```
/******/ (function(modules) { // webpackBootstrap
/******/        // The module cache
/******/        var installedModules = {};

/******/        // The require function
/******/        function __webpack_require__(moduleId) {

/******/                // Check if module is in cache
/******/                if(installedModules[moduleId])
/******/                        return installedModules[moduleId].exports;

/******/                // Create a new module (and put it into the cache)
/******/                var module = installedModules[moduleId] = {
/******/                        exports: {},
/******/                        id: moduleId,
/******/                        loaded: false
/******/                };

/******/                // Execute the module function
/******/                modules[moduleId].call(module.exports, module, module.exports,
__webpack_require__);

/******/                // Flag the module as loaded
/******/                module.loaded = true;

/******/                // Return the exports of the module
/******/                return module.exports;
/******/        }


/******/        // expose the modules object (__webpack_modules__)
/******/        __webpack_require__.m = modules;

/******/        // expose the module cache
/******/        __webpack_require__.c = installedModules;

/******/        // __webpack_public_path__
/******/        __webpack_require__.p = "";

/******/        // Load entry module and return exports
/******/        return __webpack_require__(0);
/******/ })
/************************************************************************/
/******/ ([
/* 0 */
/***/ function(module, exports, __webpack_require__) {

        /* global THREE, AFRAME, Element  */
        var cylinderTexture = __webpack_require__(1);
        var parabolicCurve = __webpack_require__(2);
        var RayCurve = __webpack_require__(3);

        if (typeof AFRAME === 'undefined') {
          throw new Error('Component attempted to register before AFRAME was available.');
        }

        if (!Element.prototype.matches) {
          Element.prototype.matches =
            Element.prototype.matchesSelector ||
            Element.prototype.mozMatchesSelector ||
            Element.prototype.msMatchesSelector ||
            Element.prototype.oMatchesSelector ||
            Element.prototype.webkitMatchesSelector ||
            function (s) {
```

```
      var matches = (this.document || this.ownerDocument).querySelectorAll(s);
      var i = matches.length;
      while (--i >= 0 && matches.item(i) !== this) { /* no-op */ }
      return i > -1;
    };
}

AFRAME.registerComponent('teleport-controls', {
  schema: {
    type: {default: 'parabolic', oneOf: ['parabolic', 'line']},
    button: {default: 'trackpad', oneOf: ['trackpad', 'trigger', 'grip', 'menu']},
    startEvents: {type: 'array'},
    endEvents: {type: 'array'},
    collisionEntities: {default: ''},
    hitEntity: {type: 'selector'},
    cameraRig: {type: 'selector'},
    teleportOrigin: {type: 'selector'},
    hitCylinderColor: {type: 'color', default: '#99ff99'},
    hitCylinderRadius: {default: 0.25, min: 0},
    hitCylinderHeight: {default: 0.3, min: 0},
    maxLength: {default: 10, min: 0, if: {type: ['line']}},
    curveNumberPoints: {default: 30, min: 2, if: {type: ['parabolic']}},
    curveLineWidth: {default: 0.025},
    curveHitColor: {type: 'color', default: '#99ff99'},
    curveMissColor: {type: 'color', default: '#ff0000'},
    curveShootingSpeed: {default: 5, min: 0, if: {type: ['parabolic']}},
    defaultPlaneSize: { default: 100 },
    landingNormal: {type: 'vec3', default: '0 1 0'},
    landingMaxAngle: {default: '45', min: 0, max: 360}
  },

  init: function () {
    var data = this.data;
    var el = this.el;
    var teleportEntity;
    var i;

    this.active = false;
    this.obj = el.object3D;
    this.hitPoint = new THREE.Vector3();
    this.rigWorldPosition = new THREE.Vector3();
    this.newRigWorldPosition = new THREE.Vector3();
    this.teleportEventDetail = {
      oldPosition: this.rigWorldPosition,
      newPosition: this.newRigWorldPosition,
      hitPoint: this.hitPoint
    };

    this.hit = false;
    this.prevHitHeight = 0;
    this.referenceNormal = new THREE.Vector3();
    this.curveMissColor = new THREE.Color();
    this.curveHitColor = new THREE.Color();
    this.raycaster = new THREE.Raycaster();

    this.defaultPlane = createDefaultPlane(this.data.defaultPlaneSize);

    teleportEntity = this.teleportEntity = document.createElement('a-entity');
    teleportEntity.classList.add('teleportRay');
    teleportEntity.setAttribute('visible', false);
    el.sceneEl.appendChild(this.teleportEntity);

    this.onButtonDown = this.onButtonDown.bind(this);
    this.onButtonUp = this.onButtonUp.bind(this);
    if (this.data.startEvents.length && this.data.endEvents.length) {
```

```
              for (i = 0; i < this.data.startEvents.length; i++) {
