unconed/MathBox.js

You can clone with HTTPS or Subversion.

Download ZIP

MathBox is a (work in progress) library for making presentation-quality math diagrams in WebGL.

1. JavaScript 99.8%
2. Other 0.2%

JavaScript Other

branch: master

Switch branches/tags

- Branches
- Tags

master

Nothing to show

Nothing to show

MathBox.js/

Cap camera theta at the poles with an epsilon because of atan2 change… …

```
…s in chrome
```

latest commit dd802725b5

unconed authored Sep 9, 2014

Permalink

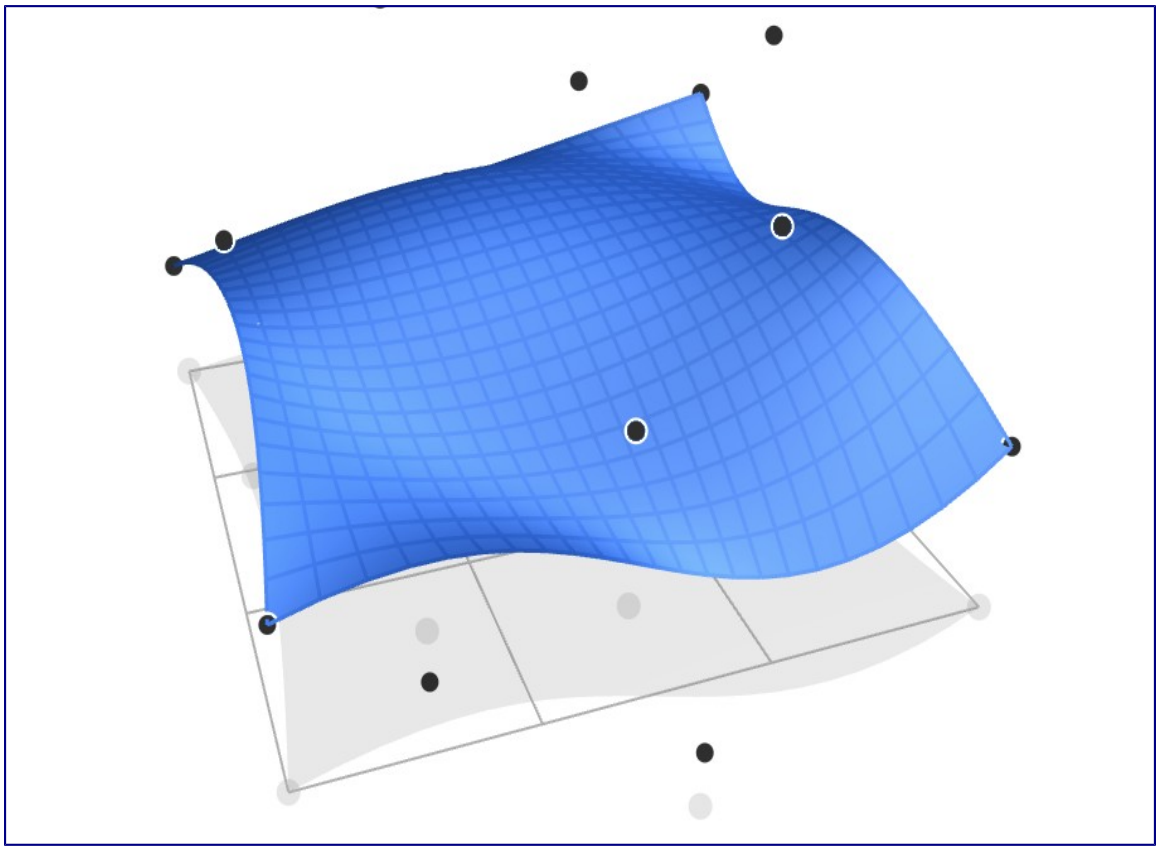
Failed to load latest commit information.

build	Cap camera theta at the poles with an epsilon because of atan2 change…	Sep 9, 2014
examples	Make orthographic camera work	Jun 9, 2014
resources	Default everything to live: true, add quicky architecture diagram, re…	Nov 14, 2012
shaders	Projective viewport	May 25, 2013
src	Cap camera theta at the poles with an epsilon because of atan2 change…	Sep 10, 2014
test	- Fix test bug	Jan 26, 2013
vendor	Dispose compat	Jun 9, 2014
.gitignore	mothereffin' mathbox	Nov 11, 2012
.gitmodules	Texture mapping for surfaces	Jan 4, 2013
CHANGES.md	Director now properly reverses timings of animations, including	Sep 15, 2013

	delays.	
IDEAS.md	Fix typo: Perhapas --> Perhaps	Sep 3, 2013
LICENSE.txt	And I suppose we need one of these	Nov 11, 2012
README.md	Adds 'Building from source' section.	Jul 17, 2014
build.sh	Projective viewport	May 25, 2013

README.md

MathBox



MathBox is a library for rendering presentation-quality math diagrams in a browser using WebGL. Built on top of Three.js and tQuery, it provides a clean API to visualize mathematical relationships and animate them smoothly.

