import { useEffect, useRef, useState } from "react";

import { mat4, quat, vec2, vec3 } from "gl-matrix";

const discVertShaderSource = `#version 300 es

uniform mat4 uWorldMatrix;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform vec3 uCameraPosition;

uniform vec4 uRotationAxisVelocity;

in vec3 aModelPosition;

in vec3 aModelNormal;

in vec2 aModelUvs;

in mat4 aInstanceMatrix;

out vec2 vUvs;

out float vAlpha;

flat out int vInstanceId;

#define PI 3.141593

void main() {

vec4 worldPosition = uWorldMatrix \* aInstanceMatrix \* vec4(aModelPosition, 1.);

// center of the disc in world space

vec3 centerPos = (uWorldMatrix \* aInstanceMatrix \* vec4(0., 0., 0., 1.)).xyz;

float radius = length(centerPos.xyz);

// skip the center vertex of the disc geometry

if (gl\_VertexID > 0) {

// stretch the disc according to the axis and velocity of the rotation

vec3 rotationAxis = uRotationAxisVelocity.xyz;

float rotationVelocity = min(.15, uRotationAxisVelocity.w \* 15.);

// the stretch direction is orthogonal to the rotation axis and the position

vec3 stretchDir = normalize(cross(centerPos, rotationAxis));

// the position of this vertex relative to the center position

vec3 relativeVertexPos = normalize(worldPosition.xyz - centerPos);

// vertices more in line with the stretch direction get a larger offset

float strength = dot(stretchDir, relativeVertexPos);

float invAbsStrength = min(0., abs(strength) - 1.);

strength = rotationVelocity \* sign(strength) \* abs(invAbsStrength \* invAbsStrength \* invAbsStrength + 1.);

// apply the stretch distortion

worldPosition.xyz += stretchDir \* strength;

}

// move the vertex back to the overall sphere

worldPosition.xyz = radius \* normalize(worldPosition.xyz);

gl\_Position = uProjectionMatrix \* uViewMatrix \* worldPosition;

vAlpha = smoothstep(0.5, 1., normalize(worldPosition.xyz).z) \* .9 + .1;

vUvs = aModelUvs;

vInstanceId = gl\_InstanceID;

}

`;

const discFragShaderSource = `#version 300 es

precision highp float;

uniform sampler2D uTex;

uniform int uItemCount;

uniform int uAtlasSize;

out vec4 outColor;

in vec2 vUvs;

in float vAlpha;

flat in int vInstanceId;

void main() {

// Calculate which item to display based on instance ID

int itemIndex = vInstanceId % uItemCount;

int cellsPerRow = uAtlasSize;

int cellX = itemIndex % cellsPerRow;

int cellY = itemIndex / cellsPerRow;

vec2 cellSize = vec2(1.0) / vec2(float(cellsPerRow));

vec2 cellOffset = vec2(float(cellX), float(cellY)) \* cellSize;

// Get texture dimensions and calculate aspect ratio

ivec2 texSize = textureSize(uTex, 0);

float imageAspect = float(texSize.x) / float(texSize.y);

float containerAspect = 1.0; // Assuming square container

// Calculate cover scale factor

float scale = max(imageAspect / containerAspect,

containerAspect / imageAspect);

// Rotate 180 degrees and adjust UVs for cover

vec2 st = vec2(vUvs.x, 1.0 - vUvs.y);

st = (st - 0.5) \* scale + 0.5;

// Clamp coordinates to prevent repeating

st = clamp(st, 0.0, 1.0);

// Map to the correct cell in the atlas

st = st \* cellSize + cellOffset;

outColor = texture(uTex, st);

outColor.a \*= vAlpha;

}

`;

class *Face* {

*/\*\**

*\* Creates a new triangle face by the indices of each vertex.*

*\** @param{*number*}a *Index of the first vertex*

*\** @param{*number*}b *Index of the second vertex*

*\** @param{*number*}c *Index of the third vertex*

*\*/*

constructor(*a*, *b*, *c*) {

this.a = a;

this.b = b;

this.c = c;

}

}

class *Vertex* {

constructor(*x*, *y*, *z*) {

this.position = vec3.*fromValues*(x, y, z);

this.normal = vec3.*create*();

this.uv = vec2.*create*();

}

}

class *Geometry* {

constructor() {

this.vertices = [];

this.faces = [];

}

*addVertex*(...*args*) {

for (let i = 0; i < args.length; i += 3) {

this.vertices.*push*(**new** *Vertex*(args[i], args[i + 1], args[i + 2]));

}

return this;

}

*addFace*(...*args*) {

for (let i = 0; i < args.length; i += 3) {

this.faces.*push*(**new** *Face*(args[i], args[i + 1], args[i + 2]));

}

return this;

}

get *lastVertex*() {

return this.vertices[this.vertices.length - 1];

}

*subdivide*(*divisions* = 1) {

const midPointCache = {};

let f = this.faces;

for (let div = 0; div < divisions; ++div) {

const newFaces = **new** *Array*(f.length \* 4);

f.*forEach*((*face*, *ndx*) => {

const mAB = this.*getMidPoint*(face.a, face.b, midPointCache);

const mBC = this.*getMidPoint*(face.b, face.c, midPointCache);

const mCA = this.*getMidPoint*(face.c, face.a, midPointCache);

const i = ndx \* 4;

newFaces[i + 0] = **new** *Face*(face.a, mAB, mCA);

newFaces[i + 1] = **new** *Face*(face.b, mBC, mAB);

newFaces[i + 2] = **new** *Face*(face.c, mCA, mBC);

newFaces[i + 3] = **new** *Face*(mAB, mBC, mCA);

});

f = newFaces;

}

this.faces = f;

return this;

}

*spherize*(*radius* = 1) {

this.vertices.*forEach*((*vertex*) => {

vec3.*normalize*(vertex.normal, vertex.position);

vec3.*scale*(vertex.position, vertex.normal, radius);

});

return this;

}

get *data*() {

return {

vertices: this.vertexData,

indices: this.indexData,

normals: this.normalData,

uvs: this.uvData,

};

}

get *vertexData*() {

return **new** *Float32Array*(

this.vertices.*flatMap*((*v*) => Array.*from*(v.position))

);

}

get *normalData*() {

return **new** *Float32Array*(this.vertices.*flatMap*((*v*) => *Array*.*from*(*v*.normal)));

}

get *uvData*() {

return **new** *Float32Array*(this.vertices.*flatMap*((*v*) => *Array*.*from*(*v*.uv)));

}

get *indexData*() {

return **new** *Uint16Array*(this.faces.*flatMap*((*f*) => [*f*.a, *f*.b, *f*.c]));

}

*getMidPoint*(*ndxA*, *ndxB*, *cache*) {

const cacheKey = *ndxA* < *ndxB* ? `k\_${*ndxB*}\_${*ndxA*}` : `k\_${*ndxA*}\_${*ndxB*}`;

if (*Object*.prototype.*hasOwnProperty*.*call*(*cache*, cacheKey)) {

return *cache*[cacheKey];

}

const a = this.vertices[*ndxA*].position;

const b = this.vertices[*ndxB*].position;

const ndx = this.vertices.length;

*cache*[cacheKey] = ndx;

this.*addVertex*(

(a[0] + b[0]) \* 0.5,

(a[1] + b[1]) \* 0.5,

(a[2] + b[2]) \* 0.5

);

return ndx;

}

}

class *IcosahedronGeometry* extends *Geometry* {

constructor() {

super();

const t = Math.*sqrt*(5) \* 0.5 + 0.5;

this.*addVertex*(

-1,

t,

0,

1,

t,

0,

-1,

-t,

0,

1,

-t,

0,

0,

-1,

t,

0,

1,

t,

0,

-1,

-t,

0,

1,

-t,

t,

0,

-1,

t,

0,

1,

-t,

0,

-1,

-t,

0,

1

).*addFace*(

0,

11,

5,

0,

5,

1,

0,

1,

7,

0,

7,

10,

0,

10,

11,

1,

5,

9,

5,

11,

4,

11,

10,

2,

10,

7,

6,

7,

1,

8,

3,

9,

4,

3,

4,

2,

3,

2,

6,

3,

6,

8,

3,

8,

9,

4,

9,

5,

2,

4,

11,

6,

2,

10,

8,

6,

7,

9,

8,

1

);

}

}

class *DiscGeometry* extends *Geometry* {

constructor(*steps* = 4, *radius* = 1) {

super();

steps = Math.*max*(4, steps);

const alpha = (2 \* Math.PI) / steps;

this.*addVertex*(0, 0, 0);

this.lastVertex.uv[0] = 0.5;

this.lastVertex.uv[1] = 0.5;

for (let i = 0; i < steps; ++i) {

const x = Math.*cos*(alpha \* i);

const y = Math.*sin*(alpha \* i);

this.*addVertex*(radius \* x, radius \* y, 0);

this.lastVertex.uv[0] = x \* 0.5 + 0.5;

this.lastVertex.uv[1] = y \* 0.5 + 0.5;

if (i > 0) {

this.*addFace*(0, i, i + 1);

}

}

this.*addFace*(0, steps, 1);

}

}

function *createShader*(*gl*, *type*, *source*) {

const shader = gl.*createShader*(type);

gl.*shaderSource*(shader, source);

gl.*compileShader*(shader);

const success = *gl*.*getShaderParameter*(shader, *gl*.COMPILE\_STATUS);

if (success) {

return shader;