                el.addEventListener(this.data.startEvents[i], this.onButtonDown);
              }
              for (i = 0; i < this.data.endEvents.length; i++) {
                el.addEventListener(this.data.endEvents[i], this.onButtonUp);
              }
            } else {
              el.addEventListener(data.button + 'down', this.onButtonDown);
              el.addEventListener(data.button + 'up', this.onButtonUp);
            }

            this.queryCollisionEntities();
          },

          update: function (oldData) {
            var data = this.data;
            var diff = AFRAME.utils.diff(data, oldData);

            // Update normal.
            this.referenceNormal.copy(data.landingNormal);

            // Update colors.
            this.curveMissColor.set(data.curveMissColor);
            this.curveHitColor.set(data.curveHitColor);

            // Create or update line mesh.
            if (!this.line ||
                'curveLineWidth' in diff || 'curveNumberPoints' in diff || 'type' in diff) {
              this.line = createLine(data);
              this.teleportEntity.setObject3D('mesh', this.line.mesh);
            }

            // Create or update hit entity.
            if (data.hitEntity) {
              this.hitEntity = data.hitEntity;
            } else if (!this.hitEntity || 'hitCylinderColor' in diff || 'hitCylinderHeight' in diff
||
                       'hitCylinderRadius' in diff) {
              // Remove previous entity, create new entity (could be more performant).
              if (this.hitEntity) { this.hitEntity.parentNode.removeChild(this.hitEntity); }
              this.hitEntity = createHitEntity(data);
              this.el.sceneEl.appendChild(this.hitEntity);
            }
            this.hitEntity.setAttribute('visible', false);

            if ('collisionEntities' in diff) { this.queryCollisionEntities(); }
          },

          remove: function () {
            var el = this.el;
            var hitEntity = this.hitEntity;
            var teleportEntity = this.teleportEntity;

            if (hitEntity) { hitEntity.parentNode.removeChild(hitEntity); }
            if (teleportEntity) { teleportEntity.parentNode.removeChild(teleportEntity); }

            el.sceneEl.removeEventListener('child-attached', this.childAttachHandler);
            el.sceneEl.removeEventListener('child-detached', this.childDetachHandler);
          },

          tick: (function () {
            var p0 = new THREE.Vector3();
            var v0 = new THREE.Vector3();
            var g = -9.8;
            var a = new THREE.Vector3(0, g, 0);
            var next = new THREE.Vector3();
            var last = new THREE.Vector3();
            var quaternion = new THREE.Quaternion();
            var translation = new THREE.Vector3();
            var scale = new THREE.Vector3();
            var shootAngle = new THREE.Vector3();
            var lastNext = new THREE.Vector3();
            var auxDirection = new THREE.Vector3();

            return function (time, delta) {
              if (!this.active) { return; }

              var matrixWorld = this.obj.matrixWorld;
              matrixWorld.decompose(translation, quaternion, scale);

              var direction = shootAngle.set(0, 0, -1)
                .applyQuaternion(quaternion).normalize();
              this.line.setDirection(auxDirection.copy(direction));
              this.obj.getWorldPosition(p0);

              last.copy(p0);

              // Set default status as non-hit
              this.teleportEntity.setAttribute('visible', true);
              this.line.material.color.set(this.curveMissColor);
              this.hitEntity.setAttribute('visible', false);
              this.hit = false;

              if (this.data.type === 'parabolic') {
                v0.copy(direction).multiplyScalar(this.data.curveShootingSpeed);

                for (var i = 0; i < this.line.numPoints; i++) {
                  var t = i / (this.line.numPoints - 1);
                  parabolicCurve(p0, v0, a, t, next);
                  // Update the raycaster with the length of the current segment last->next
                  var dirLastNext = lastNext.copy(next).sub(last).normalize();
                  this.raycaster.far = dirLastNext.length();
                  this.raycaster.set(last, dirLastNext);

                  if (this.checkMeshCollisions(i, next)) { break; }
                  last.copy(next);
                }
              } else if (this.data.type === 'line') {

next.copy(last).add(auxDirection.copy(direction).multiplyScalar(this.data.maxLength));
                this.raycaster.far = this.data.maxLength;
                this.raycaster.set(p0, direction);
                this.line.setPoint(0, p0);

                this.checkMeshCollisions(1, next);
              }
            };
          })(),

          /**
           * Run `querySelectorAll` for `collisionEntities` and maintain it with `child-attached`
           * and `child-detached` events.
