FUNCTIONS

- A curve is represented as an array of 3d points: `[ [x,y,z], [x,y,z], … ]`

```
cg.computerCurveLength(curve)    // Compute geometric curve length.
cg.copyCurve(curve)              // Duplicate a curve.
cg.resampleCurve(curve,n)        // Resample a curve to have n values.
cg.sample(curve, t, isAngle)     // Sample a curve, where 0<=t<=1. Specify if x,y,z values are radians.
cg.spline(keys,spline,ptsPerKey) // Create a spline curve. If spline arg isn't null, place results there.
```

3D GEOMETRY METHODS

```
cg.isLineIntersectPoly(A,B,P)        // Does line A…B intersect convex polyhedron bounded by plane set P?
cg.isPointInBox(P,B)                 // Is point P inside a box? B is a matrix xform of a unit cube.
cg.isRayIntersectTriangle(V,W,A,B,C) // Does ray V,W intersect the triangle with vertices A,B,C?
cg.isSphereIntersectBox(S,B)         // Does sphere S=x,y,z,r intersect a box? B is xform of unit cube.
```

MATRIX FUNCTIONS

- each matrix is represented as a flat array of 16 floating point numbers

```
cg.mAimX(vec,vec2)    // return matrix that rotates X to vec; optionally also rotate Y to vec2
cg.mAimY(vec,vec2)    // return matrix that rotates Y to vec; optionally also rotate Z to vec2
cg.mAimZ(vec,vec2)    // return matrix that rotates Z to vec; optionally also rotate X to vec2
cg.mFromQuaternion(q) // return matrix corresponding to quaternion {q.x,q.y,q.z,q.w}
cg.mHitRect(beam,rect)// does beam (xform of ray [0,0,0]=>[0,0,-1]) hit rect (xform of unit square)
cg.mIdentity()        // return identity matrix
cg.mInverse(m)        // return inverse of matrix m
cg.mMirrorZ(m)        // return a matrix that flips about z without making objects inside out
cg.mMultiply(a,b)     // return product of two matrices a and b
cg.mPerspective(fl)   // return perspective transformation along Z
cg/mProject(x,y,z)    // return projective matrix
cg.mRotateX(theta)    // return matrix that rotates about X
cg.mRotateY(theta)    // return matrix that rotates about Y
cg.mRotateZ(theta)    // return matrix that rotates about Z
cg.mScale(s)          // return matrix that scales by [s,s,s]
cg.mScale(vec)        // return matrix that scales by vec
```

```
cg.mScale(x,y,z)      // return matrix that scales by [x,y,z]
cg.mToQuaterion(m)    // return quaternion {q.x,q.y,q.z,q.w} corresponding to matrix m
cg.mTransform(m,p)    // return transform of point p by matrix m
cg.mTranslate(vec)    // return matrix that translates by vec
cg.mTranslate(x,y,z)  // return matrix that translates by [x,y,z]
cg.mTranspose(m)      // return transpose of matrix m
```

MATRIX STACK OBJECT AND METHODS

- all methods other than save and restore operate on the top matrix on the stack
- methods that act on the top matrix are built on top of the equivalent cg.* functions
- all methods other than getValue() return the matrix object

```
let matrix = cg.Matrix() // instantiate a 4x4 matrix stack object

matrix.aimX(vec)
matrix.aimY(vec)
matrix.aimZ(vec)
matrix.getValue()        // return an array of 16 floating point numbers
matrix.identity()
matrix.inverse()
matrix.project(x,y,z)
matrix.restore()         // restore matrix from the stack
matrix.rotateX(theta)
matrix.rotateY(theta)
matrix.rotateZ(theta)
matrix.save()            // push top matrix onto the stack
matrix.scale(s)
matrix.scale(vec)
matrix.scale(x,y,z)
matrix.setValue(array)   // argument is an array of 16 floating point numbers
matrix.translate(vec)
matrix.translate(x,y,z)
matrix.transpose()
```

PHYSICS OBJECTS

```
let deriv = new Deriv()  // Maintain the derivative of a changing value

deriv.get()
deriv.set(p)
deriv.update()

let spring = new Spring() // One-dimensional spring simulator

spring.getPosition()
this.setDamping(d)
this.setForce(f)
this.setMass(m)
this.update(deltaTime)

let dof = new DOF()       // One-dimensional damped oscillator simulator

this.setDamping(d)
this.setForce(f)
this.setMass(m)
this.setPosition(p)
this.update(deltaTime)
```

clay.js

API for the modeler implemented in `js/render/core/clay.js`
The root node, called `model`, is the only object that is already created for you.

You create a scene by building a tree hierarchy, using the `obj.add(...)` method.