}

console.*error*(*gl*.*getShaderInfoLog*(shader));

*gl*.*deleteShader*(shader);

return null;

}

function *createProgram*(

*gl*,

*shaderSources*,

*transformFeedbackVaryings*,

*attribLocations*

) {

const program = *gl*.*createProgram*();

[*gl*.VERTEX\_SHADER, *gl*.FRAGMENT\_SHADER].*forEach*((*type*, *ndx*) => {

const shader = *createShader*(*gl*, *type*, *shaderSources*[*ndx*]);

if (shader) *gl*.*attachShader*(program, shader);

});

if (*transformFeedbackVaryings*) {

*gl*.*transformFeedbackVaryings*(

program,

*transformFeedbackVaryings*,

*gl*.SEPARATE\_ATTRIBS

);

}

if (*attribLocations*) {

for (const attrib in *attribLocations*) {

*gl*.*bindAttribLocation*(program, *attribLocations*[attrib], attrib);

}

}

*gl*.*linkProgram*(program);

const success = *gl*.*getProgramParameter*(program, *gl*.LINK\_STATUS);

if (success) {

return program;

}

console.*error*(*gl*.*getProgramInfoLog*(program));

*gl*.*deleteProgram*(program);

return null;

}

function *makeVertexArray*(*gl*, *bufLocNumElmPairs*, *indices*) {

const va = *gl*.*createVertexArray*();

*gl*.*bindVertexArray*(va);

for (const [buffer, loc, numElem] of *bufLocNumElmPairs*) {

if (loc === -1) continue;

*gl*.*bindBuffer*(*gl*.ARRAY\_BUFFER, buffer);

*gl*.*enableVertexAttribArray*(loc);

*gl*.*vertexAttribPointer*(loc, numElem, *gl*.FLOAT, false, 0, 0);

}

if (*indices*) {

const indexBuffer = *gl*.*createBuffer*();

*gl*.*bindBuffer*(*gl*.ELEMENT\_ARRAY\_BUFFER, indexBuffer);

*gl*.*bufferData*(

*gl*.ELEMENT\_ARRAY\_BUFFER,

**new** *Uint16Array*(*indices*),

*gl*.STATIC\_DRAW

);

}

*gl*.*bindVertexArray*(null);

return va;

}

function *resizeCanvasToDisplaySize*(*canvas*) {

const dpr = Math.*min*(2, window.devicePixelRatio);

const displayWidth = Math.*round*(*canvas*.clientWidth \* dpr);

const displayHeight = Math.*round*(*canvas*.clientHeight \* dpr);

const needResize =

*canvas*.width !== displayWidth || *canvas*.height !== displayHeight;

if (needResize) {

*canvas*.width = displayWidth;

*canvas*.height = displayHeight;

}

return needResize;

}

function *makeBuffer*(*gl*, *sizeOrData*, *usage*) {

const buf = *gl*.*createBuffer*();

*gl*.*bindBuffer*(*gl*.ARRAY\_BUFFER, buf);

*gl*.*bufferData*(*gl*.ARRAY\_BUFFER, *sizeOrData*, *usage*);

*gl*.*bindBuffer*(*gl*.ARRAY\_BUFFER, null);

return buf;

}

function *createAndSetupTexture*(*gl*, *minFilter*, *magFilter*, *wrapS*, *wrapT*) {

const texture = *gl*.*createTexture*();

*gl*.*bindTexture*(*gl*.TEXTURE\_2D, texture);

*gl*.*texParameteri*(*gl*.TEXTURE\_2D, *gl*.TEXTURE\_WRAP\_S, *wrapS*);

*gl*.*texParameteri*(*gl*.TEXTURE\_2D, *gl*.TEXTURE\_WRAP\_T, *wrapT*);

*gl*.*texParameteri*(*gl*.TEXTURE\_2D, *gl*.TEXTURE\_MIN\_FILTER, *minFilter*);

*gl*.*texParameteri*(*gl*.TEXTURE\_2D, *gl*.TEXTURE\_MAG\_FILTER, *magFilter*);

return texture;

}

class *ArcballControl* {

*// flag which indicates if the user is currently dragging*

isPointerDown = false;

*// orientation of the object*

orientation = *quat*.*create*();

*// current pointer rotation as a quaternion*

pointerRotation = *quat*.*create*();

*// velocity of rotation*

rotationVelocity = 0;

*// rotation axis*

rotationAxis = vec3.*fromValues*(1, 0, 0);

*// direction to move the snap target to (in world space)*

snapDirection = vec3.*fromValues*(0, 0, -1);

*// direction of the target to move to the snap direction (in world space)*

snapTargetDirection;

EPSILON = 0.1;

IDENTITY\_QUAT = quat.*create*();

constructor(*canvas*, *updateCallback*) {

this.canvas = canvas;

this.updateCallback = updateCallback || (() => null);

this.pointerPos = vec2.*create*();

this.previousPointerPos = vec2.*create*();

this.\_rotationVelocity = 0;

this.\_combinedQuat = quat.*create*();

canvas.*addEventListener*("pointerdown", (*e*) => {

vec2.*set*(this.pointerPos, e.clientX, e.clientY);

vec2.*copy*(this.previousPointerPos, this.pointerPos);

this.isPointerDown = true;

});

canvas.*addEventListener*("pointerup", () => {

this.isPointerDown = false;

});

canvas.*addEventListener*("pointerleave", () => {

this.isPointerDown = false;

});

canvas.*addEventListener*("pointermove", (*e*) => {

if (this.isPointerDown) {

vec2.*set*(this.pointerPos, e.clientX, e.clientY);

}

});

canvas.style.touchAction = "none";

}

*update*(*deltaTime*, *targetFrameDuration* = 16) {

const timeScale = deltaTime / targetFrameDuration + 0.00001;

let angleFactor = timeScale;

let snapRotation = quat.*create*();

if (this.isPointerDown) {

const INTENSITY = 0.3 \* timeScale;

const ANGLE\_AMPLIFICATION = 5 / timeScale;

const midPointerPos = vec2.*sub*(

vec2.*create*(),

this.pointerPos,

this.previousPointerPos

);

vec2.*scale*(midPointerPos, midPointerPos, INTENSITY);

if (vec2.*sqrLen*(midPointerPos) > this.EPSILON) {

vec2.*add*(midPointerPos, this.previousPointerPos, midPointerPos);

const p = this.*#project*(midPointerPos);

const q = this.*#project*(this.previousPointerPos);

const a = vec3.*normalize*(vec3.*create*(), p);

const b = vec3.*normalize*(vec3.*create*(), q);

vec2.*copy*(this.previousPointerPos, midPointerPos);

angleFactor \*= ANGLE\_AMPLIFICATION;

this.*quatFromVectors*(a, b, this.pointerRotation, angleFactor);

} else {

quat.*slerp*(

this.pointerRotation,

this.pointerRotation,

this.IDENTITY\_QUAT,

INTENSITY

);

}

} else {

const INTENSITY = 0.1 \* timeScale;

quat.*slerp*(

this.pointerRotation,

this.pointerRotation,

this.IDENTITY\_QUAT,

INTENSITY

);

if (this.snapTargetDirection) {

const SNAPPING\_INTENSITY = 0.2;

const a = this.snapTargetDirection;

const b = this.snapDirection;

const sqrDist = vec3.*squaredDistance*(a, b);

const distanceFactor = Math.*max*(0.1, 1 - sqrDist \* 10);

angleFactor \*= SNAPPING\_INTENSITY \* distanceFactor;

this.*quatFromVectors*(a, b, snapRotation, angleFactor);

}

}

const combinedQuat = quat.*multiply*(

quat.*create*(),

snapRotation,

this.pointerRotation

);

this.orientation = quat.*multiply*(

quat.*create*(),

combinedQuat,

this.orientation

);

quat.*normalize*(this.orientation, this.orientation);

const RA\_INTENSITY = 0.8 \* timeScale;

quat.*slerp*(

this.\_combinedQuat,

this.\_combinedQuat,

combinedQuat,

RA\_INTENSITY

);

quat.*normalize*(this.\_combinedQuat, this.\_combinedQuat);

const rad = Math.*acos*(this.\_combinedQuat[3]) \* 2.0;

const s = Math.*sin*(rad / 2.0);

let rv = 0;

if (s > 0.000001) {

rv = rad / (2 \* Math.PI);

this.rotationAxis[0] = this.\_combinedQuat[0] / s;

this.rotationAxis[1] = this.\_combinedQuat[1] / s;

this.rotationAxis[2] = this.\_combinedQuat[2] / s;

}

const RV\_INTENSITY = 0.5 \* timeScale;

this.\_rotationVelocity += (rv - this.\_rotationVelocity) \* RV\_INTENSITY;

this.rotationVelocity = this.\_rotationVelocity / timeScale;

this.*updateCallback*(deltaTime);

}

*quatFromVectors*(*a*, *b*, *out*, *angleFactor* = 1) {

const axis = vec3.*cross*(vec3.*create*(), a, b);

vec3.*normalize*(axis, axis);

const d = Math.*max*(-1, Math.*min*(1, vec3.*dot*(a, b)));

const angle = Math.*acos*(d) \* angleFactor;

quat.*setAxisAngle*(out, axis, angle);

return { q: out, axis, angle };