           */
          queryCollisionEntities: function () {
            var collisionEntities;
            var data = this.data;
            var el = this.el;

            if (!data.collisionEntities) {
              this.collisionEntities = [];
              return;
            }

            collisionEntities = [].slice.call(el.sceneEl.querySelectorAll(data.collisionEntities));
            this.collisionEntities = collisionEntities;

            // Update entity list on attach.
            this.childAttachHandler = function childAttachHandler (evt) {
              if (!evt.detail.el.matches(data.collisionEntities)) { return; }
              collisionEntities.push(evt.detail.el);
            };
            el.sceneEl.addEventListener('child-attached', this.childAttachHandler);

            // Update entity list on detach.
            this.childDetachHandler = function childDetachHandler (evt) {
              var index;
              if (!evt.detail.el.matches(data.collisionEntities)) { return; }
              index = collisionEntities.indexOf(evt.detail.el);
              if (index === -1) { return; }
              collisionEntities.splice(index, 1);
            };
            el.sceneEl.addEventListener('child-detached', this.childDetachHandler);
          },

          onButtonDown: function () {
            this.active = true;
          },

          /**
           * Jump!
           */
          onButtonUp: (function () {
            const teleportOriginWorldPosition = new THREE.Vector3();
            const newRigLocalPosition = new THREE.Vector3();
            const newHandPosition = new THREE.Vector3();
            const handPosition = new THREE.Vector3();

            return function (evt) {
              if (!this.active) { return; }

              // Hide the hit point and the curve
              this.active = false;
              this.hitEntity.setAttribute('visible', false);
              this.teleportEntity.setAttribute('visible', false);

              if (!this.hit) {
                // Button released but not hit point
                return;
              }

              const rig = this.data.cameraRig || this.el.sceneEl.camera.el;
              rig.object3D.getWorldPosition(this.rigWorldPosition);
              this.newRigWorldPosition.copy(this.hitPoint);

              // If a teleportOrigin exists, offset the rig such that the teleportOrigin is above the
hitPoint
              const teleportOrigin = this.data.teleportOrigin;
              if (teleportOrigin) {
                teleportOrigin.object3D.getWorldPosition(teleportOriginWorldPosition);
                this.newRigWorldPosition.sub(teleportOriginWorldPosition).add(this.rigWorldPosition);
              }

              // Always keep the rig at the same offset off the ground after teleporting
              this.newRigWorldPosition.y = this.rigWorldPosition.y + this.hitPoint.y -
this.prevHitHeight;
              this.prevHitHeight = this.hitPoint.y;

              // Finally update the rigs position
              newRigLocalPosition.copy(this.newRigWorldPosition);
              if (rig.object3D.parent) {
                rig.object3D.parent.worldToLocal(newRigLocalPosition);
              }
              rig.setAttribute('position', newRigLocalPosition);

              // If a rig was not explicitly declared, look for hands and mvoe them proportionally as
well
              if (!this.data.cameraRig) {
                var hands = document.querySelectorAll('a-entity[tracked-controls]');
                for (var i = 0; i < hands.length; i++) {
                  hands[i].object3D.getWorldPosition(handPosition);

                  // diff = rigWorldPosition - handPosition
                  // newPos = newRigWorldPosition - diff

newHandPosition.copy(this.newRigWorldPosition).sub(this.rigWorldPosition).add(handPosition);
                  hands[i].setAttribute('position', newHandPosition);
                }
              }

              this.el.emit('teleported', this.teleportEventDetail);
            };
          })(),

          /**
           * Check for raycaster intersection.