METHODS:

```
model.animate(() => PROCEDURE) // This is called every animation frame
child = obj.add(type)          //
'sphere'|'tubeX'|'tubeY'|'tubeZ'|'cube'|'donut'
                               // For an animation node, call with no argument.
                               // Eg: hip = pelvis.add(); leg =
hip.add('tubeY');
child = obj.child(index)       // return the nth child of this node

obj = obj.aimX(vec)               // aim the object's X axis toward vec
obj = obj.aimY(vec)               // aim the object's Y axis toward vec
obj = obj.aimZ(vec)               // aim the object's Z axis toward vec
obj = obj.color([r,g,b] or r,g,b) // set color: values are between 0.0 and 1.0
obj = obj.identity()              // set to identity matrix
obj = obj.info(info)              // add extra shape info like donut thickness
obj = obj.move([x,y,z] or x,y,z)  // translate by x,y,z
obj = obj.remove(child or index)  // remove a child by reference or by index
obj = obj.scale([x,y,z] or x,y,z) // non-uniform scaling
obj = obj.scale(s)                // uniform scaling: s is a floating pt number
obj = obj.txtr(txtrUnit)          // apply texture: txtrUnit is between 0 and
14
obj = obj.turnX(radians)          // rotate about the X axis
obj = obj.turnY(radians)          // rotate about the Y axis
obj = obj.turnZ(radians)          // rotate about the Z axis
```

VALUES:

```
model.time      // time in seconds since the program started
model.deltaTime // time since previous animation frame
```

When any property is left unspecified for an object, that property is inherited from the object's parent. If the parent does not have that property, the search for a value continues all the way up to the root. All properties are defined by default for the root.

clientState: A global object that tells you the current input state of every client.

Brief descriptions of clientState methods:

```
clientState.button(id,hand,b) // Is button b pressed for a controller?
clientState.coords(id)        // The local coordinate system of this client
clientState.finger(id,hand,i) // xyz position of the tip of a finger
clientState.hand(id,hand)     // Matrix transform of this hand of this client
clientState.head(id)          // Matrix transform of the head of this client
clientState.isHand(id)        // Is this client using hand tracking?
clientState.isXR(id)          // Is this client currently in immersive mode?
clientState.pinch(id,hand,i)  // Pinch state of a finger
```

```
clientState.point(id,hand)    // [u,v] value if a hand is pointing
```

More detailed descriptions of clientState methods:

```
T_or_F = clientState.button(id,hand,b)
```

When not hand tracking, returns the press state of button `b`
When hand tracking, returns `null`

```
mat_16 = clientState.coords(id)
```

The current matrix transform of this client's local coordinate system

```
vec_3  = clientState.finger(id,hand,f)
```

When hand tracking, returns the 3D location of finger `f`
When not hand tracking, returns the 3D location of the controller's virtual ping pong ball

```
mat_16 = clientState.hand(id,hand)
```

When hand tracking, returns the matrix transform of the client's hand
When not hand tracking, returns the matrix transform of the controller

```
mat_16 = clientState.head(id)
```

The matrix transform of the client's head

```
T_or_F = clientState.isHand(id)
```

Is this client currently using hand tracking?

```
T_or_F = clientState.isXR(id)
```

Is this client currently in immersive mode?

```
T_or_F = clientState.pinch(id,hand,f)
```

When hand tracking, returns the boolean pinch state of finger `f`, or else the hand's pointing state.
When not hand tracking, returns the boolean press state of button `f-1`
`f` can be `1`,`2`,`3` or `4`, for your thumb touching your index, middle, ring or pinkie finger, respectively, or else `5` for pointing with your thumb sideways or `6` for pointing with your thumb up.
If using a controller, `draw.pinch(hand)` returns true if the trigger is pressed.

```
[u,v]  = clientState.point(id,hand)
```

If not hand tracking, or the hand isn't pointing, returns `null`.
If the hand is pointing, returns `[u,v]`, where `-1 <= u <= 1` and `-1 <= v <= 1`.
`u` measures how much the thumb is aiming up or down.

`v` measures how much the index finger is aiming up or down.

Definition of terms for clientState methods:

`T_or_F` -- A true or false value
`vec_3` -- An xyz position stored as an array of 3 floats.
`mat_16` -- A 4x4 matrix stored as a linear array of 16 floats in column-major order.
`hand` -- Valid values: 'left', 'right'
`id` -- The unique clientID of an XR client
`b` -- Which button is this? Valid values: 0,1,2,3,4,5,6
`f` -- Which finger is this? Valid values: 0,1,2,3,4 (or 5 or 6 for the `pinch` method)

g2.js

The g2 library sits on top of the 2D canvas, providing a virtual canvas that spans [-1..1, -1..1].