}

#*project*(*pos*) {

const r = 2;

const w = this.canvas.clientWidth;

const h = this.canvas.clientHeight;

const s = Math.*max*(w, h) - 1;

*// map to [-1, 1]*

const x = (2 \* pos[0] - w - 1) / s;

const y = (2 \* pos[1] - h - 1) / s;

let z = 0;

const xySq = x \* x + y \* y;

const rSq = r \* r;

if (xySq <= rSq / 2.0) {

z = Math.*sqrt*(rSq - xySq);

} else {

z = rSq / Math.*sqrt*(xySq);

}

return vec3.*fromValues*(-x, y, z);

}

}

class *InfiniteGridMenu* {

TARGET\_FRAME\_DURATION = 1000 / 60; *// 60 fps*

SPHERE\_RADIUS = 2;

#time = 0;

#deltaTime = 0;

#deltaFrames = 0;

#frames = 0;

camera = {

matrix: mat4.*create*(),

near: 0.1,

far: 40,

fov: Math.PI / 4,

aspect: 1,

position: vec3.*fromValues*(0, 0, 3),

up: vec3.*fromValues*(0, 1, 0),

matrices: {

view: mat4.*create*(),

projection: mat4.*create*(),

inversProjection: mat4.*create*(),

},

};

nearestVertexIndex = null;

smoothRotationVelocity = 0;

scaleFactor = 1.0; *// default*

movementActive = false;

constructor(

*canvas*,

*items*,

*onActiveItemChange*,

*onMovementChange*,

*onInit* = null

) {

this.canvas = canvas;

this.items = items || [];

this.onActiveItemChange = onActiveItemChange || (() => {});

this.onMovementChange = onMovementChange || (() => {});

this.*#init*(onInit);

}

*resize*() {

this.viewportSize = vec2.*set*(

this.viewportSize || vec2.*create*(),

this.canvas.clientWidth,

this.canvas.clientHeight

);

const gl = this.gl;

const needsResize = *resizeCanvasToDisplaySize*(gl.canvas);

if (needsResize) {

gl.*viewport*(0, 0, gl.drawingBufferWidth, gl.drawingBufferHeight);

}

this.*#updateProjectionMatrix*(gl);

}

*run*(*time* = 0) {

this.#deltaTime = Math.*min*(32, time - this.#time);

this.#time = time;

this.#deltaFrames = this.#deltaTime / this.TARGET\_FRAME\_DURATION;

this.#frames += this.#deltaFrames;

this.*#animate*(this.#deltaTime);

this.*#render*();

*requestAnimationFrame*((*t*) => this.*run*(t));

}

#*init*(*onInit*) {

this.gl = this.canvas.*getContext*("webgl2", {

antialias: true,

alpha: false,

});

const gl = this.gl;

if (!gl) {

throw **new** *Error*("No WebGL 2 context!");

}

this.viewportSize = *vec2*.*fromValues*(

this.canvas.clientWidth,

this.canvas.clientHeight

);

this.drawBufferSize = *vec2*.*clone*(this.viewportSize);

this.discProgram = *createProgram*(

gl,

[discVertShaderSource, discFragShaderSource],

null,

{

aModelPosition: 0,

aModelNormal: 1,

aModelUvs: 2,

aInstanceMatrix: 3,

}

);

this.discLocations = {

aModelPosition: gl.*getAttribLocation*(this.discProgram, "aModelPosition"),

aModelUvs: gl.*getAttribLocation*(this.discProgram, "aModelUvs"),

aInstanceMatrix: gl.*getAttribLocation*(

this.discProgram,

"aInstanceMatrix"

),

uWorldMatrix: gl.*getUniformLocation*(this.discProgram, "uWorldMatrix"),

uViewMatrix: gl.*getUniformLocation*(this.discProgram, "uViewMatrix"),

uProjectionMatrix: gl.*getUniformLocation*(

this.discProgram,

"uProjectionMatrix"

),

uCameraPosition: gl.*getUniformLocation*(

this.discProgram,

"uCameraPosition"

),

uScaleFactor: gl.*getUniformLocation*(this.discProgram, "uScaleFactor"),

uRotationAxisVelocity: gl.*getUniformLocation*(

this.discProgram,

"uRotationAxisVelocity"

),

uTex: gl.*getUniformLocation*(this.discProgram, "uTex"),

uFrames: gl.*getUniformLocation*(this.discProgram, "uFrames"),

uItemCount: gl.*getUniformLocation*(this.discProgram, "uItemCount"),

uAtlasSize: gl.*getUniformLocation*(this.discProgram, "uAtlasSize"),

};

this.discGeo = **new** *DiscGeometry*(56, 1);

this.discBuffers = this.discGeo.data;

this.discVAO = *makeVertexArray*(

gl,

[

[

*makeBuffer*(gl, this.discBuffers.vertices, gl.STATIC\_DRAW),

this.discLocations.aModelPosition,

3,

],

[

*makeBuffer*(gl, this.discBuffers.uvs, gl.STATIC\_DRAW),

this.discLocations.aModelUvs,

2,

],

],

this.discBuffers.indices

);

this.icoGeo = **new** *IcosahedronGeometry*();

this.icoGeo.*subdivide*(1).*spherize*(this.SPHERE\_RADIUS);

this.instancePositions = this.icoGeo.vertices.*map*((*v*) => v.position);

this.DISC\_INSTANCE\_COUNT = this.icoGeo.vertices.length;

this.*#initDiscInstances*(this.DISC\_INSTANCE\_COUNT);

this.worldMatrix = mat4.*create*();

this.*#initTexture*();

this.control = **new** *ArcballControl*(this.canvas, (*deltaTime*) =>

this.*#onControlUpdate*(deltaTime)

);

this.*#updateCameraMatrix*();

this.*#updateProjectionMatrix*(gl);

this.*resize*();

if (onInit) *onInit*(this);

}

#*initTexture*() {

const gl = this.gl;

this.tex = *createAndSetupTexture*(

gl,

gl.LINEAR,

gl.LINEAR,

gl.CLAMP\_TO\_EDGE,

gl.CLAMP\_TO\_EDGE

);

const itemCount = Math.*max*(1, this.items.length);

this.atlasSize = Math.*ceil*(Math.*sqrt*(itemCount));

const canvas = document.*createElement*("canvas");

const ctx = canvas.*getContext*("2d");

const cellSize = 512;

canvas.width = this.atlasSize \* cellSize;

canvas.height = this.atlasSize \* cellSize;

*Promise*.*all*(

this.items.*map*(

(*item*) =>

**new** *Promise*((*resolve*) => {

const img = **new** *Image*();

img.crossOrigin = "anonymous";

img.*onload* = () => *resolve*(img);

img.src = item.image;

})

)

).*then*((*images*) => {

images.*forEach*((*img*, *i*) => {

const x = (i % this.atlasSize) \* cellSize;

const y = Math.*floor*(i / this.atlasSize) \* cellSize;

ctx.*drawImage*(img, x, y, cellSize, cellSize);

});

gl.*bindTexture*(gl.TEXTURE\_2D, this.tex);

gl.*texImage2D*(

gl.TEXTURE\_2D,

0,

gl.RGBA,

gl.RGBA,

gl.UNSIGNED\_BYTE,

canvas

);

gl.*generateMipmap*(gl.TEXTURE\_2D);

});

}

#*initDiscInstances*(*count*) {

const gl = this.gl;

this.discInstances = {

matricesArray: **new** *Float32Array*(count \* 16),

matrices: [],

buffer: gl.*createBuffer*(),

};

for (let i = 0; i < count; ++i) {

const instanceMatrixArray = **new** *Float32Array*(

this.discInstances.matricesArray.buffer,

i \* 16 \* 4,

16

);

instanceMatrixArray.*set*(mat4.*create*());

this.discInstances.matrices.*push*(instanceMatrixArray);

}

gl.*bindVertexArray*(this.discVAO);

gl.*bindBuffer*(gl.ARRAY\_BUFFER, this.discInstances.buffer);

gl.*bufferData*(

gl.ARRAY\_BUFFER,

this.discInstances.matricesArray.byteLength,

gl.DYNAMIC\_DRAW

);

const mat4AttribSlotCount = 4;

const bytesPerMatrix = 16 \* 4;

for (let j = 0; j < mat4AttribSlotCount; ++j) {

const loc = this.discLocations.aInstanceMatrix + j;

gl.*enableVertexAttribArray*(loc);

gl.*vertexAttribPointer*(

loc,

4,

gl.FLOAT,

false,

bytesPerMatrix,

j \* 4 \* 4

);

gl.*vertexAttribDivisor*(loc, 1);

}

gl.*bindBuffer*(gl.ARRAY\_BUFFER, null);

gl.*bindVertexArray*(null);

}

#*animate*(*deltaTime*) {

const gl = this.gl;

this.control.*update*(deltaTime, this.TARGET\_FRAME\_DURATION);

let positions = this.instancePositions.*map*((*p*) =>

vec3.*transformQuat*(vec3.*create*(), p, this.control.orientation)

);

const scale = 0.25;

const SCALE\_INTENSITY = 0.6;

positions.*forEach*((*p*, *ndx*) => {

const s =

(Math.*abs*(p[2]) / this.SPHERE\_RADIUS) \* SCALE\_INTENSITY +

(1 - SCALE\_INTENSITY);

const finalScale = s \* scale;

const matrix = mat4.*create*();

mat4.*multiply*(

matrix,

matrix,

mat4.*fromTranslation*(mat4.*create*(), vec3.*negate*(vec3.*create*(), p))