           *
           * @param {number} Line fragment point index.
           * @param {number} Next line fragment point index.
           * @returns {boolean} true if there's an intersection.
           */
          checkMeshCollisions: function (i, next) {
            // @todo We should add a property to define if the collisionEntity is dynamic or static
            // If static we should do the map just once, otherwise we're recreating the array in
every
            // loop when aiming.
            var meshes = this.collisionEntities.map(function (entity) {
              return entity.getObject3D('mesh');
            }).filter(function (n) { return n; });
            meshes = meshes.length ? meshes : [this.defaultPlane];

            var intersects = this.raycaster.intersectObjects(meshes, true);
            if (intersects.length > 0 && !this.hit &&
                this.isValidNormalsAngle(intersects[0].face.normal)) {
              var point = intersects[0].point;

              this.line.material.color.set(this.curveHitColor);
              this.hitEntity.setAttribute('position', point);
              this.hitEntity.setAttribute('visible', true);

              this.hit = true;
              this.hitPoint.copy(intersects[0].point);

              // If hit, just fill the rest of the points with the hit point and break the loop
              for (var j = i; j < this.line.numPoints; j++) {
                this.line.setPoint(j, this.hitPoint);
              }
              return true;
            } else {
              this.line.setPoint(i, next);
```

```
	      return false;
	    }
	  },

	  isValidNormalsAngle: function (collisionNormal) {
	    var angleNormals = this.referenceNormal.angleTo(collisionNormal);
	    return (THREE.Math.RAD2DEG * angleNormals <= this.data.landingMaxAngle);
	  },
	});


	function createLine (data) {
	  var numPoints = data.type === 'line' ? 2 : data.curveNumberPoints;
	  return new RayCurve(numPoints, data.curveLineWidth);
	}

	/**
	 * Create mesh to represent the area of intersection.
	 * Default to a combination of torus and cylinder.
	 */
	function createHitEntity (data) {
	  var cylinder;
	  var hitEntity;
	  var torus;

	  // Parent.
	  hitEntity = document.createElement('a-entity');
	  hitEntity.className = 'hitEntity';

	  // Torus.
	  torus = document.createElement('a-entity');
	  torus.setAttribute('geometry', {
	    primitive: 'torus',
	    radius: data.hitCylinderRadius,
	    radiusTubular: 0.01
	  });
	  torus.setAttribute('rotation', {x: 90, y: 0, z: 0});
	  torus.setAttribute('material', {
	    shader: 'flat',
	    color: data.hitCylinderColor,
	    side: 'double',
	    depthTest: false
	  });
	  hitEntity.appendChild(torus);

	  // Cylinder.