To create a g2 instance:

```
let g2 = new G2(do_not_animate_flag [, width [, height]]);
```

If you only want to draw a non-changing image (like fixed labels and diagrams), set the `do_not_animate_flag` to `true`. This will result in much more efficient rendering. Note: `do_not_animate_flag` defaults to `false`, `width` defaults to `1024`, `height` defaults to `width`.

To use a g2 canvas as a texture source:

```
model.txtrSrc(txtr_unit, g2.getCanvas());
```

Note: If the g2 `do_not_animate_flag` is set, then `txtr` is smart enough to optimize performance by not redownloading the canvas image at every animation frame.

Methods of g2:

```
g2.addWidget(obj, type, x, y, color, label, action, size) // add a widget
g2.arrow(a,b)             // draw an arrow from a to b
g2.barChart(x,y,w,h, values, labels, colors)              // draw a bar chart
g2.clear()                // clear the canvas
g2.clock(x,y,w,h)         // draw a clock
g2.computeUVZ(objMatrix) // get the location and depth of a matrix
g2.drawOval(x,y,w,h)      // draw an oval that fits within a rectangle
g2.drawPath(path)         // draw a path, given points [[x0,y0],[x1,y1],...]
g2.drawRect(x,y,w,h,r)    // draw a rectangle. Add radius r for rounded corners
g2.drawWidgets(obj)       // draw the widgets associated with an object
g2.fillOval(x,y,w,h)      // fill an oval that fits within a rectangle
g2.fillPath(path)         // fill a path, given points [[x0,y0],[x1,y1],...]
g2.fillRect(x,y,w,h,r)    // fill a rectangle. Add radius r for rounded corners
g2.getCanvas()            // return the actual canvas.
g2.getContext()           // get the 2D context
```

```
g2.getUVZ(obj)           // get location and depth of an object in the g2 space
g2.line(a,b)             // draw a line from a to b
g2.lineWidth(w)          // set line width
g2.mouseState()          // get the state of the mouse
g2.noise(true/false)     // option to add noise to drawings for hand-drawn
effect
g2.setColor(color,dim)   // set the drawing color
g2.setFont(f)            // set the text font
g2.text(text,x,y,alignment,rotation)                    // draw text
g2.textHeight(h)         // set text height
g2.update()              // this must be called every animation frame
```

Textures

The texturing API consists of a single function:

```
model.setTxtr(src, do_not_animate)
```

The texture `src` can either be a text string, which indicates a texture file, or else an HTML canvas, which indicates a dynamically changing canvas image. If you are on a computer (not a VR/XR headset), then the texture source can be your computer's video camera.

If the texture `src` is a canvas, and `do_not_animate` is set to `true`, then the canvas texture image is only downloaded once to the GPU. For static canvas content (like fixed labels and diagrams) this flag should be set, because it is much more efficient at runtime.

Input System

`inputEvents.js` is located at `js/render/core/inputEvents.js`
`inputEvents` is a global variable accessible by all scenes.

Hand positions

To get a hand position as an `[x,y,z]` vector, use:

```
inputEvents.pos('left')
inputEvents.pos('right')
```

To get hand positions in world space:

```
leftHand.setMatrix
(cg.mMultiply(clay.inverseRootMatrix,controllerMatrix.left));
rightHand.setMatrix(cg.mMultiply(clay.inverseRootMatrix,controllerMatrix.right)
);
```

In the above example, `leftHand` and `rightHand` are objects in the scene.

Head position

To get self head position, use:

```
cg.mMultiply(clay.inverseRootMatrix,cg.mix(clay.root().inverseViewMatrix(0),

clay.root().inverseViewMatrix(1),.5));
```

The code above takes the average position of the two eyes. It returns a matrix.
We can use it like this:

```
head.setMatrix(cg.mMultiply(clay.inverseRootMatrix,
                                  cg.mix(clay.root().inverseViewMatrix(0),

clay.root().inverseViewMatrix(1),.5)));
```

Trigger button status

To get the trigger button status for either hand, use:

```
inputEvents.isPressed('left');
inputEvents.isPressed('right');
```

Events

```
inputEvents.onMove = hand => {}; // on a hand moving
inputEvents.onPress = hand => {}; // on trigger pressed for a hand
inputEvents.onDoublePress = hand => {}; // on both triggers pressed
inputEvents.onClick = hand => {}; // on trigger clicked for a hand
inputEvents.onDrag = hand => {}; // on a hand moving while trigger is pressed
inputEvents.onRelease = hand => {}; // on trigger released for a hand
```

Shaders

Custom Shader Syntax

You can find an example in `demoShaderNew.js`

To create a custom shader for an object, do:

```
model.customShader(`
<custom vertex shader fields/functions>
----------------
<custom vertex shader main>
****************
<custom fragment shader fields/functions>
----------------
<custom fragment shader main>
`);
```

The `"----------"` line is the separator between declarations and the main function.