);

mat4.*multiply*(

matrix,

matrix,

mat4.*targetTo*(mat4.*create*(), [0, 0, 0], p, [0, 1, 0])

);

mat4.*multiply*(

matrix,

matrix,

mat4.*fromScaling*(mat4.*create*(), [finalScale, finalScale, finalScale])

);

mat4.*multiply*(

matrix,

matrix,

mat4.*fromTranslation*(mat4.*create*(), [0, 0, -this.SPHERE\_RADIUS])

);

mat4.*copy*(this.discInstances.matrices[ndx], matrix);

});

gl.*bindBuffer*(gl.ARRAY\_BUFFER, this.discInstances.buffer);

gl.*bufferSubData*(gl.ARRAY\_BUFFER, 0, this.discInstances.matricesArray);

gl.*bindBuffer*(gl.ARRAY\_BUFFER, null);

this.smoothRotationVelocity = this.control.rotationVelocity;

}

#*render*() {

const gl = this.gl;

gl.*useProgram*(this.discProgram);

gl.*enable*(gl.CULL\_FACE);

gl.*enable*(gl.DEPTH\_TEST);

gl.*clearColor*(0, 0, 0, 0);

gl.*clear*(gl.COLOR\_BUFFER\_BIT | gl.DEPTH\_BUFFER\_BIT);

gl.*uniformMatrix4fv*(

this.discLocations.uWorldMatrix,

false,

this.worldMatrix

);

gl.*uniformMatrix4fv*(

this.discLocations.uViewMatrix,

false,

this.camera.matrices.view

);

gl.*uniformMatrix4fv*(

this.discLocations.uProjectionMatrix,

false,

this.camera.matrices.projection

);

gl.*uniform3f*(

this.discLocations.uCameraPosition,

this.camera.position[0],

this.camera.position[1],

this.camera.position[2]

);

gl.*uniform4f*(

this.discLocations.uRotationAxisVelocity,

this.control.rotationAxis[0],

this.control.rotationAxis[1],

this.control.rotationAxis[2],

this.smoothRotationVelocity \* 1.1

);

*// NEW UNIFORMS*

gl.*uniform1i*(this.discLocations.uItemCount, this.items.length);

gl.*uniform1i*(this.discLocations.uAtlasSize, this.atlasSize);

gl.*uniform1f*(this.discLocations.uFrames, this.#frames);

gl.*uniform1f*(this.discLocations.uScaleFactor, this.scaleFactor);

gl.*uniform1i*(this.discLocations.uTex, 0);

gl.*activeTexture*(gl.TEXTURE0);

gl.*bindTexture*(gl.TEXTURE\_2D, this.tex);

gl.*bindVertexArray*(this.discVAO);

gl.*drawElementsInstanced*(

gl.TRIANGLES,

this.discBuffers.indices.length,

gl.UNSIGNED\_SHORT,

0,

this.DISC\_INSTANCE\_COUNT

);

}

#*updateCameraMatrix*() {

mat4.*targetTo*(

this.camera.matrix,

this.camera.position,

[0, 0, 0],

this.camera.up

);

mat4.*invert*(this.camera.matrices.view, this.camera.matrix);

}

#*updateProjectionMatrix*(*gl*) {

this.camera.aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;

const height = this.SPHERE\_RADIUS \* 0.35;

const distance = this.camera.position[2];

if (this.camera.aspect > 1) {

this.camera.fov = 2 \* Math.*atan*(height / distance);

} else {

this.camera.fov = 2 \* Math.*atan*(height / this.camera.aspect / distance);

}

mat4.*perspective*(

this.camera.matrices.projection,

this.camera.fov,

this.camera.aspect,

this.camera.near,

this.camera.far

);

mat4.*invert*(

this.camera.matrices.inversProjection,

this.camera.matrices.projection

);

}

#*onControlUpdate*(*deltaTime*) {

const timeScale = *deltaTime* / this.TARGET\_FRAME\_DURATION + 0.0001;

let damping = 5 / timeScale;

let cameraTargetZ = 3;

const isMoving =

this.control.isPointerDown ||

Math.*abs*(this.smoothRotationVelocity) > 0.01;

if (isMoving !== this.movementActive) {

this.movementActive = isMoving;

this.*onMovementChange*(isMoving);

}

if (!this.control.isPointerDown) {

const nearestVertexIndex = this.*#findNearestVertexIndex*();

const itemIndex = nearestVertexIndex % Math.*max*(1, this.items.length);

this.*onActiveItemChange*(itemIndex);

const snapDirection = *vec3*.*normalize*(

*vec3*.*create*(),

this.*#getVertexWorldPosition*(nearestVertexIndex)

);

this.control.snapTargetDirection = snapDirection;

} else {

cameraTargetZ += this.control.rotationVelocity \* 80 + 2.5;

damping = 7 / timeScale;

}

this.camera.position[2] +=

(cameraTargetZ - this.camera.position[2]) / damping;

this.*#updateCameraMatrix*();

}

#*findNearestVertexIndex*() {

const n = this.control.snapDirection;

const inversOrientation = *quat*.*conjugate*(

*quat*.*create*(),

this.control.orientation

);

const nt = *vec3*.*transformQuat*(*vec3*.*create*(), n, inversOrientation);

let maxD = -1;

let nearestVertexIndex;

for (let i = 0; i < this.instancePositions.length; ++i) {

const d = *vec3*.*dot*(nt, this.instancePositions[i]);

if (d > maxD) {

maxD = d;

nearestVertexIndex = i;

}

}

return nearestVertexIndex;

}

#*getVertexWorldPosition*(*index*) {

const nearestVertexPos = this.instancePositions[*index*];

return *vec3*.*transformQuat*(

*vec3*.*create*(),

nearestVertexPos,

this.control.orientation

);

}

}

const defaultItems = [

{

image: "https://picsum.photos/900/900?grayscale",

link: "https://google.com/",

title: "",

description: "",

},

];

export default function *InfiniteMenu*({ *items* = [] }) {

const canvasRef = *useRef*(null);

const [activeItem, *setActiveItem*] = *useState*(null);

const [isMoving, *setIsMoving*] = *useState*(false);

*useEffect*(() => {

const canvas = canvasRef.current;

let sketch;

const *handleActiveItem* = (*index*) => {

const itemIndex = *index* % *items*.length;

*setActiveItem*(*items*[itemIndex]);

};

if (canvas) {

sketch = **new** *InfiniteGridMenu*(

canvas,

*items*.length ? *items* : defaultItems,

*handleActiveItem*,

*setIsMoving*,

(*sk*) => *sk*.*run*()

);

}

const *handleResize* = () => {

if (sketch) {

sketch.*resize*();

}

};

window.*addEventListener*("resize", *handleResize*);

*handleResize*();

return () => {

window.*removeEventListener*("resize", *handleResize*);

};

}, [*items*]);

const *handleButtonClick* = () => {

if (!activeItem?.link) return;

if (activeItem.link.*startsWith*("http")) {

window.*open*(activeItem.link, "\_blank");

} else {

console.*log*("Internal route:", activeItem.link);

}

};

return (

<div className="relative w-full h-full">

<canvas

id="infinite-grid-menu-canvas"

ref={canvasRef}

className="cursor-grab w-full h-full overflow-hidden object-cover relative outline-none active:cursor-grabbing"

/>

{activeItem && (

<>

{*/\* Title \*/*}

<h2

className={`

select-none

absolute

font-black

[font-size:4rem]

left-[1.6em]

top-1/2

sm:block

hidden

transform

translate-x-[20%]

-translate-y-1/2

transition-all

ease-[cubic-bezier(0.25,0.1,0.25,1.0)]

${

isMoving

? "opacity-0 pointer-events-none duration-[100ms]"

: "opacity-100 pointer-events-auto duration-[500ms]"

}

`}

>

{activeItem.title}

</h2>

{*/\* Description \*/*}

<p

className={`

select-none

absolute

max-w-[10ch]

text-[1.5rem]

top-1/2

sm:block

hidden

right-[1%]

transition-all

ease-[cubic-bezier(0.25,0.1,0.25,1.0)]

${

isMoving

? "opacity-0 pointer-events-none duration-[100ms] translate-x-[-60%] -translate-y-1/2"

: "opacity-100 pointer-events-auto duration-[500ms] translate-x-[-90%] -translate-y-1/2"

}

`}

>

{activeItem.description}

</p>

{*/\* Action Button \*/*}

<div

onClick={*handleButtonClick*}

className={`

absolute

left-1/2

import { useEffect, useRef, useState } from "react";

import { mat4, quat, vec2, vec3 } from "gl-matrix";

const discVertShaderSource = `#version 300 es

uniform mat4 uWorldMatrix;

uniform mat4 uViewMatrix;

uniform mat4 uProjectionMatrix;

uniform vec3 uCameraPosition;

uniform vec4 uRotationAxisVelocity;

in vec3 aModelPosition;

in vec3 aModelNormal;

in vec2 aModelUvs;

in mat4 aInstanceMatrix;

out vec2 vUvs;

out float vAlpha;

flat out int vInstanceId;

#define PI 3.141593

void main() {

vec4 worldPosition = uWorldMatrix \* aInstanceMatrix \* vec4(aModelPosition, 1.);