MathBox can be operated manually using a tQuery-like API, or used in presentation mode with the included Director. This lets you to feed in a script of actions and step through them one by one, generating an automatic 'rollback' script to allow you to step backwards.

MathBox was initially created for the conference talk "Making Things with Maths" at Full Frontal 2012 and is still a work in progress. There are a couple of important functionality gaps, such as labelling

being limited to axes and inability to re-use styles easily.

I've only tested thoroughly in Google Chrome, but Firefox seems to work well, with Opera at 90% functionality.

Examples

- Viewport, axis, vectors with labeling
- Bezier surface (slideshow)
- Complex numbers (slideshow)
- Complex exponentiation 4D -> 3D
- Circle equation (x^2 + y^2 = 1) complex 4D -> 3D
- Circle equation (x^2 + y^2 = 1) complex 4D -> 3D (animated projection)
- Distance from a point to a curve
- Surface/plane intersection
- Procedural landscape (slideshow)
- Rational function on the projective line
- Projective viewport transform
- Warping cartesian into cylindrical coordinates
- Texture mapping a surface
- Cross-eyed 3D
- Orthographic camera

Try It Out

You can try MathBox online right away. Simply open any of the examples above, or use this empty template and open the JavaScript console (Chrome: Ctrl-Alt-J / Cmd-Opt-J). You can interact with MathBox using the `mathbox` object and any of the methods documented below, for example:

github.com/unconed/MathBox.js/

```
  color: 0xa0a0a0,
  ticks: 5,
  lineWidth: 2,
  size: .05,
  labels: true,
})
.axis({
  id: 'y-axis',
  axis: 1,
  color: 0xa0a0a0,
  ticks: 5,
  lineWidth: 2,
  size: .05,
  labels: true,
  zero: false,
})
.axis({
  id: 'z-axis',
  axis: 2,
  color: 0xa0a0a0,
  ticks: 5,
  lineWidth: 2,
  size: .05,
  zero: false,
  labels: true,
})
// Grid
.grid({
  id: 'my-grid',
  axis: [0, 2],
  color: 0xc0c0c0,
  lineWidth: 1,
})

// Curve, explicit function
.curve({
  id: 'my-curve',
  domain: [-3, 3],
  expression: function (x) { return Math.cos(x); },
  line: true,
  points: true,
  lineWidth: 2,
})

// Curve, parametric function
.curve({
  id: 'my-circle',
  domain: [-3, 3],
  expression: function (x) { return [Math.sin(x)*2, Math.cos(x)*2]; },
  line: true,
  points: true,
  lineWidth: 2,
})
```

Configuration

All mathbox arguments are optional. The following options are available for configuration in addition to the normal tQuery world options:

```
{
  // Whether to allow mouse control of the camera.
  cameraControls: true,
  // Override the class to use for mouse controls.
  controlClass: ThreeBox.OrbitControls,
  // Whether to show the mouse cursor.
  // When set to false, the cursor auto-hides after a short delay.
  cursor: true,
  // Whether to track resizing of the containing element.
  elementResize: true,
  // Enable fullscreen mode with 'f' (browser support is buggy)
  fullscreen: true,
  // Render at scaled resolution, e.g. scale 2 is half the width/height.
  // Fractional values allowed.
  scale: 1,
  // Enable screenshot taking with 'p'
  screenshot: true,
  // Show FPS stats in the corner
  stats: true,
}
```

Manipulation

By giving objects an ID, you can manipulate them, using CSS-like selectors, i.e. `#id`. IDs must not contain spaces or punctuation other than `-` and _. You can also select all objects of a certain type, e.g. `axis`, `camera`, `grid` or `viewport`.

```
// Get all properties of the X axis object.
var properties = mathbox.get('#x-axis');

// Make all axes red
mathbox.set('axis', { color: 0xff0000 });

// Make Z axis thicker (1000ms animation)
mathbox.animate('#z-axis', { lineWidth: 10 }, { duration: 1000 });

// Clone the Z axis and move it to the side (500ms animation, 200ms delay)
mathbox.clone('#z-axis', { id: 'copy', mathPosition: [0, 1, 0] }, { duration: 500,
delay: 200 });
```

You can inspect the scene by calling:

```
var primitives = mathbox.select('*');
```

which returns an array of matching elements (`*`, `#id` or `type`).