	  cylinder = document.createElement('a-entity');
	  cylinder.setAttribute('position', {x: 0, y: data.hitCylinderHeight / 2, z: 0});
	  cylinder.setAttribute('geometry', {
	    primitive: 'cylinder',
	    segmentsHeight: 1,
	    radius: data.hitCylinderRadius,
	    height: data.hitCylinderHeight,
	    openEnded: true
	  });
	  cylinder.setAttribute('material', {
	    shader: 'flat',
	    color: data.hitCylinderColor,
	    side: 'double',
	    src: cylinderTexture,
	    transparent: true,
	    depthTest: false
	  });
	  hitEntity.appendChild(cylinder);
```

```
     return hitEntity;
    }

    function createDefaultPlane (size) {
     var geometry;
     var material;

     geometry = new THREE.PlaneBufferGeometry(100, 100);
     geometry.rotateX(-Math.PI / 2);
     material = new THREE.MeshBasicMaterial({color: 0xffff00});
     return new THREE.Mesh(geometry, material);
    }


/***/ },
/* 1 */
/***/ function(module, exports) {

    module.exports =
'url(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAAQCAYAAADXnxW3AAAACXBIWXMAAAsTAAALEwEAmpwYAA
AKT2lDQ1BQaG90b3Nob3AgSUNDIHByb2ZpbGUAAHjanVNnVFPpFj333vRCS4iAlEtvUhUIIFJCi4AUkSYqIQkQSoghodkVUcERRUU
EG8igiAOOjoCMFVEsDIoK2AfkIaKOg6OIisr74Xuja9a89+bN/rXXPues852zzwfACAyWSDNRNYAMqUIeEeCDx8TG4eQuQIEKJHAA
EAizZCFz/SMBAPh+PDwrIsAHvgABeNMLCADATZvAMByH/w/qQplcAYCEAcB0kThLCIAUAEB6jkKmAEBGAYCdmCZTAKAEAGDLY2LjA
FAtAGAnf+bTAICd+Jl7AQBblCEVAaCRACATZYhEAGg7AKzPVopFAFgwABRmS8Q5ANgtADBJV2ZIALC3AMDOEAuyAAgMADBRiIUpAA
R7AGDIIyN4AISZABRG8lc88SuuEOcqAAB4mbI8uSQ5RYFbCC1xB1dXLh4ozkkXKxQ2YQJhmkAuwnmZGTKBNA/g88wAAKCRFRHgg/P
9eM4Ors7ONo62Dl8t6r8G/yJiYuP+5c+rcEAAAOF0ftH+LC+zGoA7BoBt/qIl7gRoXgugdfeLZrIPQLUAoOnaV/Nw+H48PEWhkLnZ
2eXk5NhKxEJbYcpXff5nwl/AV/1s+X48/Pf14L7iJIEyXYFHBPjgwsz0TKUcz5IJhGLc5o9H/LcL//wd0yLESWK5WCoU41EScY5Em
ozzMqUiiUKSKcUl0v9k4t8s+wM+3zUAsGo+AXuRLahdYwP2SycQWHTA4vcAAPK7b8HUKAgDgGiD4c93/+8//UegJQCAZkmScQAAXk
QkLlTKsz/HCAAARKCBKrBBG/TBGCzABhzBBdzBC/xgNoRCJMTCQhBCCmSAHHJgKayCQiiGzbAdKmAv1EAdNMBRaIaTcA4uwlW4Dj1
wD/phCJ7BKLyBCQRByAgTYSHaiAFiilgjjggXmYX4IcFIBBKLJCDJiBRRIkuRNUgxUopUIFVIHfI9cgI5h1xGupE7yAAygvyGvEcx
lIGyUT3UDLVDuag3GoRGogvQZHQxmo8WoJvQcrQaPYw2oefQq2gP2o8+Q8cwwOgYBzPEbDAuxsNCsTgsCZNjy7EirAyrxhqwVqwDu
4n1Y8+xdwQSgUXACTYEd0IgYR5BSFhMWE7YSKggHCQ0EdoJNwkDhFHCJyKTqEu0JroR+cQYYjIxh1hILCPWEo8TLxB7iEPENyQSiU
MyJ7mQAkmxpFTSEtJG0m5SI+ksqZs0SBojk8naZGuyBzmULCAryIXkneTD5DPkG+Qh8lsKnWJAcaT4U+IoUspqShnlEOU05QZlmDJ
BVaOaUt2ooVQRNY9aQq2htlKvUYeoEzR1mjnNgxZJS6WtopXTGmgXaPdpr+h0uhHdlR5Ol9BX0svpR+iX6AP0dwwNhgNjAPdpr+h0uhHdlR5Ol9BX0svpR+iX6AP0dwwNhhWDx4hnKBmb