// center of the disc in world space

vec3 centerPos = (uWorldMatrix \* aInstanceMatrix \* vec4(0., 0., 0., 1.)).xyz;

float radius = length(centerPos.xyz);

// skip the center vertex of the disc geometry

if (gl\_VertexID > 0) {

// stretch the disc according to the axis and velocity of the rotation

vec3 rotationAxis = uRotationAxisVelocity.xyz;

float rotationVelocity = min(.15, uRotationAxisVelocity.w \* 15.);

// the stretch direction is orthogonal to the rotation axis and the position

vec3 stretchDir = normalize(cross(centerPos, rotationAxis));

// the position of this vertex relative to the center position

vec3 relativeVertexPos = normalize(worldPosition.xyz - centerPos);

// vertices more in line with the stretch direction get a larger offset

float strength = dot(stretchDir, relativeVertexPos);

float invAbsStrength = min(0., abs(strength) - 1.);

strength = rotationVelocity \* sign(strength) \* abs(invAbsStrength \* invAbsStrength \* invAbsStrength + 1.);

// apply the stretch distortion

worldPosition.xyz += stretchDir \* strength;

}

// move the vertex back to the overall sphere

worldPosition.xyz = radius \* normalize(worldPosition.xyz);

gl\_Position = uProjectionMatrix \* uViewMatrix \* worldPosition;

vAlpha = smoothstep(0.5, 1., normalize(worldPosition.xyz).z) \* .9 + .1;

vUvs = aModelUvs;

vInstanceId = gl\_InstanceID;

}

`;

const discFragShaderSource = `#version 300 es

precision highp float;

uniform sampler2D uTex;

uniform int uItemCount;

uniform int uAtlasSize;

out vec4 outColor;

in vec2 vUvs;

in float vAlpha;

flat in int vInstanceId;

void main() {

// Calculate which item to display based on instance ID

int itemIndex = vInstanceId % uItemCount;

int cellsPerRow = uAtlasSize;

int cellX = itemIndex % cellsPerRow;

int cellY = itemIndex / cellsPerRow;

vec2 cellSize = vec2(1.0) / vec2(float(cellsPerRow));

vec2 cellOffset = vec2(float(cellX), float(cellY)) \* cellSize;

// Get texture dimensions and calculate aspect ratio

ivec2 texSize = textureSize(uTex, 0);

float imageAspect = float(texSize.x) / float(texSize.y);

float containerAspect = 1.0; // Assuming square container

// Calculate cover scale factor

float scale = max(imageAspect / containerAspect,

containerAspect / imageAspect);

// Rotate 180 degrees and adjust UVs for cover

vec2 st = vec2(vUvs.x, 1.0 - vUvs.y);

st = (st - 0.5) \* scale + 0.5;

// Clamp coordinates to prevent repeating

st = clamp(st, 0.0, 1.0);

// Map to the correct cell in the atlas

st = st \* cellSize + cellOffset;

outColor = texture(uTex, st);

outColor.a \*= vAlpha;

}

`;

class *Face* {

*/\*\**

*\* Creates a new triangle face by the indices of each vertex.*

*\** @param{*number*}a *Index of the first vertex*

*\** @param{*number*}b *Index of the second vertex*

*\** @param{*number*}c *Index of the third vertex*

*\*/*

constructor(*a*, *b*, *c*) {

this.a = a;

this.b = b;

this.c = c;

}

}

class *Vertex* {

constructor(*x*, *y*, *z*) {

this.position = vec3.*fromValues*(x, y, z);

this.normal = vec3.*create*();

this.uv = vec2.*create*();

}

}

class *Geometry* {

constructor() {

this.vertices = [];

this.faces = [];

}

*addVertex*(...*args*) {

for (let i = 0; i < args.length; i += 3) {

this.vertices.*push*(**new** *Vertex*(args[i], args[i + 1], args[i + 2]));

}

return this;

}

*addFace*(...*args*) {

for (let i = 0; i < args.length; i += 3) {

this.faces.*push*(**new** *Face*(args[i], args[i + 1], args[i + 2]));

}

return this;

}

get *lastVertex*() {

return this.vertices[this.vertices.length - 1];

}

*subdivide*(*divisions* = 1) {

const midPointCache = {};

let f = this.faces;

for (let div = 0; div < divisions; ++div) {

const newFaces = **new** *Array*(f.length \* 4);

f.*forEach*((*face*, *ndx*) => {

const mAB = this.*getMidPoint*(face.a, face.b, midPointCache);

const mBC = this.*getMidPoint*(face.b, face.c, midPointCache);

const mCA = this.*getMidPoint*(face.c, face.a, midPointCache);

const i = ndx \* 4;

newFaces[i + 0] = **new** *Face*(face.a, mAB, mCA);

newFaces[i + 1] = **new** *Face*(face.b, mBC, mAB);

newFaces[i + 2] = **new** *Face*(face.c, mCA, mBC);

newFaces[i + 3] = **new** *Face*(mAB, mBC, mCA);

});

f = newFaces;

}

this.faces = f;

return this;

}

*spherize*(*radius* = 1) {

this.vertices.*forEach*((*vertex*) => {

vec3.*normalize*(vertex.normal, vertex.position);

vec3.*scale*(vertex.position, vertex.normal, radius);

});

return this;

}

get *data*() {

return {

vertices: this.vertexData,

indices: this.indexData,

normals: this.normalData,

uvs: this.uvData,

};

}

get *vertexData*() {

return **new** *Float32Array*(

this.vertices.*flatMap*((*v*) => Array.*from*(v.position))

);

}

get *normalData*() {

return **new** *Float32Array*(this.vertices.*flatMap*((*v*) => *Array*.*from*(*v*.normal)));

}

get *uvData*() {

return **new** *Float32Array*(this.vertices.*flatMap*((*v*) => *Array*.*from*(*v*.uv)));

}

get *indexData*() {

return **new** *Uint16Array*(this.faces.*flatMap*((*f*) => [*f*.a, *f*.b, *f*.c]));

}

*getMidPoint*(*ndxA*, *ndxB*, *cache*) {

const cacheKey = *ndxA* < *ndxB* ? `k\_${*ndxB*}\_${*ndxA*}` : `k\_${*ndxA*}\_${*ndxB*}`;

if (*Object*.prototype.*hasOwnProperty*.*call*(*cache*, cacheKey)) {

return *cache*[cacheKey];

}

const a = this.vertices[*ndxA*].position;

const b = this.vertices[*ndxB*].position;

const ndx = this.vertices.length;

*cache*[cacheKey] = ndx;

this.*addVertex*(

(a[0] + b[0]) \* 0.5,

(a[1] + b[1]) \* 0.5,

(a[2] + b[2]) \* 0.5

);

return ndx;

}

}

class *IcosahedronGeometry* extends *Geometry* {

constructor() {

super();

const t = Math.*sqrt*(5) \* 0.5 + 0.5;

this.*addVertex*(

-1,

t,

0,

1,

t,

0,

-1,

-t,

0,

1,

-t,

0,

0,

-1,

t,

0,

1,

t,

0,

-1,

-t,

0,

1,

-t,

t,

0,

-1,

t,

0,

1,

-t,

0,

-1,

-t,

0,

1

).*addFace*(

0,

11,

5,

0,

5,

1,

0,

1,

7,

0,

7,

10,

0,

10,

11,

1,

5,

9,

5,

11,

4,

11,

10,

2,

10,

7,

6,

7,

1,

8,

3,

9,

4,

3,

4,

2,

3,

2,

6,

3,

6,

8,

3,

8,

9,

4,

9,

5,

2,

4,

11,

6,

2,

10,

8,

6,

7,

9,

8,

1

);

}

}

class *DiscGeometry* extends *Geometry* {

constructor(*steps* = 4, *radius* = 1) {

super();

steps = Math.*max*(4, steps);

const alpha = (2 \* Math.PI) / steps;

this.*addVertex*(0, 0, 0);

this.lastVertex.uv[0] = 0.5;

this.lastVertex.uv[1] = 0.5;

for (let i = 0; i < steps; ++i) {

const x = Math.*cos*(alpha \* i);

const y = Math.*sin*(alpha \* i);

this.*addVertex*(radius \* x, radius \* y, 0);

this.lastVertex.uv[0] = x \* 0.5 + 0.5;

this.lastVertex.uv[1] = y \* 0.5 + 0.5;

if (i > 0) {

this.*addFace*(0, i, i + 1);

}

}

this.*addFace*(0, steps, 1);

}

}

function *createShader*(*gl*, *type*, *source*) {

const shader = gl.*createShader*(type);

gl.*shaderSource*(shader, source);

gl.*compileShader*(shader);

const success = *gl*.*getShaderParameter*(shader, *gl*.COMPILE\_STATUS);

if (success) {

return shader;

}

console.*error*(*gl*.*getShaderInfoLog*(shader));

*gl*.*deleteShader*(shader);

return null;

}

function *createProgram*(

*gl*,

*shaderSources*,

*transformFeedbackVaryings*,

*attribLocations*

) {

const program = *gl*.*createProgram*();

[*gl*.VERTEX\_SHADER, *gl*.FRAGMENT\_SHADER].*forEach*((*type*, *ndx*) => {

const shader = *createShader*(*gl*, *type*, *shaderSources*[*ndx*]);

if (shader) *gl*.*attachShader*(program, shader);