Styles

All MathBox primitives support the following styles, in addition to their specific properties:

```
{
color: 0x123456, // Color in hex
opacity: 1,      // Opacity
lineWidth: 2,    // Line width for curves and wireframes
pointSize: 5,    // Point size for point rendering

map: null,       // Texture Map (pass in THREE.Texture)
mapColor: 0,     // Strength of texture map color (0 - 1)
mapOpacity: 0,   // Strength of texture map mask (0 - 1)

mathScale: [1, 1, 1],    // Scale transform in math-space
mathRotation: [0, 0, 0], // Euler-angle rotation in math-space
mathPosition: [0, 0, 0], // Position shift in math-space

worldScale: [1, 1, 1],    // Scale transform in world-space
worldRotation: [0, 0, 0], // Euler-angle rotation in world-space
worldPosition: [0, 0, 0], // Position shift in world-space

zIndex: 0.0, // Z bias which pushes the object forward (+) or backward (-)
}
```

Styles are grouped under a separate `style` property for each primitive, e.g.

```
mathbox.get('#x-axis').style.color
```

Note however that styles are converted into and stored as Three.js objects, such as `THREE.Color` and `THREE.Vector3`. You can pass these back into MathBox, in fact, the array-based notation for the styles above is just a convenient shorthand.

However, when using `mathbox.set()` and `mathbox.animate()`, you can omit the `style` key and pass in a flat object mixing both styles and options:

```
mathbox.set('#x-axis', { color: 0xff0000, arrow: false });
mathbox.animate('#y-axis', { opacity: 0.5 });
```

Viewports

Each mathbox scene has an associated viewport. This sets up a specific mathematical coordinate grid. Viewports support morphing between various coordinate grids in a mathematically correct way.

The following viewport types are available:

Cartesian Regular linear XYZ.

```
.viewport({
  type: 'cartesian',
  range: [[-1, 1], [-1, 1], [-1, 1]], // Range in X, Y, Z
  scale: [1, 1, 1],                   // Scale in X, Y, Z
```

```
  rotation: [0, 0, 0],                // Viewport rotation in Euler angles
  position: [0, 0, 0],                // Viewport position in XYZ
})
```

Projective Applies a 4x4 homogeneous/projective transform.

```
.viewport({
  type: 'projective',
  range: [[-1, 1], [-1, 1], [-1, 1]], // Range in X, Y, Z
  scale: [1, 1, 1],                   // Scale in X, Y, Z
  rotation: [0, 0, 0],                // Viewport rotation in Euler angles
  position: [0, 0, 0],                // Viewport position in XYZ
  projective: [[1, 0, 0, 0],          // 4x4 projective transform
               [0, 1, 0, 0],
               [0, 0, 1, 0],
               [0, 0, 0, 1]],
})
```

Polar Polar coordinate grid in radians. X is angle, Y is radius, Z is ordinary depth. Also useful for visualizing complex operations in polar representation.

```
.viewport({
  type: 'polar',
  range: [[-π, π], [-1, 1], [-1, 1]], // Range in X, Y, Z (Angle, Radius, Depth)
  polar: 1,                           // Morph between cartesian (0) and polar (1)
  fold:  1,                           // Fold the angles by this factor
  power: 1,                           // Apply this power to the radius
  helix: 0,                           // Separate the complex plane into a helix by
this amount
  scale: [1, 1, 1],                   // Scale in X, Y, Z
  rotation: [0, 0, 0],                // Viewport rotation in Euler angles
  position: [0, 0, 0],                // Viewport position in XYZ
})
```

Sphere Spherical coordinate grid in radians. X is longitude, Y is latitude, Z is radius.

```
.viewport({
  type: 'sphere',
  range: [[-π, π], [-π/2, π/2], [-1, 1]], // Range in X, Y, Z (Longitude, Latitude,
Radius)
  sphere: 1,                              // Morph between cartesian (0) and
spherical (1)
  scale: [1, 1, 1],                       // Scale in X, Y, Z
  rotation: [0, 0, 0],                    // Viewport rotation in Euler angles
  position: [0, 0, 0],                    // Viewport position in XYZ
})
```

Note that for 2D viewports, you can just pass in 2 elements rather than 3 for each vector.

Camera

For 3D graphs, you'll often want to move the camera around. For simplicity, the camera is a simple orbiter that always looks at a particular point in space. It uses spherical coordinates (i.e.

latitude/longitude) to position itself around the point of interest.