GAcYZxl3GK+YTKYZ04sZx1QwNzHrmOeZD5lvVVgqtip8FZHKCpVKlSaVGyovVKmqpqreqgtV81XLVI+pXlN9rkZVM1PjqQnUlqtVq
p1Q61MbU2epO6iHqmeob1Q/pH5Z/YkGWcNMw09DpFGgsV/jvMYgC2MZs3gsIWsNq4Z1gTXEJrHN2Xx2KruY/R27iz2qqaE5QzNKM1
ezUvOUZj8H45hx+Jx0TgnnKKeX836K3hTvKeIpG6Y0TLkxZVxrqpaXllirSKtRq0frvTau7aedpr1Fu1n7gQ5Bx0onXCdHZ4/OBZ3
nU9lT3acKpxZNPTr1ri6qa6UbobtEd79up+6Ynr5egJ5Mb6feeb3n+hx9L/1U/W36p/VHDFgGswwkBtsMzhg8xTVxbzwdL8fb8VFD
XcNAQ6VhlWGX4YSRudE8o9VGjUYPjGnGXOMk423GbcajJgYmISZLTepN7ppSTbmmKaY7TDtMx83MzaLN1pk1mz0x1zLnm+eb15vft
2BaeFostqi2uGVJsuRaplnutrxuhVo5WaVYVVpds0atna0l1rutu6cRp7lOk06rntZnw7Dxtsm2qbcZsOXYBtuutm22fWFnYhdnt8
Wuw+6TvZN9un2N/T0HDYfZDqsdWh1+c7RyFDpWOt6azpzuP33F9JbpL2dYzxYzDP2DPjthPLKcRpnVOb00dnF2e5c4PziIuJS4LLLpc
+Lpsbxt3IveRKdPVxXeF60vWdm7Obwu2o26/uNu5p7ofcn8w0nymeWTNz0MPIQ+BR5dE/C5+VMGvfrH5PQ0+BZ7XnIy9jL5FXrdew
t6V3qvdh7xc+9j5yn+M+4zw33jLeWV/MN8C3yLfLT8Nvnl+F30N/I/9k/3r/0QCngCUBZwOJgUGBWwL7+Hp8Ib+OPzrbZfay2e1Bj
KC5QRVBj4KtguXBrSFoyOyQrSH355jOkc5pDoVQfujW0Adh5mGLw34MJ4WHhVeGP45wiFga0TGXNXfR3ENz30T6RJZE3ptnMU85ry
1KNSo+qi5qPNo3ujS6P8YuZlnM1VidWElsSxw5LiquNm5svt/87fOH4p3iC+N7F5gvyF1weaHOwvSFpxapLhIsOpZATIhOOJTwQRA
qqBaMJfITdyWOCnnCHcJnIi/RNtGI2ENcKh5O8kgqTXqS7JG8NXkkxTOlLOW5hCepkLxMDUzdmzqeFpp2IG0yPTq9MYOSkZBxQqoh
TZO2Z+pn5mZ2y6xlhbL+xW6Lty8elQfJa7OQrAVZLQq2QqboVFoo1yoHsmdlV2a/zYnKOZarnivN7cyzytuQN5zvn//tEsIS4ZK2p
YZLVy0dWOa9rGo5sjxxedsK4K2Km3VT6vtV5eufr0mek1rgV7ByoLBtQFr6wtVCuWFfevc1+1dT1gvWd+1YfqGnR
s+FYmKrhTbF5cVf9go3HjlG4dvyr+Z3JS0qavEuWTPZtJm6ebeLZ5bDpaql+aXDm4N2dq0Dd9WtO319kXbL5fNKNu7g7ZDuaO/PLi
8ZafJzs07P1SkVPRU+lQ27tLdtWHX+G7R7ht7vPY07NXbW7z3/T7JvttVAVVN1WbVZftJ+7P3P66Jqun4lvttXa1ObXHtxwOSA/0H
Iw6217nU1R3SPVRSj9Yr60cOxx++/p3vdy0NNg1VjZzG4iNwRHnk6fcJ3/ceDTradox7rOEH0x92HWcdL2pCmvKaRptTmvtbYlu6T
8w+0dbq3nr8R9sfD5w0PFl5SvNUyWna6YLTk2fyz4ydlZ19fi753GDborZ752PO32oPb++6EHTh0kX/i+c7vDvOXPK4dPKy2+UTV7
hXmq86X23qdOo8/pPTT8e7nLuarrlca7nuer21e2b36RueN87d9L158Rb/1tWeOT3dvfN6b/fF9/XfFt1+cif9zsu72Xcn7q28T7x
f9EDtQdlD3YfVP1v+3Njv3H9qwHeg89HcR/cGhYPP/pH1jw9DBY+Zj8uuGDYbrnjg+OTniP3L96fynQ89kzyaeF/6i/suuFxYvfvjV
69fO0ZjRoZfyl5O/bXyl/erA6xmv28bCxh6+yXgzMV70VvvtwXfcdx3vo98PT+R8IH8o/2j5sfVT0Kf7kxmTk/8EA5jz/GMzLdsAA
AAgY0hSTQAAeiUAAICDAAD5/wAAgOkAAHUwAADqYAAAOpgAABdvkl/FRgAAADJJREFUeNpEx7ENgDAAAzArK0JA6f8X9oewlcWStU
1wBGdwB08wgjeYm79jc2nbYH0DAC/+CORJxO5fAAAAAElFTkSuQmCC)';


/***/ },
/* 2 */
/***/ function(module, exports) {

    /* global THREE */
    // Parabolic motion equation, y = p0 + v0*t + 1/2at^2
```

```