});

if (*transformFeedbackVaryings*) {

*gl*.*transformFeedbackVaryings*(

program,

*transformFeedbackVaryings*,

*gl*.SEPARATE\_ATTRIBS

);

}

if (*attribLocations*) {

for (const attrib in *attribLocations*) {

*gl*.*bindAttribLocation*(program, *attribLocations*[attrib], attrib);

}

}

*gl*.*linkProgram*(program);

const success = *gl*.*getProgramParameter*(program, *gl*.LINK\_STATUS);

if (success) {

return program;

}

console.*error*(*gl*.*getProgramInfoLog*(program));

*gl*.*deleteProgram*(program);

return null;

}

function *makeVertexArray*(*gl*, *bufLocNumElmPairs*, *indices*) {

const va = *gl*.*createVertexArray*();

*gl*.*bindVertexArray*(va);

for (const [buffer, loc, numElem] of *bufLocNumElmPairs*) {

if (loc === -1) continue;

*gl*.*bindBuffer*(*gl*.ARRAY\_BUFFER, buffer);

*gl*.*enableVertexAttribArray*(loc);

*gl*.*vertexAttribPointer*(loc, numElem, *gl*.FLOAT, false, 0, 0);

}

if (*indices*) {

const indexBuffer = *gl*.*createBuffer*();

*gl*.*bindBuffer*(*gl*.ELEMENT\_ARRAY\_BUFFER, indexBuffer);

*gl*.*bufferData*(

*gl*.ELEMENT\_ARRAY\_BUFFER,

**new** *Uint16Array*(*indices*),

*gl*.STATIC\_DRAW

);

}

*gl*.*bindVertexArray*(null);

return va;

}

function *resizeCanvasToDisplaySize*(*canvas*) {

const dpr = Math.*min*(2, window.devicePixelRatio);

const displayWidth = Math.*round*(*canvas*.clientWidth \* dpr);

const displayHeight = Math.*round*(*canvas*.clientHeight \* dpr);

const needResize =

*canvas*.width !== displayWidth || *canvas*.height !== displayHeight;

if (needResize) {

*canvas*.width = displayWidth;

*canvas*.height = displayHeight;

}

return needResize;

}

function *makeBuffer*(*gl*, *sizeOrData*, *usage*) {

const buf = *gl*.*createBuffer*();

*gl*.*bindBuffer*(*gl*.ARRAY\_BUFFER, buf);

*gl*.*bufferData*(*gl*.ARRAY\_BUFFER, *sizeOrData*, *usage*);

*gl*.*bindBuffer*(*gl*.ARRAY\_BUFFER, null);

return buf;

}

function *createAndSetupTexture*(*gl*, *minFilter*, *magFilter*, *wrapS*, *wrapT*) {

const texture = *gl*.*createTexture*();

*gl*.*bindTexture*(*gl*.TEXTURE\_2D, texture);

*gl*.*texParameteri*(*gl*.TEXTURE\_2D, *gl*.TEXTURE\_WRAP\_S, *wrapS*);

*gl*.*texParameteri*(*gl*.TEXTURE\_2D, *gl*.TEXTURE\_WRAP\_T, *wrapT*);

*gl*.*texParameteri*(*gl*.TEXTURE\_2D, *gl*.TEXTURE\_MIN\_FILTER, *minFilter*);

*gl*.*texParameteri*(*gl*.TEXTURE\_2D, *gl*.TEXTURE\_MAG\_FILTER, *magFilter*);

return texture;

}

class *ArcballControl* {

*// flag which indicates if the user is currently dragging*

isPointerDown = false;

*// orientation of the object*

orientation = *quat*.*create*();

*// current pointer rotation as a quaternion*

pointerRotation = *quat*.*create*();

*// velocity of rotation*

rotationVelocity = 0;

*// rotation axis*

rotationAxis = vec3.*fromValues*(1, 0, 0);

*// direction to move the snap target to (in world space)*

snapDirection = vec3.*fromValues*(0, 0, -1);

*// direction of the target to move to the snap direction (in world space)*

snapTargetDirection;

EPSILON = 0.1;

IDENTITY\_QUAT = quat.*create*();

constructor(*canvas*, *updateCallback*) {

this.canvas = canvas;

this.updateCallback = updateCallback || (() => null);

this.pointerPos = vec2.*create*();

this.previousPointerPos = vec2.*create*();

this.\_rotationVelocity = 0;

this.\_combinedQuat = quat.*create*();

canvas.*addEventListener*("pointerdown", (*e*) => {

vec2.*set*(this.pointerPos, e.clientX, e.clientY);

vec2.*copy*(this.previousPointerPos, this.pointerPos);

this.isPointerDown = true;

});

canvas.*addEventListener*("pointerup", () => {

this.isPointerDown = false;

});

canvas.*addEventListener*("pointerleave", () => {

this.isPointerDown = false;

});

canvas.*addEventListener*("pointermove", (*e*) => {

if (this.isPointerDown) {

vec2.*set*(this.pointerPos, e.clientX, e.clientY);

}

});

canvas.style.touchAction = "none";

}

*update*(*deltaTime*, *targetFrameDuration* = 16) {

const timeScale = deltaTime / targetFrameDuration + 0.00001;

let angleFactor = timeScale;

let snapRotation = quat.*create*();

if (this.isPointerDown) {

const INTENSITY = 0.3 \* timeScale;

const ANGLE\_AMPLIFICATION = 5 / timeScale;

const midPointerPos = vec2.*sub*(

vec2.*create*(),

this.pointerPos,

this.previousPointerPos

);

vec2.*scale*(midPointerPos, midPointerPos, INTENSITY);

if (vec2.*sqrLen*(midPointerPos) > this.EPSILON) {

vec2.*add*(midPointerPos, this.previousPointerPos, midPointerPos);

const p = this.*#project*(midPointerPos);

const q = this.*#project*(this.previousPointerPos);

const a = vec3.*normalize*(vec3.*create*(), p);

const b = vec3.*normalize*(vec3.*create*(), q);

vec2.*copy*(this.previousPointerPos, midPointerPos);

angleFactor \*= ANGLE\_AMPLIFICATION;

this.*quatFromVectors*(a, b, this.pointerRotation, angleFactor);

} else {

quat.*slerp*(

this.pointerRotation,

this.pointerRotation,

this.IDENTITY\_QUAT,

INTENSITY

);

}

} else {

const INTENSITY = 0.1 \* timeScale;

quat.*slerp*(

this.pointerRotation,

this.pointerRotation,

this.IDENTITY\_QUAT,

INTENSITY

);

if (this.snapTargetDirection) {

const SNAPPING\_INTENSITY = 0.2;

const a = this.snapTargetDirection;

const b = this.snapDirection;

const sqrDist = vec3.*squaredDistance*(a, b);

const distanceFactor = Math.*max*(0.1, 1 - sqrDist \* 10);

angleFactor \*= SNAPPING\_INTENSITY \* distanceFactor;

this.*quatFromVectors*(a, b, snapRotation, angleFactor);

}

}

const combinedQuat = quat.*multiply*(

quat.*create*(),

snapRotation,

this.pointerRotation

);

this.orientation = quat.*multiply*(

quat.*create*(),

combinedQuat,

this.orientation

);

quat.*normalize*(this.orientation, this.orientation);

const RA\_INTENSITY = 0.8 \* timeScale;

quat.*slerp*(

this.\_combinedQuat,

this.\_combinedQuat,

combinedQuat,

RA\_INTENSITY

);

quat.*normalize*(this.\_combinedQuat, this.\_combinedQuat);

const rad = Math.*acos*(this.\_combinedQuat[3]) \* 2.0;

const s = Math.*sin*(rad / 2.0);

let rv = 0;

if (s > 0.000001) {

rv = rad / (2 \* Math.PI);

this.rotationAxis[0] = this.\_combinedQuat[0] / s;

this.rotationAxis[1] = this.\_combinedQuat[1] / s;

this.rotationAxis[2] = this.\_combinedQuat[2] / s;

}

const RV\_INTENSITY = 0.5 \* timeScale;

this.\_rotationVelocity += (rv - this.\_rotationVelocity) \* RV\_INTENSITY;

this.rotationVelocity = this.\_rotationVelocity / timeScale;

this.*updateCallback*(deltaTime);

}

*quatFromVectors*(*a*, *b*, *out*, *angleFactor* = 1) {

const axis = vec3.*cross*(vec3.*create*(), a, b);

vec3.*normalize*(axis, axis);

const d = Math.*max*(-1, Math.*min*(1, vec3.*dot*(a, b)));

const angle = Math.*acos*(d) \* angleFactor;

quat.*setAxisAngle*(out, axis, angle);

return { q: out, axis, angle };

}

#*project*(*pos*) {

const r = 2;

const w = this.canvas.clientWidth;

const h = this.canvas.clientHeight;

const s = Math.*max*(w, h) - 1;

*// map to [-1, 1]*

const x = (2 \* pos[0] - w - 1) / s;

const y = (2 \* pos[1] - h - 1) / s;

let z = 0;

const xySq = x \* x + y \* y;

const rSq = r \* r;

if (xySq <= rSq / 2.0) {

z = Math.*sqrt*(rSq - xySq);

} else {

z = rSq / Math.*sqrt*(xySq);

}

return vec3.*fromValues*(-x, y, z);