The `"*********"` line is the separator between vertex and fragment shaders.

If you only want a custom fragment shader, you may omit the first `"---------"`. (because of compatibility issues, you may also omit the `"**********"` separator.)

If you only want a custom vertex shader, you may omit the second `"---------"`.

Note: the number of `'-'` and `'*'` for the separators does not matter as long as its length is greater than 3.

Custom Shader Example

```
let obj = model.add('sphere');
obj.flag("myObj");

model.customShader(`
    uniform int myObj;
—---------------
if(myObj == 1){
    apos.xyz += noise(apos.xyz + uTime * .5) * aNor * .5;
    pos.xyz = obj2Clip(apos.xyz);
}
***************
uniform highp int myObj;
—---------------
if(myObj == 1)
    color *= .5 + noise(1.5 * vAPos);
`);
```

This custom shader first morphs the sphere by the noise function, then applies a texture to the sphere using the noise function.

Passing Data into Custom Shaders

You can use `model.setUniform()` to pass data into custom shaders.

Built-In Variables and Functions

Vertex Shader Variables

`float uTime` - Time since startup.
`mat4 uProj, uView, uModel, uInvModel` - Matrices.
`vec3 aPos` - Position in object space.
`vec3 pos` - Position in clip space.
`vec2 aUV` - UV in object space.
`vec3 aNor, aTan, aBi` - Normal, tangent, and bitangent in object space.
`vec3 worldPosition, worldNormal, worldTangent` - Position, normal, and tangent in world space.

Vertex Shader Functions

`vec3 obj2World(vec3 p)` - Transforms a point from object space to world space.
`vec3 obj2Clip(vec3 p)` - Transforms a point from object space to clip space.
`vec3 world2Obj(vec3 p)` - Transforms a point from world space to object space.
`float noise(vec3 p)` - Sample the noise function at p.

Fragment Shader Variables

`float uTime` - Time since startup.
`mat4 uProj, uView, uModel, uInvModel` - Matrices.
`vec3 uViewPosition` - Camera position in world space.
`vec3 vAPos` - Position in object space.
`vec3 vPos` - Position in clip space.
`vec3 vNor, vTan, vBi` - Normal, tangent, and bitangent in world space.
`vec2 vUV` - UV in object space.
`vec3 worldPosition, worldNormal, worldTangent` - Position, normal, and tangent in world space.
`float uOpacity` - Opacity value used in alpha blending.

Fragment Shader Functions

`vec3 obj2World(vec3 p)` - Transforms a point from object space to world space.
`vec3 world2Obj(vec3 p)` - Transforms a point from world space to object space.
`float noise(vec3 p)` - Sample the noise function at p.

Audio APIs

Basic Spatial Audio

See `/js/scenes/demoSoundWrapper.js` for a demo.

```
createSoundSource(soundIndex, soundUrl, initialPosition, loop, volume);
```

- `soundIndex` *(number)*: A unique identifier for the sound source. Must be an integer starting from 0.
- `soundUrl` *(string)*: The relative or absolute URL path to the sound file.
- `initialPosition` *(array)*: A 3D position vector `[x, y, z]` defining the sound source's initial location in space.
- `loop` *(boolean)*: Determines if the sound should loop. Use `true` for looping playback and `false` for single playback.
- `volume` *(number)*: The volume level as a percentage, where `1.0` represents 100% volume.

```
Object = model.addAudio(soundIndex)
```

- Associates a sound source (by soundIndex) with the 3D object.

```
Object.playAudio()
```

- Starts audio playback for the associated sound source.

```
Object.stopAudio()
```

- Stops the audio playback.

Acoustic Properties Setup

See `/js/util/spatial-audio.js`.

`roomDimensions`

- The `roomDimensions` object specifies the size of the room in meters. This includes the width, height, and depth of the virtual space.

```
let roomDimensions = {
  width: 6,       // Width of the room in meters
  height: 2.5,    // Height of the room in meters
  depth: 6        // Depth of the room in meters
};
```

`roomMaterials`

- The `roomMaterials` object defines the material properties of each surface in the room. These properties influence how sound reflects and absorbs on the respective surfaces, simulating realistic acoustics.

```
let roomMaterials = {
  left: 'curtain-heavy',                  // Left wall material
  right: 'curtain-heavy',                 // Right wall material
  front: 'curtain-heavy',                 // Front wall material
  back: 'curtain-heavy',                  // Back wall material
  down: 'polished-concrete-or-tile',      // Floor material
  up: 'wood-ceiling'                      // Ceiling material
};
```