        function parabolicCurveScalar (p0, v0, a, t) {
          return p0 + v0 * t + 0.5 * a * t * t;
        }

        // Parabolic motion equation applied to 3 dimensions
        function parabolicCurve (p0, v0, a, t, out) {
          out.x = parabolicCurveScalar(p0.x, v0.x, a.x, t);
          out.y = parabolicCurveScalar(p0.y, v0.y, a.y, t);
          out.z = parabolicCurveScalar(p0.z, v0.z, a.z, t);
          return out;
        }

        module.exports = parabolicCurve;


/***/ },
/* 3 */
/***/ function(module, exports) {

        /* global THREE */
        var RayCurve = function (numPoints, width) {
          this.geometry = new THREE.BufferGeometry();
          this.vertices = new Float32Array(numPoints * 3 * 2);
          this.uvs = new Float32Array(numPoints * 2 * 2);
          this.width = width;

          this.geometry.addAttribute('position', new THREE.BufferAttribute(this.vertices,
3).setDynamic(true));

          this.material = new THREE.MeshBasicMaterial({
            side: THREE.DoubleSide,
            color: 0xff0000
          });

          this.mesh = new THREE.Mesh(this.geometry, this.material);
          this.mesh.drawMode = THREE.TriangleStripDrawMode;

          this.mesh.frustumCulled = false;
          this.mesh.vertices = this.vertices;

          this.direction = new THREE.Vector3();
          this.numPoints = numPoints;
        };

        RayCurve.prototype = {
          setDirection: function (direction) {
            var UP = new THREE.Vector3(0, 1, 0);
            this.direction
              .copy(direction)
              .cross(UP)
              .normalize()
              .multiplyScalar(this.width / 2);
          },

          setWidth: function (width) {
            this.width = width;
          },

          setPoint: (function () {
            var posA = new THREE.Vector3();
            var posB = new THREE.Vector3();

            return function (i, point) {
              posA.copy(point).add(this.direction);
              posB.copy(point).sub(this.direction);
              var idx = 2 * 3 * i;
              this.vertices[idx++] = posA.x;
              this.vertices[idx++] = posA.y;
              this.vertices[idx++] = posA.z;

              this.vertices[idx++] = posB.x;
              this.vertices[idx++] = posB.y;
              this.vertices[idx++] = posB.z;

              this.geometry.attributes.position.needsUpdate = true;
            };
          })()
        };

        module.exports = RayCurve;


/***/ }
/******/ ]);
```