}

}

class *InfiniteGridMenu* {

TARGET\_FRAME\_DURATION = 1000 / 60; *// 60 fps*

SPHERE\_RADIUS = 2;

#time = 0;

#deltaTime = 0;

#deltaFrames = 0;

#frames = 0;

camera = {

matrix: mat4.*create*(),

near: 0.1,

far: 40,

fov: Math.PI / 4,

aspect: 1,

position: vec3.*fromValues*(0, 0, 3),

up: vec3.*fromValues*(0, 1, 0),

matrices: {

view: mat4.*create*(),

projection: mat4.*create*(),

inversProjection: mat4.*create*(),

},

};

nearestVertexIndex = null;

smoothRotationVelocity = 0;

scaleFactor = 1.0; *// default*

movementActive = false;

constructor(

*canvas*,

*items*,

*onActiveItemChange*,

*onMovementChange*,

*onInit* = null

) {

this.canvas = canvas;

this.items = items || [];

this.onActiveItemChange = onActiveItemChange || (() => {});

this.onMovementChange = onMovementChange || (() => {});

this.*#init*(onInit);

}

*resize*() {

this.viewportSize = vec2.*set*(

this.viewportSize || vec2.*create*(),

this.canvas.clientWidth,

this.canvas.clientHeight

);

const gl = this.gl;

const needsResize = *resizeCanvasToDisplaySize*(gl.canvas);

if (needsResize) {

gl.*viewport*(0, 0, gl.drawingBufferWidth, gl.drawingBufferHeight);

}

this.*#updateProjectionMatrix*(gl);

}

*run*(*time* = 0) {

this.#deltaTime = Math.*min*(32, time - this.#time);

this.#time = time;

this.#deltaFrames = this.#deltaTime / this.TARGET\_FRAME\_DURATION;

this.#frames += this.#deltaFrames;

this.*#animate*(this.#deltaTime);

this.*#render*();

*requestAnimationFrame*((*t*) => this.*run*(t));

}

#*init*(*onInit*) {

this.gl = this.canvas.*getContext*("webgl2", {

antialias: true,

alpha: false,

});

const gl = this.gl;

if (!gl) {

throw **new** *Error*("No WebGL 2 context!");

}

this.viewportSize = *vec2*.*fromValues*(

this.canvas.clientWidth,

this.canvas.clientHeight

);

this.drawBufferSize = *vec2*.*clone*(this.viewportSize);

this.discProgram = *createProgram*(

gl,

[discVertShaderSource, discFragShaderSource],

null,

{

aModelPosition: 0,

aModelNormal: 1,

aModelUvs: 2,

aInstanceMatrix: 3,

}

);

this.discLocations = {

aModelPosition: gl.*getAttribLocation*(this.discProgram, "aModelPosition"),

aModelUvs: gl.*getAttribLocation*(this.discProgram, "aModelUvs"),

aInstanceMatrix: gl.*getAttribLocation*(

this.discProgram,

"aInstanceMatrix"

),

uWorldMatrix: gl.*getUniformLocation*(this.discProgram, "uWorldMatrix"),

uViewMatrix: gl.*getUniformLocation*(this.discProgram, "uViewMatrix"),

uProjectionMatrix: gl.*getUniformLocation*(

this.discProgram,

"uProjectionMatrix"

),

uCameraPosition: gl.*getUniformLocation*(

this.discProgram,

"uCameraPosition"

),

uScaleFactor: gl.*getUniformLocation*(this.discProgram, "uScaleFactor"),

uRotationAxisVelocity: gl.*getUniformLocation*(

this.discProgram,

"uRotationAxisVelocity"

),

uTex: gl.*getUniformLocation*(this.discProgram, "uTex"),

uFrames: gl.*getUniformLocation*(this.discProgram, "uFrames"),

uItemCount: gl.*getUniformLocation*(this.discProgram, "uItemCount"),

uAtlasSize: gl.*getUniformLocation*(this.discProgram, "uAtlasSize"),

};

this.discGeo = **new** *DiscGeometry*(56, 1);

this.discBuffers = this.discGeo.data;

this.discVAO = *makeVertexArray*(

gl,

[

[

*makeBuffer*(gl, this.discBuffers.vertices, gl.STATIC\_DRAW),

this.discLocations.aModelPosition,

3,

],

[

*makeBuffer*(gl, this.discBuffers.uvs, gl.STATIC\_DRAW),

this.discLocations.aModelUvs,

2,

],

],

this.discBuffers.indices

);

this.icoGeo = **new** *IcosahedronGeometry*();

this.icoGeo.*subdivide*(1).*spherize*(this.SPHERE\_RADIUS);

this.instancePositions = this.icoGeo.vertices.*map*((*v*) => v.position);

this.DISC\_INSTANCE\_COUNT = this.icoGeo.vertices.length;

this.*#initDiscInstances*(this.DISC\_INSTANCE\_COUNT);

this.worldMatrix = mat4.*create*();

this.*#initTexture*();

this.control = **new** *ArcballControl*(this.canvas, (*deltaTime*) =>

this.*#onControlUpdate*(deltaTime)

);

this.*#updateCameraMatrix*();

this.*#updateProjectionMatrix*(gl);

this.*resize*();

if (onInit) *onInit*(this);

}

#*initTexture*() {

const gl = this.gl;

this.tex = *createAndSetupTexture*(

gl,

gl.LINEAR,

gl.LINEAR,

gl.CLAMP\_TO\_EDGE,

gl.CLAMP\_TO\_EDGE

);

const itemCount = Math.*max*(1, this.items.length);

this.atlasSize = Math.*ceil*(Math.*sqrt*(itemCount));

const canvas = document.*createElement*("canvas");

const ctx = canvas.*getContext*("2d");

const cellSize = 512;

canvas.width = this.atlasSize \* cellSize;

canvas.height = this.atlasSize \* cellSize;

*Promise*.*all*(

this.items.*map*(

(*item*) =>

**new** *Promise*((*resolve*) => {

const img = **new** *Image*();

img.crossOrigin = "anonymous";

img.*onload* = () => *resolve*(img);

img.src = item.image;

})

)

).*then*((*images*) => {

images.*forEach*((*img*, *i*) => {

const x = (i % this.atlasSize) \* cellSize;

const y = Math.*floor*(i / this.atlasSize) \* cellSize;

ctx.*drawImage*(img, x, y, cellSize, cellSize);

});

gl.*bindTexture*(gl.TEXTURE\_2D, this.tex);

gl.*texImage2D*(

gl.TEXTURE\_2D,

0,

gl.RGBA,

gl.RGBA,

gl.UNSIGNED\_BYTE,

canvas

);

gl.*generateMipmap*(gl.TEXTURE\_2D);

});

}

#*initDiscInstances*(*count*) {

const gl = this.gl;

this.discInstances = {

matricesArray: **new** *Float32Array*(count \* 16),

matrices: [],

buffer: gl.*createBuffer*(),

};

for (let i = 0; i < count; ++i) {

const instanceMatrixArray = **new** *Float32Array*(

this.discInstances.matricesArray.buffer,

i \* 16 \* 4,

16

);

instanceMatrixArray.*set*(mat4.*create*());

this.discInstances.matrices.*push*(instanceMatrixArray);

}

gl.*bindVertexArray*(this.discVAO);

gl.*bindBuffer*(gl.ARRAY\_BUFFER, this.discInstances.buffer);

gl.*bufferData*(

gl.ARRAY\_BUFFER,

this.discInstances.matricesArray.byteLength,

gl.DYNAMIC\_DRAW

);

const mat4AttribSlotCount = 4;

const bytesPerMatrix = 16 \* 4;

for (let j = 0; j < mat4AttribSlotCount; ++j) {

const loc = this.discLocations.aInstanceMatrix + j;

gl.*enableVertexAttribArray*(loc);

gl.*vertexAttribPointer*(

loc,

4,

gl.FLOAT,

false,

bytesPerMatrix,

j \* 4 \* 4

);

gl.*vertexAttribDivisor*(loc, 1);

}

gl.*bindBuffer*(gl.ARRAY\_BUFFER, null);

gl.*bindVertexArray*(null);

}

#*animate*(*deltaTime*) {

const gl = this.gl;

this.control.*update*(deltaTime, this.TARGET\_FRAME\_DURATION);

let positions = this.instancePositions.*map*((*p*) =>

vec3.*transformQuat*(vec3.*create*(), p, this.control.orientation)

);

const scale = 0.25;

const SCALE\_INTENSITY = 0.6;

positions.*forEach*((*p*, *ndx*) => {

const s =

(Math.*abs*(p[2]) / this.SPHERE\_RADIUS) \* SCALE\_INTENSITY +

(1 - SCALE\_INTENSITY);

const finalScale = s \* scale;

const matrix = mat4.*create*();

mat4.*multiply*(

matrix,

matrix,

mat4.*fromTranslation*(mat4.*create*(), vec3.*negate*(vec3.*create*(), p))

);

mat4.*multiply*(

matrix,

matrix,

mat4.*targetTo*(mat4.*create*(), [0, 0, 0], p, [0, 1, 0])

);

mat4.*multiply*(

matrix,

matrix,

mat4.*fromScaling*(mat4.*create*(), [finalScale, finalScale, finalScale])

);

mat4.*multiply*(

matrix,

matrix,

mat4.*fromTranslation*(mat4.*create*(), [0, 0, -this.SPHERE\_RADIUS])