By default the camera is positioned to face the X/Y plane at a distance of 3.5 units, which gives the default viewport a slight margin.

```
.camera({
  orbit: 3.5,        // Distance from the center
  phi: τ/4,          // Longitude angle in XZ, in radians, relative to 0 degrees on
the X axis
  theta: 0,          // Latitude angle towards Y, in radians, relative to the XZ
plane.
  lookAt: [0, 0, 0], // Point of focus in space
})
```

Primitives

You can add the following items into the scene by invoking these methods:

Axis

```
.axis({
  axis: 0,           // 0 = X, 1 = Y, 2 = Z
  offset: [0, 0, 0], // Shift axis position
  n: 2,              // Number of points on axis line (set to higher for curved
viewports)
  ticks: 10,         // Approximate number of ticks on axis (ticks are spaced at
sensible units).
  tickUnit: 1,       // Base unit for ticks. Set to π e.g. to space ticks at
multiples of π.
  tickScale: 10,     // Integer denoting the base for recursive division. 2 =
binary, 10 = decimal
  arrow: true,       // Whether to include an arrow on the axis
  size: .07,         // Size of the arrow relative to the stage
})
```

Bezier Curve

```
.bezier({
  n: 64,                        // Number of points
  domain: [0, 1],               // Domain, expressed in interpolation space
  data: null,                   // Array of control points, each an array of 2 or
3 elements
  order: 3,                     // Order of bezier curve (1-3)
  expression: function (x, i) {  // Live expression for data points.
    return 0;                   // Return single value or array of 2/3 elements.
  },
  points: false,                // Whether to draw points
  line: true,                   // Whether to draw lines
})
```

Bezier Surface

```
.bezierSurface({
  n: [ 64, 64 ],                     // Number of points in each direction
```

```
  domain: [[0, 1], [0, 1]],            // X/Y Domain in interpolation space
  data: null,                          // Array of array of control points, each an
array of 2 or 3 elements
  order: 3,                            // (unsupported, must be 3)
  expression: function (x, y, i, j) {  // Live expression for data points.
    return 0;                          // Return single value or array of 2/3
elements.
  },
  points: false,                       // Whether to draw points
  line: false,                         // Whether to draw wireframe lines
  mesh: true,                          // Whether to draw a solid mesh
  doubleSided: true,                   // Whether the mesh is double sided
  flipSided: false,                    // Whether to flip a single sided mesh
  shaded: true,                        // Whether to shade the surface
})
```

Curve

```
.curve({
  n: 64,                          // Number of points
  domain: [0, 1],                 // Input domain
  data: null,                     // Array of data points, each an array of 2 or 3
elements
  expression: function (x, i) {   // Live expression for data points.
    return 0;                     // Return single value or array of 2/3 elements.
  },
  points: false,                  // Whether to draw points
  line: true,                     // Whether to draw lines
})
```

Grid

```
.grid({
  axis: [ 0, 1 ],          // Primary and secondary grid axis (0 = X, 1 = Y, 2 = Z)
  offset: [0, 0, 0],       // Shift grid position
  show: [ true, true ],    // Show horizontal and vertical direction
  n: 2,                    // Number of points on grid line (set to higher for
curved viewports)
  ticks: [ 10, 10 ],       // Approximate number of ticks on axis (ticks are spaced
at sensible units).
  tickUnit: [ 1, 1],       // Base unit for ticks on each axis. Set to π e.g. to
space ticks at multiples of π.
  tickScale: [ 10, 10 ],   // Integer denoting the base for recursive division on
each axis. 2 = binary, 10 = decimal
})
```

Surface

```
.surface({
  n: [ 64, 64 ],                       // Number of points in each direction
  domain: [[0, 1], [0, 1]],            // X/Y Input domain
  data: null,                          // Array of array of data points, each an
array of 2 or 3 elements
  expression: function (x, y, i, j) {  // Live expression for data points.
    return 0;                          // Return single value or array of 2/3
elements.
```

```
  },
  points: false,                          // Whether to draw points
  line: false,                            // Whether to draw wireframe lines
  mesh: true,                             // Whether to draw a solid mesh
  doubleSided: true,                      // Whether the mesh is double sided
  flipSided: false,                       // Whether to flip a single sided mesh
  shaded: true,                           // Whether to shade the surface
})
```

Vector

```
.vector({
  n: 1,                                 // Number of vectors
  data: null,                           // Array of alternating start and end points,
                                        // each an array of 2 or 3 elements
  expression: function (i, end) {       // Live expression for start/end points.
    return 0;                           // Return single value or array of 2/3
elements.
  },
  line: true,                           // Whether to draw vector lines
  arrow: true,                          // Whether to draw arrowheads
  size: .07,                            // Size of the arrowhead relative to the stage
})
```

Building from source

After an initial clone, all submodules will have to be initialized and updated.

```
$ git submodule update --init --recursive
```

Build using

```
$ ./build.sh
```

To build without minification set SKIP_MINIFY

```
$ SKIP_MINIFY=true ./build.sh
```

Or export it to disable it for the rest of the session

```
$ export SKIP_MINIFY=true
```

Contributions

MathBox created by Steven Wittens - http://acko.net/

Contributors:

- So8res (Nate Soares)
- johan (Johan Sundström)
- waldir (Waldir Pimenta)

- fourplusone (Matthias Bartelmeß)
- hugoferreira (Hugo Ferreira)
- EvgenyAgafonchikov (Evgeny Agafonchikov)