);

mat4.*copy*(this.discInstances.matrices[ndx], matrix);

});

gl.*bindBuffer*(gl.ARRAY\_BUFFER, this.discInstances.buffer);

gl.*bufferSubData*(gl.ARRAY\_BUFFER, 0, this.discInstances.matricesArray);

gl.*bindBuffer*(gl.ARRAY\_BUFFER, null);

this.smoothRotationVelocity = this.control.rotationVelocity;

}

#*render*() {

const gl = this.gl;

gl.*useProgram*(this.discProgram);

gl.*enable*(gl.CULL\_FACE);

gl.*enable*(gl.DEPTH\_TEST);

gl.*clearColor*(0, 0, 0, 0);

gl.*clear*(gl.COLOR\_BUFFER\_BIT | gl.DEPTH\_BUFFER\_BIT);

gl.*uniformMatrix4fv*(

this.discLocations.uWorldMatrix,

false,

this.worldMatrix

);

gl.*uniformMatrix4fv*(

this.discLocations.uViewMatrix,

false,

this.camera.matrices.view

);

gl.*uniformMatrix4fv*(

this.discLocations.uProjectionMatrix,

false,

this.camera.matrices.projection

);

gl.*uniform3f*(

this.discLocations.uCameraPosition,

this.camera.position[0],

this.camera.position[1],

this.camera.position[2]

);

gl.*uniform4f*(

this.discLocations.uRotationAxisVelocity,

this.control.rotationAxis[0],

this.control.rotationAxis[1],

this.control.rotationAxis[2],

this.smoothRotationVelocity \* 1.1

);

*// NEW UNIFORMS*

gl.*uniform1i*(this.discLocations.uItemCount, this.items.length);

gl.*uniform1i*(this.discLocations.uAtlasSize, this.atlasSize);

gl.*uniform1f*(this.discLocations.uFrames, this.#frames);

gl.*uniform1f*(this.discLocations.uScaleFactor, this.scaleFactor);

gl.*uniform1i*(this.discLocations.uTex, 0);

gl.*activeTexture*(gl.TEXTURE0);

gl.*bindTexture*(gl.TEXTURE\_2D, this.tex);

gl.*bindVertexArray*(this.discVAO);

gl.*drawElementsInstanced*(

gl.TRIANGLES,

this.discBuffers.indices.length,

gl.UNSIGNED\_SHORT,

0,

this.DISC\_INSTANCE\_COUNT

);

}

#*updateCameraMatrix*() {

mat4.*targetTo*(

this.camera.matrix,

this.camera.position,

[0, 0, 0],

this.camera.up

);

mat4.*invert*(this.camera.matrices.view, this.camera.matrix);

}

#*updateProjectionMatrix*(*gl*) {

this.camera.aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;

const height = this.SPHERE\_RADIUS \* 0.35;

const distance = this.camera.position[2];

if (this.camera.aspect > 1) {

this.camera.fov = 2 \* Math.*atan*(height / distance);

} else {

this.camera.fov = 2 \* Math.*atan*(height / this.camera.aspect / distance);

}

mat4.*perspective*(

this.camera.matrices.projection,

this.camera.fov,

this.camera.aspect,

this.camera.near,

this.camera.far

);

mat4.*invert*(

this.camera.matrices.inversProjection,

this.camera.matrices.projection

);

}

#*onControlUpdate*(*deltaTime*) {

const timeScale = *deltaTime* / this.TARGET\_FRAME\_DURATION + 0.0001;

let damping = 5 / timeScale;

let cameraTargetZ = 3;

const isMoving =

this.control.isPointerDown ||

Math.*abs*(this.smoothRotationVelocity) > 0.01;

if (isMoving !== this.movementActive) {

this.movementActive = isMoving;

this.*onMovementChange*(isMoving);

}

if (!this.control.isPointerDown) {

const nearestVertexIndex = this.*#findNearestVertexIndex*();

const itemIndex = nearestVertexIndex % Math.*max*(1, this.items.length);

this.*onActiveItemChange*(itemIndex);

const snapDirection = *vec3*.*normalize*(

*vec3*.*create*(),

this.*#getVertexWorldPosition*(nearestVertexIndex)

);

this.control.snapTargetDirection = snapDirection;

} else {

cameraTargetZ += this.control.rotationVelocity \* 80 + 2.5;

damping = 7 / timeScale;

}

this.camera.position[2] +=

(cameraTargetZ - this.camera.position[2]) / damping;

this.*#updateCameraMatrix*();

}

#*findNearestVertexIndex*() {

const n = this.control.snapDirection;

const inversOrientation = *quat*.*conjugate*(

*quat*.*create*(),

this.control.orientation

);

const nt = *vec3*.*transformQuat*(*vec3*.*create*(), n, inversOrientation);

let maxD = -1;

let nearestVertexIndex;

for (let i = 0; i < this.instancePositions.length; ++i) {

const d = *vec3*.*dot*(nt, this.instancePositions[i]);

if (d > maxD) {

maxD = d;

nearestVertexIndex = i;

}

}

return nearestVertexIndex;

}

#*getVertexWorldPosition*(*index*) {

const nearestVertexPos = this.instancePositions[*index*];

return *vec3*.*transformQuat*(

*vec3*.*create*(),

nearestVertexPos,

this.control.orientation

);

}

}

const defaultItems = [

{

image: "https://picsum.photos/900/900?grayscale",

link: "https://google.com/",

title: "",

description: "",

},

];

export default function *InfiniteMenu*({ *items* = [] }) {

const canvasRef = *useRef*(null);

const [activeItem, *setActiveItem*] = *useState*(null);

const [isMoving, *setIsMoving*] = *useState*(false);

*useEffect*(() => {

const canvas = canvasRef.current;

let sketch;

const *handleActiveItem* = (*index*) => {

const itemIndex = *index* % *items*.length;

*setActiveItem*(*items*[itemIndex]);

};

if (canvas) {

sketch = **new** *InfiniteGridMenu*(

canvas,

*items*.length ? *items* : defaultItems,

*handleActiveItem*,

*setIsMoving*,

(*sk*) => *sk*.*run*()

);

}

const *handleResize* = () => {

if (sketch) {

sketch.*resize*();

}

};

window.*addEventListener*("resize", *handleResize*);

*handleResize*();

return () => {

window.*removeEventListener*("resize", *handleResize*);

};

}, [*items*]);

const *handleButtonClick* = () => {

if (!activeItem?.link) return;

if (activeItem.link.*startsWith*("http")) {

window.*open*(activeItem.link, "\_blank");

} else {

console.*log*("Internal route:", activeItem.link);

}

};

return (

<div className="relative w-full h-full">

<canvas

id="infinite-grid-menu-canvas"

ref={canvasRef}

className="cursor-grab w-full h-full overflow-hidden object-cover relative outline-none active:cursor-grabbing"

/>

{activeItem && (

<>

{*/\* Title \*/*}

<h2

className={`

select-none

absolute

font-black

[font-size:4rem]

left-[1.6em]

top-1/2

sm:block

hidden

transform

translate-x-[20%]

-translate-y-1/2

transition-all

ease-[cubic-bezier(0.25,0.1,0.25,1.0)]

${

isMoving

? "opacity-0 pointer-events-none duration-[100ms]"

: "opacity-100 pointer-events-auto duration-[500ms]"

}

`}

>

{activeItem.title}

</h2>

{*/\* Description \*/*}

<p

className={`

select-none

absolute

max-w-[10ch]

text-[1.5rem]

top-1/2

sm:block

hidden

right-[1%]

transition-all

ease-[cubic-bezier(0.25,0.1,0.25,1.0)]

${

isMoving

? "opacity-0 pointer-events-none duration-[100ms] translate-x-[-60%] -translate-y-1/2"

: "opacity-100 pointer-events-auto duration-[500ms] translate-x-[-90%] -translate-y-1/2"

}

`}

>

{activeItem.description}

</p>

{*/\* Action Button \*/*}

<div

onClick={*handleButtonClick*}

className={`

absolute

left-1/2

z-10

w-[60px]

h-[60px]

grid

place-items-center

bg-[#00ffff]

border-black

rounded-full

cursor-pointer

transition-all

ease-[cubic-bezier(0.25,0.1,0.25,1.0)]

${

isMoving

? "bottom-[-80px] opacity-0 pointer-events-none duration-[100ms] scale-0 -translate-x-1/2"

: "bottom-[3.8em] opacity-100 pointer-events-auto duration-[500ms] scale-100 -translate-x-1/2"

}

`}

>

<p className="select-none relative text-[#060606] top-[2px] text-[26px]">

&#x2197;

</p>

</div>

</>

)}

</div>

);

}

z-10

w-[60px]

h-[60px]

grid

place-items-center

bg-[#00ffff]

border-black

rounded-full

cursor-pointer

transition-all

ease-[cubic-bezier(0.25,0.1,0.25,1.0)]

${

isMoving

? "bottom-[-80px] opacity-0 pointer-events-none duration-[100ms] scale-0 -translate-x-1/2"

: "bottom-[3.8em] opacity-100 pointer-events-auto duration-[500ms] scale-100 -translate-x-1/2"

}

`}

>

<p className="select-none relative text-[#060606] top-[2px] text-[26px]">

&#x2197;

</p>

</div>

</>

)}

</div>

);

}